Томка Ю.Я.

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ МОВОЮ C#

НАВЧАЛЬНИЙ ПОСІБНИК

КАФЕДРА
КОМП'ЮТЕРНИХ
НАУК

Міністерство освіти і науки України

Чернівецький національний університет імені Юрія Федьковича

Ю.Я. Томка

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ МОВОЮ C#

Навчальний посібник

Видання 2-ге, перероблене і доповнене

видання електронне

Чернівці

ЧЕРНІВЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ЮРІЯ ФЕДЬКОВИЧА

2026

УДК 004.432.2
Т-56

Рекомендовано до друку вченою радою Навчально-наукового інституту фізико-технічних та комп'ютерних наук Чернівецького національного університету імені Юрія Федьковича
(протокол № 10 від 30 червня 2026 р.)

Рецензент:
Литвин В.В., доктор технічних наук, професор, професор кафедри інформаційних систем і мереж Національного університету «Львівська політехніка»
Газдюк К.П., доктор філософії (PhD), доцент, завідувач кафедри програмного забезпечення комп'ютерних систем Чернівецького національного університету імені Юрія Федьковича

Томка Ю.Я.
Т-56 Об'єктно-орієнтоване програмування мовою C# : навч. посіб. / Ю.Я. Томка. – Вид. 2-ге, перероб. і доп. – Чернівці : Чернівецький національний університет імені Юрія Федьковича, 2026. – 1220 с.

У навчальному посібнику в деталях приділено увагу можливостям мови C#, платформи .NET та об'єктно-орієнтованій парадигмі створення коду. Матеріал супроводжується лаконічними практичними прикладами.
Посібник призначений для підготовки здобувачів в галузі знань F "Інформаційні технології" спеціальності F3 "Комп'ютерні науки" та всіх бажаючих.

УДК 004.432.2

©Чернів. нац. ун-т , 2026
©Томка Ю.Я. , 2026

ЗМІСТ

Розділ 1. Вступ....12
1.1. Роль платформи....12
1.2. .NET Framework та сучасний .NET....18
1.3. Керований та некерований код....22
1.4. JIT-компіляція....27
Питання для самоконтролю — Розділ 1. Вступ....34
Розділ 2. Основи програмування на C#....38
2.1. Структура програми....38
2.2. Змінні та константи....43
2.3. Літерали....47
2.4. Типи даних....53
2.5. Консольне введення-виведення....61
2.6. Арифметичні операції....66
2.7. Порозрядні операції....71
2.8. Операції присвоєння....76
2.9. Перетворення базових типів....80
2.10. Явні та неявні перетворення....85
2.11. Умовні вирази....93
2.12. Конструкція if..else та тернарна операція....96
2.13. Цикли....101
2.14. Масиви....107
2.15. Завдання з масивами....113
2.16. Методи....116
2.17. Параметри методів....121
2.18. Повернення значення та оператор return....125
2.19. Передача параметрів за посиланням та значенням. Вихідні параметри....133

2.20. Масив параметрів та ключове слово params 138
2.21. Рекурсивні функції 142
2.22. Локальні функції 147
2.23. Конструкція switch 151
2.24. Перерахування enum 156
Питання для самоконтролю — Розділ 2. Основи програмування на C# 161
Розділ 3. Класи, структури та простір імен 180
3.1. Класи та об'єкти 180
3.2. Конструктори, ініціалізатори та деконструктори 192
3.3. Клас Program та метод Main. Програми верхнього рівня 203
3.4. Структури 209
3.5. Типи значень та типи посилань 217
Питання для самоконтролю — Розділ 3. Класи, структури та простір імен 224
Розділ 4. Об'єктно-орієнтоване програмування 228
4.1. Успадкування 228
4.2. Перетворення типів 240
4.3. Віртуальні методи та властивості 247
4.4. Приховування методів та властивостей 255
4.5. Відмінність перевизначення та приховування методів 260
4.6. Абстрактні класи та члени класів 265
4.7. Клас System.Object та його методи 270
4.8. Узагальнення 275
4.9. Обмеження узагальнень 282
4.10. Наслідування узагальнених типів 291
Питання для самоконтролю — Розділ 4. Об'єктно-орієнтоване програмування 296
Розділ 5. Обробка винятків 306
5.1. Конструкція try..catch..finally 306

5.2. Блок catch та фільтри винятків....312
5.3. Типи винятків. Клас Exception....319
5.4. Генерація винятку та оператор throw....324
5.5. Створення класів винятків....330
5.6. Пошук блоку catch при обробці винятків....335
Питання для самоконтролю — Розділ 5. Обробка винятків....340
Розділ 6. Делегати, події та лямбди....347
6.1. Делегати....347
6.2. Лямбди....364
6.3. Події....370
6.4. Коваріантність та контраваріантність делегатів....382
6.5. Делегати Action, Predicate та Func....389
Питання для самоконтролю — Розділ 6. Делегати, події та лямбди....397
Розділ 7. Інтерфейси....403
7.1. Визначення інтерфейсів....403
7.2. Застосування інтерфейсів....413
7.3. Явна реалізація інтерфейсів....421
7.4. Успадкування інтерфейсів....431
7.5. Інтерфейси в узагальненнях....440
7.6. Копіювання об'єктів. Інтерфейс ICloneable....446
7.7. Коваріантність та контраваріантність узагальнених інтерфейсів....454
Питання для самоконтролю — Розділ 7. Інтерфейси....461
Розділ 8. Додаткові можливості ООП у C#....468
8.1. Визначення операторів....468
8.2. Визначення інкременту та декременту....475
8.3. Перевантаження операцій перетворення типів....481
8.4. Індексатори....486

8.5. Змінні-посилання та повернення посилання 492
8.6. Методи розширення 497
8.7. Часткові класи та методи 502
8.8. Анонімні типи 508
8.9. Кортежі 513
8.10. Records 518
Питання для самоконтролю — Розділ 8. Додаткові можливості ООП у C# 525
Розділ 9. Pattern matching 535
9.1. Патерн типів 535
9.2. Патерн властивостей 543
9.3. Патерни кортежів 548
9.4. Позиційний патерн 552
9.5. Реляційний та логічний патерни 555
9.6. Патерни списків 559
Питання для самоконтролю — Розділ 9. Pattern matching 563
Розділ 10. Колекції 570
10.1. Список List<T> 570
10.2. Двозв'язаний список LinkedList<T> 576
10.3. Черга Queue<T> 583
10.4. Stack<T> 589
10.5. Словник Dictionary<K, V> 595
10.6. Клас ObservableCollection<T> 606
10.7. Інтерфейси IEnumerable<T> та IEnumerator<T> 611
10.8. Ітератори та yield return 618
Питання для самоконтролю — Розділ 10. Колекції 623
Розділ 11. Робота з рядками 631
11.1. Рядки та клас String 631
11.2. Операції з рядками 639

11.3. Форматування та інтерполяція рядків 648
11.4. Клас StringBuilder 655
11.5. Регулярні вирази 664
Питання для самоконтролю — Розділ 11. Робота з рядками 674
Розділ 12. Робота з датами та часом 679
12.1. Структура DateTime 679
12.2. Налаштування формату часу та дати 687
12.3. DateOnly та TimeOnly 694
12.4. Структура TimeSpan 701
12.5. DateTimeOffset та часові пояси 707
Питання для самоконтролю — Розділ 12. Робота з датами та часом 714
Розділ 13. Додаткові класи та структури .NET 719
13.1. Відкладена ініціалізація та тип Lazy<T> 719
13.2. Клас Math 725
13.3. Клас Convert 732
13.4. Клас Array 740
13.5. Span<T> та ReadOnlySpan<T> 747
13.6. Індекси та діапазони (Index та Range) 753
Питання для самоконтролю — Розділ 13. Додаткові класи та структури .NET 758
Розділ 14. Рефлексія 764
14.1. Введення у рефлексію. Клас System.Type 764
14.2. Застосування рефлексії та дослідження типів 773
14.3. Дослідження методів та конструкторів за допомогою рефлексії 782
14.4. Дослідження полів та властивостей за допомогою рефлексії 794
14.5. Динамічне завантаження збірок та пізнє зв'язування 803
14.6. Атрибути у .NET 813
Питання для самоконтролю — Розділ 14. Рефлексія 823

Розділ 15. Багатопоточність 829
15.1. Введення у багатопоточність. Клас Thread....829
15.2. Створення потоків. ThreadStart та ParameterizedThreadStart ..841
15.3. Синхронізація потоків. Оператор lock та клас Lock 847
15.4. Клас Monitor....857
15.5. Клас AutoResetEvent 864
15.6. Клас Mutex 869
15.7. Клас Semaphore та SemaphoreSlim....875
Питання для самоконтролю — Розділ 15. Багатопоточність 881
Розділ 16. Паралельне програмування та TPL....888
16.1. Клас Task. Основи TPL 888
16.2. Вкладені завдання та масиви завдань. Task<T> 896
16.3. Продовження завдань. ContinueWith 901
16.4. Клас Parallel 909
16.5. Скасування завдань. CancellationToken....915
Питання для самоконтролю — Розділ 16. Паралельне програмування та TPL 922
Розділ 17. Асинхронне програмування 928
17.1. async та await. Асинхронні методи 928
17.2. Типи повернення async-методів....933
17.3. Послідовне та паралельне виконання. Task.WhenAll та Task.WhenAny....940
17.4. Обробка помилок в async-методах 948
17.5. Скасування async-операцій. CancellationToken 958
17.6. Асинхронні потоки. IAsyncEnumerable<T> 966
Питання для самоконтролю — Розділ 17. Асинхронне програмування 973
Розділ 18. Робота з файловою системою....980
18.1. Класи Path, File та Directory 980
18.2. Класи FileInfo та DirectoryInfo 988

18.3. Stream та FileStream. Потоковий доступ до файлів.....997
18.4. StreamReader та StreamWriter. Текстові потоки.....1007
18.5. BinaryReader та BinaryWriter. Бінарні потоки.....1016
18.6. System.Text.Json. Серіалізація та десеріалізація.....1024
Питання для самоконтролю — Розділ 18. Файлова система та потоки.....1034
Розділ 19. Серіалізація та десеріалізація. JSON та XML.....1040
19.1. System.Text.Json — поглиблено.....1040
19.2. XML-формат та XmlDocument. DOM-модель.....1052
19.3. XDocument та LINQ to XML.....1060
19.4. XmlReader та XmlWriter. Потоковий XML.....1068
19.5. XmlSerializer — атрибутна серіалізація.....1078
19.6. XPath та вибір між JSON і XML.....1087
Питання для самоконтролю — Розділ 19. Серіалізація JSON та XML.....1096
Розділ 20. SOLID Принципи.....1102
20.1. Вступ до SOLID. Проблеми без архітектурних принципів..1102
20.2. SRP — Принцип єдиної відповідальності.....1109
20.3. OCP — Принцип відкритості/закритості.....1119
20.4. LSP — Принцип підстановки Ліскова.....1130
20.5. ISP — Принцип розподілу інтерфейсів.....1141
20.6. DIP — Принцип інверсії залежностей.....1150
Питання для самоконтролю — Розділ 20. Принципи SOLID.....1162
Розділ 21. Generic Host та Dependency Injection.....1169
21.1. Generic Host — архітектура та життєвий цикл додатку.....1169
21.2. IServiceCollection — реєстрація та розпізнавання сервісів..1180
21.3. Часи життя сервісів — Singleton, Scoped, Transient.....1192
21.4. Options Pattern — типізована конфігурація з IOptions<T>.....1202

Питання для самоконтролю — Розділ 21. Generic Host та Dependency Injection.....1213

Розділ 1. Вступ

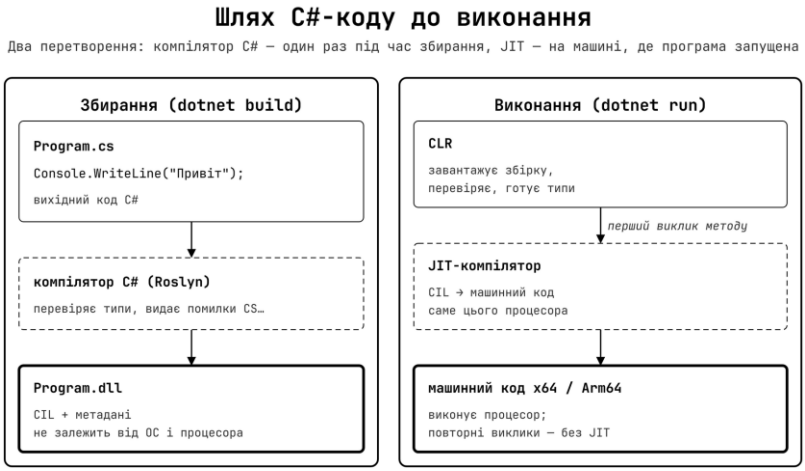
1.1. Роль платформи

Коли говорять про C#, майже завжди поруч з'являється слово .NET. Це не випадково: **C# — це мова програмування**, а **.NET — це платформа**, на якій ця мова найчастіше використовується. Мова описує синтаксис, типи, класи, методи, оператори та правила написання програм. Платформа дає середовище виконання, бібліотеки, інструменти збирання, систему пакетів і набір технологій для різних типів застосунків.

Тому C# і .NET — не одне й те саме. C# — це мова. .NET — це екосистема, у якій ця мова розкривається найповніше. Порівняти можна так: C# — це мова книги, а .NET — бібліотека, видавництво і типографія разом.

Шлях коду від рядка до виконання

Код, написаний програмістом, не виконується процесором безпосередньо. Він проходить кілька перетворень:



Шлях C#-коду від вихідного файлу до машинного коду

1. **Компілятор C#** перетворює .cs-файл у проміжне подання — **CIL** (Common Intermediate Language) разом із метаданими. Результат зберігається у збірці (.dll або .exe).
2. **CLR** (Common Language Runtime) завантажує збірку під час запуску.
3. **JIT-компілятор** (Just-In-Time) перетворює CIL на машинний код конкретного процесора — саме в момент першого виклику методу.

Така архітектура дає дві переваги одночасно: **кросплатформовість** (одна збірка запускається на Windows, Linux і macOS) і **продуктивність** (машинний код адаптований до конкретного процесора).

Перший рядок C#, який пише кожен студент:

```
C#
1  using System;
2
3  Console.WriteLine("Привіт, .NET!");
4  Console.WriteLine($"Платформа:
   {System.Runtime.InteropServices.RuntimeInformation.FrameworkDescription}");
```

— вже використовує бібліотеку платформи. Тип Console не є частиною синтаксису мови; це клас стандартної бібліотеки .NET.

Що дає .NET програмісту

.NET бере на себе роботу, яку в низькорівневих мовах програміст змушений контролювати самостійно. Платформа не просто запускає програму — вона організовує її виконання:

Що робить .NET	Що це означає на практиці
Автоматичне керування пам'яттю (GC)	Не потрібно вручну виділяти і звільняти пам'ять для об'єктів
JIT-компіляція	Код оптимізується під конкретний процесор під час запуску
Перевірка типів	Багато помилок виявляється ще на етапі компіляції
Єдина модель винятків	Помилки обробляються уніфіковано через try...catch

Що робить .NET	Що це означає на практиці
Стандартна бібліотека	Готові класи для рядків, колекцій, файлів, мережі, JSON тощо
Кросплатформовість	Один і той самий код працює на Windows, Linux, macOS

Для студента це означає просту річ: **на початку курсу можна зосередитися на логіці програми, класах, методах і структурі коду**, а не на ручному виділенні пам'яті та деталях роботи з операційною системою.

Підтримка кількох мов

CLR не прив'язана лише до C#. На платформі .NET також використовують **F#**, **Visual Basic** та інші мови, які компілюються у спільне проміжне подання CIL. Завдяки цьому різні мови можуть:

- використовувати одні й ті самі бібліотеки .NET;
- взаємодіяти в межах одного рішення;
- покладатися на єдиний runtime і GC.

Наприклад, одна частина великої системи може бути написана на C#, інша — на F#, а спільні бібліотеки — використовуватися обома. У реальних проєктах такий підхід зустрічається, коли різні команди обирають мову під конкретні задачі.

Кросплатформовість

Сучасний .NET є кросплатформовим. Застосунки на C# можна створювати для Windows, Linux, macOS, хмарних середовищ, контейнерів, мобільних і вебсценаріїв. Звісно, не кожна технологія працює всюди: WPF і Windows Forms орієнтовані на Windows, тоді як ASP.NET Core, консольні застосунки та серверні рішення запускаються на будь-якій підтримуваній ОС.

Для навчального курсу це важливо практично: студент може писати код у Windows, Linux або macOS, використовувати .NET CLI, запускати консольні програми й працювати без прив'язки до конкретної IDE.

Стандартна бібліотека класів

Одна з найсильніших сторін .NET — велика стандартна бібліотека. У ній уже є готові типи для роботи з рядками, датами, колекціями, файлами, потоками, мережею, JSON, базами даних, асинхронністю та багатьма іншими задачами.

```C#
using System;
using System.Collections.Generic;

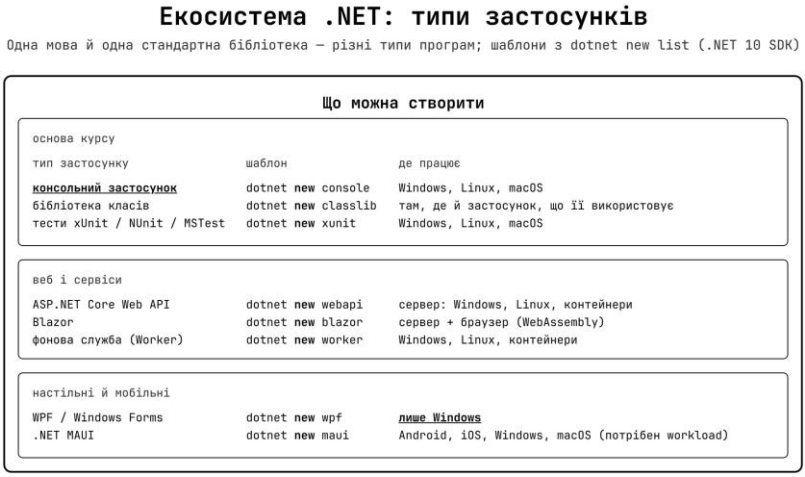
// Рядки
string name = "Іван Петренко";
Console.WriteLine(name.ToUpper());
Console.WriteLine(name.Length.ToString());

// Дати
DateTime today = DateTime.Today;
Console.WriteLine(today.ToString("dd.MM.yyyy"));

// Колекції
List<string> diagnoses = new List<string> { "Гіпертонія", "Бронхіт", "Діабет" };
Console.WriteLine($"Діагнозів: {diagnoses.Count.ToString()}");
foreach (string d in diagnoses)
    Console.WriteLine($"  – {d}");
```

Всі ці можливості — string, DateTime, List<T> — не є частиною синтаксису мови C#; це класи стандартної бібліотеки .NET.

Різноманітність технологій



Екосистема .NET: типи застосунків

На .NET можна створювати різні типи програм:

— **Консольні застосунки** — початок будь-якого навчання; дозволяють зосередитися на логіці

— **Бібліотеки класів** — код, який використовують інші проєкти

— **ASP.NET Core** — вебзастосунки та REST API

— **Blazor** — інтерактивні вебінтерфейси на C# замість JavaScript

— **WPF / Windows Forms** — настільні програми для Windows

— **.NET MAUI** — кросплатформові мобільні та настільні застосунки

— **Worker Service** — фонові служби та мікросервіси

У цьому курсі ми починаємо з консольних застосунків — вони дозволяють не відволікатися на інтерфейс і зосередитися на головному: типах, об'єктах, класах, методах, інкапсуляції, наслідуванні та поліморфізмі.

Три рівні, які важливо розрізняти

У процесі вивчення C# і .NET важливо чітко розуміти, до якого рівня належить та чи інша річ:

4. **Синтаксис C#** — if, for, class, new, override, оператори, лямбди, LINQ
5. **Стандартна бібліотека .NET** — Console, List<T>, DateTime, Math, File
6. **Архітектура власної програми** — класи, що ви пишете самостійно: Patient, Doctor, Clinic

Плутанина між цими рівнями — поширена помилка на початку. Наприклад, питання «чому в int є методи, як у класу?» стосується рівня платформи (у .NET int є псевдонімом типу System.Int32, у якого є методи ToString, Parse тощо), а не синтаксису мови. При цьому System.Int32 — не клас, а **структура** (значимий тип); різницю між ними розглянемо в розділі 3.

Автоматичне керування пам'яттю

Важлива особливість .NET — автоматичне збирання сміття (garbage collection, GC). Коли програма створює об'єкти, вони займають пам'ять у купі. Коли об'єкти більше не використовуються — CLR може звільнити цю пам'ять автоматично, без участі програміста.

```C#
using System;

// Об'єкти створюються в купі
string s1 = new string('A', 100);
string s2 = new string('B', 100);

Console.WriteLine($"s1 довжина: {s1.Length.ToString()}");
Console.WriteLine($"s2 довжина: {s2.Length.ToString()}");

// Примусовий виклик GC — лише для демонстрації, у реальному коді не потрібен
GC.Collect();
Console.WriteLine("GC викликано — непотрібна пам'ять звільнена автоматично.");
```

Це не означає, що про ресурси взагалі не треба думати. Файли, мережеві підключення та підключення до баз даних все одно потрібно закривати коректно. Але більшість повсякденної роботи з пам'яттю для об'єктів бере на себе платформа.

Для порівняння: у C++ програміст сам відповідає за те, коли виділити пам'ять (malloc/new) і коли її звільнити (free/delete). У C# типовий навчальний код виглядає простіше, бо CLR контролює життєвий цикл керованих об'єктів.

Підсумок

— **C#** — мова програмування з багатою системою типів, ООП, LINQ, async/await

— **.NET** — платформа: середовище виконання (CLR), стандартна бібліотека та екосистема технологій

— **CLR** виконує керований код: JIT-компілює CIL, керує пам'яттю через GC, обробляє винятки

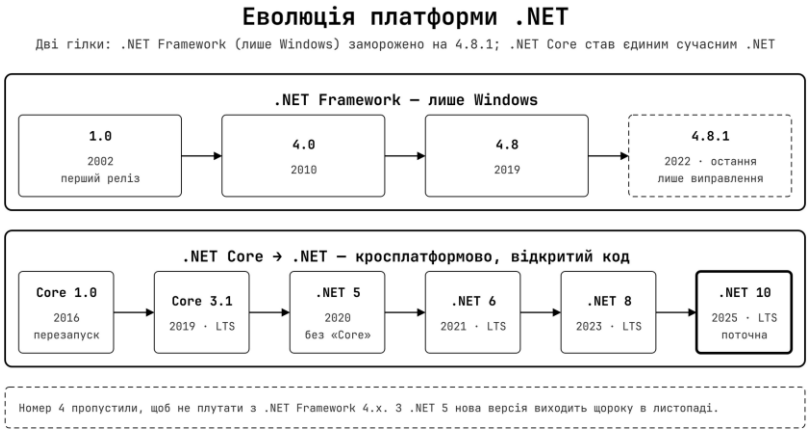
— **Стандартна бібліотека** дає готові класи для типових задач — рядки, колекції, файли, мережа

— Для цього курсу .NET — робоче середовище, у якому ми поступово переходимо від простих інструкцій до повноцінного об'єктно-орієнтованого проєктування

1.2. .NET Framework та сучасний .NET

Коли кажуть «.NET», мають на увазі різні речі залежно від контексту. Щоб не плутатися, важливо розуміти, як платформа розвивалася і чим відрізняються її гілки.

Еволюція платформи



Еволюція платформи .NET від Framework до сучасного .NET

.NET Framework — класична Windows-платформа

.NET Framework — це перша і найстаріша гілка платформи, що з'явилася у 2002 році. Вона тісно пов'язана з Windows і досі використовується у великих корпоративних системах, старих вебзастосунках на ASP.NET і настільних програмах на Windows Forms або WPF.

Ключові характеристики .NET Framework:

— **Тільки Windows** — не запускається на Linux чи macOS

— Тісна інтеграція з реєстром і API операційної системи

— Остання версія — 4.8.x; нові функції C# і runtime більше не додаються

— Залишається підтримуваною для сумісності, але не є напрямом нового розвитку

Якщо в організації є велика програма, написана багато років тому на .NET Framework, — вона цілком може й далі на ньому працювати. Але для нових проєктів .NET Framework обирати не варто.

.NET Core — кросплатформовий перезапуск

Щоб зробити платформу відкритою і кросплатформовою, Microsoft у 2016 році запустила **.NET Core** — нову гілку з нуля. Вона могла запускатися на Windows, Linux і macOS, мала відкритий вихідний код і краще підходила для сучасних серверних і хмарних застосунків.

.NET Core пройшов версії 1, 2, 3 і у 2020 році злився із основною гілкою під назвою **.NET 5** — без слова «Core». Саме тоді дві розрізнені платформи об'єдналися в одну.

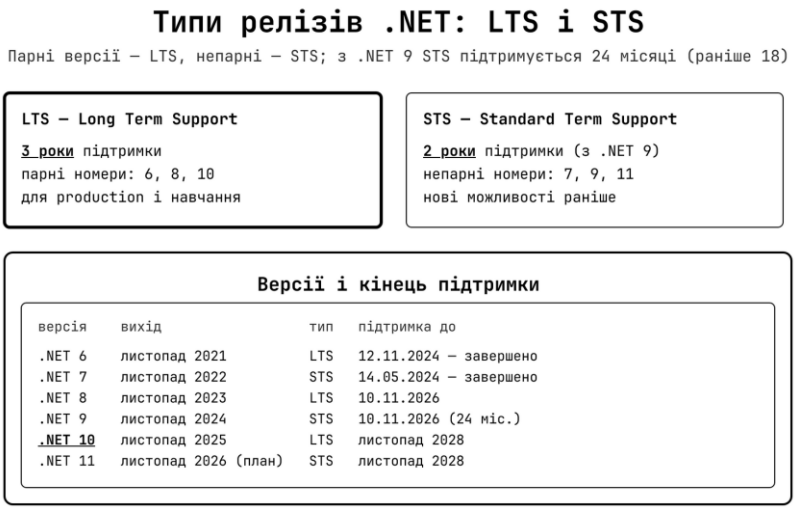
Сучасний .NET — єдина платформа

Починаючи з .NET 5, платформа розвивається як єдина кросплатформова екосистема. Версії виходять щороку в листопаді:

Версія	Рік	Тип	Статус
.NET 5	2020	STS	підтримку завершено (травень 2022)
.NET 6	2021	LTS	підтримку завершено (листопад 2024)

Версія	Рік	Тип	Статус
.NET 7	2022	STS	підтримку завершено (травень 2024)
.NET 8	2023	LTS	підтримується до 10 листопада 2026
.NET 9	2024	STS	підтримується до 10 листопада 2026
.NET 10	2025	LTS	**поточна LTS**

LTS та STS релізи



Типи релізів .NET: LTS та STS

.NET має два типи релізів:

— **LTS (Long Term Support)** — версії з довшим строком підтримки (3 роки). Парні номери: 6, 8, 10. Підходять для стабільних навчальних і production-проєктів.

— **STS (Standard Term Support)** — версії з коротшим строком підтримки. Непарні номери: 7, 9, 11. Швидше приносять нові можливості. До .NET 8 включно STS підтримувалися 18 місяців; у вересні 2025 року Microsoft подовжила строк до **24 місяців**, починаючи з .NET 9. Тому .NET 9 і .NET 8 завершують підтримку в один день — 10 листопада 2026 року.

Для навчального курсу: орієнтуйтеся на актуальну LTS-версію. Станом на 2026 рік це **.NET 10**.

SDK vs Runtime

Для розробки потрібен не лише runtime, а **SDK**:

	Runtime	**SDK**
Що вміє	Запускати готові програми	Компілювати, збирати, тестувати, публікувати
Для кого	Кінцевий користувач	Розробник
Що включає	CLR, бібліотеки	Runtime + компілятор + CLI + шаблони

Перевірити встановлену версію SDK:

```
Bash
1  dotnet --version
```

Переглянути всі встановлені SDK:

```
Bash
1  dotnet --list-sdks
```

Створити і запустити консольний проєкт:

```
Bash
1  dotnet new console -n MyCourse
2  cd MyCourse
3  dotnet run
```

Це важливо: .NET — не тільки Visual Studio. Працювати з C# можна через командний рядок на будь-якій операційній системі.

Що обирати для нових проєктів

Для нових проєктів — завжди сучасний .NET. .NET Framework має сенс лише тоді, коли треба підтримувати стару Windows-систему або технологію, що прив'язана виключно до цієї гілки.

У цьому курсі під словом «.NET» мається на увазі сучасна кросплатформова платформа. Якщо в тексті згадується .NET Framework — це лише для пояснення різниці між гілками або для розуміння спадщини старих систем.

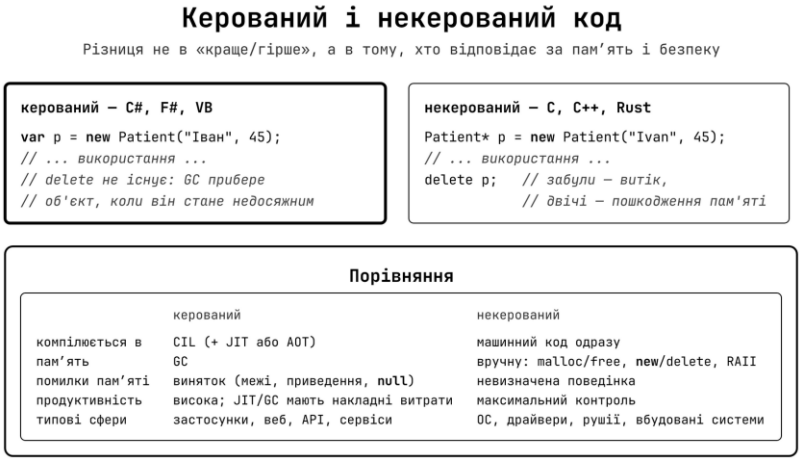
Підсумок

	.NET Framework	Сучасний .NET
Платформи	Тільки Windows	Windows, Linux, macOS
Розвиток	Заморожено на 4.8	Активний, щороку нова версія
Відкритий код	Частково	Повністю (GitHub)
Нові проєкти	Не рекомендується	✓
Поточна LTS	—	.NET 10

1.3. Керований та некерований код

Програму, написану на C# для .NET, називають **керованим кодом** (managed code). Це означає, що її виконання відбувається під контролем CLR — Common Language Runtime. CLR не просто запускає програму: вона завантажує збірки, перевіряє типи, виконує JIT-компіляцію, керує пам'яттю, обробляє винятки і надає інші служби виконання.

Порівняння: керований vs некерований



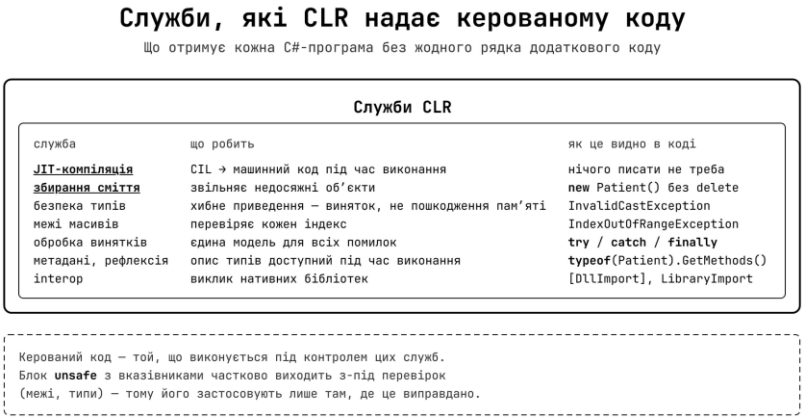
Порівняння керованого та некерованого коду

Керований код — це код, що компілюється у проміжну мову CIL і виконується під контролем CLR. До керованих мов .NET належать C#, F# і Visual Basic.

Некерований код — це код, виконання якого CLR не контролює. Типові приклади — програми та бібліотеки на C, C++ або Rust, що компілюються безпосередньо у машинний код для конкретної платформи.

Це не означає, що некерований код «гірший». Він необхідний для системного програмування, драйверів, ігрових рушіїв, високопродуктивних бібліотек і роботи з апаратним забезпеченням. Але він вимагає більше дисципліни: програміст сам відповідає за виділення і звільнення пам'яті, безпеку вказівників і управління ресурсами.

Що CLR дає керованому коду



Служби, які CLR надає керованому коду

Керований C#-код автоматично отримує від CLR повний набір служб:

— **JIT-компіляція** — CIL перетворюється у машинний код під конкретний процесор під час виконання

— **Garbage Collection** — CLR автоматично звільняє пам'ять від об'єктів, що більше не використовуються

— **Перевірка типів** — CLR перевіряє коректність операцій над типами; некоректне приведення спричиняє виняток, а не корупцію пам'яті

— **Обробка винятків** — єдина модель try/catch/finally для будь-яких помилок виконання

— **Метадані та рефлексія** — код може аналізувати власну структуру під час виконання

— **Interop** — керований код може викликати нативні бібліотеки через P/Invoke

```C#
using System;

// Демонстрація: CLR автоматично керує пам'яттю
// Об'єкти створюються у купі, GC прибирає їх коли вони більше недоступні

string s = new string('X', 50);
```

```
 7  Console.WriteLine($"Рядок: {s.Substring(0, 10)}...");
 8  Console.WriteLine($"Довжина: {s.Length.ToString()}");
 9
10  // Після виходу зі scope — GC може звільнити пам'ять автоматично
11  // У C++ тут потрібен був би explicit delete або RAII
12
13  long memBefore = GC.GetTotalMemory(false);
14  Console.WriteLine($"Пам'ять CLR: {memBefore.ToString()} байт");
```

Порівняння з C++

Щоб краще відчути різницю, порівняємо концептуально роботу з пам'яттю:

```
C#
 1  using System;
 2
 3  // C#: пам'ять керується CLR автоматично
 4  Patient p = new Patient("Іван Петренко", 45);
 5  Console.WriteLine(p.ToString());
 6  // Після виходу зі scope — GC прибере об'єкт, коли він стане недосяжним
 7
 8  class Patient
 9  {
10      public string Name { get; }
11      public int Age { get; }
12      public Patient(string name, int age) { Name = name; Age = age; }
13      public override string ToString() => $"Пацієнт: {Name}, {Age} р.";
14  }
```

У C++ еквівалентний код виглядав би так (псевдокод — не C#):

```
1  Patient* p = new Patient("Ivan", 45);  // виділення у купі
2  // ... використання ...
3  delete p;   // ОБОВ'ЯЗКОВО звільнити — інакше витік пам'яті
```

У C# рядок delete просто не існує. CLR відстежує, коли об'єкт стає недосяжним, і прибирає його автоматично. Для студента це означає, що в безпечному (не unsafe) коді C# помилка «звернення до звільненої пам'яті» неможлива в принципі, а витік через «забутий delete» — теж. Проте витік пам'яті в C# усе ж можливий в іншій формі: якщо на непотрібний об'єкт залишається посилання (у статичній колекції, у непідписаному обробнику події), GC вважає його досяжним і не прибирає. GC звільняє лише **недосяжні** об'єкти — а про досяжність відповідає програміст.

Взаємодія з некерованим кодом (Interop)

.NET не ізольований від зовнішнього світу. Керований C#-код може викликати функції з нативних бібліотек (.dll на Windows, .so на Linux) через механізм **P/Invoke** (Platform Invocation Services):

```
C#
1  // Виклик нативної Windows API функції з C#
2  [System.Runtime.InteropServices.DllImport("user32.dll")]
3  static extern int MessageBox(IntPtr hWnd, string text, string caption, int
   type);
```

При переході між керованим і некерованим кодом важливо правильно описати типи і правила виклику. Це відповідальна зона: помилки тут можуть бути складнішими для діагностики, ніж звичайні винятки C#. Для початку курсу interop не потрібен — він з'являється лише у специфічних сценаріях.

Unsafe-код у C#

C# дозволяє писати **небезпечний код** (unsafe code) — ділянки, де можна працювати з вказівниками безпосередньо. Такий код позначається ключовим словом unsafe:

```
C#
1  unsafe
2  {
3      int x = 42;
4      int* ptr = &x;          // вказівник на змінну
5      Console.WriteLine(*ptr); // розіменування
6  }
```

Unsafe-код залишається частиною .NET-програми, але виходить за межі звичайної безпечної моделі CLR. Він потрібен лише для дуже специфічних сценаріїв: high-performance буферів, взаємодії з апаратурою, оптимізацій на рівні байтів.

На початку навчання unsafe-код не потрібен. Навпаки — краще спершу добре засвоїти типи, об'єкти, класи, методи й колекції. До низькорівневих можливостей варто переходити лише тоді, коли зрозуміло, яку конкретну проблему вони розв'язують.

Підсумок

	Керований код (C#)	Некерований код (C/C++)
Виконання	Під контролем CLR	Безпосередньо ОС
Пам'ять	Автоматично (GC)	Вручну (malloc/free)
Безпека типів	Гарантується CLR	Відповідальність програміста
Продуктивність	Висока (JIT-оптимізації)	Максимальна (нативний код)
Типові сфери	Прикладна розробка, веб, API	Системне ПЗ, драйвери, рушії

> Для цього курсу ми працюємо виключно з керованим кодом C# — він є природним середовищем для вивчення об'єктно-орієнтованого програмування і не вимагає турботи про деталі керування пам'яттю.

1.4. JIT-компіляція

Код C#, написаний програмістом, не виконується процесором безпосередньо. Він проходить два окремих етапи перетворення, перш ніж стати машинними інструкціями. Розуміння цих етапів допомагає пояснити, чому .NET поєднує кросплатформовість з високою продуктивністю.

Два кроки до виконання

Крок 1 — компілятор C# (під час розробки):

Файл .cs обробляє компілятор і перетворює його на **CIL** (Common Intermediate Language) — проміжну мову платформи. Результат зберігається у збірці: файл .dll або .exe. CIL — це не машинний код жодного конкретного процесора; це платформонезалежні інструкції, зрозумілі CLR.

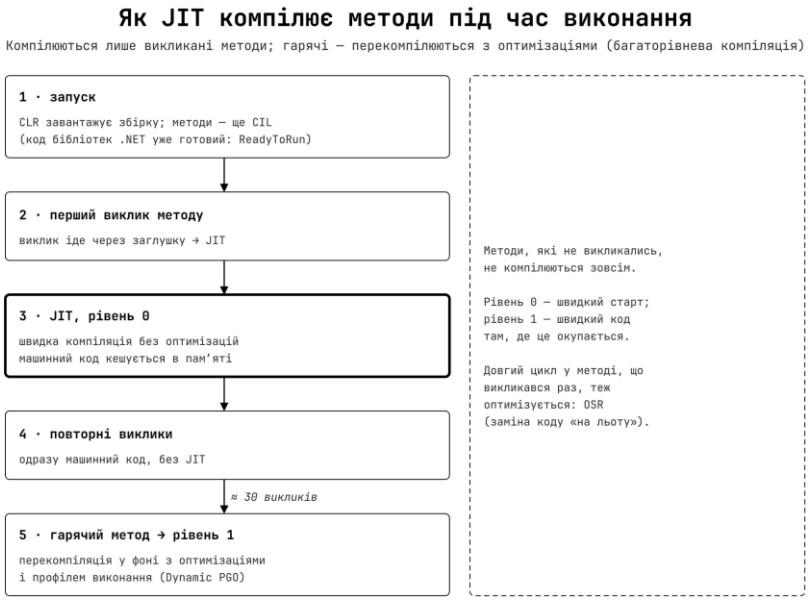
Разом із CIL збірка містить **метадані** — повний опис типів, методів, полів, параметрів. Завдяки метаданим CLR розуміє структуру коду без потреби в заголовкових файлах (на відміну від C++).

Крок 2 — JIT-компілятор (під час виконання):

JIT (Just-In-Time) — «саме вчасно». Коли програма запущена і певний метод викликається **вперше**, JIT-компілятор перетворює його CIL на машинний код конкретного процесора. Скомпільований код кешується: при повторних викликах він виконується напряму, без повторної компіляції.

У сучасному .NET (з .NET Core 3.0) цей процес **багаторівневий** (tiered compilation). Перший раз JIT компілює метод швидко, майже без оптимізацій (рівень 0) — щоб програма стартувала якнайшвидше. Якщо метод виявляється «гарячим» (приблизно після 30 викликів), він перекомпілюється у фоні з повними оптимізаціями (рівень 1), а з .NET 8 — ще й з урахуванням зібраного під час роботи профілю (Dynamic PGO); наступні виклики йдуть уже в оптимізований код. Методи з довгими циклами, які викликаються рідко, оптимізуються «на льоту» через заміну коду посеред виконання (OSR). Крім того, бібліотеки самого .NET постачаються в форматі **ReadyToRun** — з уже готовим машинним кодом, тому значна частина коду при старті взагалі не проходить через JIT.

Покроковий процес JIT



Як JIT компілює методи під час виконання

Ключовий момент: JIT не компілює всю програму при запуску. Він компілює лише ті методи, які **фактично викликаються** під час конкретного сеансу роботи. Методи, до яких програма не звертається, залишаються у вигляді CIL і не витрачають час на компіляцію.

Демонстрація: що відбувається при запуску

```C#
using System;

Console.WriteLine("Крок 1: програма запустилась.");
Console.WriteLine("Крок 2: CLR завантажила збірку.");
Console.WriteLine("Крок 3: JIT скомпілював Main при першому виклику.");

SayHello();  // JIT компілює SayHello при першому зверненні
SayHello();  // вже виконується кешований машинний код — без JIT

void SayHello()
{
    Console.WriteLine("Привіт від JIT-скомпільованого методу!");
}
```

```
14
15  // UnusedMethod існує в збірці, але JIT її не чіпає —
16  // бо вона не викликається під час цього запуску
17  void UnusedMethod()
18  {
19      Console.WriteLine("Цей метод не буде JIT-скомпільований.");
20  }
```

Перший виклик SayHello() запускає JIT-компіляцію цього методу. Другий виклик — вже йде напряму у нативний код (рівня 0; до рівня 1 метод дійде, лише якщо його викликатимуть часто).

CIL і метадані — що всередині збірки

Збірка .dll — це не просто набір інструкцій. Вона містить дві принципово різні речі:

Що	Для чого
CIL-інструкції	Платформонезалежний проміжний код, що JIT перетворює у машинний
Метадані	Опис типів, методів, полів, параметрів, атрибутів

Метадані — це те, що дозволяє C# мати **рефлексію**: програма може запитати у CLR «які методи є у цьому класі?» прямо під час виконання:

```
C#
 1  using System;
 2
 3  // Рефлексія: читаємо метадані власного класу під час виконання
 4  Type t = typeof(Patient);
 5  Console.WriteLine($"Клас: {t.Name}");
 6  Console.WriteLine("Методи:");
 7  foreach (var m in t.GetMethods())
 8      Console.WriteLine($"  {m.Name}");
 9
10  class Patient
11  {
12      public string Name { get; set; } = "";
13      public int Age { get; set; }
14      public Patient(string name, int age) { Name = name; Age = age; }
15      public override string ToString() => $"{Name}, {Age} p.";
16      public void Register() => Console.WriteLine($"{Name} зареєстрований.");
17  }
```

Це можливо саме завдяки метаданим у збірці. У мовах без керованого середовища (C, C++) такої можливості за замовчуванням немає.

Переваги JIT-підходу

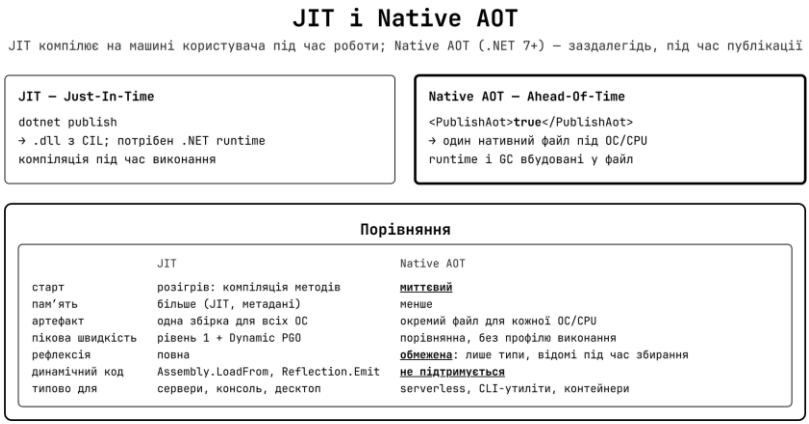
— **Кросплатформовість**: одна і та сама збірка запускається на будь-якій ОС, де є .NET runtime

— **Адаптація до CPU**: JIT знає, на якому процесорі запускається код, і може використовувати специфічні інструкції (AVX, SSE)

— **Ощадливість**: компілюється лише код, що реально виконується

— **Оптимізації runtime**: JIT бачить реальну поведінку програми — які типи насправді приходять у віртуальні виклики, які гілки виконуються частіше. Завдяки Dynamic PGO він може, наприклад, девіртуалізувати виклик для найчастішого типу й вбудувати (inline) його тіло. Сам inlining доступний і статичним компіляторам (C++, Native AOT); перевага JIT — у рішеннях на основі профілю виконання

JIT vs AOT



Порівняння JIT та AOT компіляції

Окрім JIT, у сучасному .NET існує **AOT** (Ahead-Of-Time) — компіляція наперед, ще до запуску:

— **JIT**: збірка з CIL розповсюджується, машинний код генерується на машині користувача під час виконання

— **AOT**: нативний бінарний файл генерується заздалегідь; при запуску нічого компілювати не треба

AOT корисний для:

— **AWS Lambda / хмарні функції** — де важливий час холодного старту

— **.NET MAUI і iOS** — де JIT взагалі заборонений платформою

— **Blazor WebAssembly** — де код виконується у браузері (за замовчуванням там працює інтерпретатор IL; AOT-компіляція у WebAssembly вмикається окремо і пришвидшує обчислення ціною більшого розміру завантаження)

— **CLI-утиліт** — де потрібен один нативний бінарний файл без залежностей

Для більшості серверних і консольних застосунків JIT є стандартним і оптимальним вибором. AOT потрібен у специфічних сценаріях і вносить певні обмеження (наприклад, на рефлексію).

NativeAOT — нативний бінарний файл без JIT

NativeAOT (.NET 7+) — спеціальний режим публікації, що компілює весь застосунок у **нативний машинний код** заздалегідь, ще до розгортання. Результат — єдиний виконуваний файл без залежності від встановленого .NET runtime на машині користувача. Саме середовище виконання (GC, обробка винятків, типи) нікуди не зникає — воно вбудовується у файл; відсутній лише JIT. NativeAOT для консольних застосунків з'явився в .NET 7, для ASP.NET Core — у .NET 8.

Щоб опублікувати застосунок у режимі NativeAOT:

```xml
<!-- У файлі .csproj -->
<PropertyGroup>
  <PublishAot>true</PublishAot>
</PropertyGroup>
```

```bash
dotnet publish -r win-x64 -c Release
# або
dotnet publish -r linux-x64 -c Release
```

Результат — один нативний .exe (на Windows) або бінарний файл (на Linux), що запускається без встановленого .NET.

Показник	JIT	NativeAOT
Час старту	Десятки–сотні мс (JIT-розігрів; залежить від обсягу коду)	**Одиниці–десятки мс**
Пам'ять	Більше (CLR, JIT-інфраструктура)	**Значно менше**
Розмір артефакту	Мала збірка + великий runtime	Один великий файл
Рефлексія	Повна підтримка	**Обмежена** (потрібні Source Generators)
Динамічний код	Assembly.LoadFrom, Reflection.Emit	**Не підтримується**; Activator.CreateInstance — лише для типів, відомих під час збирання
Платформа	Кросплатформово (одна збірка)	Окремий бінарний для кожної ОС/CPU

Коли NativeAOT вигідний:

— **Мікросервіси і serverless** (AWS Lambda, Azure Functions): час холодного старту критичний

— **CLI-утиліти**: один файл без залежностей, зручно розповсюджувати

— **IoT та вбудовані системи**: обмежена пам'ять, відсутність .NET runtime

— **Мобільні платформи (iOS)**: JIT заборонений платформою

Обмеження NativeAOT: NativeAOT не може компілювати код, що залежить від рефлексії з невідомими типами. Бібліотеки, що через рефлексію перебирають властивості й методи довільних типів під час виконання (класичні серіалізатори, DI-контейнери), потребують адаптації: компілятор AOT і тримінг видаляють код, використання якого не видно під час збирання. Саме тому JSON Source Generators (19.1) і інші генератори розроблені як AOT-сумісна альтернатива до рефлексивного підходу.

Підсумок

Поняття	Суть
CIL	Проміжна мова; результат роботи компілятора C#
Збірка (.dll/.exe)	CIL + метадані; кросплатформовий артефакт розгортання
Метадані	Опис типів і методів; основа рефлексії та перевірки типів
JIT	Перетворює CIL у машинний код під час першого виклику методу
Кеш JIT	Скомпільований код зберігається; повторні виклики — без JIT
AOT	Альтернатива JIT: компіляція до запуску; нативний бінарний файл
NativeAOT	AOT-режим .NET 7+: один нативний файл, швидкий старт, без JIT і без встановленого runtime (runtime вбудовано у файл)

> C#-код проходить шлях: .cs → компілятор → CIL у збірці → JIT при виконанні → машинний код процесора. Завдяки цьому ланцюжку .NET поєднує зручність високорівневої мови, кросплатформовість і продуктивність, близьку до нативного коду. NativeAOT — альтернативний шлях для сценаріїв, де JIT-розігрів неприйнятний.

Питання для самоконтролю — Розділ 1. Вступ

1.1. Роль платформи

1. Що таке C# і що таке .NET? Чи є це одне й те саме? Поясніть, чому мову програмування і платформу важливо розрізняти.
2. Які три кроки проходить код C# від вихідного файлу до виконання на процесорі? Опишіть кожен крок і назвіть його результат.

3. Клас Console — це частина синтаксису C# чи частина стандартної бібліотеки .NET? Обґрунтуйте свою відповідь.
4. Поясніть, що таке три рівні розуміння коду: синтаксис мови, стандартна бібліотека і власна архітектура програми. Наведіть по одному прикладу для кожного рівня.
5. Чому CIL-збірка, скомпільована на Windows, може виконуватися на Linux без перекомпіляції? Яка частина .NET забезпечує цю можливість?
6. Порівняйте керування пам'яттю в C# і в C++. Що робить GC і чому наявність GC не означає, що про ресурси взагалі не треба думати?
7. На .NET можна розробляти програми різних типів: консольні, вебзастосунки, мобільні. Чому у навчальному курсі починають саме з консольних застосунків?
8. Виконайте в уяві такий код і поясніть, що відбудеться:

```
C#
1  GC.Collect();
2  Console.WriteLine("Готово");
```

Чи потрібно викликати GC.Collect() у звичайному коді? Обґрунтуйте.

1.2. .NET Framework та сучасний .NET

9. Що таке .NET Framework і чим він відрізняється від сучасного .NET? Назвіть як мінімум три принципові відмінності.
10. Яка подія у 2020 році об'єднала .NET Core і .NET Framework у єдину платформу? Що змінилося після цього злиття?
11. Чому для нових проєктів не рекомендується обирати .NET Framework, навіть якщо він підтримується й досі?
12. Поясніть різницю між LTS і STS-версіями .NET. Яку версію варто обирати для навчального курсу і чому?
13. Чим відрізняється .NET SDK від .NET Runtime? Що потрібно встановити, щоб розробляти програми, і що потрібно для запуску готової програми?
14. Які команди .NET CLI використовуються для: перевірки встановленої версії SDK, перегляду всіх SDK, створення і запуску консольного проєкту? Запишіть команди правильно.

15. Порівняйте .NET Framework і сучасний .NET за критеріями: підтримувані операційні системи, відкритість вихідного коду, темп розвитку, рекомендація для нових проєктів.
16. Що станеться, якщо програма, написана на .NET Framework 4.8, буде скомпільована й запущена на машині з Linux, де встановлений лише сучасний .NET 10? Поясніть чому.

1.3. Керований та некерований код

17. Що таке керований код і що таке некерований код? Що відрізняє їх виконання на рівні середовища?
18. Які служби автоматично надає CLR коду, написаному на C#? Перелічіть щонайменше чотири і коротко поясніть кожну.
19. Порівняйте C# і C++ за способом роботи з пам'яттю. Що відбувається в C#, коли об'єкт стає недосяжним? Що потрібно зробити програмісту в C++?
20. Чи означає «некерований код» поганий або небезпечний код? Наведіть три сфери, де некерований код є необхідним і обґрунтованим вибором.
21. Що таке P/Invoke? В яких сценаріях керований C#-код потребує виклику нативних бібліотек? Чому це «відповідальна зона»?
22. Що таке unsafe-код у C#? Поясніть, чим він відрізняється від звичайного керованого коду і коли його варто застосовувати.
23. Розгляньте наступний псевдокод на C++:

```
1  Patient* p = new Patient("Ivan", 45);
2  // ... використання ...
3  // програміст забув написати delete p;
```

Що станеться в C++? Чому така сама проблема неможлива в C#?

24. Заповніть таблицю порівняння керованого та некерованого коду за такими критеріями: контроль виконання, керування пам'яттю, безпека типів, типові сфери застосування.

1.4. JIT-компіляція

25. Що таке CIL (Common Intermediate Language)? Чому CIL — не машинний код, але програма все одно виконується швидко?

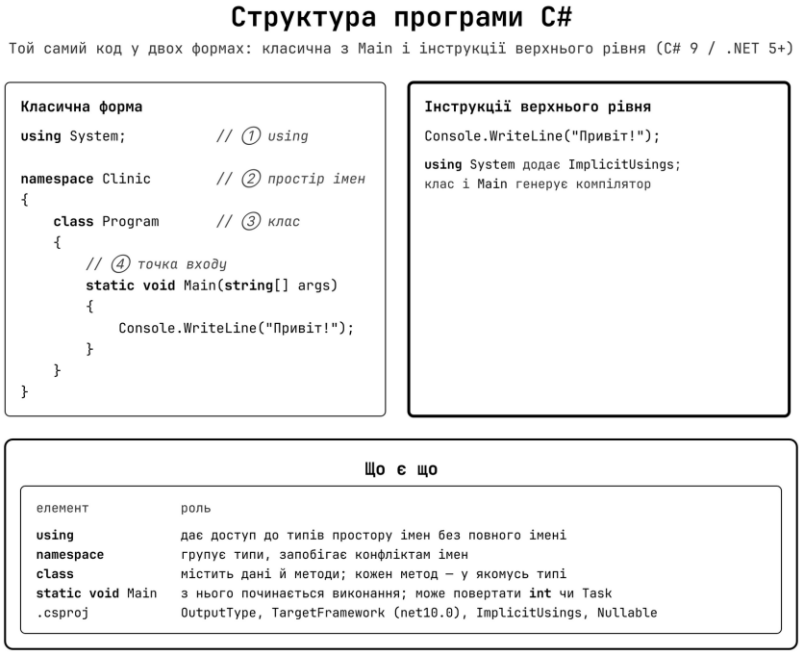
26. Поясніть різницю між двома кроками перетворення коду C# до машинного коду: що відбувається під час розробки і що — під час виконання?
27. Що відбувається, коли метод у .NET-програмі викликається **вперше**? А коли він викликається **вдруге**? Чому це важливо для продуктивності?
28. Чому JIT не компілює всю програму одразу при запуску? Яка перевага такого підходу?
29. Що таке метадані збірки і яку роль вони відіграють? Наведіть приклад того, що стає можливим завдяки метаданим і чого немає в C/C++ за замовчуванням.
30. Порівняйте JIT і AOT компіляцію: у чому суть кожного підходу, які сценарії підходять для JIT, а які — для AOT?
31. Що таке NativeAOT і яким є результат публікації у цьому режимі? Назвіть два обмеження NativeAOT і поясніть, чому вони виникають.
32. Заповніть таблицю порівняння: JIT vs NativeAOT за критеріями — час старту, обсяг пам'яті, підтримка рефлексії, кросплатформовість артефакту.

Розділ 2. Основи програмування на C#

2.1. Структура програми

Виконання програми

Весь код програми мовою C# міститься у файлах з розширенням .cs. За замовчуванням у проекті, який створюється у Visual Studio або через .NET CLI, вже є один файл — Program.cs. Саме він є точкою входу програми: виконання починається звідси.



Структура програми C#

Починаючи з C# 9 та .NET 5, у цьому файлі не обов'язково оголошувати клас Program і метод Main — компілятор додає їх неявно. Достатньо написати код напряму:

```
C#
```

```
1  using System;
2
3  Console.WriteLine("Hello, World!");
4  Console.WriteLine("Привіт від C#!");
```

Це називається **top-level statements** — можливість писати код верхнього рівня без додаткових обгорток. Саме цей стиль використовується у сучасних версіях .NET і в цьому курсі.

До C# 9 програма виглядала ось так — з явним класом і методом Main:

```
C#
1  using System;
2
3  class Program
4  {
5      static void Main(string[] args)
6      {
7          Console.WriteLine("Hello, World!");
8      }
9  }
```

Обидва варіанти компілюються однаково — різниця лише у кількості синтаксичного «шуму». У навчальному коді ми будемо використовувати сучасний стиль без зайвих обгорток.

Інструкції

Базовим будівельним блоком програми є **інструкції** (statements). Інструкція — це одна закінчена дія: оголошення змінної, присвоєння значення, виклик методу, умовний перехід, повернення з методу тощо. Кожна інструкція в C# завершується крапкою з комою (;). Цей знак вказує компілятору, де закінчується одна інструкція і починається наступна.

Наприклад, виклик методу Console.WriteLine — це одна інструкція:

```
C#
1  using System;
2
3  Console.WriteLine("Перша інструкція");
4  Console.WriteLine("Друга інструкція");
5  Console.WriteLine("Третя інструкція");
```

Кожен рядок тут є окремою інструкцією. Якщо забути крапку з комою — компілятор негайно повідомить про синтаксичну помилку, і програма не скомпілюється. Це одна з найпоширеніших помилок початківців.

Кілька інструкцій можуть бути об'єднані в один **блок коду**. Блок коду береться у фігурні дужки, а інструкції розміщуються між ними:

```
C#
1  using System;
2
3  {
4      Console.WriteLine("Перший блок – рядок 1");
5      Console.WriteLine("Перший блок – рядок 2");
6  }
```

Тут блок коду містить дві інструкції. Один блок може містити інший блок — вкладення не обмежене:

```
C#
 1  using System;
 2
 3  {
 4      Console.WriteLine("Зовнішній блок");
 5      {
 6          Console.WriteLine("Внутрішній блок");
 7          {
 8              Console.WriteLine("Глибоко вкладений блок");
 9          }
10      }
11  }
```

На практиці блоки коду з'являються скрізь: у тілах методів, умовах if, циклах for та while, класах. Розуміння того, що блок — це просто згрупований набір інструкцій у дужках, допомагає читати будь-який C#-код.

Реєстрозалежність

C# є **реєстрозалежною** мовою (case-sensitive). Це означає, що різниця між великими і малими літерами суттєва: однакові послідовності символів, але з різним регістром, сприймаються як абсолютно різні ідентифікатори.

Наприклад, для виведення тексту на консоль використовується метод Console.WriteLine. Якщо написати console.writeline або Console.writeline — компілятор видасть помилку, тому що такі ідентифікатори не існують. Регістр першої літери та регістр W у WriteLine є частиною імені і обов'язкові.

```
C#
 1  using System;
 2
 3  // Правильно:
 4  Console.WriteLine("Рядок 1");
```

```
 5
 6  // Нижче — помилки (закоментовано, бо не скомпілювалися б):
 7  // console.WriteLine("помилка — console з малої");
 8  // Console.writeline("помилка — writeline з малої");
 9  // CONSOLE.WRITELINE("помилка — усе великими");
10
11  Console.WriteLine("Реєстрозалежність — важлива особливість C#");
```

Те саме стосується змінних. Ідентифікатори name, Name і NAME — це три різні змінні з точки зору компілятора:

```
C#
1  using System;
2
3  string name = "Іван";
4  string Name = "Марія";
5
6  Console.WriteLine(name);
7  Console.WriteLine(Name);
```

На практиці давати змінним імена, що відрізняються лише регістром, не рекомендується — це заплутує код. Натомість у C# прийнятий чіткий стиль іменування: **PascalCase** для класів і методів (Console, WriteLine, Patient, GetAge) та **camelCase** для локальних змінних і параметрів (name, patientAge, systolicPressure).

Коментарі

Важливою частиною програмного коду є **коментарі**. Вони не є частиною програми — при компіляції коментарі повністю ігноруються і не впливають на виконання. Проте коментарі роблять код зрозумілішим, допомагаючи пояснити логіку, наміри або застереження.

У C# є два типи коментарів: **однорядковий** та **багаторядковий**.

Однорядковий коментар починається з подвійного слішу // і триває до кінця рядка. Усе, що написане після // на тому самому рядку, компілятор ігнорує:

```
C#
1  using System;
2
3  int pulse = 72;           // пульс у ударах за хвилину
4  int systolic = 120;       // систолічний артеріальний тиск
5  int diastolic = 80;       // діастолічний артеріальний тиск
6
7  Console.WriteLine($"Пульс: {pulse.ToString()} уд/хв");
8  Console.WriteLine($"Тиск: {systolic.ToString()}/{diastolic.ToString()} мм
   рт.ст.");
```

Багаторядковий коментар поміщається між символами /* і */. Він може займати довільну кількість рядків і зручний для великих пояснень або тимчасового виключення блоку коду:

```C#
using System;

/*
    Програма демонструє базову структуру коду C#.
    Тут можна написати будь-який текст на кількох рядках.
    Компілятор повністю проігнорує цей блок.
*/

Console.WriteLine("Код після багаторядкового коментаря виконується нормально.");
```

Хороший коментар пояснює **чому** написано саме так — не **що** робить код (це видно з самого коду), а **чому** саме таке рішення обрано, яке застереження є або яку бізнес-логіку відображає ця ділянка.

Файл проекту (.csproj)

У кожному проекті C# є файл конфігурації з розширенням .csproj. Його ім'я збігається з назвою проекту. Цей файл у форматі XML описує загальні параметри збирання та виконання проекту.

За замовчуванням файл виглядає приблизно так:

```XML
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

Розберемо основні параметри:

- **OutputType** — визначає тип вихідного артефакту. Значення Exe означає, що буде створено виконуваний файл .exe, який запускається безпосередньо. Значення Library — що буде створено файл .dll, який використовується іншими проектами як бібліотека.
- **TargetFramework** — задає версію .NET, під яку компілюється проект. Значення net10.0 означає .NET 10. Саме цією версією визначається, які можливості мови та бібліотеки доступні у проекті.

— **ImplicitUsings** — при значенні enable компілятор автоматично додає підключення найпоширеніших просторів імен (System, System.Collections.Generic, System.Linq тощо). Завдяки цьому не треба вручну писати using System; на початку кожного файлу. Саме тому у прикладах цього курсу ви бачите using System; у runnable-блоках, але у реальних проектах це часто пишеться автоматично.

— **Nullable** — при значенні enable компілятор аналізує код і попереджає, якщо є ризик звернення до null-посилання. Це допомагає уникнути одного з найчастіших джерел помилок у реальних системах — NullReferenceException.

На початку навчання файл .csproj можна не чіпати — він налаштовується один раз при створенні проекту. Але розуміння його структури стає важливим, коли виникає потреба змінити цільову версію .NET, підключити сторонній пакет через NuGet або налаштувати параметри компілятора.

2.2. Змінні та константи

Що таке змінна

Для зберігання даних у програмі застосовуються **змінні**. Змінна — це іменована область пам'яті, в якій зберігається значення певного типу. Кожна змінна має три характеристики: **тип** (що вона може зберігати), **ім'я** (як до неї звертатися) і **значення** (що вона зберігає в конкретний момент).

Тип змінної визначає, яку саме інформацію вона здатна зберігати: ціле число, дробове число, символ, рядок, логічне значення. Крім того, тип впливає на те, які операції можна застосовувати до змінної і скільки пам'яті вона займає.

Оголошення змінної

Перед тим як використовувати змінну, її необхідно **оголосити**. Синтаксис оголошення:

```
тип ім'я_змінної;
```

Спочатку вказується тип, потім ім'я. Наприклад, оголосимо змінну типу string (рядок) з іменем name:

```
C#
using System;
```

```
2
3  string name;
4  name = "Іван Петренко";
5
6  Console.WriteLine(name);
```

Тут у першому рядку змінна name лише оголошена — пам'ять виділена, але значення ще не задано. У другому рядку їй присвоюється рядок "Іван Петренко". Оскільки тип змінної string, їй можна присвоювати лише рядкові значення — спроба присвоїти число або логічне значення призведе до помилки компіляції.

Ініціалізація змінної

Також можна відразу при оголошенні присвоїти змінній початкове значення. Цей прийом називається **ініціалізацією**:

```
C#
 1  using System;
 2
 3  string name     = "Марія Сидоренко";
 4  int    age      = 32;
 5  double weight   = 58.5;
 6  bool   admitted = true;
 7
 8  Console.WriteLine($"Пацієнт: {name}");
 9  Console.WriteLine($"Вік: {age.ToString()} р.");
10  Console.WriteLine($"Вага: {weight.ToString()} кг");
11  Console.WriteLine($"Госпіталізований: {admitted.ToString()}");
```

Це найпоширеніший спосіб роботи зі змінними — оголошення та ініціалізація в одному рядку. Варто пам'ятати: якщо спробувати прочитати значення змінної до того, як воно було присвоєне, компілятор видасть помилку «використання неініціалізованої змінної».

Зміна значення

Відмінною рисою змінних є те, що у програмі можна багаторазово змінювати їх значення. Нове значення просто замінює попереднє:

```
C#
 1  using System;
 2
 3  string diagnosis = "Первинний огляд";
 4  Console.WriteLine($"Статус: {diagnosis}");
 5
 6  diagnosis = "Гіпертонія I ст.";
 7  Console.WriteLine($"Статус: {diagnosis}");
 8
 9  diagnosis = "Гіпертонія I ст. — лікування розпочато";
10  Console.WriteLine($"Статус: {diagnosis}");
```

Щоразу, коли виконується присвоєння diagnosis = "...", нове значення записується на місце попереднього. Попереднє значення при цьому безповоротно замінюється.

Правила іменування змінних

Ім'я змінної — це **ідентифікатор**, і воно підпорядковується правилам мови C#:

- ім'я може містити букви, цифри та символ підкреслення _;
- перший символ імені має бути буквою або символом підкреслення — але не цифрою;
- в імені не повинно бути пробілів та знаків пунктуації;
- ім'я не може збігатися з ключовим словом мови C# (int, string, class, new та ін.).

Хоча формально ім'я може бути будь-яким, слід давати змінним **описові імена**, що відображають їх призначення. Змінна systolicPressure розповідає про себе сама; змінна x або temp — ні.

У C# для локальних змінних і параметрів методів прийнятий стиль **camelCase**: перше слово з малої літери, кожне наступне — з великої:

```
C#
1  using System;
2
3  string patientName    = "Олег Бойко";
4  int    patientAge     = 47;
5  double bodyMassIndex  = 26.3;
6  bool   hasHighPressure = true;
7
8  Console.WriteLine($"{patientName}, {patientAge.ToString()} p.,
   IMT={bodyMassIndex.ToString()}, гіпертонія={hasHighPressure.ToString()}");
```

Неявна типізація: var

C# дозволяє не вказувати тип явно, якщо компілятор може визначити його самостійно за значенням правої частини. Для цього замість типу вказується ключове слово var:

```
C#
1  using System;
2
3  var name        = "Іван Петренко";  // компілятор виводить: string
4  var age         = 45;               // int
5  var temperature = 36.6;             // double
6  var isAdmitted  = false;            // bool
7
8  Console.WriteLine($"{name}, {age.ToString()} p., t={temperature.ToString()},
   госпіталізований={isAdmitted.ToString()}");
```

Важливо розуміти: var — це не динамічний тип і не object. Компілятор визначає конкретний тип **один раз** на основі виразу праворуч і надалі перевіряє всі операції з цією змінною так само суворо, як якщо б тип був вказаний явно. Після того як тип виведено — він фіксований.

Є обмеження: змінну з var не можна спочатку оголосити, а потім ініціалізувати — компілятор не зможе вивести тип:

```
C#
1  var x;    // помилка — немає значення, тип невідомий
2  x = 10;
```

Також не можна ініціалізувати var значенням null — з тієї самої причини. Тип null сам по собі не несе інформації про тип.

Константи

Відмінною особливістю змінних є те, що їх значення можна змінювати в процесі виконання програми. Але іноді нам потрібні значення, які не повинні змінюватись ніколи — вони відомі наперед і мають залишатися незмінними протягом усього виконання. Для таких даних у C# передбачені **константи**.

Константа оголошується з ключовим словом const перед типом і **обов'язково ініціалізується** при оголошенні:

```
C#
 1  using System;
 2
 3  const double MaxNormalTemperature = 37.0;
 4  const int    MinPulse             = 60;
 5  const int    MaxPulse             = 100;
 6  const string ClinicName           = "Міська клінічна лікарня №3";
 7
 8  double patientTemp  = 38.3;
 9  int    patientPulse = 92;
10
11  Console.WriteLine($"Клініка: {ClinicName}");
12  Console.WriteLine($"Норма температури: до {MaxNormalTemperature.ToString()}
    °C");
13  Console.WriteLine($"Норма пульсу: {MinPulse.ToString()}–{MaxPulse.ToString()}
    уд/хв");
14
15  if (patientTemp > MaxNormalTemperature)
16      Console.WriteLine($"Температура {patientTemp.ToString()} °C — підвищена!");
17
18  if (patientPulse >= MinPulse && patientPulse <= MaxPulse)
19      Console.WriteLine($"Пульс {patientPulse.ToString()} уд/хв — у нормі.");
```

Якщо спробувати змінити значення константи після її оголошення — компілятор видасть помилку:

```
C#
1  const double MaxNormalTemperature = 37.0;
2  MaxNormalTemperature = 36.6; // помилка компіляції — константі не можна
   присвоїти нове значення
```

Також неможливо оголосити константу без початкового значення:

```
C#
1  const string ClinicName; // помилка — константа не ініціалізована
```

Нерідко для констант використовують назви в стилі **PascalCase** (MaxPulse, ClinicName). Це умовність, а не вимога мови, але вона допомагає одразу відрізнити константи від змінних у коді.

Якщо нам треба зберігати у програмі деякі дані, які можуть змінюватися, — визначаємо їх як змінні. Якщо дані відомі заздалегідь і не повинні змінюватися — оголошуємо їх константами. Це покращує читабельність коду: читач одразу бачить, що перед ним фіксоване значення, яке є частиною бізнес-правил або фізичних обмежень системи.

2.3. Літерали

Літерали — це фіксовані значення, записані безпосередньо у вихідному коді програми. Коли ми пишемо 42, 3.14, "Привіт" або true — все це літерали. На відміну від змінних, значення якої може змінюватись під час виконання, літерал завжди означає одне й те саме конкретне значення, відоме ще на етапі компіляції.

Літерали можна передавати змінним як значення, передавати як аргументи методів або використовувати безпосередньо у виразах. У C# є кілька видів літералів: **логічні**, **цілочисленні**, **речові**, **символьні**, **рядкові** та спеціальний літерал null.

Логічні літерали

Є лише два логічних літерали — true (істина) і false (хибність). Вони відповідають типу bool і є єдиними допустимими значеннями для логічних змінних:

```
C#
1  using System;
```

```
2
3  bool isAdmitted   = true;
4  bool hasAllergies = false;
5
6  Console.WriteLine(isAdmitted.ToString());
7  Console.WriteLine(hasAllergies.ToString());
8  Console.WriteLine(true.ToString());
9  Console.WriteLine(false.ToString());
```

Цілочисленні літерали

Цілочисленні літерали представляють цілі числа — позитивні та негативні. Вони можуть бути виражені у трьох системах числення: **десятковій**, **двійковій** та **шістнадцятковій**.

З числами у десятковій формі все звично — вони використовуються в повсякденному житті:

```
C#
1  using System;
2
3  Console.WriteLine(-11);
4  Console.WriteLine(5);
5  Console.WriteLine(505);
```

Числа у **двійковій** формі передуються символами 0b, після яких іде набір з нулів та одиниць:

```
C#
1  using System;
2
3  Console.WriteLine(0b11);     // 3
4  Console.WriteLine(0b1011);   // 11
5  Console.WriteLine(0b100001); // 33
```

Для запису числа у **шістнадцятковій** формі використовуються символи 0x, після яких іде набір символів від 0 до 9 і від A до F:

```
C#
1  using System;
2
3  Console.WriteLine(0x0A); // 10
4  Console.WriteLine(0xFF); // 255
5  Console.WriteLine(0xA1); // 161
```

Починаючи з C# 7, у цілочисленних літералах можна використовувати символ підкреслення _ як розділювач розрядів для покращення читабельності. Компілятор його повністю ігнорує:

```
C#
```

```
1  using System;
2
3  int million   = 1_000_000;
4  int cardNumber = 1234_5678;
5
6  Console.WriteLine(million.ToString());
7  Console.WriteLine(cardNumber.ToString());
```

Речові літерали

Речові літерали представляють дробові числа. Перша форма — числа з фіксованою крапкою, де дробову частину відокремлено від цілої крапкою:

```
C#
1  using System;
2
3  Console.WriteLine(3.14.ToString());
4  Console.WriteLine(100.001.ToString());
5  Console.WriteLine((-0.38).ToString());
```

Також речові літерали можуть задаватися в **експоненційній формі** MEp, де M — мантиса, E — ознака степеня, p — порядок (степінь десяти). Наприклад, 3.2e3 означає $3.2 \times 10^3 = 3200$:

```
C#
1  using System;
2
3  Console.WriteLine(3.2e3.ToString());   // 3200
4  Console.WriteLine(1.2e-1.ToString());  // 0.12
5  Console.WriteLine(5.0e6.ToString());   // 5000000
```

За замовчуванням всі речові літерали є значеннями типу double. Якщо потрібно явно вказати тип float або decimal, використовуються суфікси: f або F для float, m або M для decimal:

```
C#
1  using System;
2
3  double d = 3.14;       // double за замовчуванням
4  float  f = 3.14f;      // суфікс f → float
5  decimal m = 3.14m;     // суфікс m → decimal
6
7  Console.WriteLine(d.ToString());
8  Console.WriteLine(f.ToString());
9  Console.WriteLine(m.ToString());
```

Тип decimal особливо важливий для фінансових розрахунків: він зберігає число в десятковій системі, тому десяткові дроби на кшталт 0.1 чи 19.99 представляються **точно** — без похибок двійкового

представлення, характерних для float і double (де 0.1 + 0.2 дає 0.30000000000000004). Водночас decimal не позбавлений округлення взагалі: точність обмежена 28–29 значущими цифрами, і дроби, що не мають скінченного десяткового запису, округлюються — 1m / 3m * 3m дає 0.9999999999999999999999999999, а не 1.

Символьні літерали

Символьні літерали представляють одиночний символ. Символ береться в **одинарні** лапки і відповідає типу char. Кожен символ кодується у Unicode і займає 2 байти.

Найпростіший вид — звичайні символи:

```
C#
1  using System;
2
3  char letter = 'A';
4  char digit  = '2';
5  char ukr    = 'I';
6
7  Console.WriteLine(letter.ToString());
8  Console.WriteLine(digit.ToString());
9  Console.WriteLine(ukr.ToString());
```

Окрему групу складають **керуючі послідовності** — спеціальні комбінації символів, що починаються зі зворотного слішу \. Вони інтерпретуються компілятором не як два символи, а як один спеціальний:

```
C#
 1  using System;
 2
 3  char newline = '\n'; // переклад рядка
 4  char tab     = '\t'; // горизонтальна табуляція
 5  char bslash  = '\\'; // зворотний сліш
 6  char squote  = '\''; // одинарна лапка
 7
 8  Console.Write("Перший рядок");
 9  Console.Write(newline);
10  Console.Write("Другий рядок");
11  Console.Write(newline);
12  Console.WriteLine("Рядок з\tтабуляцією");
```

Символи також можна визначати через шістнадцятковий код, вказуючи його після \x. Це код UTF-16 (а не лише ASCII): '\x0410' — кириличне А. На відміну від \u, після \x може стояти **від однієї до чотирьох** шістнадцяткових цифр, тому в рядках ця послідовність

підступна: "\x41BC" — це один символ \u41BC, а не A і BC. Для рядків надійніше використовувати \u з рівно чотирма цифрами:

```
C#
1  using System;
2
3  Console.WriteLine('\x78'.ToString()); // x
4  Console.WriteLine('\x5A'.ToString()); // Z
```

Ще один спосіб — визначення через **коди Unicode**. Після \u вказується чотиризначний шістнадцятковий код символу з таблиці Unicode:

```
C#
1  using System;
2
3  Console.WriteLine('P'.ToString()); // P (кирилиця)
4  Console.WriteLine('C'.ToString()); // C (кирилиця)
5  Console.WriteLine('I'.ToString()); // I (латиниця)
```

Рядкові літерали

Рядкові літерали представляють рядки — послідовності символів. Рядок береться в **подвійні** лапки і відповідає типу string:

```
C#
1  using System;
2
3  Console.WriteLine("hello");
4  Console.WriteLine("Привіт, світ!");
5  Console.WriteLine("hello world");
```

Якщо всередині рядка необхідно використати подвійну лапку, вона екранується зворотним слішем \":

```
C#
1  using System;
2
3  Console.WriteLine("Клініка \"Надія\" – міський медичний центр.");
```

У рядках також можна використовувати керуючі послідовності. Зокрема, \n здійснює перехід на новий рядок:

```
C#
1  using System;
2
3  Console.WriteLine("Пацієнт: Іван Петренко\nВік: 45 р.\nДіагноз: Гіпертонія");
```

Якщо рядок містить багато зворотних слішів (наприклад, шляхи до файлів), зручно використовувати **verbatim-рядок** з префіксом @. У такому рядку зворотний сліш не є спеціальним символом і не потребує екранування:

```
C#
using System;

string path1 = "C:\\Users\\Doctor\\Patients\\record.txt";  // звичайний рядок
string path2 = @"C:\Users\Doctor\Patients\record.txt";   // verbatim — те саме

Console.WriteLine(path1);
Console.WriteLine(path2);
```

Verbatim-рядок також може бути багаторядковим — переноси рядка всередині @"..." включаються у значення як є:

```
C#
using System;

string record = @"Пацієнт: Марія Сидоренко
Вік: 38 р.
Діагноз: Бронхіт
Лікар: Коваль О.В.";

Console.WriteLine(record);
```

null

Окремий літерал null представляє **відсутність посилання** — тобто посилання, яке не вказує на жодний об'єкт. По суті, null означає «немає значення» або «значення невідоме».

Значення null може бути присвоєне будь-якій змінній **reference-типу** — рядкам, масивам, об'єктам класів:

```
C#
using System;

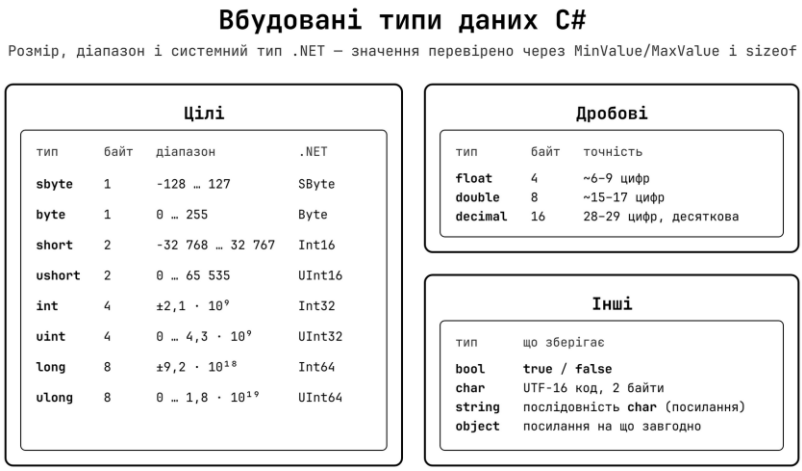
string middleName = null; // по батькові не вказано

if (middleName == null)
    Console.WriteLine("По батькові не вказано.");
else
    Console.WriteLine($"По батькові: {middleName}");
```

Спроба звернутися до члена або методу змінної, що містить null, спричинить виняток NullReferenceException під час виконання — одну з найпоширеніших помилок у програмуванні. Саме тому перевірка на null перед використанням є важливою звичкою.

2.4. Типи даних

Як і в багатьох мовах програмування, у C# є власна система типів даних, яка використовується для створення змінних. **Тип даних** визначає внутрішнє представлення даних у пам'яті, діапазон значень, які може приймати змінна, і допустимі операції, які можна застосовувати над змінною. Саме тому, оголошуючи змінну, ми зобов'язані вказати її тип — компілятор використовує цю інформацію для перевірки коректності всіх операцій із цією змінною.



Базові типи даних C#

У мові C# є такі вбудовані базові типи даних:

bool

Тип bool зберігає логічне значення — true (істина) або false (хибність). Представлений системним типом System.Boolean.

```
C#
1  using System;
2
3  bool isAdmitted  = true;
4  bool hasAllergies = false;
5
6  Console.WriteLine(isAdmitted.ToString());
7  Console.WriteLine(hasAllergies.ToString());
```

byte

Тип byte зберігає ціле беззнакове число від **0 до 255** і займає **1 байт**. Представлений системним типом System.Byte. Зручний для зберігання малих невід'ємних чисел — наприклад, значень компонентів кольору, відсотків, флагів.

```
C#
1  using System;
2
3  byte age    = 45;
4  byte volume = 255;
5  byte zero   = 0;
6
7  Console.WriteLine(age.ToString());
8  Console.WriteLine(volume.ToString());
```

sbyte

Тип sbyte (signed byte) зберігає ціле число зі знаком від **-128 до 127** і займає **1 байт**. Представлений системним типом System.SByte. Відрізняється від byte тим, що може зберігати від'ємні значення.

```
C#
1  using System;
2
3  sbyte positive = 102;
4  sbyte negative = -101;
5
6  Console.WriteLine(positive.ToString());
7  Console.WriteLine(negative.ToString());
```

short

Тип short зберігає ціле число зі знаком від **-32 768 до 32 767** і займає **2 байти**. Представлений системним типом System.Int16.

```
C#
1  using System;
2
3  short systolic  = 120;
4  short diastolic = 80;
5
6  Console.WriteLine($"Тиск: {systolic.ToString()}/{diastolic.ToString()} мм
   рт.ст.");
```

ushort

Тип ushort (unsigned short) зберігає ціле беззнакове число від **0 до 65 535** і займає **2 байти**. Представлений системним типом System.UInt16.

```C#
using System;

ushort heartRate = 72;
ushort maxValue  = 65535;

Console.WriteLine(heartRate.ToString());
Console.WriteLine(maxValue.ToString());
```

int

Тип int зберігає ціле число зі знаком від **-2 147 483 648 до 2 147 483 647** і займає **4 байти**. Представлений системним типом System.Int32. Це **найпоширеніший цілочисленний тип** у C# — за замовчуванням усі цілі літерали (наприклад, 42) мають саме тип int.

```C#
using System;

int a = 10;
int b = 0b101; // двійкова форма, b = 5
int c = 0xFF;  // шістнадцяткова форма, c = 255

Console.WriteLine(a.ToString());
Console.WriteLine(b.ToString());
Console.WriteLine(c.ToString());
```

uint

Тип uint (unsigned int) зберігає ціле беззнакове число від **0 до 4 294 967 295** і займає **4 байти**. Представлений системним типом System.UInt32. Для явного позначення uint-літерала використовується суфікс u або U.

```C#
using System;

uint a = 10U;
uint b = 0b101;
uint c = 0xFF;

Console.WriteLine(a.ToString());
Console.WriteLine(b.ToString());
Console.WriteLine(c.ToString());
```

long

Тип long зберігає ціле число зі знаком від **-9 223 372 036 854 775 808 до 9 223 372 036 854 775 807** і займає **8 байт**. Представлений системним типом System.Int64. Використовується тоді, коли діапазон int недостатній. Для явного позначення long-літерала — суфікс l або L.

```C#
```

```
1  using System;
2
3  long recordsCount = 9_000_000_000L;
4  long negative     = -10L;
5
6  Console.WriteLine(recordsCount.ToString());
7  Console.WriteLine(negative.ToString());
```

ulong

Тип ulong (unsigned long) зберігає ціле беззнакове число від **0 до 18 446 744 073 709 551 615** і займає **8 байт**. Представлений системним типом System.UInt64. Суфікс — ul або UL.

```
C#
1  using System;
2
3  ulong a = 10UL;
4  ulong b = 0b101;
5  ulong c = 0xFF;
6
7  Console.WriteLine(a.ToString());
8  Console.WriteLine(b.ToString());
9  Console.WriteLine(c.ToString());
```

float

Тип float зберігає число з плаваючою точкою приблизно від **-3.4×10^{38} до 3.4×10^{38}** і займає **4 байти**. Представлений системним типом System.Single. Забезпечує приблизно 7 значущих цифр. Для позначення float-літерала обов'язковий суфікс f або F.

```
C#
1  using System;
2
3  float weight      = 72.5f;
4  float temperature = 36.6F;
5
6  Console.WriteLine(weight.ToString());
7  Console.WriteLine(temperature.ToString());
```

double

Тип double зберігає число з плаваючою точкою приблизно від **$\pm 5.0 \times 10^{-324}$ до $\pm 1.7 \times 10^{308}$** і займає **8 байт**. Представлений системним типом System.Double. Забезпечує приблизно 15–16 значущих цифр. Це **найпоширеніший тип для дробових чисел** — за замовчуванням усі речові літерали (наприклад, 3.14) мають тип double.

```
C#
1  using System;
```

```
2
3  double pi          = 3.14159265358979;
4  double temperature = 36.6;
5  double bmi         = 22.5;
6
7  Console.WriteLine(pi.ToString());
8  Console.WriteLine(temperature.ToString());
9  Console.WriteLine(bmi.ToString());
```

decimal

Тип decimal зберігає десяткове дробове число з дуже високою точністю — 28–29 значущих цифр — і займає **16 байт**. Представлений системним типом System.Decimal. Діапазон від **±1.0 × 10^{-28} до ±7.9 × 10^{28}**. Для позначення decimal-літерала обов'язковий суфікс m або M.

На відміну від float і double, тип decimal не має похибок округлення при роботі з десятковими дробами, тому він **незамінний у фінансових розрахунках**, де точність кожного знаку критична.

```
C#
1  using System;
2
3  decimal price     = 1005.80m;
4  decimal discount  = 100.50M;
5  decimal total     = price - discount;
6
7  Console.WriteLine(price.ToString());
8  Console.WriteLine(discount.ToString());
9  Console.WriteLine(total.ToString());
```

char

Тип char зберігає **одиночний символ** у кодуванні Unicode і займає **2 байти**. Представлений системним типом System.Char. Символьні літерали беруться в одинарні лапки.

Точніше, char — це одна **кодова одиниця UTF-16** (значення від 0 до 65 535). Більшість символів, зокрема латиниця й кирилиця, вміщуються в одну таку одиницю. Але символи поза базовою площиною Unicode — емодзі, деякі ієрогліфи — кодуються **парою** одиниць (сурогатною парою) і в один char не вміщуються: "🩺".Length дорівнює 2.

```
C#
1  using System;
2
3  char a = 'A';
4  char b = '\x5A'; // Z через шістнадцятковий код
5  char c = 'I';    // кириличний символ
6
7  Console.WriteLine(a.ToString());
```

```
8  Console.WriteLine(b.ToString());
9  Console.WriteLine(c.ToString());
```

string

Тип string зберігає **послідовність символів Unicode** довільної довжини. Представлений системним типом System.String. Рядкові літерали беруться в подвійні лапки. На відміну від усіх попередніх типів, string є **reference-типом** — змінна зберігає не саме значення, а посилання на об'єкт у купі.

```
C#
1  using System;
2
3  string firstName = "Іван";
4  string lastName  = "Петренко";
5  string fullName  = firstName + " " + lastName;
6
7  Console.WriteLine(fullName);
8  Console.WriteLine($"Довжина: {fullName.Length.ToString()} символів");
```

object

Тип object може зберігати значення **будь-якого типу даних** — як value-типів, так і reference-типів. Представлений системним типом System.Object, який є **базовим класом для всіх типів і класів .NET**. Займає 4 байти на 32-розрядній платформі та 8 байт — на 64-розрядній.

```
C#
 1  using System;
 2
 3  object a = 22;
 4  object b = 3.14;
 5  object c = "hello code";
 6  object d = true;
 7
 8  Console.WriteLine(a.ToString());
 9  Console.WriteLine(b.ToString());
10  Console.WriteLine(c.ToString());
11  Console.WriteLine(d.ToString());
```

Хоча object здається зручним — «приймає все» — на практиці він використовується обережно, бо втрачається типобезпека і можливі накладні витрати при boxing/unboxing (про це детально у темі узагальнень).

Комплексний приклад

Визначимо кілька змінних різних типів і виведемо їх значення на консоль:

```
C#
 1  using System;
 2
 3  string name       = "Іван Петренко";
 4  int    age        = 45;
 5  bool   isAdmitted = false;
 6  double weight     = 78.65;
 7  decimal bill      = 1250.50m;
 8
 9  Console.WriteLine($"Ім'я: {name}");
10  Console.WriteLine($"Вік: {age.ToString()}");
11  Console.WriteLine($"Вага: {weight.ToString()}");
12  Console.WriteLine($"Госпіталізований: {isAdmitted.ToString()}");
13  Console.WriteLine($"Рахунок: {bill.ToString()} грн");
```

Використання суфіксів

При присвоєнні значень слід пам'ятати: всі речові літерали за замовчуванням мають тип double, а всі цілочисленні — тип int. Щоб явно вказати інший тип, використовуються суфікси:

```
C#
 1  using System;
 2
 3  float   f = 3.14F;
 4  decimal d = 1005.8M;
 5  uint    u = 10U;
 6  long    l = 20L;
 7  ulong  ul = 30UL;
 8
 9  Console.WriteLine(f.ToString());
10  Console.WriteLine(d.ToString());
11  Console.WriteLine(u.ToString());
12  Console.WriteLine(l.ToString());
13  Console.WriteLine(ul.ToString());
```

Системні типи

Назва вбудованого типу — це скорочення відповідного системного типу .NET. Наприклад, int і System.Int32 є повністю еквівалентними:

```
C#
1  using System;
2
3  int           a = 4;
4  System.Int32  b = 4;
5
6  Console.WriteLine(a.ToString());
```

```
7  Console.WriteLine(b.ToString());
8  Console.WriteLine((a == b).ToString()); // true
```

Обидва записи абсолютно рівнозначні — компілятор сприймає їх однаково. На практиці зазвичай вживають скорочені псевдоніми (int, string, bool), оскільки вони стисліші і звичніші.

Неявна типізація

У C# можна не вказувати тип змінної явно, якщо компілятор може визначити його самостійно з виразу праворуч. Для цього замість назви типу вказується ключове слово var:

```
C#
 1  using System;
 2
 3  var hello = "Hello to World";
 4  var c     = 20;
 5  var d     = 3.14;
 6  var flag  = true;
 7
 8  Console.WriteLine(hello.GetType().Name); // String
 9  Console.WriteLine(c.GetType().Name);     // Int32
10  Console.WriteLine(d.GetType().Name);     // Double
11  Console.WriteLine(flag.GetType().Name);  // Boolean
```

Оскільки за замовчуванням цілочисленні значення розглядаються як int, а речові — як double, то c отримає тип int, а d — тип double. Тип виводиться **один раз** при оголошенні і фіксується — надалі змінна поводиться точно так само, як якби тип був вказаний явно.

Неявно типізована змінна має кілька обмежень. По-перше, не можна спочатку оголосити змінну, а потім ініціалізувати — компілятор не знатиме, який тип вивести:

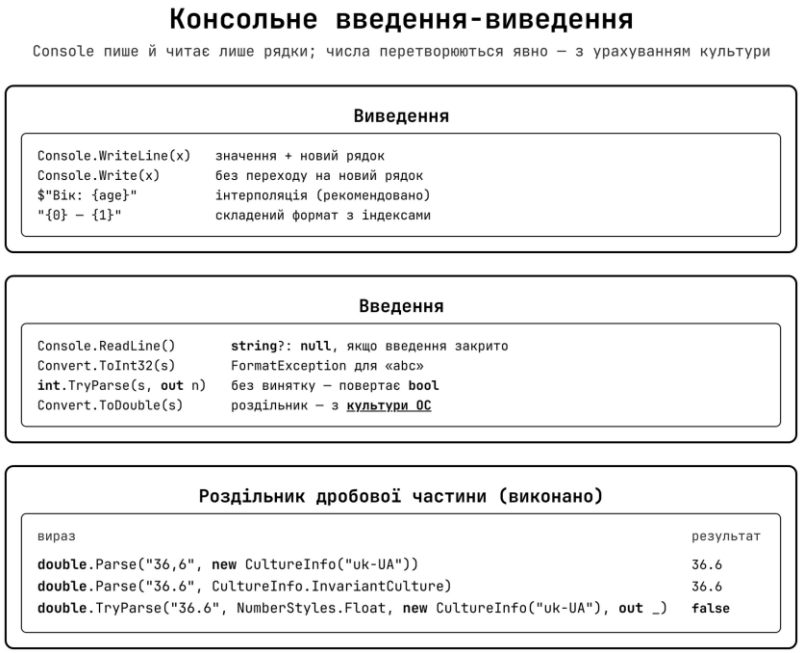
```
C#
1  // Цей код працює:
2  int a;
3  a = 20;
4
5  // Цей код не працює — помилка компіляції:
6  var c;
7  c = 20;
```

По-друге, не можна ініціалізувати var значенням null, оскільки null сам по собі не несе інформації про тип:

```
C#
1  // Цей код не працює — помилка компіляції:
2  var c = null;
```

2.5. Консольне введення-виведення

Будь-яка програма, яка взаємодіє з користувачем, повинна вміти отримувати від нього дані і повертати результат. Найпростішим і найбільш доступним механізмом такої взаємодії є **консоль** — текстовий інтерфейс, у якому програма виводить рядки тексту і читає те, що вводить користувач з клавіатури. У C# для роботи з консоллю призначений статичний клас Console з простору імен System. Він надає методи для виведення даних (Write, WriteLine), отримання введення (ReadLine, ReadKey) та налаштування параметрів консольного вікна.



Консольне введення-виведення

Виведення даних на консоль

Основним методом виведення є Console.WriteLine. Він приймає значення довільного типу, перетворює його на рядок і виводить на консоль, після чого автоматично переходить на новий рядок. Саме тому

кожен наступний виклик Console.WriteLine починає свій вивід з нового рядка.

```
C#
 1  using System;
 2
 3  string name     = "Олена Бойченко";
 4  int    age      = 29;
 5  double weight   = 63.4;
 6  bool   admitted = true;
 7
 8  Console.WriteLine(name);
 9  Console.WriteLine(age.ToString());
10  Console.WriteLine(weight.ToString());
11  Console.WriteLine(admitted.ToString());
```

Зверніть увагу: числові та логічні змінні передаються у Console.WriteLine через .ToString() — явне перетворення на рядок. Це корисна звичка, яка унаочнює, що відбувається перетворення типу, і усуває двозначність у деяких ситуаціях.

Console.Write — вивід без переходу на новий рядок

Якщо потрібно вивести кілька значень в одному рядку, не розриваючи його, застосовується метод Console.Write. Він діє так само, як Console.WriteLine, але **не додає символ переходу на новий рядок** після виводу. Наступне виведення продовжиться з того ж рядка, де завершилось попереднє.

```
C#
1  using System;
2
3  Console.Write("Пацієнт: ");
4  Console.Write("Олена Бойченко");
5  Console.Write(", вік: ");
6  Console.Write(29.ToString());
7  Console.WriteLine(" р.");  // лише тут — перехід на новий рядок
8  Console.WriteLine("Запис завершено.");
```

Таке поєднання Write і WriteLine зручне, коли рядок формується з кількох шматків і перехід потрібен лише наприкінці.

Інтерполяція рядків

Коли потрібно вивести кілька змінних в одному реченні, вручну об'єднувати рядки методами Write незручно. Значно елегантнішим є механізм **інтерполяції рядків**, доступний починаючи з C# 6. Перед рядком ставиться символ $, а всередині рядка змінні або вирази

вказуються у фігурних дужках {...}. Під час виконання компілятор підставляє значення замість кожної вставки.

```
C#
1  using System;
2
3  string name   = "Ярослав Коваль";
4  int    age    = 41;
5  double height = 1.82;
6
7  Console.WriteLine($"Пацієнт: {name}, вік: {age} р., зріст: {height} м");
8  Console.WriteLine($"Через рік пацієнту виповниться {age + 1} рік");
```

У фігурні дужки можна вкладати не лише змінні, але й будь-які вирази: арифметичні операції, виклики методів, тернарні оператори. Компілятор обчислить вираз і вставить результат у рядок. Це робить інтерполяцію дуже потужним інструментом для формування повідомлень.

Форматні рядки з плейсхолдерами

Інший спосіб підставляти значення в рядок — **складений формат** (composite formatting). У рядку розміщуються числові плейсхолдери виду {0}, {1}, {2} і т. д., а після рядка через кому вказуються значення, які треба підставити замість них. Нумерація плейсхолдерів починається з нуля і відповідає порядку аргументів.

```
C#
1  using System;
2
3  string name   = "Марія Сидоренко";
4  int    age    = 35;
5  double weight = 58.2;
6
7  Console.WriteLine("Ім'я: {0}, вік: {1} р., вага: {2} кг", name, age, weight);
```

Важливо розуміти, що {0} замінюється першим аргументом після рядка, {1} — другим і так далі. Порядок у рядку та порядок аргументів можуть навіть відрізнятися: якщо написати {1} першим, то виведеться другий аргумент. Цей механізм корисний, коли один аргумент треба вивести в кількох місцях рядка — досить кількох посилань на той самий індекс.

Нині в сучасному C# перевага надається інтерполяції ($"..."), оскільки вона наочніша і надійніша — компілятор одразу бачить вирази всередині дужок і може повідомити про помилку. Складений формат залишається в мові для сумісності та деяких сценаріїв локалізації.

Зчитування даних з консолі

Для отримання введення від користувача призначений метод Console.ReadLine(). Він **зупиняє виконання програми**, очікуючи, доки користувач не натисне Enter, і повертає весь введений рядок як значення типу string?.

```
C#
1  using System;
2
3  Console.Write("Введіть ваше ім'я: ");
4  string? name = Console.ReadLine();
5  Console.WriteLine($"Привіт, {name}!");
```

Зверніть увагу на тип string? — це рядок, який також може мати значення null. Метод Console.ReadLine() повертає null, коли потік введення закритий і зчитувати нічого. На практиці при звичайній роботі з клавіатурою null не виникає, але компілятор вимагає, щоб це було відображено в типі змінної. Детально про nullable-типи мова піде в наступних темах.

Зчитування числових даних

Метод Console.ReadLine() завжди повертає рядок — навіть якщо користувач ввів 42. Щоб отримати числове значення, рядок треба перетворити на відповідний тип. Для цього зручно використовувати методи класу Convert:

- Convert.ToInt32(рядок) — перетворює рядок у ціле число int;
- Convert.ToDouble(рядок) — перетворює у дробове число double;
- Convert.ToDecimal(рядок) — перетворює у точне десяткове число decimal;
- Convert.ToBoolean(рядок) — перетворює рядок "True" або "False" у bool.

```
C#
 1  using System;
 2
 3  Console.Write("Введіть ім'я пацієнта: ");
 4  string? name = Console.ReadLine();
 5
 6  Console.Write("Введіть вік: ");
 7  int age = Convert.ToInt32(Console.ReadLine());
 8
 9  Console.Write("Введіть вагу (кг): ");
10  double weight = Convert.ToDouble(Console.ReadLine());
11
12  Console.Write("Введіть суму рахунку: ");
```

```
decimal bill = Convert.ToDecimal(Console.ReadLine());

Console.WriteLine($"Пацієнт: {name}");
Console.WriteLine($"Вік: {age.ToString()} р.");
Console.WriteLine($"Вага: {weight.ToString()} кг");
Console.WriteLine($"Рахунок: {bill.ToString()} грн");
```

Якщо користувач введе текст там, де очікується число, Convert.ToInt32 кине виняток FormatException. Тому у реальних програмах введення числових значень захищають конструкцією try-catch або методом int.TryParse — про них мова піде в наступних розділах.

Культура та роздільник дробової частини

Важливий нюанс при введенні дробових чисел: роздільник між цілою і дробовою частиною залежить від **поточної культури операційної системи**. В українській та більшості європейських культурах роздільником є кома (,), в американській — крапка (.). Якщо операційна система налаштована на українську культуру, а користувач вводить 1.5 з крапкою — Convert.ToDouble може видати помилку або неправильний результат. Розробнику важливо враховувати цю відмінність і або чітко документувати формат введення, або явно вказувати культуру при перетворенні.

Комплексний приклад: реєстрація пацієнта

Об'єднаємо отримані знання в одну програму, яка збирає дані про нового пацієнта:

```
C#
using System;

Console.WriteLine("=== Реєстрація пацієнта ===");

Console.Write("Прізвище та ім'я: ");
string? fullName = Console.ReadLine();

Console.Write("Вік: ");
int age = Convert.ToInt32(Console.ReadLine());

Console.Write("Температура тіла (°C): ");
double temperature = Convert.ToDouble(Console.ReadLine());

Console.Write("Госпіталізований (true/false): ");
bool admitted = Convert.ToBoolean(Console.ReadLine());

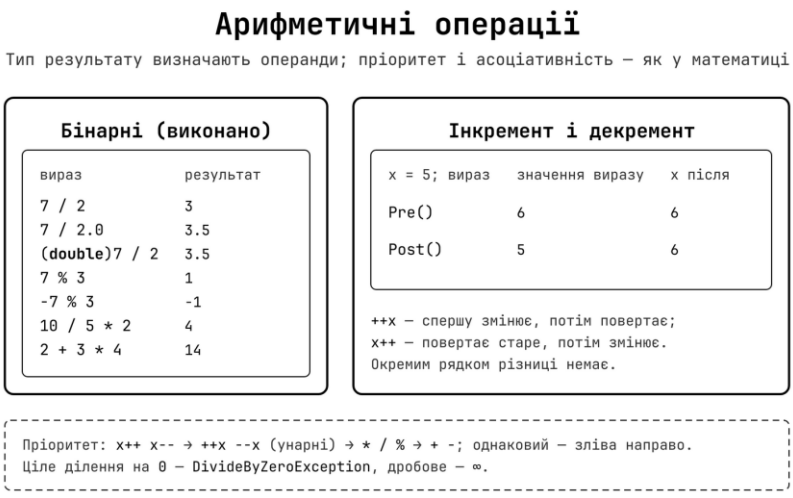
Console.WriteLine();
Console.WriteLine("--- Картка пацієнта ---");
```

```
Console.WriteLine($"Ім'я:          {fullName}");
Console.WriteLine($"Вік:           {age.ToString()} р.");
Console.WriteLine($"Температура:   {temperature.ToString()} °C");
Console.WriteLine($"Госпіталізований: {admitted.ToString()}");
```

Ця програма демонструє повний цикл взаємодії: запит до користувача через Console.Write, зчитування відповіді через Console.ReadLine, перетворення на потрібний тип і вивід підсумкового результату з форматуванням.

2.6. Арифметичні операції

Операції — це фундаментальний механізм обчислень у будь-якій мові програмування. **Операція** — це дія над одним або кількома значеннями, що дає новий результат. Значення, над якими виконується операція, називаються **операндами**. Залежно від кількості операндів розрізняють унарні (один операнд), бінарні (два операнди) та тернарні (три операнди) операції. У цьому розділі розглянемо арифметичні операції — найпоширіший клас операцій, що безпосередньо відповідають математичним діям.



Арифметичні операції C#

Бінарні арифметичні операції

Бінарні арифметичні операції виконуються над двома операндами і повертають нове числове значення.

Додавання + складає два значення. Якщо обидва операнди є цілими числами — результат цілий, якщо хоча б один дробовий — результат дробовий:

```
C#
1  using System;
2
3  int heartRate    = 72;
4  int addedBeats   = 15;
5  int newRate      = heartRate + addedBeats;
6
7  Console.WriteLine($"Пульс після навантаження: {newRate.ToString()} уд/хв");
```

Віднімання - обчислює різницю двох значень:

```
C#
1  using System;
2
3  double normalTemp  = 36.6;
4  double patientTemp = 38.2;
5  double excess      = patientTemp - normalTemp;
6
7  Console.WriteLine($"Температура вище норми на: {excess.ToString()} °C");
```

Множення * обчислює добуток двох значень:

```
C#
1  using System;
2
3  double dailyDose = 0.5;   // мл
4  int    days      = 10;
5  double totalDose = dailyDose * days;
6
7  Console.WriteLine($"Загальна доза за курс: {totalDose.ToString()} мл");
```

Ділення / має важливу особливість, яку необхідно засвоїти. Якщо обидва операнди є **цілими числами**, результат теж буде цілим — дробова частина відкидається без округлення. Тобто 10 / 3 дасть 3, а не 3.333.... Якщо результат присвоюється змінній double, це не рятує — сама операція вже виконалась як ціла:

```
C#
1  using System;
2
3  double wrongResult  = 10 / 3;     // обидва літерали — int, результат 3
4  double rightResult  = 10.0 / 3.0; // хоча б один дробовий — результат 3.33...
```

```
5
6  Console.WriteLine($"Цілочисленне ділення: {wrongResult.ToString()}");
7  Console.WriteLine($"Дробове ділення: {rightResult.ToString()}");
```

Щоб отримати дробовий результат при діленні цілих змінних, достатньо явно привести один з операндів до double:

```
C#
1  using System;
2
3  int patients  = 7;
4  int doctors   = 2;
5  double load   = (double)patients / doctors;
6
7  Console.WriteLine($"Навантаження: {load.ToString()} пацієнтів на лікаря");
```

Залишок від ділення % повертає залишок після цілочисленного ділення. Ця операція корисна для перевірки парності числа, циклічного переходу по масиву та багатьох інших задач:

```
C#
1  using System;
2
3  int totalPatients = 17;
4  int bedsPerWard   = 5;
5  int remainder     = totalPatients % bedsPerWard;
6
7  Console.WriteLine($"Залишок після розподілу по палатах: {remainder.ToString()}
   пацієнт(ів)");
8  Console.WriteLine($"17 ділиться на 5 рівно? {(17 % 5 == 0).ToString()}");
```

Унарні операції: інкремент і декремент

Окрему категорію становлять унарні операції, що змінюють значення на одиницю. **Інкремент ++** збільшує значення змінної на 1, **декремент --** — зменшує на 1. Обидві операції мають дві форми.

Префіксна форма (++x, --x) спочатку змінює значення змінної, а потім повертає вже змінене значення як результат. Тобто в момент, коли вираз обчислюється, змінна вже містить нове значення.

Постфіксна форма (x++, x--) спочатку повертає поточне (старе) значення змінної, а вже потім змінює її. Тобто в момент обчислення виразу береться старе значення, і лише після цього змінна оновлюється.

Ця різниця є критично важливою, коли результат операції використовується одночасно з інкрементом або декрементом:

```
C#
1  using System;
2
3  int x1 = 5;
```

```
int z1 = ++x1;   // спочатку x1 стає 6, потім z1 = 6
Console.WriteLine($"Префіксний ++: x1={x1.ToString()}, z1={z1.ToString()}");

int x2 = 5;
int z2 = x2++;   // спочатку z2 = 5 (старе значення), потім x2 стає 6
Console.WriteLine($"Постфіксний ++: x2={x2.ToString()}, z2={z2.ToString()}");
```

Те саме стосується декременту:

```
C#
using System;

int a = 10;
int b = --a;   // a спочатку стає 9, потім b = 9
Console.WriteLine($"Префіксний --: a={a.ToString()}, b={b.ToString()}");

int c = 10;
int e = c--;   // e = 10 (старе значення), потім c стає 9
Console.WriteLine($"Постфіксний --: c={c.ToString()}, e={e.ToString()}");
```

Коли інкремент або декремент стоять окремо в рядку — різниця між формами не має значення, оскільки результат операції нікуди не присвоюється. Обидва записи i++ і ++i однаково збільшать і на 1.

Пріоритет операцій

Коли в одному виразі поєднується кілька операцій, результат залежить від порядку їх виконання. Цей порядок визначається **пріоритетом** (precedence). Арифметичні операції в C# мають такий пріоритет (від вищого до нижчого):

1. Постфіксний інкремент/декремент (x++, x--), а також префіксний (++x, --x);
2. Множення, ділення, залишок (*, /, %);
3. Додавання, віднімання (+, -).

Операції з вищим пріоритетом виконуються першими. Якщо пріоритет однаковий — визначає **асоціативність**: арифметичні оператори є **лівоасоціативними**, тобто виконуються зліва направо. Наприклад, вираз 10 / 5 * 2 трактується як (10 / 5) * 2 = 4, а не 10 / (5 * 2) = 1.

Проілюструємо пріоритет на прикладі:

```
C#
using System;

int a = 3;
int b = 5;
int c = 40;
```

```
 6
 7  // Порядок виконання: спочатку постфіксний c--, потім b * a, потім відніманн
 8  int d = c-- - b * a;
 9  // Після операції: c=39 (декремент відбувся), a=3, b=5, d = 40 - 15 = 25
10  Console.WriteLine($"a={a.ToString()}  b={b.ToString()}  c={c.ToString()}
    d={d.ToString()}");
```

Дужки дозволяють явно змінити порядок обчислень, незалежно від пріоритету. Все, що знаходиться всередині дужок, обчислюється першим:

```
C#
 1  using System;
 2
 3  int a = 3;
 4  int b = 5;
 5  int c = 40;
 6
 7  // Тепер дужки змінюють порядок: спочатку --b, потім c - b, потім * a
 8  int d = (c - (--b)) * a;
 9  // --b: b стає 4; c - 4 = 36; 36 * 3 = 108
10  Console.WriteLine($"a={a.ToString()}  b={b.ToString()}  c={c.ToString()}
    d={d.ToString()}");
```

Асоціативність операторів

Питання асоціативності виникає тоді, коли в одному виразі стоять підряд кілька операторів **одного пріоритету**. Наприклад:

```
C#
1  using System;
2
3  int result = 10 / 5 * 2;
4  // Ліва асоціативність: (10 / 5) * 2 = 2 * 2 = 4
5  Console.WriteLine(result.ToString()); // 4
```

Якби оператори / і * були правоасоціативними, вираз трактувався б як 10 / (5 * 2) = 1. Але арифметичні оператори в C# лівоасоціативні, тому лівий оператор виконується першим. Це звична математична поведінка, що відповідає правилам виконання операцій у рядку зліва направо.

Комплексний приклад: клінічні розрахунки

```
C#
 1  using System;
 2
 3  double weight = 80.0;   // кг
 4  double height = 1.75;   // м
 5
```

```
// Індекс маси тіла: вага / зріст²
double bmi = weight / (height * height);

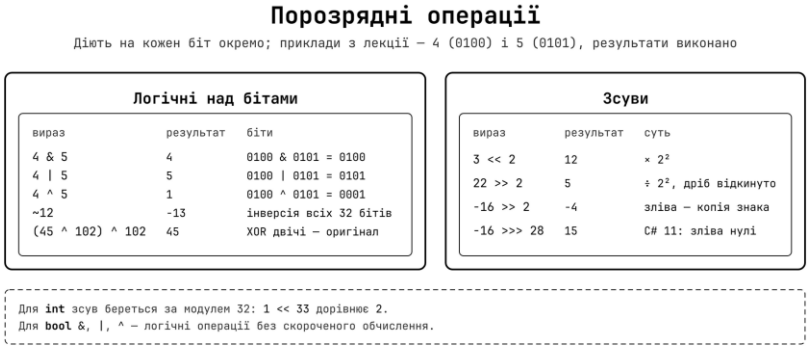
int systolic  = 130;
int diastolic = 85;

// Пульсовий тиск
int pulsePressure = systolic - diastolic;

Console.WriteLine($"Вага: {weight.ToString()} кг, зріст: {height.ToString()} м");
Console.WriteLine($"ІМТ: {bmi.ToString()} кг/м²");
Console.WriteLine($"Систолічний: {systolic.ToString()}, діастолічний: {diastolic.ToString()}");
Console.WriteLine($"Пульсовий тиск: {pulsePressure.ToString()} мм рт.ст.");
```

2.7. Порозрядні операції

Порозрядні (або побітові) операції — це особливий клас операцій, які виконуються не над числом як цілим значенням, а безпосередньо над його окремими **двійковими розрядами (бітами)**. Щоб зрозуміти порозрядні операції, необхідно мати уявлення про те, як числа зберігаються в пам'яті у вигляді послідовності нулів і одиниць. Наприклад, число 5 у двійковому поданні — це 101, число 12 — це 00001100. Порозрядні операції працюють саме з цими позиціями (розрядами) і формують результат, обробляючи кожен розряд незалежно.



Порозрядні операції C#

Побітовий AND — операція &

Операція & (AND, кон'юнкція) виконує **побітове логічне множення**: для кожного розряду результат дорівнює 1 лише тоді, коли в **обох** операндів цей розряд дорівнює 1. Якщо хоча б в одного операнда розряд дорівнює 0 — результат 0.

Розглянемо приклад з числами 4 (двійк. 100) та 5 (двійк. 101):

```
1    100   (4)
2  & 101   (5)
3  -----
4    100   (4)
```

Перший розряд: 1 & 1 = 1. Другий: 0 & 0 = 0. Третій: 0 & 1 = 0. Результат — 100, тобто 4.

```
C#
1  using System;
2
3  int x1 = 2; // двійк. 010
4  int y1 = 5; // двійк. 101
5  Console.WriteLine($"2 & 5 = {(x1 & y1).ToString()}"); // 010 & 101 = 000 = 0
6
7  int x2 = 4; // двійк. 100
8  int y2 = 5; // двійк. 101
9  Console.WriteLine($"4 & 5 = {(x2 & y2).ToString()}"); // 100 & 101 = 100 = 4
```

Операція AND часто використовується для **перевірки або скидання окремих бітів** у числі — наприклад, щоб з'ясувати, чи встановлений певний прапорець у бітовому полі.

Побітовий OR — операція |

Операція | (OR, диз'юнкція) повертає 1 у розряді, якщо **хоча б у одного** з операндів цей розряд дорівнює 1. Тільки якщо обидва розряди дорівнюють 0 — результат 0.

```
C#
1  using System;
2
3  int x1 = 2; // двійк. 010
4  int y1 = 5; // двійк. 101
5  Console.WriteLine($"2 | 5 = {(x1 | y1).ToString()}"); // 010 | 101 = 111 = 7
6
7  int x2 = 4; // двійк. 100
8  int y2 = 5; // двійк. 101
9  Console.WriteLine($"4 | 5 = {(x2 | y2).ToString()}"); // 100 | 101 = 101 = 5
```

Операція OR застосовується для **встановлення бітів** — примусового переведення окремих розрядів у 1.

Побітовий XOR — операція ^

Операція ^ (XOR, виключне АБО) повертає 1, якщо розряди в операндів **різні**: один — 0, інший — 1. Якщо обидва розряди однакові (обидва 0 або обидва 1) — результат 0.

Ця операція має цікаву властивість: якщо застосувати XOR двічі з тим самим ключем, отримаємо вихідне значення. Саме на цьому ґрунтується найпростіший алгоритм симетричного шифрування:

```
C#
 1  using System;
 2
 3  int value   = 45;   // число, яке треба "зашифрувати" (двійк. 101101)
 4  int key     = 102;  // ключ шифрування (двійк. 1100110)
 5
 6  int encrypted = value ^ key;
 7  Console.WriteLine($"Зашифровано: {encrypted.ToString()}"); // 75
 8
 9  int decrypted = encrypted ^ key;
10  Console.WriteLine($"Розшифровано: {decrypted.ToString()}"); // 45
```

Перше застосування XOR «змішує» число з ключем. Друге застосування того самого XOR з тим самим ключем відновлює оригінал. Це відбувається тому, що (x ^ k) ^ k = x — XOR сам є своєю оберненою операцією.

Побітове заперечення — операція ~

Операція ~ (NOT, інверсія) є **унарною** — вона приймає один операнд і **інвертує** всі його розряди: 0 стає 1, 1 стає 0. Для числа 12 (двійк. 00001100) результатом буде 11110011:

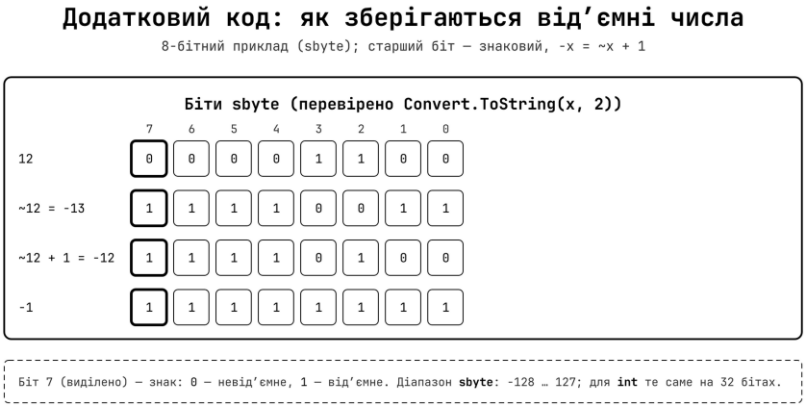
```
C#
1  using System;
2
3  int x = 12;       // двійк. 00001100
4  int inv = ~x;
5  Console.WriteLine($"~12 = {inv.ToString()}"); // -13
```

Результат –13 пояснюється тим, як у C# зберігаються від'ємні цілі числа.

Подання від'ємних чисел: додатковий код

У C# для знакових цілих типів застосовується **додатковий код** (two's complement). У цій системі **старший (лівий) розряд є знаковим**: якщо

він 0 — число додатне, якщо 1 — від'ємне. Решта розрядів разом зі знаковим кодують величину числа у спеціальному форматі.



Подання від'ємних чисел у доповнювальному коді

Щоб отримати від'ємне число з позитивного, виконують два кроки: інвертують усі розряди (~x) і додають одиницю. Тому -12 представляється як ~12 + 1:

```
C#
1  using System;
2
3  int x  = 12;
4  int y  = ~x;   // інверсія всіх розрядів
5  y += 1;        // +1 → отримуємо від'ємне число
6  Console.WriteLine($"~12 + 1 = {y.ToString()}"); // -12
```

Саме тому ~12 дає не просто «інвертовані біти» як беззнакове число, а -13: інверсія 12 дає двійкове значення, яке в signed-форматі читається як -13. Тобто ~x = -(x + 1) — ця формула точно описує поведінку NOT для будь-якого знакового цілого.

Операції зсуву: << і >>

Операції зсуву переміщують усі біти числа **вліво** або **вправо** на задану кількість позицій.

Зсув вліво << додає праворуч нулі, «вштовхуючи» біти числа ліворуч. Кожне зміщення на одну позицію вліво рівнозначне **множенню на 2**. Зсув на n позицій — множенню на 2^n:

```
C#
 1  using System;
 2
 3  int a = 4;   // двійк. 000100
 4  int b = a << 1; // 001000 = 8   (4 * 2)
 5  int c = a << 2; // 010000 = 16  (4 * 4)
 6  int e = a << 3; // 100000 = 32  (4 * 8)
 7
 8  Console.WriteLine($"4 << 1 = {b.ToString()}");
 9  Console.WriteLine($"4 << 2 = {c.ToString()}");
10  Console.WriteLine($"4 << 3 = {e.ToString()}");
```

Зсув вправо >> переміщує біти праворуч; відповідні ліві позиції заповнюються знаковим бітом (для від'ємних чисел — одиницями, для додатних — нулями). Кожне зміщення вправо рівнозначне **цілочисленному діленню на 2**:

```
C#
1  using System;
2
3  int a = 16;  // двійк. 10000
4  int b = a >> 1; // 01000 = 8   (16 / 2)
5  int c = a >> 2; // 00100 = 4   (16 / 4)
6
7  Console.WriteLine($"16 >> 1 = {b.ToString()}");
8  Console.WriteLine($"16 >> 2 = {c.ToString()}");
```

Зсув є більш ефективним, ніж арифметичне ділення або множення, оскільки процесор виконує його за одну команду. Проте використовувати зсуви замість * і / варто лише там, де це справді критично для продуктивності — в решті випадків ясність коду важливіша.

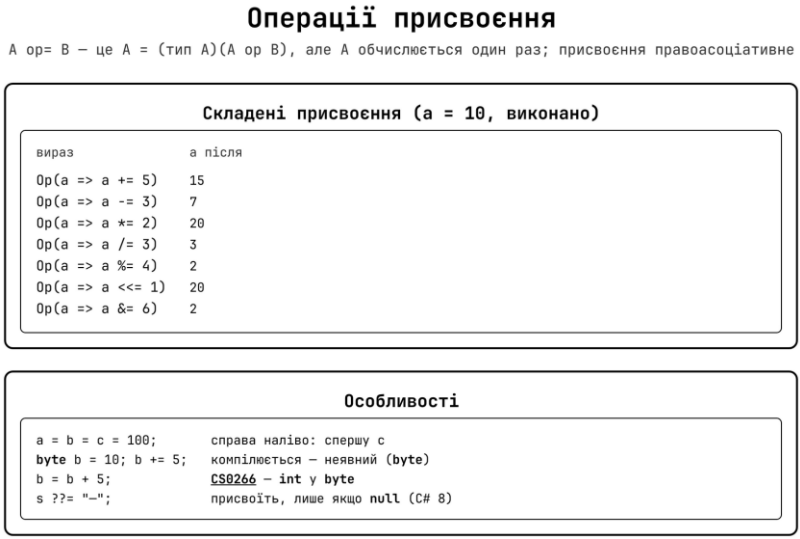
Розглянемо зсув числа, яке не є степенем двійки:

```
C#
1  using System;
2
3  int a = 22;  // двійк. 10110
4  int b = a << 2; // 1011000 = 88
5  int c = a >> 2; // 101 = 5 (дробова частина відкидається)
6
7  Console.WriteLine($"22 << 2 = {b.ToString()}");
8  Console.WriteLine($"22 >> 2 = {c.ToString()}");
```

При зсуві вправо дробова частина ділення втрачається — так само, як і при цілочисленному поділі. Тому 22 >> 2 дає 5, а не 5.5.

2.8. Операції присвоєння

Операція присвоєння — одна з найфундаментальніших операцій у програмуванні. Вона записує значення у змінну, задаючи або оновлюючи її стан. В операції присвоєння беруть участь два операнди: **лівий** (змінна, якій присвоюється значення) і **правий** (вираз, значення якого присвоюється). Лівий операнд завжди повинен бути іменованим виразом, що допускає модифікацію, — зазвичай це змінна або елемент масиву. Звичайний літерал або константа лівим операндом бути не може.



Операції присвоєння C#

Базове присвоєння =

Найпростіша операція присвоєння — =. Вона обчислює вираз праворуч і записує результат у змінну ліворуч:

```
C#
1  using System;
2
3  int patientAge    = 45;
4  double temperature = 37.2;
5  string diagnosis  = "Гіпертонія I ст.";
6
```

```
7  Console.WriteLine($"Вік: {patientAge.ToString()}");
8  Console.WriteLine($"Температура: {temperature.ToString()} °C");
9  Console.WriteLine($"Діагноз: {diagnosis}");
```

Операція присвоєння має **низький пріоритет**: спочатку повністю обчислюється вираз правого операнда, і лише потім результат записується у змінну. Це дозволяє без будь-яких проблем писати a = b * 5 + 2 — множення і додавання виконаються першими.

Множинне присвоєння

C# дозволяє присвоювати одне і те саме значення кільком змінним в одному виразі. Для цього операції присвоєння ставляться підряд. Оскільки присвоєння є **правоасоціативним**, вони виконуються праворуч наліво:

```
C#
1  using System;
2
3  int a, b, c;
4  a = b = c = 100;
5
6  Console.WriteLine($"a={a.ToString()}  b={b.ToString()}  c={c.ToString()}");
```

Тут спочатку виконується c = 100, потім b = c (тобто b = 100), і нарешті a = b (тобто a = 100). Усі три змінні отримують значення 100. Це зручно для ініціалізації групи лічильників, прапорців або результатів одним значенням.

Те саме стосується і складніших виразів правого операнда:

```
C#
1  using System;
2
3  int x, y, z;
4  x = y = z = 34 * 2 / 4;   // Обчислюється вираз 34*2/4=17, потім присвоюється
5
6  Console.WriteLine($"x={x.ToString()}  y={y.ToString()}  z={z.ToString()}");
```

Складені операції присвоєння

Дуже поширена ситуація: змінна бере участь в операції і результат записується назад у ту саму змінну. Наприклад, score = score + 10. Для таких випадків C# пропонує **складені (compound) оператори присвоєння** — скорочені форми, де лівий операнд неявно присутній і з правого боку.

Арифметичні складені присвоєння

Кожен арифметичний оператор має свій відповідний оператор складеного присвоєння:

Оператор	Еквівалент	Дія
A += B	A = A + B	Додавання та присвоєння
A -= B	A = A - B	Віднімання та присвоєння
A *= B	A = A * B	Множення та присвоєння
A /= B	A = A / B	Ділення та присвоєння
A %= B	A = A % B	Залишок та присвоєння

Застосуємо їх послідовно до однієї змінної, спостерігаючи за зміною її значення:

```C#
using System;

int dose = 10;
Console.WriteLine($"Початкова доза: {dose.ToString()} мг");

dose += 5;
Console.WriteLine($"Після += 5:  {dose.ToString()} мг");

dose -= 3;
Console.WriteLine($"Після -= 3:  {dose.ToString()} мг");

dose *= 2;
Console.WriteLine($"Після *= 2:  {dose.ToString()} мг");

dose /= 4;
Console.WriteLine($"Після /= 4:  {dose.ToString()} мг");

dose %= 3;
Console.WriteLine($"Після %= 3:  {dose.ToString()} мг");
```

Такий стиль запису не лише коротший, але й зменшує ризик помилок: немає небезпеки написати a = b + 5 замість a = a + 5, якщо обидві змінні мають схожі імена.

Побітові складені присвоєння

Аналогічні скорочені форми є й для порозрядних операцій та операцій зсуву:

```C#
using System;

int flags = 0b00001010;  // початковий стан бітів

flags |= 0b00000001;     // встановити біт 0 → 0b00001011
Console.WriteLine($"Після |=: {flags.ToString()}");

flags &= 0b00001111;     // зберегти лише молодші 4 біти → 0b00001011
Console.WriteLine($"Після &=: {flags.ToString()}");

flags ^= 0b00001001;     // інвертувати біти 0 і 3 → 0b00000010
Console.WriteLine($"Після ^=: {flags.ToString()}");

flags <<= 3;             // зсунути вліво на 3 → 0b00010000 = 16
Console.WriteLine($"Після <<=3: {flags.ToString()}");

flags >>= 1;             // зсунути вправо на 1 → 0b00001000 = 8
Console.WriteLine($"Після >>=1: {flags.ToString()}");
```

Побітові складені присвоєння використовуються при роботі з бітовими прапорцями — компактним способом зберігання кількох булевих значень в одному цілому числі.

Правоасоціативність операцій присвоєння

На відміну від арифметичних операцій (які лівоасоціативні), всі операції присвоєння в C# є **правоасоціативними**: вираз, що стоїть праворуч, обчислюється першим і передає своє значення лівому. Це дозволяє будувати ланцюжки присвоєнь:

```C#
using System;

int a = 8;
int b = 6;

int c = a += b -= 5;
// Покроково:
// 1. b -= 5  →  b = 6 - 5 = 1
// 2. a += b  →  a = 8 + 1 = 9
// 3. c = a   →  c = 9

Console.WriteLine($"a={a.ToString()}, b={b.ToString()}, c={c.ToString()}");
```

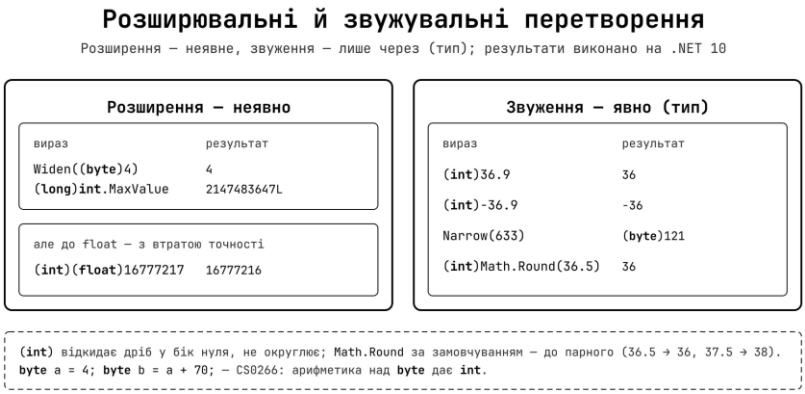
Розуміння правоасоціативності важливе при читанні таких ланцюжків: слід читати їх зсередини назовні або справа наліво, крок за кроком обчислюючи кожне підвираз.

Практичний приклад: накопичення витрат

```
C#
 1  using System;
 2
 3  decimal totalCost = 0m;
 4
 5  decimal consultationFee = 350m;
 6  decimal analysisFee     = 180m;
 7  decimal medicationCost  = 420m;
 8
 9  totalCost += consultationFee;
10  totalCost += analysisFee;
11  totalCost += medicationCost;
12
13  decimal discount = totalCost * 0.1m;
14  totalCost -= discount;
15
16  Console.WriteLine($"Консультація:    {consultationFee.ToString()} грн");
17  Console.WriteLine($"Аналізи:         {analysisFee.ToString()} грн");
18  Console.WriteLine($"Ліки:            {medicationCost.ToString()} грн");
19  Console.WriteLine($"Знижка 10%:     -{discount.ToString()} грн");
20  Console.WriteLine($"Підсумок:        {totalCost.ToString()} грн");
```

2.9. Перетворення базових типів

Програми рідко працюють лише з даними одного типу. На практиці часто виникає потреба передати значення одного типу туди, де очікується інший: зберегти byte у змінну int, або навпаки — помістити int у byte. Саме тут вступає в дію **перетворення типів** (type conversion). Розуміння того, коли і як відбуваються перетворення, є необхідним для написання коректного і передбачуваного коду.



Перетворення базових типів

Чому потрібне перетворення типів

Розглянемо просту ситуацію: є змінна типу byte зі значенням 4. Ми хочемо додати до неї число і записати результат у нову byte-змінну:

```
C#
1  using System;
2
3  byte a = 4;
4  int  b = a + 70;  // це працює: результат int, записується в int
5
6  Console.WriteLine(b.ToString()); // 74
```

Але якщо спробувати записати результат у byte-змінну:

```
C#
1  byte a = 4;
2  byte b = a + 70;  // помилка компіляції!
```

Компілятор відмовить у цьому навіть попри те, що число 74 чудово вписується в діапазон byte (0–255). Причина полягає в тому, що результат операції додавання для цілих типів з розрядністю не більше int завжди повертається як int — 4-байтове ціле. Компілятор не може без явної вказівки записати 4-байтовий результат у 1-байтову змінну, оскільки це потенційно небезпечна операція.

Операція явного приведення типу

Щоб вирішити цю проблему, застосовується **явне приведення типів** (explicit cast). Синтаксис простий: перед значенням або виразом у круглих дужках вказується тип, до якого потрібно привести:

```
(цільовий_тип) вираз
```

Застосуємо до попереднього прикладу:

```
C#
using System;

byte a = 4;
byte b = (byte)(a + 70);

Console.WriteLine(b.ToString()); // 74
```

Дужки навколо (a + 70) важливі: без них операція приведення застосується тільки до a, а потім до результату (byte)a + 70 знову буде складання двох значень і повернення int. Зовнішні дужки гарантують, що спочатку обчислюється вся сума, і вже потім виконується приведення всього результату до byte.

Розширювальні перетворення (widening)

Розширювальне перетворення (widening conversion) відбувається, коли значення типу з меншим розміром у пам'яті записується у змінну типу з більшим розміром. Наприклад, byte займає 1 байт (8 біт), а int — 4 байти (32 біти). Записати byte в int — це розширення: місця більше, ніж потрібно, і дані точно поміщаються без втрат.

Розширювальне перетворення виконується **автоматично** і не потребує жодних явних дій з боку розробника — компілятор сам додає необхідні нулі зліва:

```
C#
using System;

byte   a = 200;
int    b = a;
long   c = b;
double e = c;

Console.WriteLine($"byte  {a.ToString()}  → int  {b.ToString()}");
Console.WriteLine($"int   {b.ToString()} → long {c.ToString()}");
Console.WriteLine($"long  {c.ToString()} → double {e.ToString()}");
```

Внутрішньо відбувається **доповнення нулями** (zero extension) для беззнакових типів: ліві біти заповнюються нулями. Для знакових типів відбувається **sign extension**: ліві біти заповнюються копією знакового біту — нулями для позитивних чисел і одиницями для від'ємних. Саме завдяки цьому від'ємні числа залишаються від'ємними після розширення:

```
C#
1  using System;
2
3  sbyte negative = -4;   // двійк. 11111100
4  short wider    = negative;  // двійк. 1111111111111100
5
6  Console.WriteLine($"sbyte {negative.ToString()} → short {wider.ToString()}");
```

Звужувальні перетворення (narrowing)

Звужувальне перетворення (narrowing conversion) — це зворотна ситуація: значення типу з більшим розміром записується в змінну меншого типу. Наприклад, int у byte, або double у int. Тут вже немає гарантії, що значення поміститься без втрат — старші біти просто відкидаються.

Саме тому звужувальне перетворення вимагає **явного приведення** типу: розробник повинен свідомо підтвердити, що розуміє можливу втрату даних і бере за неї відповідальність.

```
C#
1  using System;
2
3  ushort a = 300;
4  byte   b = (byte)a;    // старший байт відкидається!
5
6  Console.WriteLine($"ushort {a.ToString()} → byte {b.ToString()}"); // 44, не
   300!
```

При перетворенні double на int відкидається дробова частина — без округлення:

```
C#
1  using System;
2
3  double temperature = 36.9;
4  int    truncated   = (int)temperature;  // 36, не 37!
5
6  Console.WriteLine($"double {temperature.ToString()} → int
   {truncated.ToString()}");
```

Важливо усвідомити: (int) не округлює, а **відкидає** дробову частину. Число 36.9 перетвориться не в 37, а в 36. Для коректного математичного округлення до найближчого цілого потрібно використовувати Math.Round.

Автоматичні безпечні перетворення

Компілятор C# виконує розширювальні перетворення автоматично — тільки в тих випадках, коли значення гарантовано не вийде за межі цільового типу. Ось основні ланцюжки таких перетворень (повний перелік — у розділі 2.10):

```
1  byte  → short  → int  → long  → decimal
2  int   → double
3  short → float  → double
4  char  → int    → long
```

«Не вийде за межі» не означає «без втрат». Перетворення int, uint, long, ulong → float і long, ulong → double теж неявні, але можуть **втратити точність**: float має лише 24 біти мантиси, double — 53. Наприклад, float f = 16777217; зберігає 16777216, і компілятор не попереджає про це.

Звернімо увагу на особливість пари double і decimal: незважаючи на те, що decimal займає 16 байт (проти 8 у double) і забезпечує вищу точність — перетворення між ними **не є автоматичним ні в одному напрямку**. Обидва типи потребують явного приведення:

```
C#
 1  using System;
 2
 3  double  d = 3.14;
 4  decimal m = (decimal)d;
 5
 6  decimal price = 99.99m;
 7  double  p     = (double)price;
 8
 9  Console.WriteLine($"double → decimal: {m.ToString()}");
10  Console.WriteLine($"decimal → double: {p.ToString()}");
```

Причина полягає в тому, що double і decimal мають фундаментально різну внутрішню організацію: double — це IEEE 754 бінарне число з плаваючою точкою, а decimal — десяткове число з фіксованою точністю. Пряме перетворення між ними може призвести до незначних похибок, тому компілятор змушує розробника зробити це явно — щоб він розумів, що таке перетворення відбувається.

Приклад: обчислення з перетвореннями типів

```C#
using System;

int totalPatients = 37;
int malePatients  = 14;

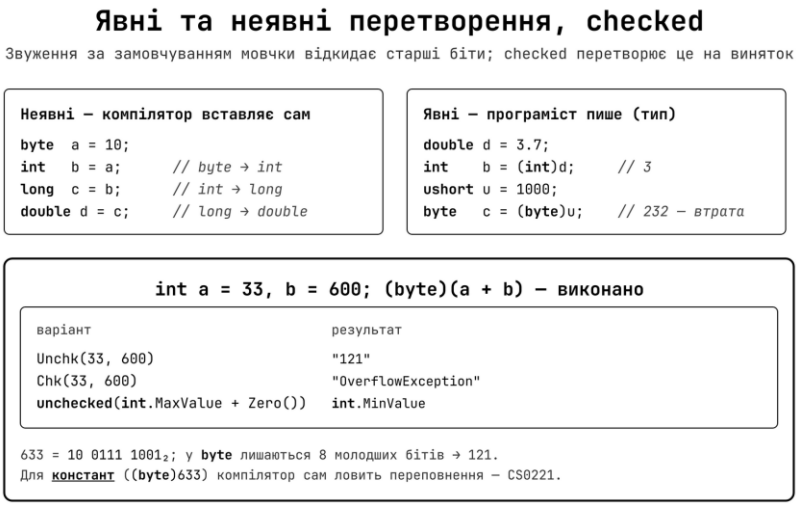
// Явне приведення для отримання дробового результату ділення
double maleRatio = (double)malePatients / totalPatients * 100;

Console.WriteLine($"Загалом пацієнтів: {totalPatients.ToString()}");
Console.WriteLine($"Чоловіків: {malePatients.ToString()}");
Console.WriteLine($"Частка чоловіків: {maleRatio.ToString()} %");

// Зворотне: дробовий результат → ціла частина (дробова відкидається)
int wholePercent = (int)maleRatio;
Console.WriteLine($"У цілих відсотках: {wholePercent.ToString()} %");
```

2.10. Явні та неявні перетворення

У попередньому розділі ми з'ясували, що перетворення типів буває розширювальним і звужувальним. Але є ще одна важлива класифікація: **явні** (explicit) та **неявні** (implicit) перетворення. Ця класифікація описує не напрямок перетворення (ширше чи вужче), а те, хто відповідає за його виконання — компілятор автоматично чи розробник вручну.



Явні та неявні перетворення

Неявні перетворення

Неявне перетворення (implicit conversion) — це таке перетворення, яке компілятор виконує **автоматично**, без жодних явних вказівок у коді. Розробнику достатньо просто записати значення у змінну іншого типу, і все відбувається само собою.

Неявні перетворення можливі лише в тих випадках, коли вони **гарантовано безпечні щодо діапазону** — тобто коли жодне значення вихідного типу не може вийти за межі цільового типу. Саме тому компілятор без вагань виконує, наприклад, byte → int: будь-яке значення byte (0–255) без проблем вписується в int (–2 млрд до +2 млрд).

```
C#
1  using System;
2
3  byte   age  = 45;
4  int    ageI = age;    // byte → int: неявно, безпечно
5  long   ageL = ageI;  // int → long: неявно, безпечно
6  double ageD = ageL;  // long → double: неявно, безпечно
7
8  Console.WriteLine($"byte {age.ToString()} → int {ageI.ToString()} → long
   {ageL.ToString()} → double {ageD.ToString()}");
```

Розширення зі знаком (sign extension)

Коли неявно розширюється беззнаковий тип (наприклад, byte), ліворуч додаються **нулі**. Це називається доповненням нулями. Але коли розширюється **знаковий** тип (sbyte, short, int), механізм інший: ліворуч додається копія **знакового розряду** (старшого біту). Для позитивних чисел знаковий розряд дорівнює 0 — додаються нулі. Для від'ємних він дорівнює 1 — додаються одиниці. Завдяки цьому від'ємне число після розширення залишається від'ємним і зберігає своє значення:

Розглянемо розширення позитивного числа. Знаковий біт 0 — ліворуч додаються нулі:

```
C#
1  using System;
2
3  sbyte positiveSmall = 4;
4  short positiveWide  = positiveSmall; // 00000100 → 0000000000000100
5
6  Console.WriteLine($"sbyte {positiveSmall.ToString()} → short
   {positiveWide.ToString()}");
```



Sign extension позитивного числа: sbyte 4 → short

Розширення від'ємного числа: знаковий біт 1 — ліворуч додаються одиниці, завдяки чому від'ємне значення зберігається:

```
C#
1  using System;
2
3  sbyte negativeSmall = -4;
4  short negativeWide  = negativeSmall; // 11111100 → 1111111111111100
5
6  Console.WriteLine($"sbyte {negativeSmall.ToString()} → short
   {negativeWide.ToString()}");
```



Sign extension від'ємного числа: sbyte -4 → short

Без цього механізму розширення знакових від'ємних чисел давало б абсолютно невірний результат — замість -4 отримали б якесь величезне позитивне число.

Явні перетворення

Явне перетворення (explicit conversion) — це перетворення, яке компілятор **не може виконати автоматично** без явної вказівки розробника. Причина в тому, що такі перетворення потенційно небезпечні: не для всіх значень вихідного типу гарантується коректний результат у цільовому типі. Розробник повинен сам сказати: «Я розумію ризик і беру відповідальність».

Синтаксис явного приведення:

```
1  (цільовий_тип) вираз
```

```
C#
 1  using System;
 2
 3  int    a = 4;
 4  int    b = 6;
 5  byte   c = (byte)(a + b);  // int → byte, явне
 6
 7  double pi    = 3.14159;
 8  int    piInt = (int)pi;    // double → int, дробова частина відкидається
 9
10  Console.WriteLine($"(byte)(4 + 6) = {c.ToString()}");
11  Console.WriteLine($"(int)3.14159 = {piInt.ToString()}");
```

Безпечні ланцюжки неявних перетворень

Компілятор дозволяє неявні перетворення, де значення гарантовано вміщується в цільовий тип. Основні ланцюжки:

```
1  byte  → short → int → long → decimal
2  int   → double
3  short → float  → double
4  char  → int    → long
```

Це спрощена картина. Повний перелік неявних числових перетворень за специфікацією C#:

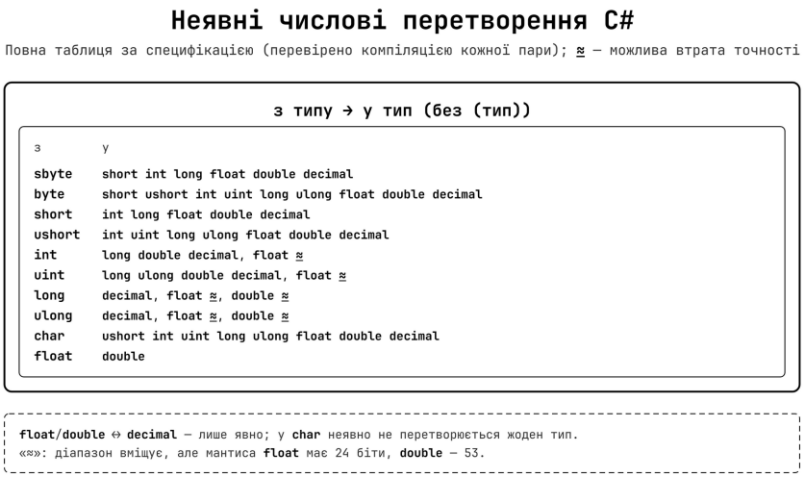
Звідки	Куди (неявно)
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal

Звідки	Куди (неявно)
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal
char	ushort, int, uint, long, ulong, float, double, decimal
float	double

Гарантія стосується **діапазону**, а не точності. Перетворення цілих типів у float (з int, uint, long, ulong) і в double (з long, ulong) можуть мовчки втратити молодші цифри: у float лише 24 біти мантиси, у double — 53.

```
C#
1  using System;
2
3  int    big = 16777217;          // 2^24 + 1
4  float  f   = big;                // неявно, без попередження
5  Console.WriteLine($"int → float: {((int)f).ToString()}");    // 16777216
6
7  long   huge = 9007199254740993;  // 2^53 + 1
8  double d    = huge;              // неявно, без попередження
9  Console.WriteLine($"long → double: {((long)d).ToString()}"); // 9007199254740992
```

Зверніть увагу також на відсутні перетворення: зі знакових типів у беззнакові (int → uint, sbyte → byte) неявного шляху немає — від'ємне значення не вміщується в беззнаковий тип.



Таблиця безпечних неявних перетворень

Усе, що виходить за межі цих ланцюжків, вимагає явного (тип). Зокрема, хоча decimal технічно «більший» за double (28–29 значущих цифр проти 15–16), пряме неявне перетворення між ними **не підтримується ні в якому напрямку**. Ці типи принципово різні за структурою: double — це IEEE 754 бінарне число з плаваючою точкою, а decimal — десяткове число з фіксованою точністю. Пряме перетворення між ними може призвести до незначних похибок, тому компілятор змушує розробника зробити це явно:

```C#
using System;

double  d = 4.5;
decimal m = (decimal)d;   // обов'язково явне

decimal price = 199.99m;
double  p     = (double)price;  // теж обов'язково явне

Console.WriteLine($"double {d.ToString()} → decimal {m.ToString()}");
Console.WriteLine($"decimal {price.ToString()} → double {p.ToString()}");
```

Втрата даних при явному перетворенні

Явне перетворення до типу меншої розрядності нерідко призводить до **втрати даних**. Якщо значення не вписується в цільовий тип — старші біти просто відкидаються, і результат виявляється невірним:

```
C#
1  using System;
2
3  int  a = 33;
4  int  b = 600;
5  byte c = (byte)(a + b);   // 633 не вписується в byte (0–255)
6
7  Console.WriteLine($"(byte)(33 + 600) = {c.ToString()}"); // 121, а не 633!
```

Чому 121? Число 633 у двійковому вигляді — 10 01111001. Тип byte зберігає лише молодші 8 біт: 01111001 — це 121. Старший біт 1 відкинутий. Програма не повідомить про помилку — вона просто продовжить роботу з неправильним значенням. Саме це й є найнебезпечнішим у переповненні: помилка залишається прихованою.

Ключове слово checked

Щоб захиститись від мовчазного переповнення, C# надає ключове слово **checked**. Воно змушує середовище виконання перевіряти, чи не виходить результат арифметичної операції або приведення за межі цільового типу. Якщо виходить — кидається виняток OverflowException.

```
C#
 1  using System;
 2
 3  try
 4  {
 5      int  a = 33;
 6      int  b = 600;
 7      byte c = checked((byte)(a + b));  // кине OverflowException
 8      Console.WriteLine(c.ToString());
 9  }
10  catch (OverflowException ex)
11  {
12      Console.WriteLine($"Переповнення: {ex.Message}");
13  }
```

Без checked — програма мовчки дасть 121. З checked — виняток зупинить виконання в місці переповнення, і ми одразу побачимо проблему. Це особливо важливо в математично критичних місцях — розрахунках доз ліків, фінансових транзакціях або алгоритмах, де точність є необхідністю.

Існує також протилежне ключове слово **unchecked** — воно явно вказує, що переповнення дозволене і не має кидати виняток. Для виразів, що обчислюються під час виконання, це стандартна поведінка за замовчуванням (якщо в проєкті не ввімкнено <CheckForOverflowUnderflow>). **Константні** вирази компілятор

перевіряє завжди: (byte)633 чи (byte)(33 + 600) з літералами дають помилку компіляції CS0221, а int.MaxValue + 1 — CS0220; саме тому в прикладах вище значення зберігаються у змінних a і b (явний unchecked((byte)633) цю перевірку знімає і дає 121). Крім того, unchecked може знадобитися всередині checked-блоку, якщо для якоїсь конкретної операції переповнення є навмисним:

```
C#
1  using System;
2
3  int big    = int.MaxValue;
4  int result = unchecked(big + 1);  // wraparound навмисно
5
6  Console.WriteLine($"int.MaxValue + 1 (unchecked) = {result.ToString()}");
```

checked і unchecked можна застосовувати як до окремого виразу (через дужки), так і до цілого блоку коду:

```
C#
1  checked
2  {
3      // Усі арифметичні операції в цьому блоці перевіряються
4      int x = int.MaxValue;
5      int y = x + 1;  // OverflowException
6  }
```

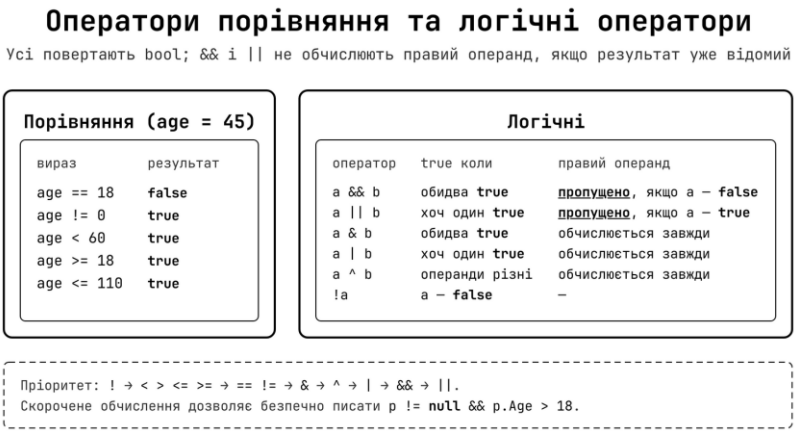
На практиці checked доцільно вмикати для критичних обчислень, де переповнення неприпустиме і повинно бути негайно виявлене, а не призводити до прихованих помилок у даних.

Комплексний приклад: зважений розрахунок дози

```
C#
 1  using System;
 2
 3  double weightKg  = 72.5;
 4  double dosePerKg = 0.8;   // мг/кг
 5
 6  double  totalDoseD = weightKg * dosePerKg;
 7  decimal totalDoseM = (decimal)totalDoseD;   // явне: double → decimal
 8
 9  Console.WriteLine($"Вага пацієнта:  {weightKg.ToString()} кг");
10  Console.WriteLine($"Доза:           {dosePerKg.ToString()} мг/кг");
11  Console.WriteLine($"Разова доза (double):  {totalDoseD.ToString()} мг");
12  Console.WriteLine($"Разова доза (decimal): {totalDoseM.ToString()} мг");
13
14  int doseMgInt = (int)totalDoseM;
15  Console.WriteLine($"Округлено (відкидання): {doseMgInt.ToString()} мг");
```

2.11. Умовні вирази

Програми рідко виконуються лінійно від початку до кінця. Зазвичай потрібно приймати рішення: перевірити, чи вік пацієнта вписується в допустимий діапазон, чи є значення тиску критичним, чи не перевищено добову норму ліків. Для цього в C# існують **умовні вирази** — вирази, що повертають логічне значення типу bool: true (істина) або false (хибність). До умовних виразів належать оператори порівняння та логічні оператори.



Умовні вирази C#: оператори порівняння та логічні оператори

Оператори порівняння

Оператори порівняння порівнюють два операнди і повертають bool. Якщо твердження відповідає дійсності — результат true, інакше — false.

Оператор	Значення	Приклад
==	Рівність	age == 18
!=	Нерівність	age != 0
<	Менше ніж	age < 60
>	Більше ніж	age > 18

Оператор	Значення	Приклад
<=	Менше або рівно	age <= 110
>=	Більше або рівно	age >= 0

Оператори <, >, <=, >= мають вищий пріоритет ніж == і !=.

```
C#
 1  using System;
 2
 3  int age        = 45;
 4  int criticalBp = 180;
 5  int bpSystolic = 155;
 6
 7  Console.WriteLine($"Вік == 45: {(age == 45).ToString()}");
 8  Console.WriteLine($"Вік != 18: {(age != 18).ToString()}");
 9  Console.WriteLine($"Тиск < критичного: {(bpSystolic <
    criticalBp).ToString()}");
10  Console.WriteLine($"Вік >= 60 (пенсіонер): {(age >= 60).ToString()}");
```

Логічні оператори

Логічні оператори об'єднують кілька умовних виразів в один. Усі вони приймають операнди типу bool і повертають bool.

&& — логічне І (AND, кон'юнкція)

Повертає true лише тоді, коли **обидва** операнди дорівнюють true. Якщо хоч один — false, результат false.

```
C#
 1  using System;
 2
 3  int age        = 45;
 4  int bpSystolic = 155;
 5
 6  bool isAdultNormalBp = age >= 18 && bpSystolic < 140;
 7  bool isAdultHighBp   = age >= 18 && bpSystolic >= 140;
 8
 9  Console.WriteLine($"Дорослий і нормальний тиск: {isAdultNormalBp.ToString()}");
10  Console.WriteLine($"Дорослий і підвищений тиск: {isAdultHighBp.ToString()}");
```

|| — логічне АБО (OR, диз'юнкція)

Повертає true, якщо **хоча б один** операнд дорівнює true. false лише коли обидва — false.

```
C#
1  using System;
```

```
2
3  int age = 8;
4
5  bool needsSpecialCare = age < 12 || age >= 70;
6  Console.WriteLine($"Потребує особливого догляду:
   {needsSpecialCare.ToString()}");
```

! — логічне НЕ (NOT, заперечення)

Унарний оператор: інвертує значення. !true → false, !false → true.

```
C#
1  using System;
2
3  bool isStableCondition = true;
4  bool isUnstable = !isStableCondition;
5  Console.WriteLine($"Нестабільний стан: {isUnstable.ToString()}");
```

^ — виключне АБО (XOR)

Повертає true, якщо операнди **різні**: один true, інший false. Якщо обидва однакові — повертає false.

```
C#
1  using System;
2
3  bool hasFever    = true;
4  bool hasCough    = true;
5
6  bool onlyOneSym  = hasFever ^ hasCough;
7  Console.WriteLine($"Лише один симптом з двох: {onlyOneSym.ToString()}");
```

| і & — операції без скорочення

C# також має пари | (логічне АБО) і & (логічне І), які на відміну від || і && **завжди обчислюють обидва операнди**. Це принципова різниця.

У виразі x || y: якщо x вже є true, обчислення y пропускається — результат і так true. Аналогічно x && y: якщо x є false, обчислення y пропускається. Це називається **скороченим обчисленням** (short-circuit evaluation) і підвищує продуктивність.

```
C#
 1  using System;
 2
 3  int bpSystolic = 130;
 4  int bpDiastolic = 85;
 5
 6  // && — якщо bpSystolic <= 140 вже false, bpDiastolic не перевіряється
 7  bool isOptimal = bpSystolic <= 130 && bpDiastolic <= 80;
 8  Console.WriteLine($"Оптимальний тиск: {isOptimal.ToString()}");
 9
10  // & — обидва операнди завжди обчислюються
11  bool alsoOptimal = bpSystolic <= 130 & bpDiastolic <= 80;
```

```
12  Console.WriteLine($"Той самий результат: {alsoOptimal.ToString()}");
```

Оператори | і & зазвичай використовують для порозрядних операцій над цілими числами (що розглядалося в розділі 2.7). Для логічних перевірок умов застосовуйте && і || — вони ефективніші.

Комбінування умов

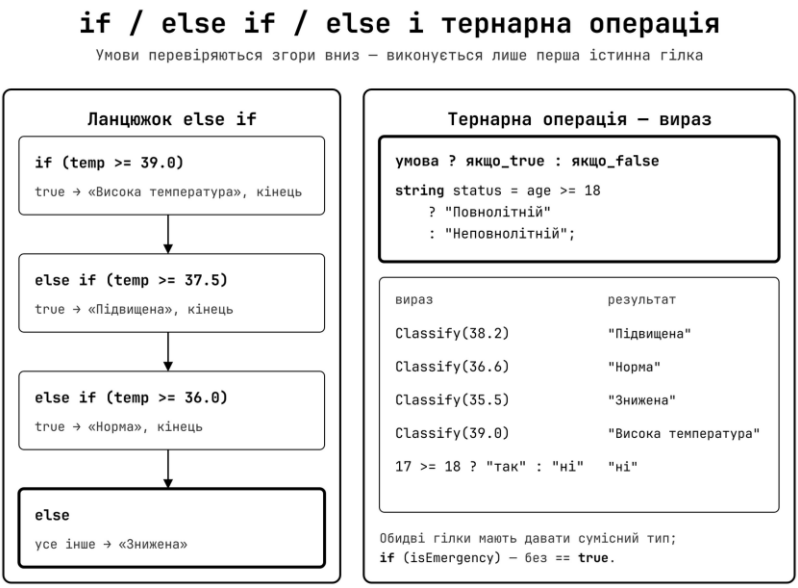
Умови можна поєднувати в складні вирази. Пріоритет: спочатку !, потім <, >, <=, >=, потім ==, !=, потім &, потім ^, потім |, потім &&, і нарешті ||. Для ясності завжди використовуйте дужки:

```
C#
 1  using System;
 2
 3  int age        = 45;
 4  int bpSystolic = 165;
 5  bool hasDiabetes = true;
 6
 7  // Пацієнт у зоні ризику: або дуже високий тиск, або (старший 40 і діабет)
 8  bool highRisk = bpSystolic >= 160 || (age > 40 && hasDiabetes);
 9  Console.WriteLine($"Зона ризику: {highRisk.ToString()}");
10
11  // Допускається до операції: немає критичного тиску І не діабетик, або вік < 30
12  bool canOperate = (bpSystolic < 180 && !hasDiabetes) || age < 30;
13  Console.WriteLine($"Допуск до операції: {canOperate.ToString()}");
```

Результат умовного виразу — значення типу bool. Його можна зберігати у змінну, передавати в конструкції if, використовувати як аргумент методу або відразу виводити на екран.

2.12. Конструкція if..else та тернарна операція

Умовні вирази (розділ 2.11) повертають true або false — але самі по собі нічого не роблять. Щоб програма реагувала на результат перевірки і виконувала різні дії залежно від умови, використовуються **умовні конструкції**. Найпоширеніша з них — if...else.



Конструкція if/else та тернарна операція

Блок if

Найпростіша форма — блок if без гілки else. Якщо умова в дужках дорівнює true, виконується тіло блоку; якщо false — тіло пропускається і програма продовжується далі:

```
C#
1  if (умова)
2  {
3      // виконується лише коли умова == true
4  }
```

```
C#
 1  using System;
 2
 3  int bpSystolic = 165;
 4
 5  if (bpSystolic >= 160)
 6  {
 7      Console.WriteLine("Увага: критично підвищений артеріальний тиск!");
 8      Console.WriteLine("Необхідна негайна консультація лікаря.");
 9  }
10
11  Console.WriteLine("Огляд завершено.");
```

Якщо тіло блоку складається лише з однієї інструкції, фігурні дужки можна опустити. Проте для читабельності та уникнення помилок краще завжди їх писати:

```
C#
1  if (bpSystolic >= 160)
2      Console.WriteLine("Критично підвищений тиск!");  // одна інструкція
```

Блок else

Блок else виконується, якщо умова блоку if дорівнює false. Таким чином одна з двох гілок завжди виконається:

```
C#
 1  using System;
 2
 3  int age = 15;
 4
 5  if (age >= 18)
 6  {
 7      Console.WriteLine("Пацієнт повнолітній. Реєстрація без обмежень.");
 8  }
 9  else
10  {
11      Console.WriteLine("Пацієнт неповнолітній. Потрібна згода батьків.");
12  }
```

Ланцюжок else if

Коли варіантів більше двох, до конструкції додаються гілки else if. CLR перевіряє умови **зверху вниз** і виконує першу гілку, умова якої є true. Решта гілок при цьому ігноруються:

```
C#
 1  using System;
 2
 3  int bpSystolic = 145;
 4
 5  if (bpSystolic < 120)
 6  {
 7      Console.WriteLine("Тиск: оптимальний.");
 8  }
 9  else if (bpSystolic < 130)
10  {
11      Console.WriteLine("Тиск: нормальний.");
12  }
13  else if (bpSystolic < 140)
14  {
15      Console.WriteLine("Тиск: підвищено-нормальний. Спостереження.");
16  }
17  else if (bpSystolic < 160)
18  {
```

```
    Console.WriteLine("Тиск: гіпертонія I ступеня. Консультація.");
}
else if (bpSystolic < 180)
{
    Console.WriteLine("Тиск: гіпертонія II ступеня. Лікування.");
}
else
{
    Console.WriteLine("Тиск: гіпертонічний криз! Екстрена допомога!");
}
```

Останній блок else є необов'язковим, але виступає «страхувальною» гілкою, що обробляє всі інші випадки. В медичних системах рекомендується завжди його вказувати.

Вкладені if

Умовні конструкції можна вкладати одна в одну. При цьому важливо стежити за відступами, щоб не заплутатись у структурі:

```
C#
using System;

int age        = 72;
int bpSystolic = 170;

if (age >= 60)
{
    Console.WriteLine("Пацієнт: похилого віку.");

    if (bpSystolic >= 160)
    {
        Console.WriteLine("Стан: підвищений ризик. Госпіталізація.");
    }
    else
    {
        Console.WriteLine("Стан: стабільний. Профілактичне спостереження.");
    }
}
else
{
    Console.WriteLine("Пацієнт: молодший вікової групи ризику.");
}
```

Тернарна операція

Тернарна (умовна) операція є компактним варіантом if...else для **виразів** — тобто там, де потрібно обрати одне з двох значень залежно від умови. Синтаксис:

```
умова ? значення_якщо_true : значення_якщо_false
```

Оператор ? приймає три операнди, звідки й назва «тернарний». Якщо умова true — результатом є другий операнд; якщо false — третій:

```
C#
 1  using System;
 2
 3  int age = 45;
 4
 5  string ageGroup = age >= 60 ? "Пенсіонер" : "Дорослий";
 6  Console.WriteLine($"Вікова група: {ageGroup}");
 7
 8  int bpSystolic = 145;
 9  string bpStatus = bpSystolic >= 140 ? "Підвищений" : "Нормальний";
10  Console.WriteLine($"Тиск: {bpStatus} ({bpSystolic.ToString()} мм рт.ст.)");
```

Тернарний оператор зручний для коротких виразів. Для складних умов або кількох дій краще використовувати повну конструкцію if...else — це підвищує читабельність і спрощує підтримку коду.

Тернарні оператори можна вкладати один в одний, але більше одного рівня вкладення суттєво знижує зрозумілість:

```
C#
 1  using System;
 2
 3  int bpSystolic = 145;
 4
 5  string bpCategory = bpSystolic < 120 ? "Оптимальний"
 6                    : bpSystolic < 140 ? "Нормальний"
 7                    : bpSystolic < 160 ? "Гіпертонія I"
 8                    : "Гіпертонія II+";
 9
10  Console.WriteLine($"Категорія тиску: {bpCategory}");
```

Умова як значення bool

В умові блоку if може стояти будь-який вираз, що повертає bool — у тому числі змінна типу bool, результат порівняння або повернене значення методу:

```
C#
 1  using System;
 2
 3  bool isEmergency   = true;
 4  int  bpSystolic    = 185;
 5
 6  if (isEmergency)
 7      Console.WriteLine("Режим екстреної допомоги активовано.");
 8
 9  if (isEmergency && bpSystolic > 180)
10      Console.WriteLine("Гіпертонічний криз — виклик реанімаційної бригади!");
```

Ніколи не пишіть if (isEmergency == true) — це надлишок. if (isEmergency) і if (!isEmergency) — коректний і лаконічний стиль.

2.13. Цикли

Часто в програмі потрібно виконати одну й ту саму дію багато разів: перебрати всіх пацієнтів у списку, підрахувати суму показників, знайти максимальне значення серед вимірювань. Для цього в C# існують **цикли** — конструкції, що повторюють блок коду доти, поки виконується певна умова.

У C# є чотири основні типи циклів:

— for — коли кількість ітерацій відома або можна її обчислити

— while — коли умова перевіряється до виконання тіла

— do...while — коли тіло повинно виконатись хоча б один раз

— foreach — для перебору колекцій (масивів, списків тощо)



Типи циклів у C#

Цикл for

Цикл for — найуніверсальніший для випадків, коли кількість ітерацій відома заздалегідь. Оголошення складається із трьох частин, розділених крапкою з комою:

```
C#
1  for (ініціалізація; умова; крок)
2  {
3      // тіло циклу
4  }
```

— **Ініціалізація** — виконується один раз перед першою ітерацією (зазвичай оголошення лічильника).

— **Умова** — перевіряється **до кожної** ітерації; якщо false — цикл завершується.

— **Крок** — виконується **після кожної** ітерації (зазвичай збільшення або зменшення лічильника).

```
C#
1  using System;
2
3  // Виводимо дані 5 пацієнтів
4  string[] patientNames = { "Іван Петренко", "Марія Сидоренко", "Олег Бойко",
   "Тетяна Коваль", "Сергій Мельник" };
5
6  for (int i = 0; i < patientNames.Length; i++)
7  {
8      Console.WriteLine($"Пацієнт {(i + 1).ToString()}: {patientNames[i]}");
9  }
```

Розберемо крок за кроком: int i = 0 — оголошуємо лічильник i з нуля. i < patientNames.Length — цикл триває, поки i не досягне довжини масиву (тобто 5). i++ — після кожної ітерації лічильник збільшується на 1. Таким чином виконується рівно 5 ітерацій: i = 0, 1, 2, 3, 4.

Кількість ітерацій не обов'язково збільшується. Цикл може йти й у зворотному напрямку:

```
C#
1  using System;
2
3  // Виводимо список у зворотному порядку
4  string[] patients = { "Іван Петренко", "Марія Сидоренко", "Олег Бойко" };
5
6  for (int i = patients.Length - 1; i >= 0; i--)
7  {
8      Console.WriteLine($"[{i.ToString()}] {patients[i]}");
9  }
```

У циклі for можна оголошувати кілька змінних та вказувати кілька виразів у кроці:

```
C#
1  using System;
2
3  // Пара лічильників — i для індексу, j для виведення з 1
4  for (int i = 0, j = 1; i < 3; i++, j++)
5  {
6      Console.WriteLine($"Запис {j.ToString()} (індекс {i.ToString()})");
7  }
```

Якщо всі три частини оголошення пропустити, отримаємо **нескінченний цикл** — він виконується доти, поки не буде примусово зупинений оператором break:

```
C#
1  for (;;)
2  {
3      // нескінченно
4  }
```

Цикл while

Цикл while перевіряє умову **перед** виконанням тіла. Якщо умова від самого початку хибна — тіло не виконається жодного разу:

```
C#
1  while (умова)
2  {
3      // тіло циклу
4  }
```

```
C#
 1  using System;
 2
 3  int temperature = 38;
 4
 5  while (temperature > 36)
 6  {
 7      Console.WriteLine($"Температура: {temperature.ToString()}°C. Жарознижуючий
    засіб.");
 8      temperature--;
 9  }
10
11  Console.WriteLine($"Температура нормалізована: {temperature.ToString()}°C.");
```

Цикл do...while

Цикл do...while виконує тіло **до** перевірки умови. Це гарантує **щонайменше одне виконання** тіла, навіть якщо умова від початку хибна:

```
C#
1  do
2  {
3      // тіло виконується до перевірки умови
4  }
5  while (умова);
```

```
C#
 1  using System;
 2
```

```
 3  int dose = 0;
 4
 5  do
 6  {
 7      dose += 50;
 8      Console.WriteLine($"Введена доза: {dose.ToString()} мг.");
 9  }
10  while (dose < 200);
11
12  Console.WriteLine("Курс завершено.");
```

Зверніть увагу на крапку з комою після while (умова) — вона обов'язкова саме в do...while. Відмінність від while: якщо початкове значення dose вже дорівнювало б 200, цикл while не виконався б жодного разу, а do...while виконається один раз (запише 250 і перевірить умову).

Цикл foreach

Цикл foreach призначений для послідовного перебору **колекцій**: масивів, списків та будь-яких об'єктів, які підтримують перебір. Змінна циклу отримує значення кожного елемента по черзі:

```
C#
1  foreach (тип змінна in колекція)
2  {
3      // змінна містить поточний елемент
4  }
```

```
C#
1  using System;
2
3  string[] diagnoses = { "Гіпертонія", "Бронхіт", "Цукровий діабет", "Анемія" };
4
5  foreach (string diagnosis in diagnoses)
6  {
7      Console.WriteLine($"Діагноз: {diagnosis}");
8  }
```

Тип змінної циклу повинен відповідати типу елементів колекції. Замість явного типу можна використовувати var — компілятор визначить тип самостійно:

```
C#
 1  using System;
 2
 3  int[] bloodPressures = { 120, 135, 148, 160, 115 };
 4  int   count          = 0;
 5
 6  foreach (var bp in bloodPressures)
 7  {
```

```
 8      if (bp >= 140)
 9          count++;
10  }
11
12  Console.WriteLine($"Пацієнтів з підвищеним тиском (≥140): {count.ToString()}");
```

Важливо: змінна циклу foreach доступна лише для читання — присвоєння їй нового значення (x = 0;) є помилкою компіляції CS1656. Тому замінити елементи масиву через foreach не вийде — для цього використовуйте for з індексом. (Якщо елементи — об'єкти reference-типу, змінювати їхні властивості через змінну циклу можна: змінна лише-для-читання — це посилання, а не сам об'єкт.) Окремо: додавати чи видаляти елементи List<T> під час його обходу foreach не можна — на наступній ітерації буде кинуто InvalidOperationException.

Оператор break

Оператор break негайно завершує цикл і передає керування першій інструкції після нього, незалежно від стану умови:

```
C#
 1  using System;
 2
 3  int[] heartRates = { 72, 68, 75, 110, 88, 65 };
 4
 5  int criticalIndex = -1;
 6  for (int i = 0; i < heartRates.Length; i++)
 7  {
 8      if (heartRates[i] > 100)
 9      {
10          criticalIndex = i;
11          break;  // знайшли перше критичне значення — зупиняємось
12      }
13  }
14
15  if (criticalIndex >= 0)
16      Console.WriteLine($"Критичний пульс виявлено на вимірюванні
    {criticalIndex.ToString()}: {heartRates[criticalIndex].ToString()} уд/хв");
17  else
18      Console.WriteLine("Критичних значень не виявлено.");
```

Оператор continue

Оператор continue пропускає решту поточної ітерації і переходить до **наступної**. Цикл при цьому **не завершується**:

```
C#
 1  using System;
 2
 3  int[] ages = { 45, -1, 32, 0, 67, 28, -5, 55 };
 4
```

```
Console.WriteLine("Коректні значення віку:");
foreach (int age in ages)
{
    if (age <= 0)
        continue;  // пропускаємо некоректні значення

    Console.WriteLine($"  {age.ToString()} р.");
}
```

Вкладені цикли

Один цикл може бути розміщений всередині іншого — так утворюються **вкладені цикли**. При цьому зовнішній цикл виконується N разів, і на кожній ітерації зовнішнього повністю відпрацьовує внутрішній. Загальна кількість ітерацій — добуток кількостей ітерацій усіх рівнів.

```C#
using System;

string[] wards   = { "Кардіологія", "Неврологія", "Ортопедія" };
string[] shifts  = { "Ранкова", "Денна", "Нічна" };

for (int w = 0; w < wards.Length; w++)
{
    Console.WriteLine($"Відділення: {wards[w]}");
    for (int s = 0; s < shifts.Length; s++)
    {
        Console.WriteLine($"  Зміна: {shifts[s]}");
    }
}
```

Вкладені цикли є основою алгоритмів обробки двовимірних масивів (матриць, таблиць), що детальніше розглядається у розділі 2.14.

Вибір типу циклу

- **for** — кількість ітерацій відома, потрібен лічильник або потрібно проходити у зворотному напрямку.
- **while** — умова відома, але кількість ітерацій — ні; тіло може не виконатись.
- **do...while** — тіло повинно виконатись принаймні один раз (введення даних, меню).
- **foreach** — перебір усіх елементів колекції від початку до кінця без зміни елементів.

2.14. Масиви

До цього моменту кожне значення зберігалося в окремій змінній. Але якщо потрібно зберегти вік 50 пацієнтів або показники тиску за 30 вимірювань — оголошувати 50 або 30 окремих змінних украй незручно. Для зберігання наборів однотипних значень призначені **масиви**.

Масив — це послідовність фіксованого розміру, що зберігає елементи **одного типу** в суміжних комірках пам'яті. Після оголошення кількість елементів масиву не змінюється.

Оголошення та ініціалізація масиву

Синтаксис оголошення відрізняється від звичайної змінної наявністю квадратних дужок після типу:

```
тип[] назва = new тип[розмір];
```

```
C#
using System;

// Масив з 5 цілих чисел (всі елементи = 0 за замовчуванням)
int[] bloodPressures = new int[5];

// Присвоїмо значення
bloodPressures[0] = 120;
bloodPressures[1] = 135;
bloodPressures[2] = 148;
bloodPressures[3] = 115;
bloodPressures[4] = 160;

Console.WriteLine($"Перший вимір: {bloodPressures[0].ToString()} мм рт.ст.");
Console.WriteLine($"П'ятий вимір: {bloodPressures[4].ToString()} мм рт.ст.");
```

Масив також можна ініціалізувати значеннями відразу при оголошенні. Усі варіанти еквівалентні:

```
C#
using System;

int[] ages1 = new int[4] { 45, 32, 67, 28 };  // явний розмір і значення
int[] ages2 = new int[]  { 45, 32, 67, 28 };  // розмір виводиться автоматично
int[] ages3 = new[]      { 45, 32, 67, 28 };  // скорочений запис
int[] ages4 =            { 45, 32, 67, 28 };  // найкоротший варіант

Console.WriteLine($"Всі масиви мають довжину {ages1.Length.ToString()}");
Console.WriteLine($"Перший елемент: {ages1[0].ToString()}");
```

Аналогічно для рядків:

```
C#
using System;
```

```
2
3  string[] patientNames = { "Іван Петренко", "Марія Сидоренко", "Олег Бойко" };
4
5  foreach (string name in patientNames)
6      Console.WriteLine(name);
```

Індекси та доступ до елементів



Одновимірний масив: структура, індекси та Length

Кожен елемент масиву має **індекс** — ціле число, яке вказує його позицію. **Нумерація починається з нуля**: перший елемент має індекс 0, другий — 1, останній — довжина - 1.

```
C#
1  using System;
2
3  int[] ages = { 45, 32, 67, 28, 55 };
4
5  Console.WriteLine($"Перший (індекс 0): {ages[0].ToString()}");
6  Console.WriteLine($"Третій (індекс 2): {ages[2].ToString()}");
7  Console.WriteLine($"Останній (індекс 4): {ages[4].ToString()}");
```

Елементи можна змінювати через індекс:

```
C#
1  using System;
2
3  int[] temps = { 36, 37, 39, 38, 36 };
4  Console.WriteLine($"До корекції: {temps[2].ToString()}°C");
5
6  temps[2] = 38;  // виправили помилку введення
7  Console.WriteLine($"Після корекції: {temps[2].ToString()}°C");
```

Якщо вказати індекс за межами масиву, виникне виняток IndexOutOfRangeException під час виконання:

```
C#
 1  using System;
 2
 3  try
 4  {
 5      int[] values = { 1, 2, 3 };
 6      Console.WriteLine(values[10].ToString()); // помилка!
 7  }
 8  catch (IndexOutOfRangeException ex)
 9  {
10      Console.WriteLine($"Помилка: {ex.Message}");
11  }
```

Властивість Length

Кожен масив має властивість Length, що повертає загальну кількість його елементів. Її важливо використовувати у циклах замість «магічних чисел»:

```
C#
 1  using System;
 2
 3  int[] heartRates = { 72, 68, 75, 88, 65 };
 4
 5  Console.WriteLine($"Кількість вимірів: {heartRates.Length.ToString()}");
 6
 7  // Сума всіх значень
 8  int sum = 0;
 9  for (int i = 0; i < heartRates.Length; i++)
10      sum += heartRates[i];
11
12  double avg = (double)sum / heartRates.Length;
13  Console.WriteLine($"Середній пульс: {avg.ToString("F1")} уд/хв");
```

Оператор ^ (індекс від кінця)

Починаючи з C# 8.0, можна звертатися до елементів масиву **від кінця** за допомогою оператора ^. ^1 — останній елемент, ^2 — передостанній тощо:

```
C#
1  using System;
2
3  int[] readings = { 120, 135, 148, 160, 115 };
4
5  Console.WriteLine($"Останній вимір:         {readings[^1].ToString()}");
6  Console.WriteLine($"Передостанній вимір:    {readings[^2].ToString()}");
7  Console.WriteLine($"Те саме через Length:   {readings[readings.Length -
   1].ToString()}");
```

readings[^1] — це лаконічний еквівалент readings[readings.Length - 1].

Перебір масиву

foreach

Для послідовного перебору всіх елементів найбільш читабельним є foreach:

```
C#
1  using System;
2
3  double[] temperatures = { 36.6, 37.2, 38.5, 37.0, 36.8 };
4
5  foreach (double temp in temperatures)
6  {
7      string status = temp >= 38.0 ? "Гарячка!" : "Норма";
8      Console.WriteLine($"{temp.ToString("F1")}°C — {status}");
9  }
```

for

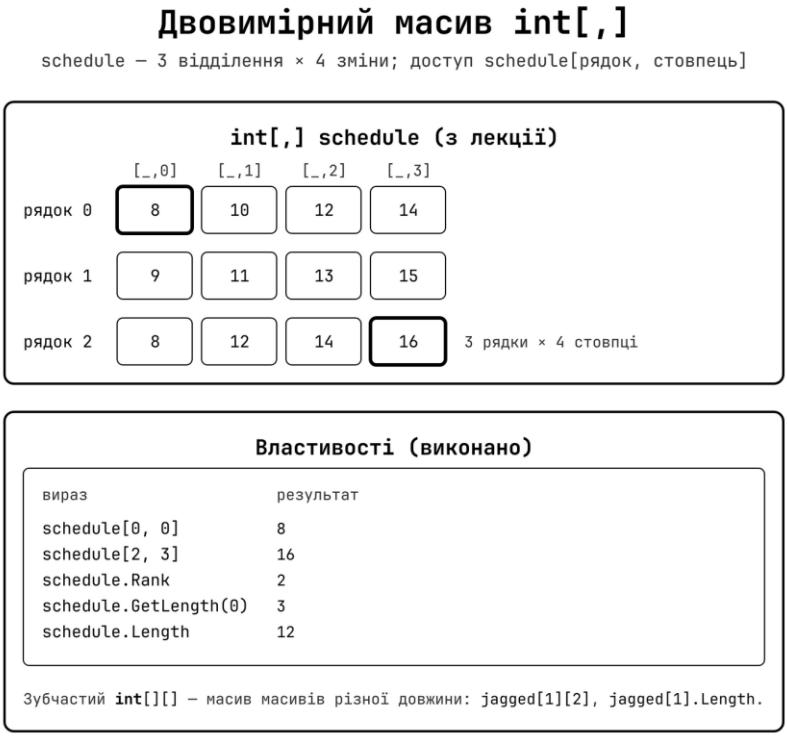
Якщо потрібен індекс або зміна елементів — використовуйте for:

```
C#
 1  using System;
 2
 3  int[] bps = { 120, 145, 155, 130, 160 };
 4
 5  Console.WriteLine("Нормалізація значень (до 140 мм рт.ст.):");
 6  for (int i = 0; i < bps.Length; i++)
 7  {
 8      if (bps[i] > 140)
 9          bps[i] = 140;
10      Console.WriteLine($"  [{i.ToString()}] = {bps[i].ToString()}");
11  }
```

foreach **не дозволяє** змінювати елементи колекції через змінну циклу — лише читати. Тому для модифікації елементів завжди застосовуйте for.

Багатовимірні масиви

Масиви можуть бути **багатовимірними**. Найчастіше використовуються двовимірні масиви — вони представляють таблиці (матриці).



Двовимірний масив: структура, рядки та стовпці

Для оголошення двовимірного масиву в квадратних дужках вказується кома, що розділяє розміри:

```
C#
 1  using System;
 2
 3  // Розклад: 3 відділення × 4 зміни (кількість пацієнтів у зміні)
 4  int[,] schedule = {
 5      { 8, 10, 12, 14 },
 6      { 9, 11, 13, 15 },
 7      { 8, 12, 14, 16 }
```

```
};

int rows    = schedule.GetUpperBound(0) + 1; // 3
int columns = schedule.GetUpperBound(1) + 1; // 4

Console.WriteLine($"Відділень: {rows.ToString()}, Змін: {columns.ToString()}");
Console.WriteLine($"Ранкова зміна, відділення 1: {schedule[0, 0].ToString()}
пацієнтів");
Console.WriteLine($"Нічна зміна, відділення 3:   {schedule[2, 3].ToString()}
пацієнтів");

// Перебір: зовнішній цикл — рядки, внутрішній — стовпці
for (int r = 0; r < rows; r++)
{
    for (int c = 0; c < columns; c++)
    {
        Console.Write($"{schedule[r, c].ToString(),4}");
    }
    Console.WriteLine();
}
```

schedule.GetUpperBound(0) + 1 повертає кількість рядків, GetUpperBound(1) + 1 — кількість стовпців. Загальна довжина двовимірного масиву через Length — це добуток усіх розмірностей (тут: 3 × 4 = 12).

Масиви можуть мати й більшу кількість вимірів (тривимірні тощо), але на практиці частіше застосовуються одно- та двовимірні.

Зубчастий масив (масив масивів)

На відміну від прямокутного двовимірного масиву, **зубчастий масив** (jagged array) — це масив, що містить інші масиви, причому кожен підмасив може мати різну довжину. Оголошується через подвійні квадратні дужки [][]:

C#
```
using System;

// Кількість пацієнтів по днях тижня (різна кількість прийомів)
int[][] dailyPatients = {
    new int[] { 12, 8, 10 },           // Понеділок: 3 прийоми
    new int[] { 15, 12, 9, 11 },       // Вівторок:  4 прийоми
    new int[] { 10, 7 },               // Середа:    2 прийоми
};

for (int day = 0; day < dailyPatients.Length; day++)
{
    string[] dayNames = { "Понеділок", "Вівторок", "Середа" };
    Console.Write($"{dayNames[day]}: ");

    foreach (int count in dailyPatients[day])
        Console.Write($"{count.ToString()} ");

```

```
18      Console.WriteLine();
19  }
```

Зубчастий масив застосовується там, де підмасиви мають різні розміри. Якщо всі рядки однакової довжини — зручніше використовувати прямокутний [,].

Основні поняття масивів

Поняття	Опис	Приклад
Ранг	Кількість вимірів	int[,] — ранг 2
Довжина виміру	Розмір конкретного виміру	new int[3, 4] — 3 рядки, 4 стовпці
Length	Загальна кількість елементів	3 × 4 = 12
Індекс	Номер елемента (з нуля)	arr[0] — перший
^-оператор	Індекс від кінця	arr[^1] — останній

2.15. Завдання з масивами

Опанувавши цикли, умовні конструкції та масиви, розглянемо кілька практичних задач, що відображають типові операції обробки медичних числових даних. Кожна з них демонструє окремий алгоритмічний прийом: підрахунок елементів за умовою, перестановку елементів у зворотному порядку та впорядкування масиву методом бульбашкового сортування.

Підрахунок пацієнтів із підвищеним артеріальним тиском

Першим завданням є підрахунок кількості елементів масиву, які задовольняють певну умову. У клінічному контексті це може бути підрахунок пацієнтів, у яких систолічний артеріальний тиск перевищує 140 мм рт.ст. — порогове значення гіпертонії I ступеня.

Алгоритм: оголошуємо допоміжну змінну-лічильник, обнуляємо її перед циклом, потім перебираємо масив і збільшуємо лічильник щоразу, коли поточний елемент відповідає умові.

```C#
using System;

int[] bloodPressures = { 120, 155, 130, 172, 118, 145, 165, 128, 138, 180 };

int hypertensionCount = 0;
foreach (int bp in bloodPressures)
{
    if (bp >= 140)
        hypertensionCount++;
}

Console.WriteLine($"Всього вимірювань:
{bloodPressures.Length.ToString()}");
Console.WriteLine($"Підвищений тиск (≥140 мм рт.ст.):
{hypertensionCount.ToString()}");
Console.WriteLine($"У нормі:                          {(bloodPressures.Length -
hypertensionCount).ToString()}");
```

Тут hypertensionCount спочатку дорівнює нулю. На кожній ітерації foreach перевіряється умова bp >= 140. Якщо поточне значення тиску досягає або перевищує порогове — лічильник збільшується на одиницю. Після завершення циклу змінна містить шукану кількість.

Такий підхід легко адаптується до будь-якого діапазону: наприклад, щоб знайти вимірювання в межах норми (90–139), достатньо замінити умову на bp >= 90 && bp < 140.

Реверс масиву показників

Друге завдання — інверсія (реверс) масиву, тобто переставлення елементів у зворотному порядку без виділення додаткового масиву. Практично це корисно, наприклад, коли добові вимірювання температури записані від найдавніших до найновіших, але для виведення в журналі потрібен зворотний хронологічний порядок — від останнього до першого.

Алгоритм: одночасно рухаємося від початку й від кінця масиву назустріч одне одному та обмінюємо елементи місцями. Переміщаємося лише до середини — після неї кожна пара вже оброблена.

```C#
using System;

double[] temps = { 36.4, 36.7, 37.1, 37.8, 38.2, 37.5, 36.9 };

Console.Write("До реверсу: ");
foreach (double t in temps)
    Console.Write($"{t.ToString("F1")}  ");
Console.WriteLine();

```

```
int n = temps.Length;
int k = n / 2; // індекс середини — до нього треба дійти

for (int i = 0; i < k; i++)
{
    double temp   = temps[i];
    temps[i]      = temps[n - i - 1];
    temps[n - i - 1] = temp;
}

Console.Write("Після реверсу: ");
foreach (double t in temps)
    Console.Write($"{t.ToString("F1")}  ");
Console.WriteLine();
```

Ключова ідея обміну: щоб переставити два значення, потрібна тимчасова змінна temp. Без неї перший запис знищив би одне зі значень: temps[i] = temps[n - i - 1] затерло б temps[i], і при виконанні наступної строки temps[n - i - 1] = temps[i] ми записали б вже нове (а не оригінальне) значення. Саме тому спочатку зберігаємо temps[i] у temp, потім перезаписуємо temps[i], і лише потім копіюємо temp на місце temps[n - i - 1].

Якщо довжина масиву непарна (наприклад, 7 елементів), центральний елемент з індексом 3 залишається на місці — він сам собі пара, а цикл до нього не доходить (k = 3, отже і дорівнює 0, 1, 2).

Сортування масиву методом бульбашки

Третє завдання — впорядкування масиву. Медична система часто потребує відсортованих показників: наприклад, щоб визначити мінімальне та максимальне значення тиску серед групи пацієнтів або виявити, хто перебуває у критичному стані.

Найпростіший для розуміння алгоритм — **бульбашкове сортування** (bubble sort): повторювані проходи масивом, при кожному з яких більший елемент «спливає» вправо, поступаючись місцем меншому. Назва пов'язана з тим, що великі значення, як бульбашки, поступово переміщуються до кінця.

C#
```
using System;

int[] bps = { 155, 72, 180, 118, 145, 165, 130, 88, 160, 120 };

Console.Write("До сортування: ");
foreach (int bp in bps)
    Console.Write($"{bp.ToString()}  ");
Console.WriteLine();

// Бульбашкове сортування (за зростанням)
```

```
int temp;
for (int i = 0; i < bps.Length - 1; i++)
{
    for (int j = i + 1; j < bps.Length; j++)
    {
        if (bps[i] > bps[j])
        {
            temp   = bps[i];
            bps[i] = bps[j];
            bps[j] = temp;
        }
    }
}

Console.Write("Після сортування: ");
foreach (int bp in bps)
    Console.Write($"{bp.ToString()}  ");
Console.WriteLine();

Console.WriteLine($"Мінімальний тиск: {bps[0].ToString()} мм рт.ст.");
Console.WriteLine($"Максимальний тиск: {bps[^1].ToString()} мм рт.ст.");
```

Розберемо алгоритм детально. Зовнішній цикл for (int i = 0; i < bps.Length - 1; i++) послідовно вибирає кожен елемент, крім останнього — він стане найбільшим сам по собі після проходів. Внутрішній цикл for (int j = i + 1; j < bps.Length; j++) починається **з наступного після і елемента** і порівнює обраний елемент bps[i] з кожним іншим праворуч від нього. Якщо bps[i] > bps[j] — більший елемент ліворуч, і їх треба поміняти місцями. Обмін виконується через тимчасову змінну temp — за тим самим принципом, що й у задачі з реверсом.

Після завершення сортування масив впорядкований за зростанням. Мінімальний елемент — bps[0], максимальний — bps[^1] (останній елемент через оператор ^).

Бульбашкове сортування простіше для початкового розуміння, але не є найефективнішим для великих масивів. У промисловому коді для впорядкування застосовують методи Array.Sort() і Array.Sort(array, comparer), що реалізують більш ефективні алгоритми. Проте ручна реалізація сортування демонструє важливий алгоритмічний принцип: розбиття задачі на підзадачі за допомогою вкладених циклів.

2.16. Методи

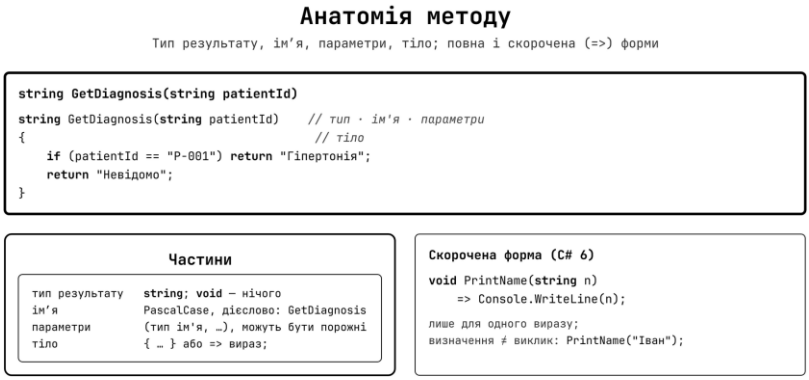
Якщо змінна зберігає значення, то **метод** зберігає поведінку — іменований блок коду, що виконує певні дії. Методи дозволяють структурувати програму: замість того щоб копіювати одну й ту саму логіку в різних місцях, її записують один раз у методі, а потім викликають скільки завгодно разів звідусіль.

Насправді методи вже зустрічалися нам із самого початку: Console.WriteLine() — це метод, визначений у класі Console. Тепер розглянемо, як створювати власні методи.

Синтаксис визначення методу

Загальний синтаксис оголошення методу такий:

```
[модифікатори] тип_результату НазваМетоду([параметри])
{
    // тіло методу
}
```



Анатомія методу: тип результату, назва, параметри, тіло

— **Тип результату** — тип значення, яке метод повертає після виконання. Якщо метод нічого не повертає, вказується void.

— **Назва методу** — ідентифікатор за тими самими правилами, що й назви змінних. За угодою C# методи іменуються з великої літери (PascalCase): PrintDiagnosis, GetPatientAge, RegisterVisit.

— **Параметри** — вхідні дані, які метод отримує при виклику (розглядаються в розділі 2.17). Якщо параметрів немає — дужки залишаються порожніми.

— **Тіло** — блок інструкцій у фігурних дужках, що виконуються при кожному виклику методу.

— **Модифікатори** необов'язкові; їх призначення розглядається в наступних розділах при вивченні класів і ООП.

Метод типу void

Найпростіший варіант — метод, що виконує дії, але нічого не повертає. Тип результату для таких методів — void:

```
C#
1  using System;
2
3  void PrintWelcome()
4  {
5      Console.WriteLine("=== Клінічна система реєстрації ===");
6      Console.WriteLine("Вітаємо! Введіть дані пацієнта.");
7  }
8
9  PrintWelcome();
```

Тут PrintWelcome — метод без параметрів і без значення, що повертається. У його тілі дві інструкції виведення. Виклик PrintWelcome() виконує обидві.

Важливо: **визначення** методу (блок коду з іменем і тілом) не виконує його автоматично. Для виконання необхідний **виклик** — рядок із назвою методу та круглими дужками.

Виклик методів

Для виклику методу вказується його назва та круглі дужки зі значеннями параметрів (якщо вони є). Якщо параметрів немає — дужки залишаються порожніми:

```
1  НазваМетоду();
```

Головна перевага методів — **повторне використання**: один і той самий метод можна викликати будь-яку кількість разів у будь-якому місці програми:

```
C#
 1  using System;
 2
 3  void PrintSeparator()
 4  {
 5      Console.WriteLine("──────────────────────────────────");
 6  }
 7
 8  void PrintHeader()
 9  {
10      PrintSeparator();
11      Console.WriteLine("  Журнал прийому пацієнтів");
12      PrintSeparator();
13  }
14
15  PrintHeader();
```

```
Console.WriteLine("Пацієнт: Іван Петренко");
PrintSeparator();
Console.WriteLine("Пацієнт: Марія Сидоренко");
PrintSeparator();
Console.WriteLine("Пацієнт: Олег Бойко");
PrintSeparator();
```

Зверніть увагу: PrintHeader сам викликає PrintSeparator двічі всередині свого тіла. Методи можуть викликати інші методи — це дозволяє ієрархічно структурувати код.

Порядок розташування визначення методу та його виклику **не має значення**: виклик може передувати визначенню, і програма все одно спрацює коректно. Компілятор C# аналізує весь файл перед виконанням, тому всі методи відомі йому ще до того, як до них доходить виконання.

Кілька методів із умовним вибором

Програма може оголошувати будь-яку кількість методів і викликати потрібний залежно від умови. Це дозволяє чітко розподілити відповідальність між різними ділянками коду:

```C#
using System;

void PrintCriticalAlert()
{
    Console.WriteLine("[!] КРИТИЧНИЙ СТАН: негайна госпіталізація!");
    Console.WriteLine("    Виклик чергового лікаря та реанімаційної бригади.");
}

void PrintWarningAlert()
{
    Console.WriteLine("[!] Підвищені показники. Консультація лікаря.");
    Console.WriteLine("    Рекомендовано позачерговий огляд.");
}

void PrintNormalStatus()
{
    Console.WriteLine("[✓] Показники в межах норми.");
    Console.WriteLine("    Наступний плановий огляд згідно з розкладом.");
}

int bpSystolic = 175;

if (bpSystolic >= 180)
    PrintCriticalAlert();
else if (bpSystolic >= 140)
    PrintWarningAlert();
else
    PrintNormalStatus();
```

Кожен метод відповідає за одну конкретну дію — виведення відповідного повідомлення. Логіка вибору знаходиться окремо, у конструкції if...else. Такий розподіл спрощує читання та подальше редагування: щоб змінити текст попередження, достатньо відредагувати лише один метод, не торкаючись логіки вибору. Цей принцип — кожен метод відповідає лише за одну річ — є одним із фундаментальних у проектуванні програм.

Скорочений запис методу (вираз-тіло)

Якщо тіло методу складається лише з **однієї інструкції**, його можна записати у скороченій формі через оператор =>:

```
тип_результату НазваМетоду([параметри]) => інструкція;
```

Повна і скорочена форми є точними еквівалентами з погляду компілятора:

```
C#
using System;

// Повна форма
void PrintDivider()
{
    Console.WriteLine("================================");
}

// Скорочена форма — точний еквівалент
void PrintDividerShort() =>
Console.WriteLine("================================");

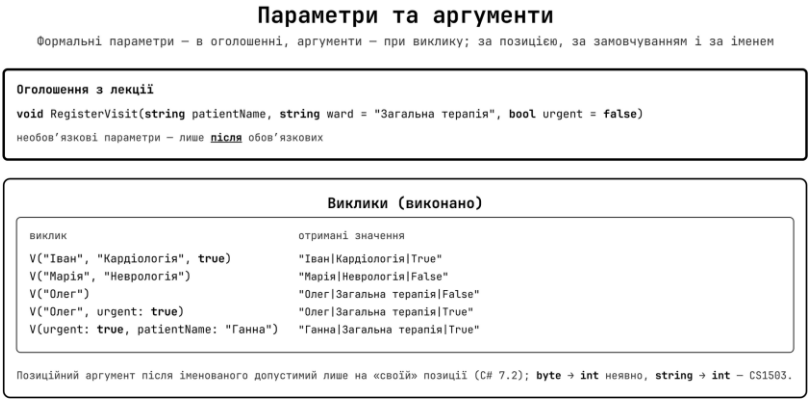
PrintDivider();
PrintDividerShort();
```

Скорочений запис => (вираз-тіло, або expression-bodied member) зустрічається у коді C# дуже часто. Він є синтаксичним цукром — не змінює поведінку методу, лише скорочує запис. Застосовується тоді, коли метод концептуально простий і однорядковий запис не знижує читабельності.

Коли тіло методу складається з кількох інструкцій — завжди використовуйте повну форму з фігурними дужками. Спроба вмістити кілька інструкцій у стрілкову форму є синтаксичною помилкою.

2.17. Параметри методів

Метод без параметрів завжди виконує одне й те саме: виводить один і той самий рядок або виконує фіксовану дію. Щоб метод міг працювати з різними даними, йому передають **параметри** — вхідні значення, що надходять ззовні під час виклику. Саме параметри роблять методи гнучкими й придатними для повторного використання.



Формальні та фактичні параметри методу

Синтаксис параметрів

Параметри оголошуються у круглих дужках після назви методу через кому. Кожен параметр складається з типу та імені:

```
тип_результату НазваМетоду(тип1 параметр1, тип2 параметр2, ...)
{
    // тіло
}
```

Наприклад, метод реєстрації пацієнта, що приймає ім'я та вік:

```C#
using System;

void RegisterPatient(string name, int age)
{
    Console.WriteLine($"Зареєстровано пацієнта: {name}, {age.ToString()} р.");
}

RegisterPatient("Іван Петренко", 45);
RegisterPatient("Марія Сидоренко", 32);
RegisterPatient("Олег Бойко", 67);
```

Метод RegisterPatient має два параметри: name типу string та age типу int. При кожному виклику він отримує нові значення та виводить рядок із ними. Один і той самий метод, три різних результати.

Формальні та фактичні параметри (аргументи)

У C# використовуються два терміни для позначення параметрів:

- **Формальні параметри** — імена та типи, що оголошуються у визначенні методу. У прикладі вище — name та age. Вони існують лише всередині тіла методу.
- **Фактичні параметри (аргументи)** — конкретні значення, що передаються при виклику. У виклику RegisterPatient("Іван Петренко", 45) аргументами є рядок "Іван Петренко" та ціле число 45.

При виклику аргументи передаються параметрам **за позицією** — зліва направо у порядку, в якому вони оголошені. Перший аргумент стає першим параметром, другий — другим тощо.

Аргументами можуть бути не тільки літерали, а й змінні та складні вирази:

```C#
using System;

void PrintMeasurement(string label, int value, string unit)
{
    Console.WriteLine($"{label}: {value.ToString()} {unit}");
}

string patientName = "Тетяна Коваль";
int bp             = 145;
int pulse          = 80;

PrintMeasurement("Артеріальний тиск", bp, "мм рт.ст.");
PrintMeasurement("Пульс",             pulse, "уд/хв");
PrintMeasurement("Вік",               bp - 100, "р.");   // вираз як аргумент
```

Аргументом bp - 100 є результат арифметичного виразу — компілятор обчислює його значення перед передачею в метод.

Якщо як аргумент передається змінна, вона **повинна мати значення** на момент виклику. Передача неініціалізованої змінної є помилкою компіляції.

Відповідність типів

Кожен аргумент повинен відповідати типу відповідного формального параметра або допускати неявне перетворення до нього. Компілятор перевіряє це ще до виконання програми:

```
C#
 1  using System;
 2
 3  void LogPressure(string patientId, int systolic)
 4  {
 5      Console.WriteLine($"Пацієнт {patientId}: тиск {systolic.ToString()} мм
    рт.ст.");
 6  }
 7
 8  byte bpByte = 138; // byte → int: неявне розширення дозволене
 9  LogPressure("P-001", bpByte);  // ✓ byte автоматично розширюється до int
10
11  LogPressure("P-002", 160);     // ✓ int-літерал → int
```

Типи byte, short, int є числами різного розміру: byte вміщується в int, тому таке перетворення безпечне й автоматичне. Проте передати рядок замість числа або навпаки неможливо — компілятор виявить це одразу:

```
C#
1  LogPressure(180, "P-003"); // ✗ Помилка: int і string поміняні місцями
```

Необов'язкові параметри зі значенням за замовчуванням

За замовчуванням при виклику методу необхідно передати значення для **кожного** оголошеного параметра. Але іноді деякі параметри мають «розумне» значення за замовчуванням і передавати їх явно незручно. C# дозволяє оголошувати **необов'язкові параметри**: якщо аргумент не передано — використовується задане значення за замовчуванням:

```
C#
 1  using System;
 2
 3  void RegisterVisit(string patientName, string ward = "Загальна терапія", bool
    urgent = false)
 4  {
 5      string urgencyTag = urgent ? " [ТЕРМІНОВО]" : "";
 6      Console.WriteLine($"{patientName} → {ward}{urgencyTag}");
 7  }
 8
 9  RegisterVisit("Іван Петренко", "Кардіологія", true);  // всі аргументи
10  RegisterVisit("Марія Сидоренко", "Неврологія");       // urgent = false за
    замовч.
```

```
11  RegisterVisit("Олег Бойко");                          // ward і urgent — за
    замовч.
```

Необов'язкові параметри **завжди розташовуються після обов'язкових**. Визначити обов'язковий параметр після необов'язкового неможливо — компілятор видасть помилку. Значення за замовчуванням повинне бути константним виразом: літерал, null, true/false або результат const-виразу.

Іменовані параметри

Зазвичай аргументи передаються за позицією — перший аргумент відповідає першому параметру. Але C# дозволяє передавати аргументи **за іменем**, явно вказуючи, якому параметру призначається значення. Синтаксис: ім'яПараметра: значення:

```
C#
 1  using System;
 2
 3  void RegisterPatient(string name, int age, string ward = "Загальна терапія",
    bool vip = false)
 4  {
 5      string vipTag = vip ? " [VIP]" : "";
 6      Console.WriteLine($"[{ward}{vipTag}] {name}, {age.ToString()} р.");
 7  }
 8
 9  // Позиційна передача (звична форма)
10  RegisterPatient("Іван Петренко", 45, "Кардіологія");
11
12  // Іменована: ward пропускаємо, vip — за іменем
13  RegisterPatient("Марія Сидоренко", 32, vip: true);
14
15  // Повністю іменована — порядок не важливий
16  RegisterPatient(age: 67, name: "Олег Бойко", ward: "Неврологія");
```

Іменована передача корисна у кількох ситуаціях:

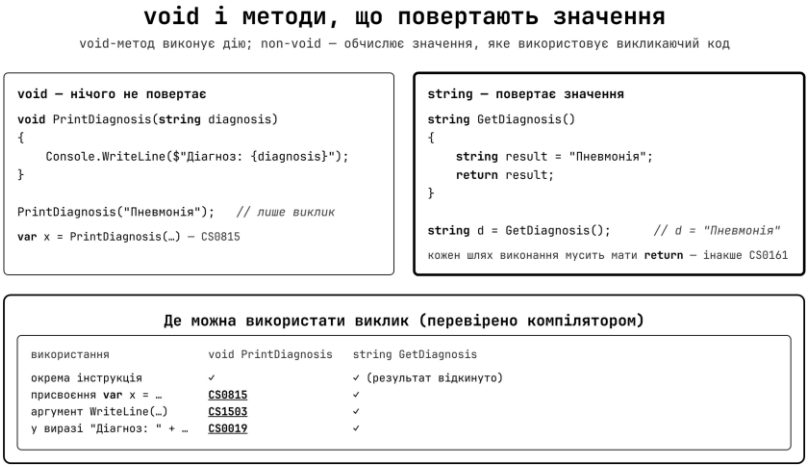
4. **Пропуск необов'язкових параметрів.** Якщо потрібно встановити лише деякі необов'язкові параметри, а не всі підряд — без іменування довелося б передавати значення за замовчуванням явно для кожного пропущеного параметра.
5. **Покращення читабельності.** Запис RegisterPatient(name: "Іван", age: 45) зрозуміліший, ніж RegisterPatient("Іван", 45) при читанні незнайомого коду.
6. **Будь-який порядок.** Іменовані аргументи можна передавати в довільному порядку, що іноді зручно логічно.

Важливе обмеження: позиційний аргумент після іменованого допустимий лише тоді, коли іменований стоїть **на своїй позиції**. До C# 7.2 правило було жорсткішим — після першого іменованого всі наступні мусили бути іменованими. Починаючи з C# 7.2, виклик RegisterPatient(name: "Іван", 45) компілюється, бо name і так перший параметр. А от RegisterPatient(age: 45, "Іван") — помилка CS8323: іменований аргумент стоїть не на своєму місці, і компілятор не може зіставити наступний позиційний.

2.18. Повернення значення та оператор return

Методи, розглянуті в попередніх розділах, виконували дії — виводили рядки на екран, оновлювали стан, реєстрували події. Але справжня потужність методів розкривається тоді, коли вони не лише щось роблять, а й **повертають результат** своєї роботи назад до коду, що їх викликав. Саме так будуються обчислення, що складаються з послідовних кроків: один метод обчислює діагноз, інший формує звіт, третій виводить його — і кожен крок отримує результат попереднього.

У C# методи поділяються на два класи залежно від того, чи повертають вони значення. Методи з типом void виконують дію і нічого не повертають. Методи з будь-яким іншим типом — string, int, bool, double тощо — зобов'язані повернути значення відповідного типу за допомогою оператора return.



Порівняння void та non-void методів

Синтаксис оператора return

Оператор return записується усередині тіла методу і складається з ключового слова та значення, що повертається:

```
return значення_що_повертається;
```

Після виконання return метод негайно завершується — жоден рядок коду нижче вже не виконається, а управління повертається до місця виклику разом із зазначеним значенням.

Повернення значення: перший приклад

Визначимо метод, який повертає рядковий опис діагнозу пацієнта. Тип методу — string, тобто він зобов'язаний повернути рядок:

```
C#
using System;

string GetPrimaryDiagnosis()
{
    return "Пневмонія, правобічна, середньої тяжкості";
}

string diagnosis = GetPrimaryDiagnosis();
Console.WriteLine($"Основний діагноз: {diagnosis}");
```

Метод GetPrimaryDiagnosis оголошений з типом string. Оператор return передає рядок назад до викликаючого коду, де він зберігається у змінній diagnosis. Компілятор знає тип заздалегідь і перевіряє відповідність ще до запуску програми.

Обов'язковість return у non-void методах

Якщо метод оголошений з типом, відмінним від void, він **обов'язково** повинен мати оператор return із відповідним значенням. Відсутність return є помилкою компіляції — програма навіть не запуститься:

```
C#
1  // ПОМИЛКА КОМПІЛЯЦІЇ — не всі шляхи виконання повертають значення
2  string GetDiagnosis()
3  {
4      Console.WriteLine("Аналіз завершено");
5      // return відсутній — компілятор відмовить
6  }
```

Компілятор C# аналізує всі можливі шляхи виконання тіла методу і перевіряє, чи кожен з них завершується оператором return. Якщо хоча б один шлях не повертає значення — це помилка.

Відповідність типів

Тип значення після return повинен точно відповідати оголошеному типу методу або допускати неявне перетворення до нього. Якщо метод оголошений як string, він не може повернути int або bool:

```
C#
1  // ПОМИЛКА — тип методу string, але повертається int
2  string GetDiagnosis()
3  {
4      return 42; // компілятор відмовить: неможливо перетворити int в string
5  }
```

```
C#
1  // ПОМИЛКА — тип методу bool, але повертається string
2  bool IsPatientAdult()
3  {
4      return "так"; // неправильно — потрібен true або false
5  }
```

Суворий контроль типів на етапі компіляції — захист від цілого класу помилок, які інакше проявились би лише під час виконання в невідповідний момент.

Використання результату методу

Значення, що повертає метод, можна **присвоїти змінній**, щоб використати його пізніше:

```C#
using System;

string GetDiagnosis(string symptom)
{
    if (symptom == "кашель")
        return "Бронхіт";
    if (symptom == "висока температура")
        return "ГРВІ";
    return "Необхідне додаткове обстеження";
}

string result = GetDiagnosis("кашель");
Console.WriteLine($"Діагноз: {result}");

string result2 = GetDiagnosis("висока температура");
Console.WriteLine($"Діагноз: {result2}");

string result3 = GetDiagnosis("запаморочення");
Console.WriteLine($"Діагноз: {result3}");
```

Кожен виклик GetDiagnosis повертає окреме значення залежно від аргументу. Змінна отримує конкретний рядок і далі може бути використана довільну кількість разів.

Результат методу також можна передати **безпосередньо як аргумент** іншому методу, не зберігаючи у проміжну змінну:

```C#
using System;

string GetDiagnosis()
{
    return "Цукровий діабет, 2 тип";
}

void PrintDiagnosisReport(string diagnosis)
{
    Console.WriteLine($"=== Медичний висновок ===");
    Console.WriteLine($"Встановлений діагноз: {diagnosis}");
    Console.WriteLine($"Документ сформовано автоматично.");
}

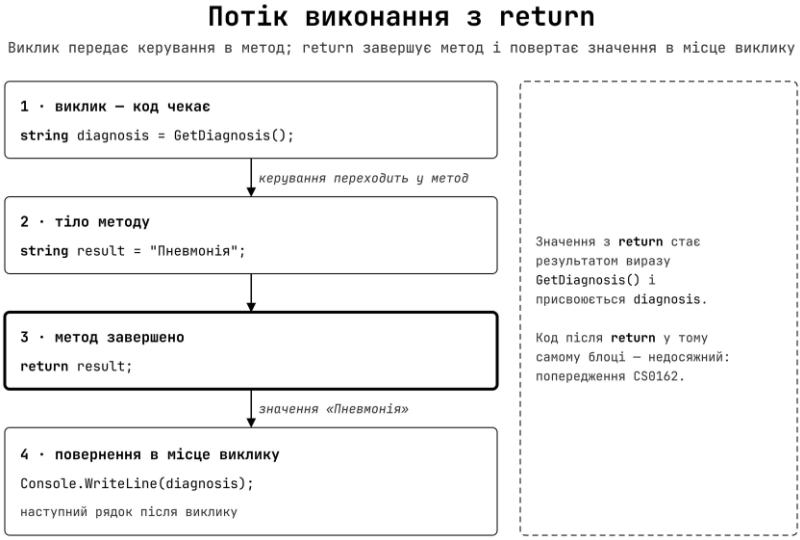
PrintDiagnosisReport(GetDiagnosis());
```

У виклику PrintDiagnosisReport(GetDiagnosis()) спочатку виконується GetDiagnosis() — його результат одразу передається параметру diagnosis методу PrintDiagnosisReport. Проміжна змінна не

потрібна. Такий запис є поширеним у C# і робить код лаконічнішим, якщо результат використовується лише один раз.

Як виконується повернення значення

Коли програма виконує виклик методу, що повертає значення, відбувається наступна послідовність: управління переходить до тіла методу, метод виконує свою логіку, натрапляє на return, передає значення назад і завершується. Викликаючий код отримує це значення в точці виклику.



Потік виконання при поверненні значення

Складні вирази у return

Після ключового слова return можна записати не лише змінну або літерал, а й **будь-який вираз** — арифметичний, логічний, виклик іншого методу. Компілятор обчислить вираз і поверне результат:

```
C#
 1  using System;
 2
 3  double CalculateBMI(double weightKg, double heightM)
 4  {
 5      return weightKg / (heightM * heightM);
```

```
}

bool IsNormalBMI(double bmi)
{
    return bmi >= 18.5 && bmi <= 24.9;
}

double bmi = CalculateBMI(72.0, 1.75);
Console.WriteLine($"Індекс маси тіла: {bmi.ToString("F1")}");
Console.WriteLine($"Норма: {IsNormalBMI(bmi).ToString()}");
```

Метод CalculateBMI повертає результат арифметичного виразу — значення обчислюється в момент виконання return. Метод IsNormalBMI повертає результат логічного виразу — true або false. Обидва методи не використовують проміжних змінних: return здатен повернути будь-який вираз відповідного типу.

Скорочені (expression-bodied) методи

Коли тіло методу складається лише з одного оператора return, C# дозволяє записати такий метод у скороченій формі за допомогою оператора => (лямбда-стрілка). Дужки і return при цьому опускаються:

```
тип_результату НазваМетоду(параметри) => вираз;
```

Наприклад, повна форма:

```
C#
string GetWard()
{
    return "Кардіологія";
}
```

Еквівалентна скороченій формі:

```
C#
string GetWard() => "Кардіологія";
```

Скорочена форма не є окремим механізмом — це лише синтаксичний цукор, який компілятор перетворює на повну форму автоматично. Поведінка ідентична. Скорочені методи широко використовуються для простих обчислень і запитів, де розгорнутий синтаксис був би надлишковим:

```
C#
using System;

string GetWard()        => "Кардіологія";
int    GetBedCount()    => 24;
```

```
 5  bool   IsEmergency()    => false;
 6
 7  double CalculateDose(double weightKg, double dosePerKg) => weightKg *
    dosePerKg;
 8
 9  string GetPatientInfo(string name, int age) =>
10      $"Пацієнт: {name}, {age.ToString()} р., відділення: {GetWard()}";
11
12  Console.WriteLine(GetWard());
13  Console.WriteLine($"Ліжок у відділенні: {GetBedCount().ToString()}");
14  Console.WriteLine($"Екстрена ситуація: {IsEmergency().ToString()}");
15  Console.WriteLine($"Доза препарату: {CalculateDose(70.0, 0.5).ToString("F1")}
    мг");
16  Console.WriteLine(GetPatientInfo("Олена Гончар", 54));
```

Як видно, скорочена форма добре читається для простих методів і дозволяє компактно визначити кілька методів поспіль. Для методів із гілками, циклами або кількома операціями слід використовувати повну форму.

Вихід із методу та недосяжний код

Оператор return не лише повертає значення — він **негайно завершує виконання методу**. Будь-який код, розташований після return у тому самому блоці виконання, є недосяжним і ніколи не виконається:

```
C#
1  string GetStatus()
2  {
3      return "Стабільний";
4      Console.WriteLine("Це рядок не виконається"); // недосяжний код
5  }
```

З синтаксичного погляду наведений код коректний, однак компілятор C# видає **попередження** (warning CS0162: «Виявлено недосяжний код»). Це сигнал про логічну помилку: програміст, можливо, мав намір виконати дію, але вона назавжди заблокована оператором return. Такий код слід прибирати.

Недосяжний код найчастіше виникає через помилку при рефакторингу — коли return переміщують, але забувають видалити рядки, що стоять після нього.

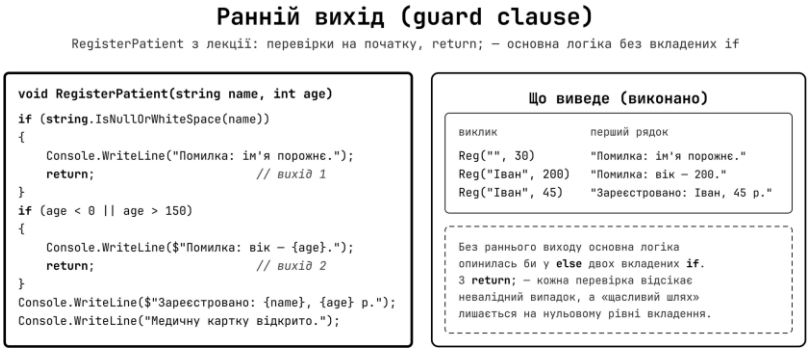
return у void-методах: ранній вихід

Оператор return можна застосовувати й у методах типу void — тих, що не повертають жодного значення. У цьому випадку після return не

вказується ніяке значення, а сам оператор використовується виключно для **дострокового завершення** методу:

```
C#
1  return;
```

Типова ситуація — **перевірка вхідних умов** на початку методу. Якщо дані некоректні, метод одразу завершується, не виконуючи основну логіку. Цей патерн має назву **Guard Clause** (охоронна умова) — серія перевірок на початку методу, кожна з яких здійснює ранній вихід при порушенні інваріанту:



Патерн Guard Clause: ранній вихід із методу

Розглянемо практичний приклад: метод реєстрації пацієнта повинен перевіряти коректність даних перед будь-якою обробкою.

```
C#
 1  using System;
 2
 3  void RegisterPatient(string name, int age)
 4  {
 5      if (string.IsNullOrWhiteSpace(name))
 6      {
 7          Console.WriteLine("Помилка: ім'я пацієнта не може бути порожнім.");
 8          return;
 9      }
10
11      if (age < 0 || age > 150)
12      {
13          Console.WriteLine($"Помилка: некоректний вік — {age.ToString()}.");
14          return;
15      }
16
```

```
17      Console.WriteLine($"Пацієнта зареєстровано: {name}, {age.ToString()} р.");
18      Console.WriteLine("Медичну картку відкрито.");
19  }
20
21  RegisterPatient("Іван Петренко", 45);
22  RegisterPatient("", 30);
23  RegisterPatient("Марія Сидоренко", -5);
24  RegisterPatient("Олег Бойко", 200);
```

Метод RegisterPatient містить дві охоронні умови. Перша перевіряє ім'я: якщо воно порожнє або складається лише з пробілів — виводить помилку і виходить. Друга перевіряє вік: якщо значення виходить за межі можливого — аналогічно завершує роботу. Тільки якщо обидві перевірки пройдено, виконується основна логіка реєстрації.

Такий підхід має кілька переваг. По-перше, основна логіка методу не загорнута у вкладені умови — вона читається прямолінійно. По-друге, кожна некоректна умова обробляється в одному місці і одразу при вході в метод. По-третє, метод сам гарантує коректність своїх передумов, а не покладається на перевірку ззовні.

Ранній вихід через return у void-методах є стандартною практикою C# і активно використовується в реальних проєктах для захисного програмування та покращення читабельності коду.

2.19. Передача параметрів за посиланням та значенням. Вихідні параметри

Коли метод отримує параметр, виникає ключове питання: що саме він отримує — незалежну копію значення чи прямий доступ до оригінальної змінної в пам'яті? Відповідь на це питання визначає поведінку всієї програми. У C# існують два принципово різні способи передачі параметрів: **за значенням** і **за посиланням**. Вибір між ними впливає на те, чи зможе метод змінити вхідні дані, скільки значень він здатний повернути, і наскільки ефективно використовується пам'ять.

Передача параметрів за значенням

Передача за значенням — це стандартний і найпростіший спосіб. Коли аргумент передається за значенням, метод отримує **окрему копію** змінної. Метод працює зі своєю копією незалежно від оригінальної змінної, і будь-які зміни всередині методу не впливають на змінну у викликаючому коді.

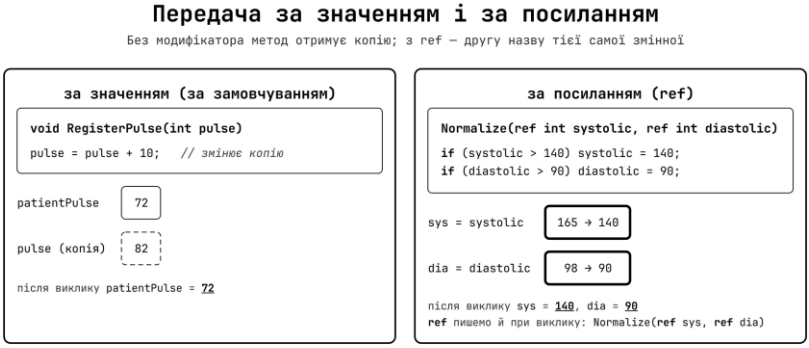
```
C#
```

```
using System;

void RegisterPulse(int pulse)
{
    pulse = pulse + 10; // змінюємо копію
    Console.WriteLine($"Пульс у методі: {pulse.ToString()} уд/хв");
}

int patientPulse = 80;
Console.WriteLine($"Пульс до виклику: {patientPulse.ToString()} уд/хв");
RegisterPulse(patientPulse);
Console.WriteLine($"Пульс після виклику: {patientPulse.ToString()} уд/хв");
```

Метод RegisterPulse отримує копію значення patientPulse. Збільшення pulse всередині методу змінює лише цю копію — вона існує окремо в пам'яті і зникає після завершення методу. Оригінальна змінна patientPulse у викликаючому коді залишається незмінною.



Різниця між передачею за значенням і за посиланням

Передача параметрів за посиланням та модифікатор ref

При передачі за посиланням метод отримує не копію, а **адресу оригінальної змінної в пам'яті**. Це означає, що параметр і оригінальна змінна вказують на одне й те саме місце в пам'яті. Будь-яка зміна параметра всередині методу відразу відображається на оригінальній змінній.

Для передачі за посиланням використовується модифікатор ref — він вказується як при оголошенні методу, так і при кожному виклику:

```
C#
using System;

void NormalizeBloodPressure(ref int systolic, ref int diastolic)
```

```
 4  {
 5      if (systolic > 140) systolic = 140;
 6      if (diastolic > 90) diastolic = 90;
 7      Console.WriteLine($"Нормалізовано:
    {systolic.ToString()}/{diastolic.ToString()} мм рт.ст.");
 8  }
 9
10  int sys = 165;
11  int dia = 98;
12  Console.WriteLine($"До: {sys.ToString()}/{dia.ToString()} мм рт.ст.");
13  NormalizeBloodPressure(ref sys, ref dia);
14  Console.WriteLine($"Після: {sys.ToString()}/{dia.ToString()} мм рт.ст.");
```

Обидві змінні sys і dia передаються за посиланням. Метод NormalizeBloodPressure безпосередньо змінює їх значення у пам'яті — після виклику в змінних буде вже скоригований тиск. Зверніть увагу: ключове слово ref потрібне і в оголошенні методу, і в кожному виклику — це явний сигнал читачу, що значення можуть змінитись.

Змінна, яка передається через ref, **обов'язково** повинна бути ініціалізована перед викликом. Компілятор не дозволить передати неініціалізовану змінну — адже метод може як читати, так і записувати значення.

Вихідні параметри. Модифікатор out

Модифікатор out схожий на ref, але з важливою відмінністю: вихідний параметр не потребує ініціалізації до виклику — він призначений виключно для **повернення значення з методу**. Натомість метод зобов'язаний присвоїти йому значення перед завершенням.

```
C#
 1  using System;
 2
 3  void CalculateVitalSigns(int age, out int maxHeartRate, out int
    targetHeartRate)
 4  {
 5      maxHeartRate = 220 - age;
 6      targetHeartRate = (int)((220 - age) * 0.75);
 7  }
 8
 9  int max;
10  int target;
11  CalculateVitalSigns(45, out max, out target);
12  Console.WriteLine($"Максимальний пульс: {max.ToString()} уд/хв");
13  Console.WriteLine($"Цільовий пульс: {target.ToString()} уд/хв");
```

Головна перевага out-параметрів — можливість повернути **кілька значень** з одного методу. Оператор return дозволяє повернути лише

одне значення, тоді як метод може мати довільну кількість out-параметрів.

Якщо метод не присвоює значення out-параметру — це помилка компіляції. Компілятор суворо відстежує, що кожна гілка виконання методу обов'язково записує значення в усі out-параметри:

```
C#
1  // ПОМИЛКА — out-параметр не отримав значення
2  void GetData(out int result)
3  {
4      Console.WriteLine("обробка...");
5      // result не присвоєно — компілятор відмовить
6  }
```

Inline-оголошення out-змінних

C# дозволяє оголошувати змінні, що передаються як out-аргументи, безпосередньо у місці виклику методу — без попереднього оголошення:

```
C#
 1  using System;
 2
 3  void AnalyzeBloodTest(
 4      double hemoglobin,
 5      out bool isAnemia,
 6      out string severity)
 7  {
 8      isAnemia = hemoglobin < 120.0;
 9      if      (hemoglobin < 70.0)  severity = "Тяжка анемія";
10      else if (hemoglobin < 90.0)  severity = "Середня анемія";
11      else if (hemoglobin < 120.0) severity = "Легка анемія";
12      else                         severity = "Норма";
13  }
14
15  AnalyzeBloodTest(85.0, out bool hasAnemia, out string grade);
16  Console.WriteLine($"Анемія: {hasAnemia.ToString()}");
17  Console.WriteLine($"Ступінь: {grade}");
18
19  AnalyzeBloodTest(130.0, out bool hasAnemia2, out string grade2);
20  Console.WriteLine($"Анемія: {hasAnemia2.ToString()}");
21  Console.WriteLine($"Ступінь: {grade2}");
```

Якщо тип out-змінної можна визначити автоматично, замість явного типу можна писати var:

```
C#
1  AnalyzeBloodTest(85.0, out var hasAnemia, out var grade);
```

Компілятор сам визначає тип змінних зі сигнатури методу. Це зручно, коли тип довгий або очевидний з контексту.

Вхідні параметри. Модифікатор in

Модифікатор in вирішує специфічну задачу: передати параметр за посиланням (уникнувши копіювання), але **заборонити його зміну** всередині методу. Це комбінація ефективності посилання і гарантії незмінності значення.

```C#
using System;

void PrintPatientSummary(in string name, in int age, in double weight)
{
    // name = "Інший";  // ПОМИЛКА — модифікатор in забороняє зміну
    Console.WriteLine($"Пацієнт: {name}");
    Console.WriteLine($"Вік: {age.ToString()} р., Вага: {weight.ToString("F1")} кг");
    double bmi = weight / ((age < 18 ? 1.6 : 1.75) * (age < 18 ? 1.6 : 1.75));
    Console.WriteLine($"ІМТ (орієнтовно): {bmi.ToString("F1")}");
}

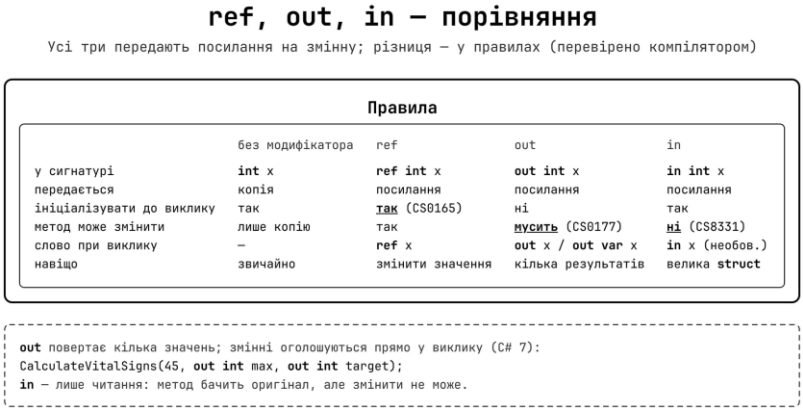
string patientName = "Олена Гончар";
int patientAge = 42;
double patientWeight = 68.5;

PrintPatientSummary(in patientName, in patientAge, in patientWeight);
```

Модифікатор in на стороні виклику є необов'язковим — компілятор розуміє що передається за посиланням і без нього. Але явне написання in корисне для читача: воно підкреслює, що змінна передається за посиланням і не буде змінена.

Практична цінність in найбільша для **великих структур** (struct з багатьма полями), де копіювання дорого коштує. Для примітивних типів (int, double) різниця непомітна.

Порівняння модифікаторів



Порівняльна таблиця ref, out, in

Коротке практичне правило вибору:

— **ref** — коли метод повинен і читати, і змінювати існуюче значення (наприклад, лічильник, накопичувач);

— **out** — коли метод повинен повернути кілька результатів, яких ще не існує;

— **in** — коли метод лише читає велику структуру і не повинен її змінювати;

— **без модифікатора** — у більшості випадків, коли метод просто використовує отримані дані.

2.20. Масив параметрів та ключове слово params

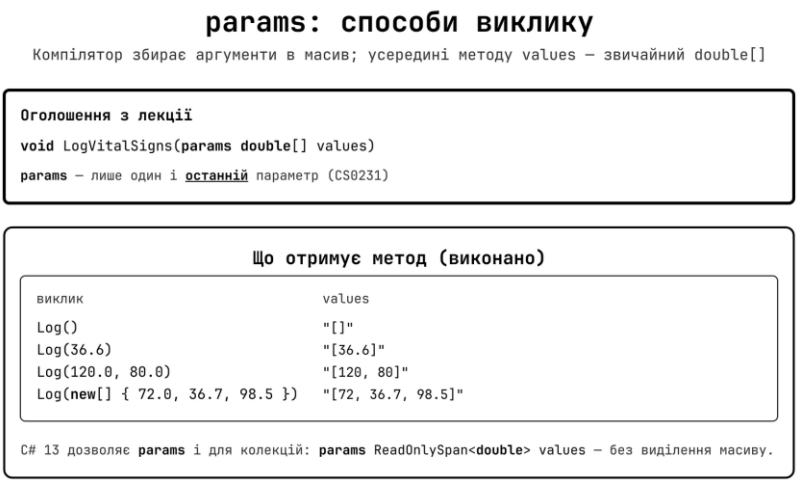
Усі методи, розглянуті раніше, мали фіксовану кількість параметрів — кожен виклик передавав рівно стільки аргументів, скільки оголошено у сигнатурі. Але іноді кількість вхідних даних заздалегідь невідома: метод для логування показників може отримати одне значення або десять, метод агрегації — будь-яку кількість чисел. Ключове слово params вирішує цю задачу, дозволяючи приймати змінну кількість аргументів одного типу.

Синтаксис params

Параметр із ключовим словом params оголошується як одновимірний масив. При виклику методу на його місці можна передати довільну кількість значень через кому, готовий масив, або взагалі нічого:

```
C#
 1  using System;
 2
 3  void LogVitalSigns(params double[] values)
 4  {
 5      Console.WriteLine($"Отримано показників: {values.Length.ToString()}");
 6      foreach (var v in values)
 7          Console.WriteLine($"  {v.ToString("F1")}");
 8  }
 9
10  LogVitalSigns(36.6);
11  LogVitalSigns(120.0, 80.0);
12  LogVitalSigns(72.0, 36.7, 98.5, 120.0);
13  LogVitalSigns(); // порожній масив — теж допустимо
```

Незалежно від способу виклику, всередині методу values — це звичайний масив double[]. Можна звертатись до values.Length, ітерувати через foreach, звертатись за індексом. Якщо аргументів не передано — values.Length дорівнює нулю, і метод коректно обробляє порожній випадок.



Способи виклику методу з params

Комбінування з іншими параметрами

Параметр params може бути не єдиним у методі. Допускається передувати йому будь-яка кількість звичайних параметрів. Вони заповнюються першими, а все що залишається потрапляє до params-масиву:

```C#
using System;

void LogMeasurements(string label, string unit, params double[] values)
{
    Console.WriteLine($"=== {label} ({unit}) ===");
    if (values.Length == 0)
    {
        Console.WriteLine("  Даних немає");
        return;
    }
    double sum = 0;
    foreach (var v in values) sum += v;
    double avg = sum / values.Length;
    Console.WriteLine($"  Кількість: {values.Length.ToString()}");
    Console.WriteLine($"  Середнє:  {avg.ToString("F1")}");
}

LogMeasurements("Артеріальний тиск", "мм рт.ст.", 120.0, 130.0, 125.0);
LogMeasurements("Температура", "°C", 36.6, 37.1, 36.8, 36.9);
LogMeasurements("Пульс", "уд/хв");
```

Перші два аргументи (label і unit) — обов'язкові фіксовані параметри, вони завжди присутні при виклику. Все що йде після них — збирається у масив values. Якщо після фіксованих параметрів нічого не передано, values буде порожнім масивом.

Важливе обмеження: params-параметр **завжди останній** у списку. Розмістити після нього будь-які інші параметри — помилка компіляції, бо компілятор не зможе відрізнити де закінчуються значення для params і починаються наступні параметри:

```C#
// ПОМИЛКА — params не може стояти не останнім
void Log(params double[] values, string label)
{
}
```

Передача готового масиву

Замість переліку окремих значень через кому, на місце params-параметра можна передати вже готовий масив. Метод не робить різниці між цими двома способами виклику:

```C#
using System;

void PrintReadings(string patientName, params double[] temps)
{
    Console.WriteLine($"Пацієнт: {patientName}");
    foreach (var t in temps)
        Console.WriteLine($"  Температура: {t.ToString("F1")} °C");
}

// Передаємо окремі значення
PrintReadings("Іван Петренко", 36.6, 37.0, 36.8);

// Передаємо масив — результат той самий
double[] morningReadings = { 36.4, 36.7, 36.9 };
PrintReadings("Марія Сидоренко", morningReadings);
```

Обидва виклики абсолютно еквівалентні — з точки зору методу він завжди отримує double[].

Масив як звичайний параметр — відмінність від params

Параметр-масив без ключового слова params — це зовсім інша конструкція. Він вимагає явно передавати масив, і не дозволяє записувати окремі значення через кому:

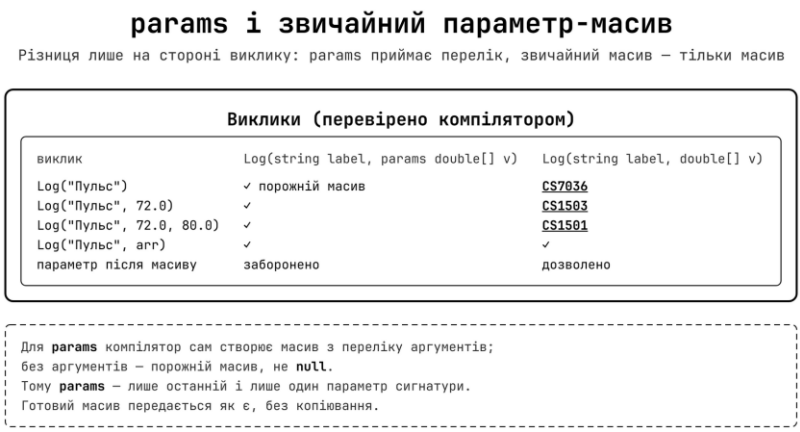
```C#
using System;

// Без params — лише масив
void AnalyzeReadings(double[] values, string label)
{
    double sum = 0;
    foreach (var v in values) sum += v;
    Console.WriteLine($"{label}: середнє = {(sum / values.Length).ToString("F1")}");
}

double[] pulses = { 72.0, 75.0, 68.0, 80.0 };
AnalyzeReadings(pulses, "Пульс");

// AnalyzeReadings(72.0, 75.0, "Пульс"); // ПОМИЛКА — потрібен масив
```

На відміну від params, після звичайного масиву-параметра можуть стояти інші параметри — компілятор точно знає, де закінчується масив (він завжди один об'єкт), і може прийняти наступний аргумент.



params vs звичайний масив-параметр

Вибір між params і звичайним масивом залежить від сценарію: params зручний у публічному API де викликаючий код не хоче щоразу створювати масив; звичайний масив-параметр кращий коли дані вже зібрані в колекцію і потрібно мати після нього ще інші параметри.

2.21. Рекурсивні функції

Рекурсія — це механізм, за якого функція викликає саму себе. На перший погляд це звучить як нескінченне зациклення, але в правильно написаній рекурсивній функції кожен наступний виклик отримує простіший варіант задачі, а вся послідовність завершується на конкретному базовому випадку. Рекурсія природно описує задачі, які мають вкладену самоподібну структуру: факторіал, числа Фібоначчі, обхід ієрархії файлів або структури даних типу дерево.

Два обов'язкових елементи рекурсивної функції

Будь-яка коректна рекурсивна функція повинна містити два структурних елементи:

7. **Базовий випадок** (base case) — умова, при якій функція повертає результат безпосередньо, без подальших рекурсивних викликів. Це «дно» рекурсії, звідки починається розгортання результатів.

8. **Рекурсивний крок** — виклик самої функції з аргументом, що наближає задачу до базового випадку. Кожен виклик повинен зменшувати складність задачі, щоб зрештою досягти базового випадку.

Якщо відсутній базовий випадок або рекурсивний крок ніколи не наближається до нього — функція рекурсуватиме нескінченно і призведе до StackOverflowException.

Рекурсивна функція факторіалу

Факторіал числа n визначається як добуток усіх натуральних чисел від 1 до n: n! = 1 × 2 × … × n. Математично його можна виразити рекурентно: n! = n × (n-1)!, і ця формула безпосередньо відображається у рекурсивному методі:

```
C#
 1  using System;
 2
 3  int Factorial(int n)
 4  {
 5      if (n == 1) return 1;        // базовий випадок
 6      return n * Factorial(n - 1); // рекурсивний крок
 7  }
 8
 9  Console.WriteLine($"Factorial(4) = {Factorial(4).ToString()}");
10  Console.WriteLine($"Factorial(5) = {Factorial(5).ToString()}");
11  Console.WriteLine($"Factorial(6) = {Factorial(6).ToString()}");
```

Базовий випадок — n == 1, де відоме значення без обчислення. Рекурсивний крок передає n - 1, наближаючись до бази з кожним викликом.

Поетапне розгортання: як виконується Factorial(4)

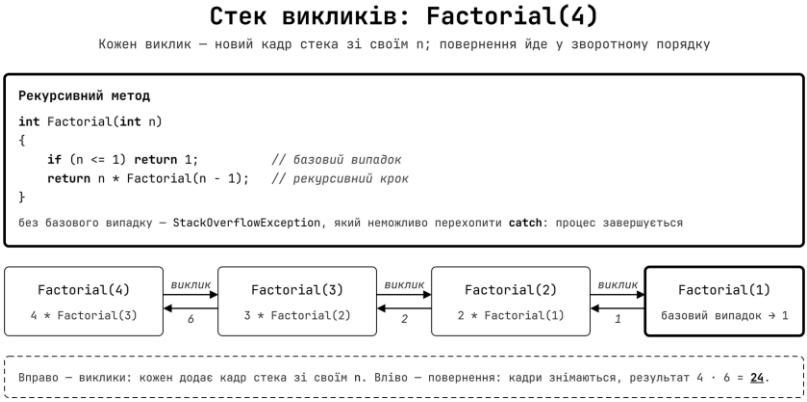
Щоб зрозуміти механізм рекурсії, розглянемо крок за кроком що відбувається при виклику Factorial(4):

9. Factorial(4): n ≠ 1, виконується return 4 * Factorial(3) — чекає результату
10. Factorial(3): n ≠ 1, виконується return 3 * Factorial(2) — чекає результату
11. Factorial(2): n ≠ 1, виконується return 2 * Factorial(1) — чекає результату
12. Factorial(1): n == 1, **базовий випадок** — повертає 1

Тепер стек розгортається у зворотньому напрямку:

— Factorial(2) отримує 1, повертає 2 × 1 = 2
— Factorial(3) отримує 2, повертає 3 × 2 = 6
— Factorial(4) отримує 6, повертає 4 × 6 = 24

Фактично виконується: 4 × 3 × 2 × 1 = 24.



Стек викликів при виконанні Factorial(4)

Кожен активний виклик функції займає місце на стеку викликів і зберігає свій локальний контекст (значення n, адресу повернення). Після досягнення базового випадку стек починає «розкручуватись» — кожна функція отримує результат від нижньої та повертає свій.

Рекурсивна функція Фібоначчі

Числа Фібоначчі — ще один класичний приклад рекурсії. Послідовність визначається формулою f(n) = f(n-1) + f(n-2), де f(0) = 0, f(1) = 1. Перші члени: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34…

C#
```
using System;

int Fibonacci(int n)
{
    if (n == 0 || n == 1) return n; // два базових випадки
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}

for (int i = 0; i <= 10; i++)
    Console.WriteLine($"f({i.ToString()}) = {Fibonacci(i).ToString()}");
```

Тут два базових випадки: f(0) = 0 і f(1) = 1. Рекурсивний крок розбиває задачу на два менших підзавдання. Зверніть увагу: для Fibonacci(5) деякі підзавдання обчислюються повторно — наприклад, Fibonacci(2) викликається кілька разів. Це класична проблема наївної рекурсії Фібоначчі, де кількість викликів зростає експоненційно.

Рекурсія та цикли — порівняння

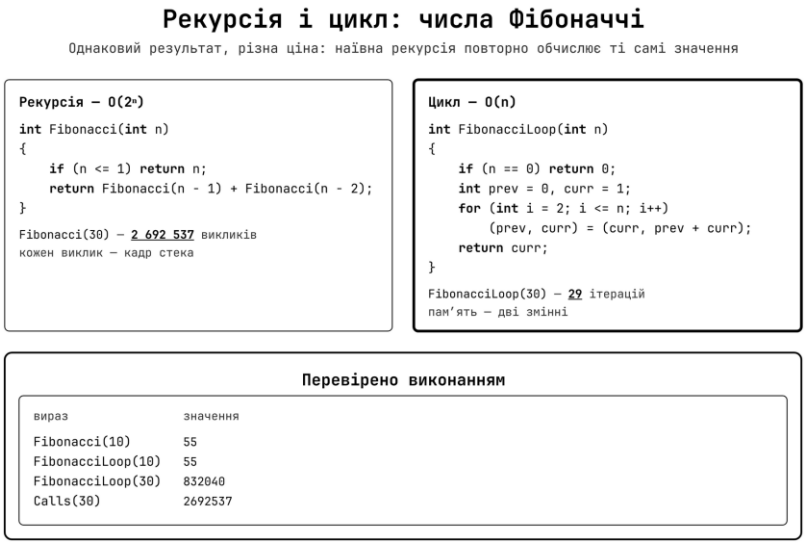
Більшість рекурсивних алгоритмів можна переписати через цикл. Ось варіант Фібоначчі без рекурсії:

```C#
using System;

int FibonacciLoop(int n)
{
    if (n == 0) return 0;
    int prev = 0;
    int curr = 1;
    for (int i = 2; i <= n; i++)
    {
        int next = prev + curr;
        prev = curr;
        curr = next;
    }
    return curr;
}

for (int i = 0; i <= 10; i++)
    Console.WriteLine($"f({i.ToString()}) = {FibonacciLoop(i).ToString()}");
```

Варіант із циклом виконується за лінійний час O(n) замість $O(2^n)$ у наївній рекурсії. Жодного повторного обчислення, жодних витрат на стек.



Рекурсія vs Цикл: коли що обирати

Коли рекурсія виправдана

Рекурсія не є поганим підходом — вона є природним і виразним рішенням для певного класу задач:

- **Ієрархічні структури**: обхід дерева каталогів, XML/JSON-документа, дерева виразів. Ітеративний обхід таких структур вимагає ручного управління власним стеком — рекурсія робить це автоматично.
- **Алгоритми «розділяй і володарюй»**: злиттєве сортування (merge sort), швидке сортування (quicksort), бінарний пошук у рекурсивному записі — усі вони природно розбивають задачу на підзадачі.
- **Математичні означення, що мають рекурентну форму**: якщо сама задача визначена рекурентно (як факторіал чи Фібоначчі), рекурсивний код читається як пряме відображення означення.

У прикладному коді медичної інформаційної системи рекурсія може знадобитись при обході ієрархії підрозділів лікарні, де кожен відділ може мати вкладені підвідділи довільної глибини. Ітерувати таку структуру через цикл значно складніше, ніж обійти рекурсивно.

Для лінійних задач — підсумовування масиву, пошук максимуму, послідовна обробка — цикл завжди кращий вибір: він швидший, використовує менше пам'яті і не загрожує StackOverflowException при великих вхідних даних.

2.22. Локальні функції

Локальна функція — це функція, оголошена безпосередньо всередині іншого методу. Вона існує і видима виключно в межах того методу, де оголошена, і не може бути викликана ззовні. Локальні функції вирішують конкретну проблему: дозволяють виокремити фрагмент логіки у власний іменований блок, не виносячи його у простір імен класу як окремий метод. Це корисно тоді, коли певна допоміжна операція потрібна лише в одному місці і немає сенсу робити її частиною публічного або навіть приватного API класу.

Навіщо потрібні локальні функції

Розглянемо метод, який аналізує результати аналізів пацієнта і перевіряє кілька показників. Без локальної функції повторювана логіка перевірки дублюється:

```
C#
 1  using System;
 2
 3  void AnalyzeBloodTest(double hemoglobin, double glucose, double cholesterol)
 4  {
 5      // перевірка гемоглобіну
 6      string hgbStatus = hemoglobin < 120 ? "Низький" : hemoglobin > 170 ?
    "Високий" : "Норма";
 7      Console.WriteLine($"Гемоглобін {hemoglobin.ToString("F1")}: {hgbStatus}");
 8
 9      // та сама логіка для глюкози і холестерину — дублювання
10      string glcStatus = glucose < 3.9 ? "Низький" : glucose > 6.1 ? "Високий" :
    "Норма";
11      Console.WriteLine($"Глюкоза {glucose.ToString("F1")}: {glcStatus}");
12
13      string chlStatus = cholesterol < 3.0 ? "Низький" : cholesterol > 5.2 ?
    "Високий" : "Норма";
14      Console.WriteLine($"Холестерин {cholesterol.ToString("F1")}: {chlStatus}");
15  }
16
17  AnalyzeBloodTest(115.0, 5.4, 6.1);
```

Логіка "Низький" / "Норма" / "Високий" повторюється тричі. Локальна функція усуває це дублювання, залишаючи код лише всередині методу:

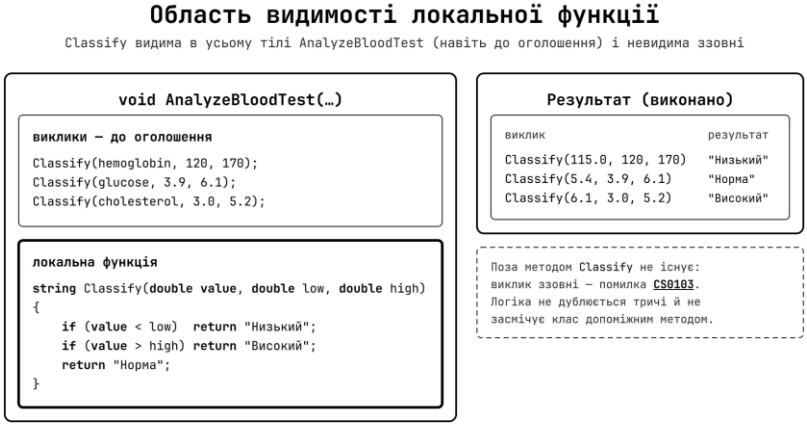
```C#
using System;

void AnalyzeBloodTest(double hemoglobin, double glucose, double cholesterol)
{
    Console.WriteLine($"Гемоглобін {hemoglobin.ToString("F1")}: {Classify(hemoglobin, 120, 170)}");
    Console.WriteLine($"Глюкоза {glucose.ToString("F1")}: {Classify(glucose, 3.9, 6.1)}");
    Console.WriteLine($"Холестерин {cholesterol.ToString("F1")}: {Classify(cholesterol, 3.0, 5.2)}");

    string Classify(double value, double low, double high)
    {
        if (value < low)  return "Низький";
        if (value > high) return "Високий";
        return "Норма";
    }
}

AnalyzeBloodTest(115.0, 5.4, 6.1);
AnalyzeBloodTest(135.0, 4.5, 4.8);
```

Локальна функція Classify визначена після того, як використовується — у C# порядок оголошення всередині методу не має значення. Вона видима у всьому тілі методу AnalyzeBloodTest, але абсолютно невидима ззовні.



Область видимості локальної функції

Доступ до змінних зовнішнього методу

Звичайна (нестатична) локальна функція має доступ до змінних і параметрів методу, всередині якого вона визначена. Ці змінні можна читати і змінювати без явної передачі як параметри:

```C#
using System;

void GeneratePatientReport(string patientName, int wardNumber)
{
    string header = $"=== Пацієнт: {patientName} (Палата
{wardNumber.ToString()}) ===";

    PrintSection("Анамнез", "Хронічний бронхіт, 5 років");
    PrintSection("Скарги", "Кашель, задишка при навантаженні");
    PrintSection("Призначення", "Інгалятор, фізіотерапія");

    void PrintSection(string title, string content)
    {
        // header — змінна зовнішнього методу, доступна без параметра
        Console.WriteLine(header);
        Console.WriteLine($"  [{title}]: {content}");
    }
}

GeneratePatientReport("Іван Петренко", 7);
```

Функція PrintSection читає змінну header зовнішнього методу напряму. Це зручно, але вимагає уваги: локальна функція може ненавмисно змінити змінні зовнішнього контексту, що ускладнює розуміння коду.

Статичні локальні функції

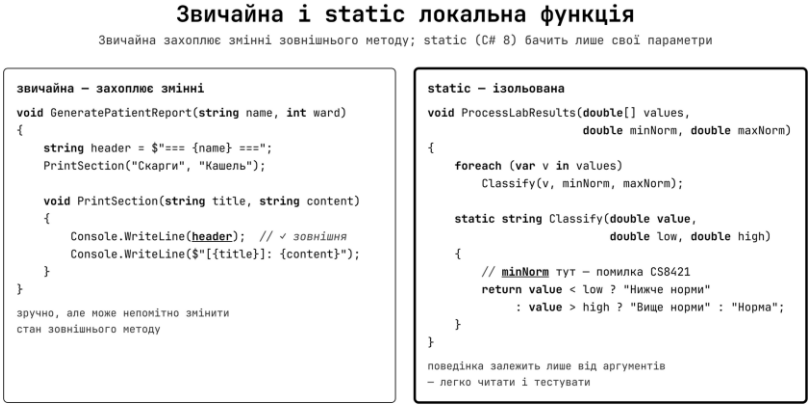
Щоб гарантовано заборонити локальній функції звертатися до змінних зовнішнього методу, її можна оголосити зі словом static. Статична локальна функція ізольована: вона бачить лише те, що отримує через свої параметри. Будь-яка спроба звернутися до змінної зовнішнього методу всередині статичної локальної функції — це помилка компіляції.

```C#
using System;

void ProcessLabResults(double[] values, double minNorm, double maxNorm)
{
    int belowCount = 0;
    int aboveCount = 0;

    foreach (var v in values)
```

```
 9      {
10          string status = Classify(v, minNorm, maxNorm);
11          Console.WriteLine($"  {v.ToString("F1")} → {status}");
12          if (status == "Нижче норми") belowCount++;
13          if (status == "Вище норми")  aboveCount++;
14      }
15
16      Console.WriteLine($"Нижче норми: {belowCount.ToString()}, Вище норми:
    {aboveCount.ToString()}");
17
18      static string Classify(double value, double low, double high)
19      {
20          // minNorm, maxNorm — НЕ доступні тут (статична функція)
21          if (value < low)  return "Нижче норми";
22          if (value > high) return "Вище норми";
23          return "Норма";
24      }
25  }
26
27  double[] glucoseReadings = { 3.5, 5.1, 7.2, 4.8, 6.8, 3.1 };
28  ProcessLabResults(glucoseReadings, 3.9, 6.1);
```



Звичайна vs static локальна функція

Перевага static локальної функції — явна гарантія ізольованості. Читаючи сигнатуру static string Classify(...), розробник одразу розуміє: ця функція не має прихованих побічних ефектів через захоплені змінні, її поведінка визначається лише аргументами. Це робить код передбачуваним і простішим для тестування.

Локальні функції проти приватних методів

Локальна функція і приватний метод класу вирішують схожу задачу — ізоляція допоміжної логіки — але мають різну область застосування:

— **Приватний метод** доступний усьому класу. Використовуйте його, якщо допоміжна логіка може знадобитися кільком методам класу.

— **Локальна функція** доступна лише одному методу. Використовуйте її, якщо логіка потрібна виключно в одному місці і виносити її в клас означало б штучно розширювати API.

Локальні функції також корисні для покращення читабельності складних методів: замість того щоб читати 80 рядків коду підряд, читач бачить короткий метод з виразними викликами іменованих локальних функцій і занурюється в деталі лише за потреби.

2.23. Конструкція switch

Конструкція switch призначена для порівняння одного виразу з набором фіксованих значень і виконання різних дій залежно від результату. Вона є структурованою альтернативою ланцюжку if / else if / else у тих ситуаціях, де потрібно обробити кілька конкретних варіантів одного значення. switch не лише читабельніший у таких сценаріях — він явно сигналізує читачу, що всі гілки обробляють одну змінну, а не довільні незалежні умови.

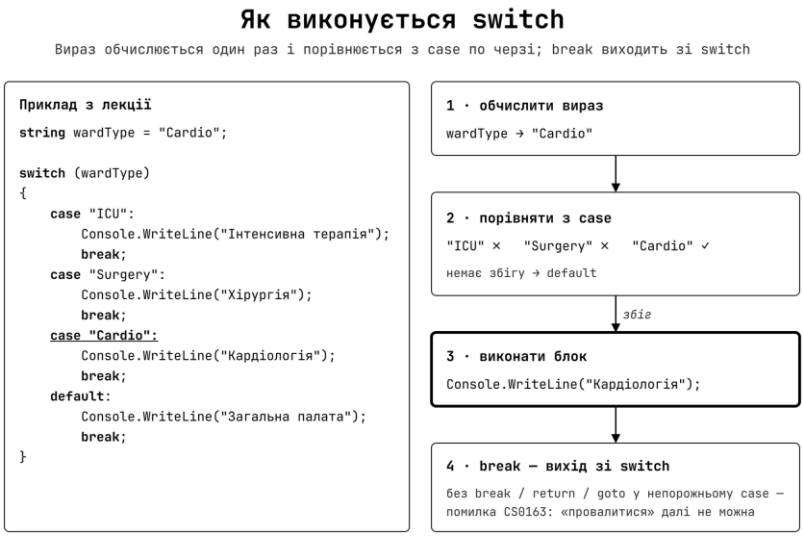
Синтаксис switch/case

Формальна структура конструкції switch:

```
switch (вираз)
{
    case значення1:
        код для значення1
        break;
    case значення2:
        код для значення2
        break;
    // ...
    default:
        код для всіх інших значень
        break;
}
```

Вираз у дужках після switch обчислюється один раз. Його значення послідовно порівнюється з кожним case. При збігу виконується

відповідний блок коду. Наприкінці кожного case обов'язково стоїть оператор переходу — найчастіше break, який зупиняє виконання та виходить із switch.



Потік виконання конструкції switch/case

Базовий приклад: тип відділення

```C#
using System;

string wardType = "Cardio";

switch (wardType)
{
    case "ICU":
        Console.WriteLine("Відділення: Інтенсивна терапія");
        break;
    case "Surgery":
        Console.WriteLine("Відділення: Хірургія");
        break;
    case "Cardio":
        Console.WriteLine("Відділення: Кардіологія");
        break;
    default:
        Console.WriteLine("Відділення: Загальна палата");
        break;
}
```

Значення "Cardio" збігається з третім case, виводиться відповідне повідомлення, break завершує конструкцію. Якщо wardType не збігається з жодним case — виконується блок default. Блок default необов'язковий, але його рекомендується додавати завжди як захисну гілку для непередбачених значень.

Кілька значень для одного блоку

Якщо кілька значень мають обробляти однаковий код, їх case-мітки можна розміщувати поспіль без break між ними:

```C#
using System;

string diagnosis = "Bronchitis";

switch (diagnosis)
{
    case "Pneumonia":
    case "Bronchitis":
    case "Pleuritis":
        Console.WriteLine("Профіль: Пульмонологія");
        break;
    case "Infarction":
    case "Arrhythmia":
        Console.WriteLine("Профіль: Кардіологія");
        break;
    default:
        Console.WriteLine("Профіль: Терапія (загальний)");
        break;
}
```

Три case-мітки підряд (Pneumonia, Bronchitis, Pleuritis) без break між ними означають: якщо значення збігається з будь-яким із них — виконується один і той самий блок. Це єдиний дозволений в C# спосіб «провалитися» між case.

Оператор goto case

Якщо після виконання одного case потрібно явно перейти до іншого (не наступного за списком), використовується goto case:

```C#
using System;

int priority = 1;

switch (priority)
{
    case 1:
```

```
 8          Console.WriteLine("КРИТИЧНИЙ пріоритет — негайна госпіталізація");
 9          goto case 2;
10      case 2:
11          Console.WriteLine("Виклик чергового лікаря");
12          break;
13      case 3:
14          Console.WriteLine("Запис на прийом");
15          break;
16      default:
17          Console.WriteLine("Стандартна черга");
18          break;
19  }
```

При priority == 1 спочатку виводиться повідомлення про критичний пріоритет, потім goto case 2 перекидає виконання до блоку case 2. Використовувати goto case слід обережно — надмірне застосування ускладнює читання.

switch у методах з return

Конструкцію switch зручно використовувати всередині методів для повернення різних значень. Оператор return у блоці case одночасно виходить із switch і з методу, тому break у таких блоках не потрібен:

```
C#
 1  using System;
 2
 3  string GetWardName(string code)
 4  {
 5      switch (code)
 6      {
 7          case "ICU":     return "Інтенсивна терапія";
 8          case "SRG":     return "Хірургія";
 9          case "CARD":    return "Кардіологія";
10          case "NEURO":   return "Неврологія";
11          default:        return "Загальне відділення";
12      }
13  }
14
15  Console.WriteLine(GetWardName("ICU"));
16  Console.WriteLine(GetWardName("CARD"));
17  Console.WriteLine(GetWardName("ORTHO"));
```

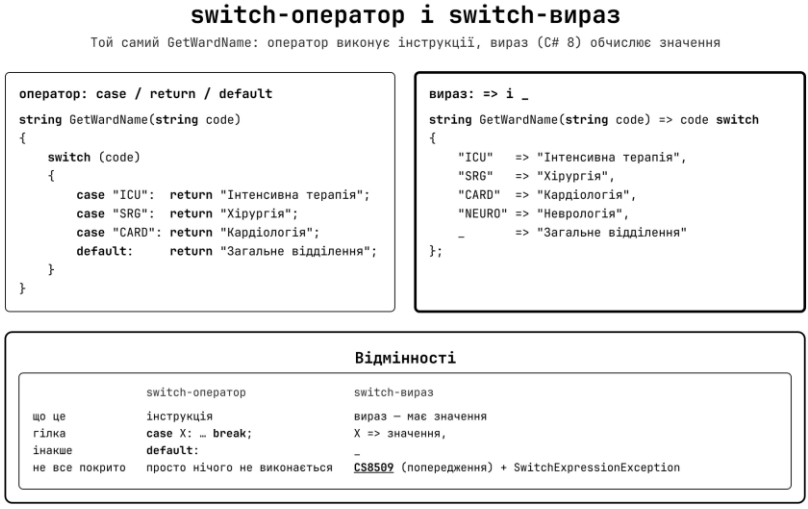
Компілятор перевіряє, що метод повертає значення при всіх можливих варіантах. Якщо відсутній default і жоден case не покриває всі можливі значення — виникне помилка компіляції.

switch-вираз (switch expression)

C# надає компактний синтаксис для switch, що повертає значення — **switch-вираз**. Замість ключового слова case і break використовується оператор =>, а замість default — символ підкреслення _:

```
C#
 1  using System;
 2
 3  string GetWardName(string code) => code switch
 4  {
 5      "ICU"   => "Інтенсивна терапія",
 6      "SRG"   => "Хірургія",
 7      "CARD"  => "Кардіологія",
 8      "NEURO" => "Неврологія",
 9      _       => "Загальне відділення"
10  };
11
12  Console.WriteLine(GetWardName("ICU"));
13  Console.WriteLine(GetWardName("CARD"));
14  Console.WriteLine(GetWardName("ORTHO"));
```

Switch-вираз є виразом, а не оператором — він обчислюється і повертає значення. Його можна використати безпосередньо в присвоєнні або у return. Результат кожної гілки — значення після =>, гілки розділяються комами, а не break.



switch-оператор vs switch-вираз

Switch-вираз вимагає **вичерпності**: компілятор перевіряє, що покриті всі можливі значення. Якщо гілка _ відсутня, компілятор видає **попередження** CS8509 («switch-вираз не обробляє всі можливі значення»), але код компілюється; і якщо під час виконання вхідне значення не збігається з жодною гілкою — буде кинуто SwitchExpressionException. Тому для рядкових і числових типів гілка _ практично завжди обов'язкова.

Ще приклад з обчисленням: switch-вираз може повертати не лише рядки, а будь-який тип:

```
C#
 1  using System;
 2
 3  double CalculateDrugDose(string drug, double weightKg)
 4  {
 5      double dosePerKg = drug switch
 6      {
 7          "Aspirin"    => 10.0,
 8          "Ibuprofen"  => 5.0,
 9          "Paracetamol"=> 15.0,
10          _            => 0.0
11      };
12      return weightKg * dosePerKg;
13  }
14
15  Console.WriteLine($"Аспірин 70 кг: {CalculateDrugDose("Aspirin",
    70.0).ToString("F1")} мг");
16  Console.WriteLine($"Ібупрофен 60 кг: {CalculateDrugDose("Ibuprofen",
    60.0).ToString("F1")} мг");
17  Console.WriteLine($"Невідомий: {CalculateDrugDose("Unknown",
    70.0).ToString("F1")} мг");
```

Switch-вираз оцінює назву препарату і повертає дозу на кілограм, після чого обчислюється повна доза. Для невідомого препарату _-гілка повертає 0.0, і результат коректно дорівнює нулю.

2.24. Перерахування enum

Перерахування (enum) — це тип даних, який визначає іменований набір логічно пов'язаних констант. Замість того щоб представляти стан, тип або категорію довільними числами чи рядками, enum дозволяє дати кожному можливому значенню зрозуміле ім'я. Це робить код самодокументованим: PatientStatus.Critical несе значно більше змісту, ніж магічне число 4 або рядок "critical", і водночас захищає від помилок — компілятор перевіряє, що змінній типу PatientStatus можна присвоїти лише одну з оголошених констант.

Оголошення enum

Перерахування оголошується за допомогою ключового слова enum, після якого йде ім'я типу та список констант через кому:

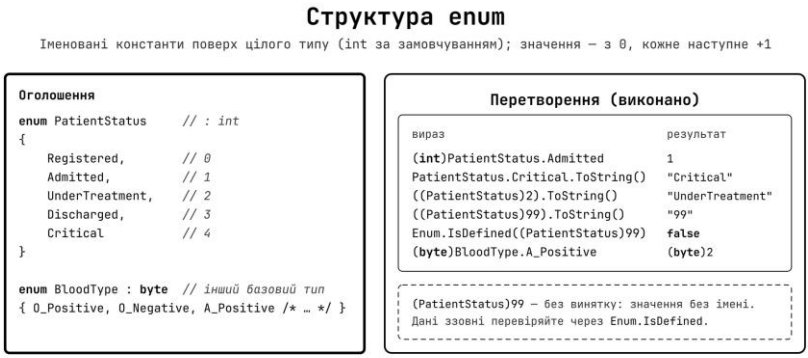
```
enum НазваПерерахування
{
    Константа1,
    Константа2,
    // ...
    КонстантаN
}
```

Оголосимо перерахування для стану пацієнта в медичній системі:

```
C#
using System;

PatientStatus status = PatientStatus.Admitted;
Console.WriteLine($"Статус пацієнта: {status.ToString()}");
Console.WriteLine($"Числове значення: {((int)status).ToString()}");

enum PatientStatus
{
    Registered,       // 0
    Admitted,         // 1
    UnderTreatment,   // 2
    Discharged,       // 3
    Critical          // 4
}
```



Структура enum: константи та числові значення

За замовчуванням константи перерахування мають тип int і автоматично отримують числові значення починаючи з нуля — кожна наступна константа на одиницю більша від попередньої.

Enum як тип: змінні і параметри

Кожне оголошене перерахування є повноцінним типом даних. Змінні, параметри методів і поля класів можуть мати тип enum так само, як і будь-який примітивний тип:

```
C#
 1  using System;
 2
 3  void PrintStatus(string patientName, PatientStatus status)
 4  {
 5      string label = status switch
 6      {
 7          PatientStatus.Registered      => "зареєстрований",
 8          PatientStatus.Admitted        => "прийнятий до відділення",
 9          PatientStatus.UnderTreatment  => "проходить лікування",
10          PatientStatus.Discharged      => "виписаний",
11          PatientStatus.Critical        => "критичний стан",
12          _                             => "невідомий статус"
13      };
14      Console.WriteLine($"{patientName}: {label}");
15  }
16
17  PrintStatus("Іван Петренко",   PatientStatus.Admitted);
18  PrintStatus("Марія Сидоренко", PatientStatus.UnderTreatment);
19  PrintStatus("Олег Бойко",      PatientStatus.Critical);
20
21  enum PatientStatus { Registered, Admitted, UnderTreatment, Discharged, Critical
    }
```

Компілятор не дозволить передати у параметр PatientStatus довільне число або рядок — тільки одну з оголошених констант. Це унеможливлює цілий клас помилок, де передається некоректне значення.

Enum і switch: природне поєднання

Перерахування і конструкція switch чудово доповнюють одне одного. switch гарантує обробку кожного варіанту, а enum обмежує множину можливих значень:

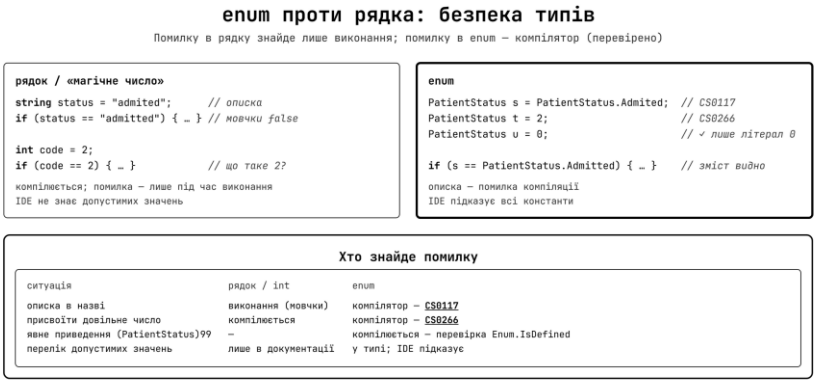
```
C#
 1  using System;
 2
 3  void DescribeWard(WardType ward)
 4  {
 5      switch (ward)
 6      {
 7          case WardType.ICU:
 8              Console.WriteLine("Інтенсивна терапія: цілодобовий моніторинг");
 9              break;
10          case WardType.Surgery:
```

```
            Console.WriteLine("Хірургія: операційні, реанімація");
            break;
        case WardType.Cardiology:
            Console.WriteLine("Кардіологія: ЕКГ, ехокардіографія");
            break;
        case WardType.Neurology:
            Console.WriteLine("Неврологія: МРТ, ЕЕГ");
            break;
        case WardType.General:
            Console.WriteLine("Загальна палата: стандартне лікування");
            break;
    }
}

DescribeWard(WardType.ICU);
DescribeWard(WardType.Cardiology);
DescribeWard(WardType.General);

enum WardType { ICU, Surgery, Cardiology, Neurology, General }
```

Чому enum краще за рядки і «магічні числа»



enum vs рядок: безпека типів

Використання рядків або чисел для представлення фіксованого набору станів має серйозні недоліки: помилки в написанні ("admited" замість "admitted") залишаються непоміченими до часу виконання; компілятор не підказує які значення взагалі допустимі; код, що читає if (status == 2), нічого не говорить про семантику значення 2. enum усуває всі ці проблеми — IDE показує підказки з усіма допустимими значеннями, компілятор відхиляє некоректні присвоєння (описка в імені константи — CS0117, присвоєння числа PatientStatus s = 2; — CS0266), а код читається як звичайна мова. Єдиний виняток — літерал

0: він неявно перетворюється на будь-який enum, тому PatientStatus s = 0; компілюється й дає Registered. Довільне число можна перетворити лише явним приведенням — про це нижче.

Тип і числові значення констант

За замовчуванням enum використовує тип int. Явно вказати інший цілий тип можна через двокрапку після назви:

```
C#
 1  using System;
 2
 3  BloodType blood = BloodType.A_Positive;
 4  Console.WriteLine($"Група крові: {blood.ToString()}");
 5  Console.WriteLine($"Код: {((byte)blood).ToString()}");
 6
 7  enum BloodType : byte   // byte: 0–255, економить пам'ять
 8  {
 9      O_Positive,   // 0
10      O_Negative,   // 1
11      A_Positive,   // 2
12      A_Negative,   // 3
13      B_Positive,   // 4
14      B_Negative,   // 5
15      AB_Positive,  // 6
16      AB_Negative   // 7
17  }
```

Допустимі типи для enum: byte, sbyte, short, ushort, int, uint, long, ulong. Для більшості застосунків int (за замовчуванням) є правильним вибором; byte або short корисні при великій кількості екземплярів enum у пам'яті.

Явне призначення числових значень

Константам можна явно призначити числові значення. Якщо вказати значення лише першій константі — решта отримують значення послідовно від неї:

```
C#
 1  using System;
 2
 3  void HandleRequest(string patientName, PriorityLevel priority)
 4  {
 5      Console.WriteLine($"{patientName}: пріоритет {((int)priority).ToString()}
    ({priority.ToString()})");
 6      if ((int)priority >= (int)PriorityLevel.High)
 7          Console.WriteLine("  → Негайне втручання!");
 8  }
 9
10  HandleRequest("Іван Петренко",   PriorityLevel.Low);
```

```
HandleRequest("Марія Сидоренко", PriorityLevel.High);
HandleRequest("Олег Бойко",      PriorityLevel.Critical);

enum PriorityLevel
{
    Low    = 1,
    Medium = 2,
    High   = 5,
    Critical = 10
}
```

Явні значення зручні, коли числові значення мають зовнішній сенс — наприклад, відповідають кодам у базі даних або протоколі обміну. Перетворення між enum і числовим типом виконується явним приведенням: (int)priority або (PriorityLevel)5.

Зворотне перетворення — з числа у константу enum — також робиться через приведення, але слід пам'ятати: якщо числове значення не відповідає жодній константі, C# не кине винятку, а поверне enum зі значенням, яке не має імені. Тому при роботі з ненадійними зовнішніми даними перед перетворенням варто перевіряти через Enum.IsDefined.

Питання для самоконтролю — Розділ 2. Основи програмування на C#

2.1. Структура програми

1. Що таке top-level statements у C# 9? Що компілятор додає неявно, коли ви пишете код без класу Program і методу Main?
2. Що буде, якщо забути крапку з комою в кінці інструкції? Що саме повідомить компілятор і на якому етапі?
3. У чому різниця між визначенням методу і його викликом? Чи виконується метод автоматично, коли ви його визначаєте?
4. C# є реєстрозалежною мовою. Що виведе наступний код (вкажіть, чи скомпілюється він взагалі)?

```
C#
string name = "Іван";
string Name = "Марія";
Console.WriteLine(name);
Console.WriteLine(Name);
```

5. Поясніть різницю між однорядковим і багаторядковим коментарем. Коли доречно використовувати кожен тип?

6. Що таке блок коду в C#? Скільки рівнів вкладеності допускається і чому блоки є фундаментальною конструкцією мови?
7. Який параметр у файлі .csproj відповідає за версію .NET? Що означає значення net10.0? Що дає параметр <Nullable>enable</Nullable>?
8. Поясніть, навіщо потрібен параметр <ImplicitUsings>enable</ImplicitUsings>. Що відбувається без нього?

2.2. Змінні та константи

9. Яке відношення між поняттями «тип», «ім'я» і «значення» змінної? Навіщо потрібен кожен з цих трьох аспектів?
10. У чому різниця між оголошенням змінної і її ініціалізацією? Що відбудеться, якщо спробувати прочитати значення змінної до ініціалізації?
11. Що виведе наступний код?

```
C#
1  string diagnosis = "Первинний огляд";
2  diagnosis = "Гіпертонія I ст.";
3  diagnosis = "Гіпертонія I ст. — лікування розпочато";
4  Console.WriteLine(diagnosis);
```

12. Перелічіть правила іменування змінних у C#. Яка угода стосовно стилю іменування (camelCase/PascalCase) для локальних змінних?
13. Що таке ключове слово var? Чи є var динамічним типом? Що відбувається після того, як компілятор вивів тип з виразу?
14. Чому наступний код не скомпілюється?

```
C#
1  var x;
2  x = 10;
```

Як правильно записати той самий намір?

15. Що таке константа? Чим вона відрізняється від змінної і коли варто використовувати const замість звичайної змінної?
16. Спроєктуйте набір констант для медичної системи: нормальна температура тіла, мінімальний і максимальний пульс, назва

клініки. Запишіть їх оголошення з дотриманням угод іменування.

2.3. Літерали

17. Що таке літерал? Чим літерал відрізняється від змінної? Наведіть приклади літералів кожного виду.
18. Запишіть число 255 у трьох системах числення: десятковій, двійковій і шістнадцятковій. Як вказати кожну форму в коді C#?
19. Що таке роздільник розрядів _ у числових літералах? Що виведе наступний код?

```
C#
1  int million = 1_000_000;
2  int cardNumber = 1234_5678;
3  Console.WriteLine(million);
4  Console.WriteLine(cardNumber);
```

20. Який тип за замовчуванням мають речові літерали без суфікса (наприклад, 3.14)? Які суфікси потрібні для float і decimal? Чому суфікс для decimal особливо важливий у фінансових розрахунках?
21. Що таке керуюча послідовність у символьних літералах? Запишіть символьні літерали для: табуляції, переходу на новий рядок, одинарної лапки, зворотного слішу.
22. У чому різниця між звичайним рядковим літералом і verbatim-рядком (@"...")? Запишіть шлях C:\Users\Doctor\file.txt обома способами.
23. Що таке null? Якому типу змінних можна присвоїти null і чому? Що відбудеться, якщо викликати метод на змінній з null?
24. Поясніть, чому результат 3.2e3 дорівнює 3200. Запишіть число 0.00015 в експоненційній формі.

2.4. Типи даних

25. Що визначає тип даних для змінної? Назвіть три речі, які тип визначає для змінної.
26. Чим відрізняються типи byte і sbyte? Для яких практичних задач підходить byte, а для яких — sbyte?
27. Який цілочисленний тип є найпоширенішим у C# і чому? Яке числове значення за замовчуванням має літерал 42?

28. У чому принципова різниця між float/double і decimal при роботі з дробовими числами? Чому decimal незамінний у фінансових розрахунках?
29. Що таке string і що таке object з точки зору системи типів .NET? Чому object може зберігати «будь-що», але використовується обережно?
30. Що означає, що int є псевдонімом System.Int32? Чи є різниця між int a = 4 і System.Int32 a = 4 для компілятора?
31. Що виведе наступний код?

```C#
var hello = "Hello";
var c     = 20;
var d     = 3.14;
Console.WriteLine(hello.GetType().Name);
Console.WriteLine(c.GetType().Name);
Console.WriteLine(d.GetType().Name);
```

32. Визначте тип, який найкраще підходить для кожної ситуації: вік пацієнта (0–120), точна сума рахунку в гривнях, пульс у ударах за хвилину (0–300), результат аналізу на гемоглобін (дробове значення), ознака госпіталізації (так/ні). Обґрунтуйте вибір.

2.5. Консольне введення-виведення

33. У чому різниця між Console.Write і Console.WriteLine? Наведіть приклад, де важливо використати саме Console.Write.
34. Що таке інтерполяція рядків у C#? Що означає символ $ перед рядком і фігурні дужки {...} всередині нього? Чи можна поміщати в дужки не лише змінні, але й вирази?
35. Порівняйте інтерполяцію рядків ($"...") і складений формат (плейсхолдери {0}, {1}). Яка перевага інтерполяції у сучасному C#?
36. Що повертає Console.ReadLine()? Чому результат має тип string?, а не просто string? Коли метод поверне null?
37. Що відбудеться, якщо користувач введе рядок "сорок п'ять" там, де програма чекає:

```C#
int age = Convert.ToInt32(Console.ReadLine());
```

Як захистити програму від такої ситуації?

38. Що таке культура операційної системи у контексті введення дробових чисел? Чому 1.5 з крапкою може призвести до помилки на машині з українською культурою?
39. Напишіть фрагмент коду, що зчитує з консолі ім'я і вік пацієнта, а потім виводить рядок у форматі: "Пацієнт: Іван Петренко, 45 р.".
40. Поясніть, чому наступний код виводить значення у форматі інтерполяції, але age.ToString() є бажаною практикою при передачі числа в рядок:

```
C#
1  int age = 45;
2  Console.WriteLine($"Вік: {age}");
3  Console.WriteLine($"Вік: {age.ToString()}");
```

2.6. Арифметичні операції

41. Що відбувається при діленні двох цілих чисел у C#? Що виведе наступний код?

```
C#
1  double wrongResult = 10 / 3;
2  double rightResult = 10.0 / 3.0;
3  Console.WriteLine(wrongResult);
4  Console.WriteLine(rightResult);
```

42. Що таке операція % (залишок від ділення)? Як за допомогою % перевірити, чи є число парним?
43. У чому різниця між префіксним (++x) і постфіксним (x++) інкрементом? Що виведе наступний код?

```
C#
1  int x = 5;
2  int z = x++;
3  Console.WriteLine(x);
4  Console.WriteLine(z);
```

44. Що таке пріоритет операцій? Яким буде результат виразу c-- - b * a, де c=40, b=5, a=3? Поясніть крок за кроком.
45. Що таке лівоасоціативність арифметичних операцій? Яким буде результат 10 / 5 * 2 і чому?
46. Як отримати дробовий результат при діленні двох цілочисленних змінних patients / doctors? Запишіть код.

47. Написати код для обчислення ІМТ (індекс маси тіла) за формулою вага / (зріст * зріст), де вага в кг і зріст у метрах. Виведіть результат.
48. Поясніть, чому x++ * 2 і ++x * 2 дають різні результати, якщо початкове значення x = 3. Обчисліть обидва вирази.

2.7. Порозрядні операції

49. Що таке порозрядні операції? На відміну від арифметичних операцій, над чим саме вони виконуються?
50. Поясніть операцію побітового AND (&). Яким буде результат 4 & 5? Покажіть двійкові подання і побітові обчислення.
51. Яка практична задача вирішується за допомогою операції OR (|)? Наведіть приклад встановлення окремого біту у числі.
52. Яка унікальна властивість операції XOR (^) робить її придатною для найпростішого шифрування? Доведіть математично, що (x ^ k) ^ k = x.
53. Що таке додатковий код (two's complement)? Чому ~12 дорівнює -13, а не іншому числу?
54. Що відбувається з числом при зсуві вліво на 1 позицію? А при зсуві вправо? Яким математичним операціям це відповідає?
55. Що виведе наступний код і чому?

```
C#
1  int a = 22;
2  int b = a >> 2;
3  Console.WriteLine(b);
```

56. Порівняйте << з множенням на 2 за ефективністю. Коли варто використовувати зсуви замість арифметичних операцій, а коли — ні?

2.8. Операції присвоєння

57. Що таке операція присвоєння і яким є порядок її виконання відносно інших операцій у виразі?
58. Що означає правоасоціативність операцій присвоєння? Що виведе наступний код?

```
C#
1  int a, b, c;
```

```
2  a = b = c = 100;
3  Console.WriteLine(a);
4  Console.WriteLine(b);
5  Console.WriteLine(c);
```

59. Перепишіть вираз score = score + 10 за допомогою складеного оператора присвоєння. Назвіть п'ять складених арифметичних операторів присвоєння.
60. Що виведе наступний код? Поясніть крок за кроком:

```
C#
1  int a = 8;
2  int b = 6;
3  int c = a += b -= 5;
4  Console.WriteLine(a);
5  Console.WriteLine(b);
6  Console.WriteLine(c);
```

61. Що робить оператор |= з бітовими прапорцями? Запишіть приклад, де ви встановлюєте нульовий біт числа за допомогою |=.
62. Чому складені оператори присвоєння (+=, -= тощо) зменшують ризик помилок порівняно зі звичайним записом a = a + b?
63. Напишіть код, що накопичує суму трьох медичних витрат (консультація, аналізи, ліки) за допомогою +=, а потім застосовує знижку 10% через -=.
64. Чи можливе таке: 3 = x? Чи можливе const double Pi = 3.14; Pi = 3.0;? Поясніть правило щодо лівого операнда присвоєння.

2.9. Перетворення базових типів

65. Чому наступний код не скомпілюється, хоча значення 74 вписується в byte?

```
C#
1  byte a = 4;
2  byte b = a + 70;
```

Як виправити цей код?

66. Що таке розширювальне перетворення (widening)? Чому воно відбувається автоматично і не потребує явного приведення?
67. Що таке звужувальне перетворення (narrowing)? Чому воно вимагає явного приведення? Що відбудеться, якщо значення не вписується в цільовий тип?

68. Що виведе наступний код і чому результат саме такий?

```
C#
1  ushort a = 300;
2  byte b = (byte)a;
3  Console.WriteLine(b);
```

69. Що робить (int) з дробовим числом? Чим це відрізняється від математичного округлення? Що виведе (int)36.9?
70. Поясніть ланцюжки автоматичних безпечних перетворень. Чому перетворення між double і decimal не є автоматичним в жодному напрямку, попри те що decimal «більший»?
71. Напишіть код, що обчислює частку чоловіків серед пацієнтів у відсотках. Використайте явне приведення для отримання дробового результату ділення.
72. Що таке sign extension (розширення зі знаком)? Чому значення sbyte -4 після перетворення в short залишається -4, а не стає великим позитивним числом?

2.10. Явні та неявні перетворення

73. У чому різниця між явним і неявним перетворенням типів? Хто відповідає за виконання кожного з них?
74. Коли компілятор дозволяє неявне перетворення? Яка гарантія повинна виконуватися, щоб перетворення вважалося безпечним?
75. Що таке ключове слово checked? Що відбудеться у наступному коді без checked і з ним?

```
C#
1  int a = 33;
2  int b = 600;
3  byte c = (byte)(a + b);
```

76. Що таке unchecked і навіщо воно потрібне, якщо це «стандартна поведінка»?
77. Що виведе наступний код і чому?

```
C#
1  int a = 33;
2  int b = 600;
3  byte c = (byte)(a + b);
4  Console.WriteLine(c);
```

Порахуйте самостійно, чому результат 121.

78. Поясніть, чому отримане в результаті переповнення неправильне значення особливо небезпечне порівняно зі звичайним винятком. У яких медичних або фінансових сценаріях це критично?
79. Чи є decimal більшим за double у сенсі точності та кількості значущих цифр? Якщо так, чому не можна неявно конвертувати double у decimal?
80. Спроєктуйте функцію CalculateDose(double weightKg, double dosePerKg), що повертає результат як decimal. Використайте явне перетворення і поясніть, де воно відбувається.

2.11. Умовні вирази

81. Що повертають оператори порівняння (==, !=, <, >, <=, >=)? Якому типу належить результат?
82. У чому різниця між && і & для логічних значень? Що таке скорочене обчислення (short-circuit evaluation) і чому воно підвищує продуктивність?
83. Поясніть оператор XOR (^) для логічних значень. Коли він повертає true?
84. Запишіть умову, що перевіряє: «пацієнт є дорослим (вік >= 18) І має підвищений тиск (систолічний >= 140) АБО є критичний стан (тиск >= 180)». Використовуйте дужки для ясності.
85. Що виведе наступний код?

```
C#
1  int age = 45;
2  int bpSystolic = 155;
3  bool highRisk = bpSystolic >= 160 || (age > 40 && bpSystolic >= 140);
4  Console.WriteLine(highRisk);
```

86. Чому if (isEmergency == true) є надлишковим записом? Як правильно записати цю умову?
87. Яку роль відіграє оператор ! (NOT)? Запишіть еквівалентний вираз для !(age >= 18) без використання !.
88. Розставте пріоритети операцій у наступному виразі і вкажіть результат (всі змінні мають конкретні значення):

```
C#
1  bool result = !hasFever || age > 60 && bpSystolic < 180;
```

```
2  // hasFever = false, age = 65, bpSystolic = 150
```

2.12. Конструкція if..else та тернарна операція

89. Що відбувається, якщо умова блоку if є false і немає блоку else? Яка частина коду виконується?
90. Як працює ланцюжок else if? Чи перевіряються всі умови підряд, чи виконання зупиняється після першого збігу?
91. Що виведе наступний код?

```
C#
1  int bpSystolic = 145;
2  if (bpSystolic < 120)      Console.WriteLine("Оптимальний");
3  else if (bpSystolic < 130) Console.WriteLine("Нормальний");
4  else if (bpSystolic < 140) Console.WriteLine("Підвищено-нормальний");
5  else if (bpSystolic < 160) Console.WriteLine("Гіпертонія I");
6  else                        Console.WriteLine("Гіпертонія II+");
```

92. Чому рекомендується завжди писати фігурні дужки навіть для однієї інструкції в тілі if? Наведіть приклад помилки, яка виникає через відсутність дужок.
93. Що таке тернарна операція? Перетворіть наступну конструкцію у тернарну:

```
C#
1  string status;
2  if (bpSystolic >= 140) status = "Підвищений";
3  else status = "Нормальний";
```

94. Що таке вкладені if? Напишіть код, що спочатку перевіряє вік (>= 60 — похилий), а потім всередині — тиск (>= 160 — критичний), і виводить відповідне повідомлення.
95. Коли доречно використовувати тернарний оператор, а коли краще залишити повну форму if...else? Наведіть приклад обох ситуацій.
96. Спроєктуйте ланцюжок if...else if...else, що класифікує ІМТ пацієнта: нижче 18.5 — дефіцит ваги, 18.5–24.9 — норма, 25.0–29.9 — надлишкова вага, 30 і вище — ожиріння.

2.13. Цикли

97. Назвіть чотири типи циклів у C#. Для якої ситуації кожен з них найкраще підходить?

98. З яких трьох частин складається заголовок циклу for? Коли виконується кожна частина?
99. Що виведе наступний код?

```
C#
1  for (int i = 5; i >= 1; i--)
2      Console.Write($"{i} ");
```

100. У чому ключова відмінність між while і do...while? Наведіть приклад ситуації, де do...while є єдиним коректним вибором.
101. Що таке break і continue? Поясніть їхній вплив на виконання циклу. Чим вони відрізняються один від одного?
102. Чому у циклі foreach не можна змінювати елементи колекції через змінну циклу? Як слід вчинити, якщо потрібна модифікація елементів?
103. Напишіть код з вкладеними циклами, що виводить таблицю множення від 1×1 до 3×3.
104. Що таке нескінченний цикл? Запишіть нескінченний цикл for і поясніть, як коректно вийти з нього за допомогою break.

2.14. Масиви

105. Що таке масив? Що означає «фіксований розмір» і «однотипні елементи» у визначенні масиву?
106. З якого індексу починається нумерація елементів масиву? Чому це важливо враховувати при звертанні до останнього елемента?
107. Що відбудеться під час виконання наступного коду?

```
C#
1  int[] values = { 1, 2, 3 };
2  Console.WriteLine(values[10]);
```

108. Які чотири способи ініціалізації масиву з конкретними значеннями допускаються в C#? Наведіть усі чотири варіанти для масиву {45, 32, 67, 28}.
109. Що таке оператор ^ (індекс від кінця)? Напишіть код, що виводить останній та передостанній елемент масиву, використовуючи ^.
110. У чому різниця між прямокутним двовимірним масивом [,] і зубчастим масивом [][]? Коли доречний кожен варіант?

111. Напишіть код, що знаходить мінімальне і максимальне значення в масиві цілих чисел, використовуючи цикл for.
112. Що таке властивість Length для одновимірного масиву і що повертає Length для двовимірного? Як отримати кількість рядків і стовпців двовимірного масиву?

2.15. Завдання з масивами

113. Опишіть алгоритм підрахунку кількості елементів масиву, що задовольняють певну умову. Яку структуру даних використовують для накопичення кількості?
114. Напишіть код, що підраховує кількість значень тиску >= 140 у масиві {120, 155, 130, 172, 118, 145}.
115. Опишіть алгоритм реверсу масиву без виділення нового масиву. Чому потрібна тимчасова змінна temp для обміну елементів?
116. Що виведе наступний код після реверсу масиву {1, 2, 3, 4, 5}? Скільки ітерацій виконає цикл і чому саме стільки?
117. Поясніть принцип роботи бульбашкового сортування. Що означає «спливання» більшого елемента?
118. Чому зовнішній цикл бульбашкового сортування виконується n-1 разів, а не n?
119. Порівняйте ручну реалізацію сортування та Array.Sort(). Коли в реальному коді краще використовувати готовий метод і чому?
120. Напишіть код, що знаходить середнє значення масиву показників температури {36.6, 37.2, 38.5, 37.0, 36.8}.

2.16. Методи

121. Що таке метод і яку задачу він вирішує? Чому важливо уникати дублювання коду?
122. Які чотири складові мають усі методи? Що означає тип void для методу?
123. У чому різниця між визначенням методу і його викликом? Чи має значення порядок: визначення до або після виклику?
124. Що таке скорочений запис методу (expression-bodied member) через =>? Коли він застосовується і яке обмеження має?
125. Що виведе наступний код?

```C#
void PrintSeparator() => Console.WriteLine("---");
void PrintHeader()
{
    PrintSeparator();
    Console.WriteLine("Журнал");
    PrintSeparator();
}
PrintHeader();
```

126. Поясніть принцип «кожен метод відповідає лише за одну річ». Чому цей принцип покращує читабельність і підтримку коду?
127. Напишіть три методи: PrintCriticalAlert, PrintWarningAlert, PrintNormalStatus, і викличте потрібний залежно від значення bpSystolic.
128. Запропонуйте, як розбити наступний складний блок коду на кілька методів: зчитування даних пацієнта, перевірка коректності даних, виведення картки пацієнта.

2.17. Параметри методів

129. У чому різниця між формальними параметрами і фактичними параметрами (аргументами)?
130. Чи можна передавати аргументи у іншому порядку, ніж оголошені параметри? Що для цього потрібно?
131. Що таке необов'язкові параметри зі значенням за замовчуванням? Де в списку параметрів вони повинні знаходитися?
132. Що виведе наступний код?

```C#
void RegisterVisit(string name, string ward = "Терапія", bool urgent = false)
{
    string tag = urgent ? " [ТЕРМІНОВО]" : "";
    Console.WriteLine($"{name} → {ward}{tag}");
}
RegisterVisit("Іван", "Кардіологія", true);
RegisterVisit("Марія", "Неврологія");
RegisterVisit("Олег");
```

133. Що таке іменовані параметри? Запишіть виклик методу RegisterPatient(string name, int age, string ward), де ward передається за іменем, а name і age — за позицією.
134. Яке обмеження існує щодо порядку іменованих і позиційних аргументів в одному виклику?

135. Що станеться, якщо передати аргумент типу string у параметр типу int? На якому етапі виявляється ця помилка?
136. Спроєктуйте метод PrintPatientCard, що приймає ім'я, вік та необов'язкові параметри: відділення (за замовчуванням «Загальне») і VIP-статус (за замовчуванням false).

2.18. Повернення значення та оператор return

137. У чому принципова різниця між методом типу void і методом з типом результату string або int?
138. Що відбувається після виконання оператора return? Чи виконається код, що стоїть після return у тому самому блоці?
139. Що таке «недосяжний код» (unreachable code)? Як компілятор реагує на нього?
140. Що таке Guard Clause (охоронна умова)? Які переваги цього патерну порівняно зі вкладеними умовами?
141. Що виведе наступний код?

```
C#
1  string GetDiagnosis(string symptom)
2  {
3      if (symptom == "кашель") return "Бронхіт";
4      if (symptom == "температура") return "ГРВІ";
5      return "Необхідне обстеження";
6  }
7  Console.WriteLine(GetDiagnosis("кашель"));
8  Console.WriteLine(GetDiagnosis("головний біль"));
```

142. Як результат одного методу можна передати безпосередньо як аргумент іншому методу без проміжної змінної? Запишіть приклад.
143. Напишіть метод CalculateBMI(double weight, double height), що повертає значення ІМТ, і метод IsNormalBMI(double bmi), що повертає true, якщо ІМТ між 18.5 і 24.9.
144. Поясніть, коли return; (без значення) використовується у void-методах. Напишіть приклад методу з двома охоронними умовами на початку.

2.19. Передача параметрів за посиланням та значенням. Вихідні параметри

145. Що відбувається з оригінальною змінною при передачі аргументу за значенням (без модифікаторів)? Що відбувається при передачі за посиланням через ref?

146. Що виведе наступний код?

```C#
void Increment(int x) { x += 10; }
int a = 5;
Increment(a);
Console.WriteLine(a);
```

147. Що виведе наступний код?

```C#
void Increment(ref int x) { x += 10; }
int a = 5;
Increment(ref a);
Console.WriteLine(a);
```

148. У чому різниця між ref і out? Яка вимога до ініціалізації змінної для кожного з них?

149. Навіщо потрібні out-параметри, якщо є оператор return? У чому їхня перевага? Наведіть приклад методу, що повертає два значення через out.

150. Що таке inline-оголошення out-змінних? Запишіть виклик методу з використанням out var.

151. Для чого призначений модифікатор in? Чим він схожий і чим відрізняється від ref?

152. Спроєктуйте метод CalculateVitalSigns(int age, out int maxHeartRate, out int targetHeartRate), що обчислює максимальний пульс (220 - вік) і цільовий (75% від максимального). Запишіть визначення і виклик методу.

2.20. Масив параметрів та ключове слово params

153. Що таке params? Яку задачу воно вирішує, що не вирішують звичайні параметри?

154. Що відбувається всередині методу з params-параметром? Чи є він звичайним масивом?

155. Які три способи можна використати для передачі значень у params-параметр? Що відбудеться, якщо не передати жодного аргументу?

156. Де в списку параметрів повинен знаходитися params-параметр? Що буде, якщо порушити це правило?
157. У чому різниця між методом з params double[] values і методом зі звичайним double[] values? Чи однаковий виклик у обох?
158. Що виведе наступний код?

```
C#
1  void Log(params double[] values)
2  {
3      Console.WriteLine($"Отримано: {values.Length}");
4  }
5  Log(36.6);
6  Log(120.0, 80.0);
7  Log();
```

159. Напишіть метод LogMeasurements(string label, params double[] values), що виводить назву і середнє значення переданих показників. Переконайтеся, що порожній масив обробляється коректно.
160. Порівняйте params і звичайний масив-параметр: коли кожен з них зручніший у публічному API?

2.21. Рекурсивні функції

161. Що таке рекурсія? Що відрізняє рекурсивну функцію від звичайної?
162. Які два обов'язкових елементи повинна містити кожна коректна рекурсивна функція? Що станеться, якщо один з них відсутній?
163. Поясніть покроково, як виконується Factorial(4). Скільки рекурсивних викликів відбудеться і в якому порядку?
164. Що таке стек викликів? Що трапляється з ним у разі нескінченної рекурсії?
165. Яке обмеження має наївна рекурсивна реалізація Фібоначчі? Чому Fibonacci(30) виконується повільно?
166. Перепишіть факторіал через цикл for. Порівняйте читабельність рекурсивного та ітеративного варіантів.
167. Для яких класів задач рекурсія є природним і виразним рішенням? Назвіть щонайменше три і поясніть чому.
168. Що виведе наступний код і скільки разів буде викликана функція f?

```C#
int f(int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;
    return f(n - 1) + f(n - 2);
}
Console.WriteLine(f(5));
```

2.22. Локальні функції

169. Що таке локальна функція і чим вона відрізняється від звичайного приватного методу класу?
170. Де у коді може бути оголошена локальна функція? Чи має значення порядок: визначення до або після місця використання?
171. Яку задачу вирішують локальні функції? Наведіть приклад, де без локальної функції виникає дублювання коду.
172. Що таке доступ до змінних зовнішнього методу (захоплення)? Чому це може бути небезпечно?
173. Що таке static локальна функція? Яку гарантію вона надає і для чого це корисно?
174. У чому перевага локальної функції над виносом коду у приватний метод класу?
175. Перепишіть наступний код з дублюванням, використовуючи локальну функцію:

```C#
void AnalyzeTest(double hgb, double glc)
{
    string hgbStatus = hgb < 120 ? "Низький" : hgb > 170 ? "Високий" : "Норма";
    string glcStatus = glc < 3.9 ? "Низький" : glc > 6.1 ? "Високий" : "Норма";
    Console.WriteLine($"Гемоглобін: {hgbStatus}");
    Console.WriteLine($"Глюкоза: {glcStatus}");
}
```

176. Коли варто оголошувати локальну функцію зі словом static? Що зміниться, якщо спробувати звернутися до змінної зовнішнього методу всередині static-локальної функції?

2.23. Конструкція switch

177. У яких ситуаціях конструкція switch є кращою альтернативою ланцюжку if...else if? Що вона повідомляє читачеві коду?

178. Навіщо потрібен оператор break наприкінці кожного блоку case? Що станеться, якщо його відсутній у конструкції switch — але є return?

179. Що виведе наступний код?

```C#
string diagnosis = "Bronchitis";
switch (diagnosis)
{
    case "Pneumonia":
    case "Bronchitis":
    case "Pleuritis":
        Console.WriteLine("Пульмонологія");
        break;
    default:
        Console.WriteLine("Терапія");
        break;
}
```

180. Що таке блок default у конструкції switch? Чи є він обов'язковим і чому рекомендується його завжди додавати?

181. Для чого використовується goto case? Напишіть приклад, де goto case логічно виправданий.

182. Що таке switch-вираз (switch expression)? Чим він відрізняється від switch-оператора синтаксично і концептуально?

183. Що виведе наступний код?

```C#
string GetWard(string code) => code switch
{
    "ICU"  => "Інтенсивна терапія",
    "CARD" => "Кардіологія",
    _      => "Загальне"
};
Console.WriteLine(GetWard("CARD"));
Console.WriteLine(GetWard("ORTHO"));
```

184. Що таке «вичерпність» switch-виразу? Що станеться, якщо відсутня гілка _ і вхідне значення не збігається з жодним case?

2.24. Перерахування enum

185. Що таке enum? Яку задачу він вирішує порівняно з використанням «магічних чисел» або рядкових констант?

186. Яке числове значення за замовчуванням отримує перша константа enum? Яке значення отримує наступна?

187. Що виведе наступний код?

```C#
enum PatientStatus { Registered, Admitted, UnderTreatment, Discharged, Critical }
PatientStatus status = PatientStatus.Admitted;
Console.WriteLine(status.ToString());
Console.WriteLine((int)status);
```

188. Чому використання enum безпечніше, ніж рядкові константи типу "admitted" для представлення станів системи? Наведіть конкретний приклад помилки, яку enum запобігає.

189. Які цілі типи можуть використовуватися як основа enum? Для яких сценаріїв варто обирати byte замість int?

190. Що відбудеться, якщо привести до enum числове значення, яке не відповідає жодній константі? Як захиститися від цього при роботі з ненадійними зовнішніми даними?

191. Напишіть enum WardType зі значеннями ICU, Surgery, Cardiology, Neurology, General і метод GetWardDescription(WardType ward), що повертає текстовий опис через switch-вираз.

192. Для enum PriorityLevel { Low = 1, Medium = 2, High = 5, Critical = 10 } запишіть умову, що перевіряє: «пріоритет є принаймні High». Поясніть, чому явні числові значення у цьому enum є осмисленим вибором.

Розділ 3. Класи, структури та простір імен

3.1. Класи та об'єкти

Проблема без класів

Розглянемо типову задачу: зберегти інформацію про пацієнтів клініки та вивести її на екран. На перший погляд здається, що достатньо кількох змінних — ім'я, телефон, вік. Але що відбувається, коли пацієнтів стає більше одного?

```
C#
 1  using System;
 2
 3  // Два пацієнти — вже шість змінних
 4  string patient1Name  = "Олена Коваль";
 5  string patient1Phone = "0501234567";
 6  int    patient1Age   = 32;
 7
 8  string patient2Name  = "Михайло Бондар";
 9  string patient2Phone = "0671234567";
10  int    patient2Age   = 45;
11
12  Console.WriteLine($"{patient1Name}, тел: {patient1Phone}, вік: {patient1Age}");
13  Console.WriteLine($"{patient2Name}, тел: {patient2Phone}, вік: {patient2Age}");
```

Уже з двома пацієнтами — шість змінних. Із десятьма — шістдесят. Їх легко переплутати: ніщо не заважає випадково записати вік у змінну телефону. Передати набір пов'язаних змінних у метод — окремий виклик, адже доведеться перераховувати кожну з них як окремий параметр. Зберегти всіх пацієнтів у масив — взагалі неможливо, бо масив містить лише один тип.

Саме тут у гру вступають **класи** — механізм, що дозволяє визначати власні типи даних, об'єднуючи пов'язані поля й методи роботи з ними в єдину структуру.

Клас і об'єкт

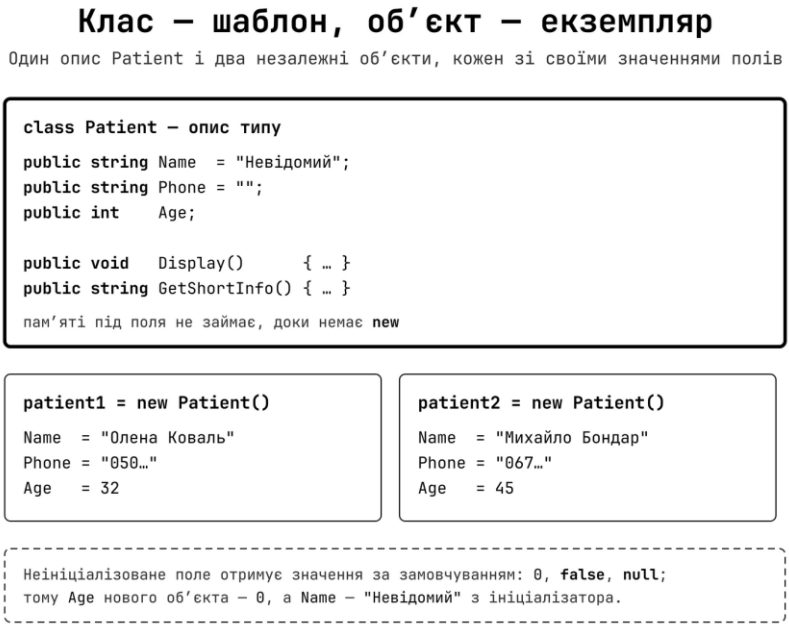
В основі об'єктно-орієнтованого програмування лежать два нерозривно пов'язані поняття.

Клас — це опис нового типу даних: він визначає, які поля (дані) та методи (поведінка) матиме кожен представник цього типу. Клас — це лише шаблон, він не займає пам'яті під конкретні дані.

Об'єкт — конкретний екземпляр класу, створений у пам'яті під час виконання програми. Кожен об'єкт зберігає власні значення полів, незалежні від інших об'єктів того самого класу.

Аналогія з повсякденного життя: клас — це бланк анкети пацієнта, однаковий для всіх. Заповнена анкета конкретної людини — це об'єкт. З одного бланку можна заповнити тисячу анкет, і кожна міститиме свої унікальні дані.

Важливо також усвідомити: у системі типів C# клас — це повноцінний тип, нарівні з int, string чи bool. Після оголошення class Patient компілятор знає тип Patient і дозволяє використовувати його скрізь, де дозволено будь-який інший тип: як тип змінної, параметра методу, елемента масиву, повернення функції.



Клас Patient як шаблон та два об'єкти-екземпляри

Оголошення класу

Клас оголошується ключовим словом class, за яким слідує ім'я та тіло у фігурних дужках:

```C#
class Patient
{
    // поля та методи
}
```

Ім'я класу записується у стилі **PascalCase** — кожне слово з великої літери: Patient, DoctorManager, AppointmentStatus. Це загальноприйнята конвенція для типів у C#.

Усередині фігурних дужок розміщується вміст класу — поля, методи, константи та інші члени.

Поля

Поля — це змінні, оголошені безпосередньо в тілі класу. Вони зберігають **стан** кожного конкретного об'єкта і доступні з будь-якого методу цього класу без передачі через параметри.

```C#
class Patient
{
    public string Name  = "Невідомий";
    public string Phone = "";
    public int    Age;
}
```

Модифікатор public означає, що поле доступне ззовні класу — з будь-якого місця програми, де є посилання на об'єкт. Це найпростіший підхід, достатній для початку; у пізніших розділах ми побачимо, чому в реальних проєктах поля роблять private і захищають їх через властивості.

Якщо поле не ініціалізоване явно, воно отримує значення за замовчуванням відповідно до свого типу. Це не випадкове «сміття» з пам'яті, а гарантована поведінка середовища виконання .NET:

```C#
using System;

Patient p = new Patient();
Console.WriteLine($"Name:  '{p.Name}'");   // рядок: null
Console.WriteLine($"Phone: '{p.Phone}'");  // рядок: ""  (ми задали "")
Console.WriteLine($"Age:   {p.Age}");      // int: 0
Console.WriteLine($"Flag:  {p.IsActive}"); // bool: false

class Patient
{
    public string Name;        // null — рядки за замовч. null
    public string Phone = "";  // "" — ми задали явно
    public int    Age;         // 0
    public bool   IsActive;    // false
```

```
15  }
```

> **Зверніть увагу:** прямий публічний доступ до полів (patient.Name = "...") — це найпростіший підхід для навчального прикладу. У реальних проєктах поля роблять private і надають доступ через властивості з валідацією. Ми прийдемо до цього в розділі про інкапсуляцію.

Методи

Методи визначають **поведінку** об'єкта — що він вміє робити. На відміну від звичайних функцій, методи класу мають прямий доступ до всіх його полів без передачі через параметри: компілятор неявно передає посилання на поточний об'єкт у кожен виклик методу.

```
C#
 1  using System;
 2
 3  Patient p = new Patient();
 4  p.Name  = "Олена Коваль";
 5  p.Phone = "0501234567";
 6  p.Age   = 32;
 7
 8  p.Display();
 9  Console.WriteLine(p.GetShortInfo());
10  Console.WriteLine("Повнолітній: " + p.IsAdult());
11
12  class Patient
13  {
14      public string Name  = "Невідомий";
15      public string Phone = "";
16      public int    Age;
17
18      // метод без повернення значення
19      public void Display()
20      {
21          Console.WriteLine($"Пацієнт: {Name}, тел: {Phone}, вік: {Age}");
22      }
23
24      // метод, що повертає рядок
25      public string GetShortInfo()
26      {
27          return $"{Name} ({Age} р.)";
28      }
29
30      // метод із логікою
31      public bool IsAdult()
32      {
33          return Age >= 18;
34      }
35  }
```

Усередині методу поля Name, Phone, Age доступні напряму — без будь-якого префікса. Явно вказати, що йдеться про поле саме цього об'єкта, а не про локальну змінну, можна через ключове слово this:

```
C#
1  public void Display()
2  {
3      // this.Name та Name — одне й те саме,
4      // але this корисний коли параметр має те ж ім'я що й поле
5      Console.WriteLine($"Пацієнт: {this.Name}, вік: {this.Age}");
6  }
```

Створення об'єкта

Оголошення класу — це лише опис. Щоб отримати реальний об'єкт у пам'яті, використовується оператор new:

```
C#
1  Patient patient1 = new Patient();
```

У цьому рядку відбуваються три речі:

1. new Patient() виділяє блок пам'яті в **купі** (heap) та ініціалізує всі поля значеннями за замовчуванням або початковими значеннями.
2. Викликається **конструктор** — спеціальний метод ініціалізації (наразі автоматичний, без параметрів).
3. Змінна patient1 отримує **посилання** — адресу розміщення об'єкта в пам'яті.

Починаючи з C# 9, якщо тип змінної вже вказаний зліва, його можна не повторювати в new:

```
C#
1  Patient patient1 = new();  // рівнозначно new Patient()
```

Точкова нотація

Доступ до полів і методів об'єкта здійснюється через **крапку** (dot notation): спочатку ім'я змінної, потім крапка, потім ім'я члена.

```
C#
 1  using System;
 2
 3  // Створюємо об'єкт і звертаємось до полів
 4  Patient patient1 = new Patient();
 5  Console.WriteLine(patient1.Name);           // Невідомий (значення за замовч.)
 6  Console.WriteLine($"Вік: {patient1.Age}");  // 0
 7
```

```
// Записуємо нові значення
patient1.Name  = "Олена Коваль";
patient1.Phone = "0501234567";
patient1.Age   = 32;

// Викликаємо методи
patient1.Display();
Console.WriteLine(patient1.GetShortInfo());

class Patient
{
    public string Name  = "Невідомий";
    public string Phone = "";
    public int    Age;

    public void Display()
    {
        Console.WriteLine($"Пацієнт: {Name}, тел: {Phone}, вік: {Age}");
    }

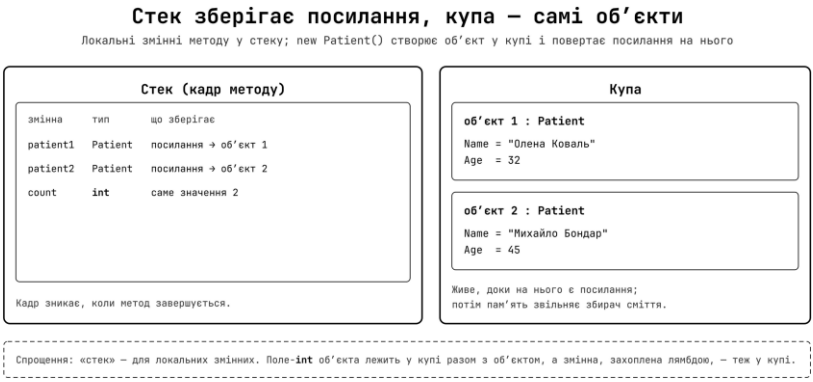
    public string GetShortInfo()
    {
        return $"{Name} ({Age} р.)";
    }
}
```

Об'єкти у пам'яті

Щоб правильно розуміти поведінку об'єктів, важливо мати уявлення про те, де і як вони зберігаються.

Стек (stack) — швидка область пам'яті для локальних змінних і параметрів методів. Стек працює за принципом LIFO: коли метод завершується, всі його локальні змінні автоматично зникають. Змінна-об'єкт (Patient patient1) живе саме тут, але вона зберігає не самі дані, а **посилання** — адресу у пам'яті, де реально знаходиться об'єкт.

Купа (heap) — велика область пам'яті для довгоживучих об'єктів. Оператор new завжди виділяє пам'ять у купі. Об'єкти живуть у купі до тих пір, поки на них є хоча б одне посилання; коли всі посилання зникають, збирач сміття (Garbage Collector) автоматично звільняє пам'ять.



Стек зберігає посилання, купа — самі об'єкти

Це пояснює два важливі факти:

— Об'єкт є доступним з будь-якого місця програми — достатньо мати посилання на нього.

— Локальна змінна може зникнути, але об'єкт у купі залишиться, якщо на нього ще посилаються.

Кілька об'єктів

Кожен об'єкт у купі незалежний: він зберігає власний набір значень полів. Зміна полів одного об'єкта жодним чином не впливає на інший.

```C#
using System;

Patient patient1 = new Patient();
patient1.Name = "Олена Коваль";
patient1.Age  = 32;

Patient patient2 = new Patient();
patient2.Name = "Михайло Бондар";
patient2.Age  = 45;

patient1.Display();
patient2.Display();

// Змінюємо вік patient1 — patient2 залишається незмінним
patient1.Age = 33;
Console.WriteLine("Після зміни patient1.Age:");
patient1.Display();
patient2.Display(); // вік 45 — не змінився

class Patient
{
```

```
    public string Name  = "Невідомий";
    public string Phone = "";
    public int    Age;

    public void Display()
    {
        Console.WriteLine($"Пацієнт: {Name}, тел: {Phone}, вік: {Age}");
    }
}
```

Дві змінні — один об'єкт

Оскільки змінна зберігає **посилання**, а не сам об'єкт, можливо присвоїти це посилання іншій змінній. У результаті обидві змінні вказуватимуть на **той самий об'єкт** у купі — і зміна через одну змінну буде видна через іншу.

```C#
using System;

Patient a = new Patient();
a.Name = "Олена Коваль";
a.Age  = 32;

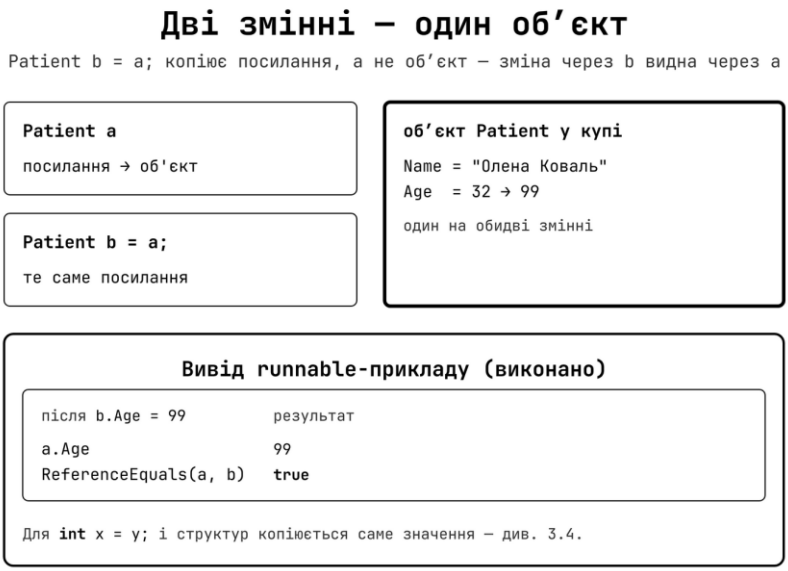
// b отримує не копію об'єкта, а те саме посилання
Patient b = a;

Console.WriteLine("Перед зміною:");
Console.WriteLine($"a.Age = {a.Age}");  // 32
Console.WriteLine($"b.Age = {b.Age}");  // 32 — той самий об'єкт

// Змінюємо через b
b.Age = 99;

Console.WriteLine("Після b.Age = 99:");
Console.WriteLine($"a.Age = {a.Age}");  // 99 — змінилось!
Console.WriteLine($"b.Age = {b.Age}");  // 99

class Patient
{
    public string Name = "Невідомий";
    public int    Age;
}
```



Дві змінні — одне посилання на один об'єкт

Це фундаментальна відмінність між **посилальними типами** (class) і **типами-значеннями** (int, bool, struct). При присвоєнні int b = a створюється незалежна копія числа. При присвоєнні Patient b = a копіюється лише посилання, сам об'єкт залишається одним.

null — відсутнє посилання

Змінна посилального типу, яка не вказує на жоден об'єкт, містить спеціальне значення null. Це не нуль, не порожній рядок і не пустий об'єкт — це явна відсутність посилання.

```C#
using System;

Patient p = null;  // посилання ні на що не вказує

// Спроба звернутися до поля через null-посилання — виняток
if (p != null)
{
    p.Display();
}
else
{
```

```
    Console.WriteLine("p не вказує на жоден об'єкт");
}

// Ще коротший запис через ?. (null-умовний оператор)
p?.Display();  // нічого не робить, якщо p == null
Console.WriteLine("Програма завершилась без помилки");

class Patient
{
    public string Name  = "Невідомий";
    public string Phone = "";
    public int    Age;

    public void Display()
    {
        Console.WriteLine($"Пацієнт: {Name}, тел: {Phone}, вік: {Age}");
    }
}
```

Спроба викликати метод або звернутися до поля через null-посилання призводить до NullReferenceException — одного з найпоширеніших винятків у програмуванні. Саме тому перевірка на null перед використанням — обов'язкова звичка.

Масив об'єктів

Оскільки Patient — це повноцінний тип, об'єкти можна зберігати у масиві так само, як числа чи рядки. Масив Patient[] зберігає **посилання** на об'єкти, тому кожен елемент спочатку дорівнює null і потребує ініціалізації.

C#
```
using System;

Patient[] patients = new Patient[3];

patients[0] = new Patient();
patients[0].Name  = "Олена Коваль";
patients[0].Phone = "0501234567";
patients[0].Age   = 32;

patients[1] = new Patient();
patients[1].Name  = "Михайло Бондар";
patients[1].Phone = "0671234567";
patients[1].Age   = 45;

patients[2] = new Patient();
patients[2].Name  = "Ірина Мельник";
patients[2].Phone = "0931234567";
patients[2].Age   = 28;

// Виводимо всіх пацієнтів у циклі
foreach (Patient p in patients)
{
    p.Display();
```

```
}

Console.WriteLine($"Всього пацієнтів: {patients.Length}");

class Patient
{
    public string Name  = "Невідомий";
    public string Phone = "";
    public int    Age;

    public void Display()
    {
        Console.WriteLine($"Пацієнт: {Name}, тел: {Phone}, вік: {Age}");
    }

    public string GetShortInfo()
    {
        return $"{Name} ({Age} р.)";
    }
}
```

Саме цю структуру — клас і масив об'єктів — ми будемо розвивати впродовж усього курсу: додаватимемо методи пошуку, сортування, фільтрації, поступово ускладнюючи систему в міру вивчення нових інструментів мови.

Константи класу

Константи зберігають значення, що не змінюється і логічно належить **всьому класу**, а не конкретному об'єкту. Оголошуються з ключовим словом const, значення присвоюється лише один раз — при оголошенні.

C#
```
using System;

// Звертаємось до константи через ім'я класу, не через об'єкт
Console.WriteLine($"Макс. довжина імені: {Patient.MaxNameLength}");
Console.WriteLine($"Телефон за замовч.:  {Patient.DefaultPhone}");

Patient p = new Patient();
p.Display();

// Patient.MaxNameLength = 50; // помилка компіляції: змінити константу не можна

class Patient
{
    // Константи — через ім'я класу, не через об'єкт
    public const int    MaxNameLength = 100;
    public const string DefaultPhone  = "не вказано";

    public string Name  = "Невідомий";
    public string Phone = DefaultPhone;  // використовуємо константу всередині класу
    public int    Age;
```

```
21
22      public void Display()
23      {
24          Console.WriteLine($"Пацієнт: {Name}, тел: {Phone}, вік: {Age}");
25      }
26  }
```

Звернення до константи здійснюється через ім'я **класу**: Patient.MaxNameLength — а не через екземпляр. Це логічно: константа не залежить від конкретного об'єкта, вона є частиною самого опису типу.

Клас в окремому файлі

У реальних проєктах кожен клас розміщується у власному файлі, ім'я якого збігається з ім'ям класу: Patient.cs. Це стандартна конвенція, що спрощує навігацію у великих кодових базах.

Patient.cs

```
C#
 1  class Patient
 2  {
 3      public string Name  = "Невідомий";
 4      public string Phone = "";
 5      public int    Age;
 6
 7      public void Display()
 8      {
 9          Console.WriteLine($"Пацієнт: {Name}, тел: {Phone}, вік: {Age}");
10      }
11
12      public string GetShortInfo()
13      {
14          return $"{Name} ({Age} р.)";
15      }
16  }
```

Program.cs

```
C#
1  Patient p = new Patient();
2  p.Name  = "Олена Коваль";
3  p.Phone = "0501234567";
4  p.Age   = 32;
5  p.Display();
```

Обидва файли належать одному проєкту — компілятор C# автоматично бачить усі класи в усіх файлах проєкту без жодного import, include чи require. Достатньо, щоб файли знаходились у тій самій директорії проєкту або підключеній до нього.

Підсумок

Поняття	Що це
class	Шаблон нового типу з полями і методами
Поле	Змінна на рівні класу, зберігає стан об'єкта
Метод	Функція, що визначає поведінку; має доступ до полів
new	Виділяє пам'ять у heap, повертає посилання
public	Поле або метод доступні ззовні класу
this	Явне посилання на поточний об'єкт усередині методу
const	Значення належить класу, незмінне; доступ через ім'я класу
null	Посилання, що не вказує на жоден об'єкт
Stack	Зберігає локальні змінні і посилання
Heap	Зберігає самі об'єкти; пам'ять звільняє GC

У наступному параграфі ми розглянемо **конструктори** — спеціальні методи, що дозволяють задавати початкові значення полів безпосередньо в момент створення об'єкта через new, а не після.

3.2. Конструктори, ініціалізатори та деконструктори

Створення конструкторів

У попередньому розділі об'єкти створювались через конструктор за замовчуванням, який C# генерує автоматично. Проте у реальній розробці майже завжди виникає потреба визначити власний конструктор — щоб об'єкт одразу отримував коректні початкові

значення і не міг існувати в «порожньому» стані. Уявіть картку пацієнта у клінічній системі: безглуздо мати об'єкт Patient, у якого ні імені, ні віку — такий стан є семантично некоректним. Конструктор вирішує цю проблему, гарантуючи, що кожен новий об'єкт одразу містить усі необхідні дані.

На рівні коду конструктор — це метод, який має те саме ім'я, що й клас, може приймати параметри, але **не має типу, що повертається** (навіть void). Варто пам'ятати: щойно ви визначаєте в класі хоча б один власний конструктор, компілятор більше не генерує конструктор за замовчуванням автоматично.

Визначимо у класі Patient найпростіший конструктор без параметрів, який встановлює початкові значення:

```C#
using System;

Patient patient = new Patient();
patient.Print(); // Ім'я: Невідомий Вік: 0

class Patient
{
    public string name;
    public int age;

    public Patient()
    {
        Console.WriteLine("Реєстрація нового пацієнта");
        name = "Невідомий";
        age = 0;
    }

    public void Print()
    {
        Console.WriteLine($"Ім'я: {name} Вік: {age}");
    }
}
```

Конструктори можуть мати модифікатори доступу. У прикладі вище конструктор оголошено з public — це дозволяє створювати об'єкти Patient з будь-якого місця програми. Якби конструктор був private, об'єкт можна було б створити лише всередині самого класу (такий прийом використовується, наприклад, у патерні Singleton).

Перевантаження конструкторів

Клас може мати кілька конструкторів — це називається **перевантаженням** (overloading). Компілятор розрізняє їх за кількістю та типами параметрів. Це зручно: залежно від того, яка інформація

відома в момент реєстрації пацієнта, можна викликати відповідний варіант конструктора.

```C#
using System;

Patient p1 = new Patient();
Patient p2 = new Patient("Олена Ковальчук");
Patient p3 = new Patient("Іван Петренко", 45);

p1.Print(); // Ім'я: Невідомий Вік: 0
p2.Print(); // Ім'я: Олена Ковальчук Вік: 0
p3.Print(); // Ім'я: Іван Петренко Вік: 45

class Patient
{
    public string name;
    public int age;

    public Patient() { name = "Невідомий"; age = 0; }
    public Patient(string n) { name = n; age = 0; }
    public Patient(string n, int a) { name = n; age = a; }

    public void Print()
    {
        Console.WriteLine($"Ім'я: {name} Вік: {age}");
    }
}
```

У прикладі визначено три конструктори — кожен приймає різну кількість параметрів. Коли ми пишемо new Patient("Іван Петренко", 45), компілятор автоматично обирає третій варіант за сигнатурою.

Починаючи з C# 9, назву типу при виклику конструктора можна опустити, якщо тип відомий з контексту:

```C#
Patient p1 = new();               // аналогічно new Patient()
Patient p2 = new("Олена");        // аналогічно new Patient("Олена")
Patient p3 = new("Іван", 45);     // аналогічно new Patient("Іван", 45)
```

Це скорочення особливо зручне при ініціалізації полів або колекцій, де тип вже вказано ліворуч.

Ключове слово this

Ключове слово this — це посилання на **поточний екземпляр** класу. Найчастіше воно використовується тоді, коли параметри конструктора (або методу) мають ті самі імена, що й поля класу. Без this компілятор не може однозначно зрозуміти, де поле, а де параметр.

```C#
using System;

Patient patient = new Patient("Марія Сидоренко", 32);
patient.Print(); // Ім'я: Марія Сидоренко Вік: 32

class Patient
{
    public string name;
    public int age;

    public Patient() { name = "Невідомий"; age = 0; }

    public Patient(string name)
    {
        this.name = name; // this.name — поле, name — параметр
        age = 0;
    }

    public Patient(string name, int age)
    {
        this.name = name;
        this.age  = age;
    }

    public void Print() => Console.WriteLine($"Ім'я: {name} Вік: {age}");
}
```

У виразі this.name = name ліва частина (this.name) означає поле поточного об'єкта, а права (name) — параметр конструктора. Якщо б параметри мали інші назви (наприклад, n та a), ключове слово this для доступу до поля не було б обов'язковим — хоча багато стилів кодування рекомендують використовувати this явно для ясності. Крім того, через this можна звертатися до будь-якого методу або поля класу з будь-якого іншого методу.

Ланцюжок виклику конструкторів

Коли в класі є кілька конструкторів, вони нерідко дублюють одну й ту саму логіку ініціалізації. Щоб уникнути дублювання, один конструктор може викликати інший за допомогою синтаксису : this(...). Це називається **ланцюжком виклику конструкторів**.

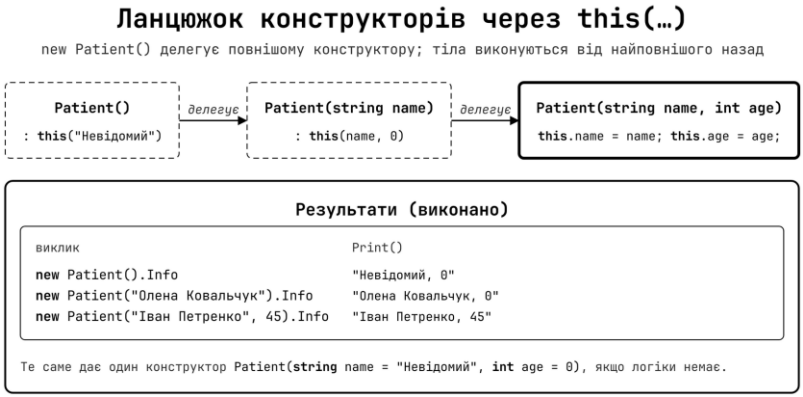
```C#
using System;

Patient p1 = new Patient();
Patient p2 = new Patient("Олена Ковальчук");
Patient p3 = new Patient("Іван Петренко", 45);

p1.Print(); // Ім'я: Невідомий Вік: 0
p2.Print(); // Ім'я: Олена Ковальчук Вік: 0
p3.Print(); // Ім'я: Іван Петренко Вік: 45
```

```
10
11  class Patient
12  {
13      public string name;
14      public int age;
15
16      public Patient() : this("Невідомий") {}
17      public Patient(string name) : this(name, 0) {}
18      public Patient(string name, int age)
19      {
20          this.name = name;
21          this.age  = age;
22      }
23
24      public void Print() => Console.WriteLine($"Ім'я: {name} Вік: {age}");
25  }
```

Важливо розуміти порядок виконання: спочатку виконується **викликаний** конструктор (той, що вказаний після :), і лише потім — тіло поточного. Тобто перший конструктор викликає другий, другий — третій, і лише після виконання третього управління повертається назад по ланцюжку.



Ланцюжок виклику конструкторів Patient через this()

Ланцюжок конструкторів особливо корисний, коли «повний» конструктор містить нетривіальну логіку валідації або ініціалізації — тоді всі спрощені конструктори просто делегують йому роботу, і ця логіка не повторюється.

Параметри за замовчуванням як альтернатива

Коли конструктори лише передають значення одне одному без додаткової логіки, увесь ланцюжок можна замінити **одним конструктором з параметрами за замовчуванням**. Це значно скорочує код і спрощує його розуміння:

```C#
using System;

Patient p1 = new Patient();
Patient p2 = new Patient("Олена Ковальчук");
Patient p3 = new Patient("Іван Петренко", 45);

p1.Print(); // Ім'я: Невідомий Вік: 0
p2.Print(); // Ім'я: Олена Ковальчук Вік: 0
p3.Print(); // Ім'я: Іван Петренко Вік: 45

class Patient
{
    public string name;
    public int age;

    public Patient(string name = "Невідомий", int age = 0)
    {
        this.name = name;
        this.age  = age;
    }

    public void Print() => Console.WriteLine($"Ім'я: {name} Вік: {age}");
}
```

Якщо при виклику конструктора аргумент для параметра не передається, автоматично підставляється значення за замовчуванням. Тобто new Patient() — це те саме, що new Patient("Невідомий", 0). Слід пам'ятати: параметри зі значеннями за замовчуванням завжди мають іти **після** параметрів без них.

Ініціалізатори об'єктів

Ініціалізатори — це синтаксична конструкція, яка дозволяє встановити значення доступних полів і властивостей об'єкта безпосередньо в момент його створення, без написання спеціального конструктора. Значення передаються у фігурних дужках після виклику конструктора:

```C#
using System;

Patient patient = new Patient { name = "Тетяна Мороз", age = 29 };
patient.Print(); // Ім'я: Тетяна Мороз Вік: 29

```

```
 6  class Patient
 7  {
 8      public string name;
 9      public int age;
10
11      public void Print() => Console.WriteLine($"Ім'я: {name} Вік: {age}");
12  }
```

При використанні ініціалізаторів слід враховувати два важливих моменти:

— За допомогою ініціалізатора можна встановити значення **лише тих полів і властивостей**, які доступні за межами класу (тобто мають модифікатор public). Закриті (private) поля залишаються недоступними.

— Ініціалізатор виконується **після** конструктора. Тому якщо і конструктор, і ініціалізатор встановлюють значення одного й того самого поля, перемагає значення з ініціалізатора.

Ініціалізатори особливо зручні, коли поле класу само по собі є об'єктом іншого класу. У такому випадку можна ініціалізувати вкладений об'єкт у тому ж виразі:

```
C#
 1  using System;
 2
 3  Patient patient = new Patient { name = "Іван Петренко", diagnosis = { title =
    "Грип" } };
 4  patient.Print(); // Ім'я: Іван Петренко Діагноз: Грип
 5
 6  class Patient
 7  {
 8      public string name;
 9      public Diagnosis diagnosis;
10
11      public Patient()
12      {
13          name      = "Невідомий";
14          diagnosis = new Diagnosis();
15      }
16
17      public void Print() => Console.WriteLine($"Ім'я: {name} Діагноз:
    {diagnosis.title}");
18  }
19
20  class Diagnosis
21  {
22      public string title = "Не встановлено";
23  }
```

Зверніть увагу на синтаксис ініціалізації вкладеного об'єкта:

```
C#
```

```
1  diagnosis = { title = "Грип" }
```

Тут не створюється новий об'єкт Diagnosis — натомість змінюється поле title вже існуючого об'єкта, який конструктор Patient() створив раніше. Саме тому такий запис можливий лише якщо конструктор уже ініціалізував поле diagnosis.

Деконструктори

Деконструктори (не плутати з деструкторами) дозволяють **розкласти об'єкт на окремі складові** — повернути кілька значень одночасно без використання кортежів або допоміжних структур. Деконструктор визначається як метод з іменем Deconstruct і параметрами out:

```
C#
 1  using System;
 2
 3  Patient patient = new Patient("Олена Ковальчук", 34);
 4
 5  (string name, int age) = patient;
 6
 7  Console.WriteLine(name);         // Олена Ковальчук
 8  Console.WriteLine(age.ToString()); // 34
 9
10  class Patient
11  {
12      string name;
13      int age;
14
15      public Patient(string name, int age)
16      {
17          this.name = name;
18          this.age  = age;
19      }
20
21      public void Deconstruct(out string patientName, out int patientAge)
22      {
23          patientName = name;
24          patientAge  = age;
25      }
26  }
```

Значення з деконструктора передаються змінним **за позицією**: перше out-значення — першій змінній, друге — другій. Насправді деконструктор — це синтаксичний цукор над явним викликом методу:

```
C#
1  string name;
2  int age;
3  patient.Deconstruct(out name, out age);
```

Обидва записи еквівалентні — компілятор перетворює деструктурування у звичайний виклик методу.

Буває, що не всі значення деконструктора потрібні. Щоб пропустити певне значення, замість змінної використовується символ відкидання _ (discard):

```
C#
 1  using System;
 2
 3  Patient patient = new Patient("Олена Ковальчук", 34);
 4
 5  (_, int age) = patient;
 6
 7  Console.WriteLine(age.ToString()); // 34
 8
 9  class Patient
10  {
11      string name;
12      int age;
13
14      public Patient(string name, int age)
15      {
16          this.name = name;
17          this.age  = age;
18      }
19
20      public void Deconstruct(out string patientName, out int patientAge)
21      {
22          patientName = name;
23          patientAge  = age;
24      }
25  }
```

Оскільки перше значення — ім'я пацієнта, яке нас не цікавить, замість змінної стоїть _. Компілятор розуміє цей знак як «це значення мені не потрібне» і не виділяє для нього змінну.

Статичний конструктор

Окрім звичайних конструкторів, C# підтримує **статичний конструктор** — він виконується один раз автоматично перед першим зверненням до класу. Він не має модифікатора доступу, не приймає параметрів і не може бути викликаний явно. Його призначення — ініціалізація статичних полів або виконання одноразових налаштувань при «пробудженні» класу.

```
C#
 1  using System;
 2
 3  Console.WriteLine("Програма запускається...");
 4
```

```
Patient p1 = new Patient("Іван Петренко");
Patient p2 = new Patient("Марія Сидоренко");
Patient p3 = new Patient("Олена Ковальчук");

Console.WriteLine($"Зареєстровано пацієнтів: {Patient.Count.ToString()}");

class Patient
{
    public static int Count;
    public string Name;

    static Patient()                    // статичний конструктор — виконується один раз
    {
        Count = 0;
        Console.WriteLine("Клінічну систему ініціалізовано.");
    }

    public Patient(string name)
    {
        Name = name;
        Count++;
    }
}
```

Зверніть увагу: статичний конструктор виконується **до** першого new Patient(...), але **після** рядка "Програма запускається..." — рівно в момент першого звернення до класу. Порядок гарантований середовищем виконання .NET.

Приватний конструктор

Конструктор може мати модифікатор private — тоді створити об'єкт класу ззовні буде неможливо. Це архітектурне рішення, яке використовується в патерні **Singleton** — коли система повинна мати рівно один екземпляр певного класу. У клінічній системі реєстр клініки логічно існує в єдиному примірнику:

C#
```
using System;

ClinicRegistry r1 = ClinicRegistry.GetInstance();
ClinicRegistry r2 = ClinicRegistry.GetInstance();

r1.AddPatient("Іван Петренко");
Console.WriteLine($"Пацієнтів: {r2.PatientCount.ToString()}");
Console.WriteLine($"Один об'єкт: {object.ReferenceEquals(r1, r2).ToString()}");

class ClinicRegistry
{
    private static ClinicRegistry _instance;
    public int PatientCount;

    private ClinicRegistry() { PatientCount = 0; }  // ззовні не викликати
```

```
16
17      public static ClinicRegistry GetInstance()
18      {
19          if (_instance == null)
20              _instance = new ClinicRegistry();
21          return _instance;
22      }
23
24      public void AddPatient(string name) { PatientCount++; }
25  }
```

Оскільки конструктор private, єдиний спосіб отримати об'єкт — через GetInstance(). Обидві змінні r1 і r2 вказують на один об'єкт — ReferenceEquals повертає true.

Первинні конструктори (C# 12)

Починаючи з C# 12, параметри конструктора можна оголошувати прямо в заголовку класу — це називається **первинним конструктором** (primary constructor). Він скорочує шаблонний код і робить оголошення класу компактнішим:

```
C#
 1  using System;
 2
 3  Patient p1 = new Patient("Іван Петренко", 45);
 4  Patient p2 = new Patient("Марія Сидоренко", 32);
 5
 6  p1.Print();
 7  p2.Print();
 8
 9  class Patient(string name, int age)   // первинний конструктор
10  {
11      public void Print()
12      {
13          Console.WriteLine($"Пацієнт: {name}, {age.ToString()} років");
14      }
15  }
```

Параметри первинного конструктора (name, age) доступні у всьому тілі класу. Якщо потрібні додаткові конструктори — вони делегують виклик первинному через this(...):

```
C#
 1  using System;
 2
 3  Patient p1 = new Patient("Іван Петренко", 45);
 4  Patient p2 = new Patient("Невідомий");
 5
 6  p1.Print();
 7  p2.Print();
 8
```

```
 9  class Patient(string name, int age)
10  {
11      public Patient(string name) : this(name, 0) { }
12
13      public void Print()
14      {
15          Console.WriteLine($"Пацієнт: {name}, {age.ToString()} років");
16      }
17  }
```

Первинні конструктори особливо зручні для простих класів-моделей, де потрібно лише зберегти кілька значень і надати до них доступ.

3.3. Клас Program та метод Main. Програми верхнього рівня

Коли ви пишете перший рядок C#-коду і натискаєте «Запустити», виникає закономірне питання: з якого саме місця починається виконання програми? На відміну від скриптових мов, де інтерпретатор просто читає файл зверху донизу, у C# є чітко визначена **точка входу** — спеціальний метод, якому середовище виконання передає управління при старті. Розуміння цього механізму важливе, бо воно пояснює, чому одні змінні і методи доступні «скрізь», а інші — лише всередині класу, і яку роль відіграє структура файлу у великих проектах.

Метод Main як точка входу

Точкою входу до програми мовою C# є метод Main. Саме з нього починається виконання, і програма обов'язково повинна містити цей метод. Класично він виглядає так:

```
C#
1  class Program
2  {
3      static void Main(string[] args)
4      {
5          Console.WriteLine("Клінічна система запущена.");
6      }
7  }
```

Метод Main завжди оголошується зі статичним модифікатором static — це означає, що він належить класу, а не конкретному об'єкту, і може бути викликаний без створення екземпляра класу. Детальніше про

static йтиметься в наступних розділах. Тип, що повертається, — void (метод нічого не повертає) або int (код завершення процесу). Параметр string[] args — масив рядків, через який можна передати аргументи при запуску програми з командного рядка.

Програми верхнього рівня

Інструкції верхнього рівня з'явилися в C# 9 (.NET 5), а починаючи з .NET 6 (Visual Studio 2022) шаблон нового консольного проєкту генерує саме такий лаконічний стартовий код:

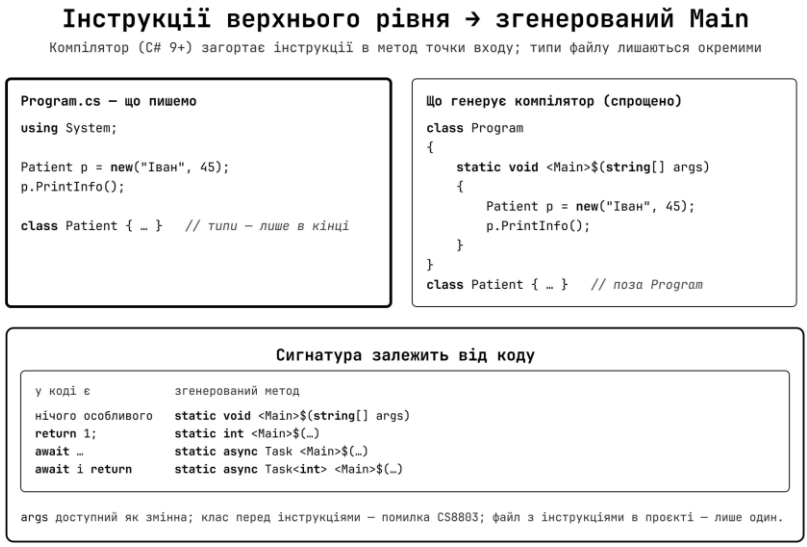
```
C#
1  // See https://aka.ms/new-console-template for more information
2  Console.WriteLine("Hello, World!");
```

Тут немає жодного визначення класу Program і методу Main — але програма повністю коректна і виконується без помилок. Це так звана **програма верхнього рівня** (top-level program), а рядки коду в ній — **інструкції верхнього рівня** (top-level statements).

Насправді компілятор C# сам генерує клас Program і метод точки входу, поміщаючи туди всі ваші інструкції верхнього рівня. Згенерований метод має службове ім'я <Main>$, недоступне з коду, а його сигнатура залежить від вмісту: void або int (якщо є return зі значенням), async Task або async Task<int> (якщо є await). Параметр args доступний як звичайна змінна. Спрощено це можна уявити так — код вище фактично еквівалентний:

```
C#
1  class Program
2  {
3      static void Main(string[] args)
4      {
5          Console.WriteLine("Hello, World!");
6      }
7  }
```

Тобто розробник звільнений від написання цього шаблонного коду — компілятор робить це автоматично. Обидва варіанти поводяться однаково; відрізняються лише ім'я згенерованого методу (<Main>$ замість Main) і модифікатор доступу класу.



Top-level statements → компілятор генерує клас Program і метод Main

Змінні, методи і класи у top-level програмах

Розміщення різних елементів у top-level програмі підпорядковується чітким правилам. Якщо ми оголошуємо змінні та локальні методи, вони потрапляють всередину згенерованого методу Main. Наприклад, ця програма верхнього рівня:

```C#
string greeting = "Ласкаво просимо до клінічної системи";

PrintGreeting(greeting);

void PrintGreeting(string message)
{
    Console.WriteLine(message);
}
```

є повністю еквівалентною такому явному запису:

```C#
class Program
{
    static void Main(string[] args)
    {
        string greeting = "Ласкаво просимо до клінічної системи";
```

```
 6
 7          PrintGreeting(greeting);
 8
 9          void PrintGreeting(string message)
10          {
11              Console.WriteLine(message);
12          }
13      }
14  }
```

Ситуація з класами інша: якщо в top-level програмі визначається новий клас, він розміщується **поза** класом Program, на рівні простору імен. Тому наступний код:

```
C#
 1  Patient patient = new("Іван Петренко", 45);
 2  patient.PrintInfo();
 3
 4  class Patient
 5  {
 6      public string Name;
 7      public int Age;
 8
 9      public Patient(string name, int age) { Name = name; Age = age; }
10      public void PrintInfo() => Console.WriteLine($"Пацієнт: {Name}, {Age}
    років");
11  }
```

еквівалентний такому розгорнутому запису:

```
C#
 1  class Program
 2  {
 3      static void Main(string[] args)
 4      {
 5          Patient patient = new("Іван Петренко", 45);
 6          patient.PrintInfo();
 7      }
 8  }
 9
10  class Patient
11  {
12      public string Name;
13      public int Age;
14
15      public Patient(string name, int age) { Name = name; Age = age; }
16      public void PrintInfo() => Console.WriteLine($"Пацієнт: {Name}, {Age}
    років");
17  }
```

Клас Patient знаходиться поза Program в обох випадках — top-level синтаксис лише приховує від нас шаблонний код Program.

Є одне важливе обмеження: **визначення типів (класів, структур тощо) повинні йти в кінці файлу**, після всіх інструкцій верхнього рівня. Якщо розмістити клас перед інструкціями — компілятор видасть помилку. Правильний порядок:

```C#
// Спочатку — інструкції верхнього рівня
Patient patient = new("Іван Петренко", 45);
patient.PrintInfo();

// Потім — визначення класів
class Patient
{
    public string Name;
    public int Age;

    public Patient(string name, int age) { Name = name; Age = age; }
    public void PrintInfo() => Console.WriteLine($"Пацієнт: {Name}, {Age} років");
}
```

Явне визначення Main та його застосування

Незважаючи на зручність top-level синтаксису, явне визначення класу Program і методу Main залишається цілком коректним і досі широко використовується. Такий підхід обов'язковий у великих проектах, де потрібно мати кілька класів і чітко контролювати структуру коду, або коли інструменти і шаблони проекту вимагають явного Main.

```C#
using System;

class Program
{
    static void Main(string[] args)
    {
        Patient patient = new Patient("Марія Сидоренко", 32);
        patient.PrintInfo();
    }
}

class Patient
{
    public string Name;
    public int Age;

    public Patient(string name, int age)
    {
        Name = name;
        Age  = age;
    }

    public void PrintInfo()
    {
```

```
25          Console.WriteLine($"Пацієнт: {Name}, {Age} років");
26      }
27  }
```

Обидва підходи — top-level і явний Main — дають однаковий скомпільований результат. Вибір між ними — питання стилю, вимог команди або розміру проекту. Для навчальних прикладів і невеликих утиліт top-level синтаксис значно скорочує обсяг шаблонного коду і дозволяє зосередитись на суті задачі.

Асинхронна точка входу: async Task Main

У сучасних .NET-програмах часто виникає потреба виконувати асинхронні операції — звертатись до мережі, читати файли, робити запити до бази даних. Асинхронність у C# будується на ключових словах async і await. Щоб використовувати await безпосередньо в точці входу, метод Main може бути оголошений як асинхронний:

```
C#
1  class Program
2  {
3      static async Task Main(string[] args)
4      {
5          await DoSomethingAsync();
6      }
7  }
```

Тип, що повертається, змінюється з void на Task (або Task<int> якщо потрібен код завершення). Це дозволяє Main «чекати» на завершення асинхронних операцій, не блокуючи потік виконання.

У top-level програмах асинхронність вмикається ще простіше — достатньо використати await прямо в коді верхнього рівня, і компілятор автоматично зробить точку входу асинхронною:

```
C#
1  using System.Net.Http;
2
3  var client = new HttpClient();
4  string result = await client.GetStringAsync("https://example.com");
5  Console.WriteLine(result);
```

Компілятор перетворить цей код на async Task Main(...) самостійно. Детально асинхронне програмування розглядається в наступних розділах курсу — тут важливо розуміти, що точка входу програми повністю підтримує цей механізм.

3.4. Структури

Поряд із класами структури представляють ще один спосіб створення власних типів даних у C#. Більше того, багато примітивних типів — int, double, bool та інші — по суті є структурами, визначеними в стандартній бібліотеці .NET. Це не випадково: структури оптимізовані для зберігання невеликих, семантично цілісних наборів даних, які логічно утворюють одне значення. Наприклад, показник артеріального тиску складається з двох чисел — систолічного і діастолічного — і не має сенсу розглядати їх окремо. Саме такі дані найкраще представляти структурою.

Ключова відмінність структури від класу полягає в тому, що **структура є типом значення** (value type), а клас — **типом посилання** (reference type). Детально про цю різницю йтиметься у розділі 3.5, але вже зараз важливо розуміти: при копіюванні структури створюється повністю незалежна копія, тоді як при копіюванні посилання на клас обидві змінні вказують на один і той самий об'єкт у пам'яті.

Визначення структури

Для визначення структури застосовується ключове слово struct:

```
C#
1  struct ім'я_структури
2  {
3      // Елементи структури
4  }
```

Після слова struct йде назва структури і далі у фігурних дужках розміщуються елементи структури — поля, методи тощо. Як і класи, структури можуть зберігати стан як поля (змінні) і визначати поведінку як методи. Наприклад, визначимо структуру BloodPressure, яка представляє показник артеріального тиску пацієнта:

```
C#
 1  struct BloodPressure
 2  {
 3      public int systolic;   // систолічний тиск (верхній)
 4      public int diastolic;  // діастолічний тиск (нижній)
 5
 6      public void Print()
 7      {
 8          Console.WriteLine($"Тиск: {systolic}/{diastolic} мм рт. ст.");
 9      }
10  }
```

Тут визначено два поля для зберігання показників тиску та метод Print для виведення результату на консоль. Для звернення до полів і методів структури застосовується точкова нотація — так само, як і для класів:

```
C#
1  об'єкт.поле_структури
2  об'єкт.метод_структури(параметри)
```

Створення об'єкта структури

Ініціалізація за допомогою конструктора

Для використання структури її необхідно ініціалізувати. Як і у випадку класів, для створення об'єктів структури застосовується виклик конструктора з оператором new. Навіть якщо в коді структури не визначено жодного конструктора, вона має щонайменше один — **конструктор за замовчуванням**, який генерується компілятором. Він не приймає параметрів і створює об'єкт зі значеннями за замовчуванням: 0 для числових полів і null для рядків.

```
C#
 1  using System;
 2
 3  BloodPressure bp = new BloodPressure(); // виклик конструктора за замовчуванням
 4  // або: BloodPressure bp = new();
 5
 6  bp.systolic  = 120; // встановлюємо значення після створення
 7  bp.diastolic = 80;
 8
 9  bp.Print(); // Тиск: 120/80 мм рт. ст.
10
11  struct BloodPressure
12  {
13      public int systolic;
14      public int diastolic;
15
16      public void Print()
17      {
18          Console.WriteLine($"Тиск: {systolic}/{diastolic} мм рт. ст.");
19      }
20  }
```

Безпосередня ініціалізація полів

Якщо всі поля структури мають модифікатор public, структуру можна ініціалізувати без виклику конструктора — просто оголосивши змінну і надавши значення всім полям вручну перед першим використанням. Компілятор вимагає, щоб **усі поля були заповнені** до звернення до

будь-якого з них або до методів структури. Якщо хоч одне поле залишиться неініціалізованим, компілятор видасть помилку.

```
C#
 1  using System;
 2
 3  BloodPressure bp; // конструктор не викликається
 4  bp.systolic  = 130;
 5  bp.diastolic = 85;
 6
 7  bp.Print(); // Тиск: 130/85 мм рт. ст.
 8
 9  struct BloodPressure
10  {
11      public int systolic;
12      public int diastolic;
13
14      public void Print()
15      {
16          Console.WriteLine($"Тиск: {systolic}/{diastolic} мм рт. ст.");
17      }
18  }
```

Ініціалізація полів за замовчуванням

Починаючи з C# 10, поля структури можна ініціалізувати безпосередньо при їх визначенні — так само, як у класі. До C# 10 це було заборонено. Якщо структура має ініціалізатори полів, вона зобов'язана оголосити хоча б один конструктор явно (інакше помилка CS8983), а щоб ініціалізатори спрацювали при new BloodPressure(), потрібен саме явний конструктор без параметрів:

```
C#
 1  using System;
 2
 3  BloodPressure bp = new BloodPressure();
 4  bp.Print(); // Тиск: 120/80 мм рт. ст.
 5
 6  struct BloodPressure
 7  {
 8      public int systolic  = 120; // значення за замовчуванням, C# 10+
 9      public int diastolic = 80;
10
11      public BloodPressure() { } // явний конструктор без параметрів обов'язковий
12
13      public void Print() => Console.WriteLine($"Тиск: {systolic}/{diastolic} мм
    рт. ст.");
14  }
```

Конструктори структури

Як і клас, структура може визначати власні конструктори з параметрами. Це дозволяє ініціалізувати об'єкт у момент створення, не

заповнюючи поля вручну після new. Починаючи з C# 10, дозволено також визначати конструктор без параметрів явно:

```
C#
 1  using System;
 2
 3  BloodPressure bp1 = new BloodPressure();
 4  BloodPressure bp2 = new BloodPressure(130, 85);
 5
 6  bp1.Print(); // Тиск: 120/80 мм рт. ст.
 7  bp2.Print(); // Тиск: 130/85 мм рт. ст.
 8
 9  struct BloodPressure
10  {
11      public int systolic;
12      public int diastolic;
13
14      public BloodPressure() // C# 10+: власний конструктор без параметрів
15      {
16          systolic  = 120;
17          diastolic = 80;
18      }
19
20      public BloodPressure(int systolic, int diastolic)
21      {
22          this.systolic  = systolic;
23          this.diastolic = diastolic;
24      }
25
26      public void Print() => Console.WriteLine($"Тиск: {systolic}/{diastolic} мм
    рт. ст.");
27  }
```

Варто зазначити: якщо в структурі є конструктор із параметрами, але немає явного конструктора без параметрів, то при виклику new() (без аргументів) все одно використовується автоматичний конструктор за замовчуванням — він встановлює всі поля в 0/null, ігноруючи будь-які значення за замовчуванням, визначені в полях.

Конструктори структури можна також будувати в ланцюжок через this(...) — за тими самими правилами, що й для класів. Починаючи з C# 11, конструктор структури взагалі не зобов'язаний ініціалізувати всі поля: поля, яким конструктор нічого не присвоїв, компілятор автоматично заповнює значеннями за замовчуванням (0, false, null; перевірено на .NET 10). До C# 11 таке поле давало помилку компіляції. Ланцюжок через this(...) при цьому працює так само, як у класах:

```
C#
 1  using System;
 2
 3  BloodPressure bp1 = new BloodPressure();
 4  BloodPressure bp2 = new BloodPressure(130);
 5  BloodPressure bp3 = new BloodPressure(140, 90);
```

```
 6
 7  bp1.Print(); // Тиск: 120/80 мм рт. ст.
 8  bp2.Print(); // Тиск: 130/80 мм рт. ст.
 9  bp3.Print(); // Тиск: 140/90 мм рт. ст.
10
11  struct BloodPressure
12  {
13      public int systolic;
14      public int diastolic;
15
16      public BloodPressure() : this(120, 80) {}
17      public BloodPressure(int systolic) : this(systolic, 80) {}
18      public BloodPressure(int systolic, int diastolic)
19      {
20          this.systolic  = systolic;
21          this.diastolic = diastolic;
22      }
23
24      public void Print() => Console.WriteLine($"Тиск: {systolic}/{diastolic} мм
    рт. ст.");
25  }
```

Ініціалізатор структури

Так само, як і для класу, можна використовувати синтаксис ініціалізатора для створення об'єкта структури. При цьому спочатку викликається конструктор без параметрів (явний або автоматичний), а потім вказаним полям надаються задані значення:

```
C#
 1  using System;
 2
 3  BloodPressure bp = new BloodPressure { systolic = 118, diastolic = 76 };
 4  bp.Print(); // Тиск: 118/76 мм рт. ст.
 5
 6  struct BloodPressure
 7  {
 8      public int systolic;
 9      public int diastolic;
10
11      public void Print() => Console.WriteLine($"Тиск: {systolic}/{diastolic} мм
    рт. ст.");
12  }
```

Копіювання структури за допомогою with

Коли потрібно створити новий об'єкт структури на основі існуючого, змінивши лише окремі поля, зручно використовувати оператор with. Він створює **копію** структури і застосовує до неї вказані зміни, не торкаючись оригіналу:

```
C#
 1  using System;
```

```
 2
 3  BloodPressure initial   = new BloodPressure { systolic = 120, diastolic = 80 };
 4  BloodPressure elevated  = initial with { systolic = 145 };
 5  BloodPressure crisis    = initial with { systolic = 180, diastolic = 110 };
 6
 7  initial.Print();  // Тиск: 120/80 мм рт. ст.
 8  elevated.Print(); // Тиск: 145/80 мм рт. ст.
 9  crisis.Print();   // Тиск: 180/110 мм рт. ст.
10
11  struct BloodPressure
12  {
13      public int systolic;
14      public int diastolic;
15
16      public void Print() => Console.WriteLine($"Тиск: {systolic}/{diastolic} мм
    рт. ст.");
17  }
```

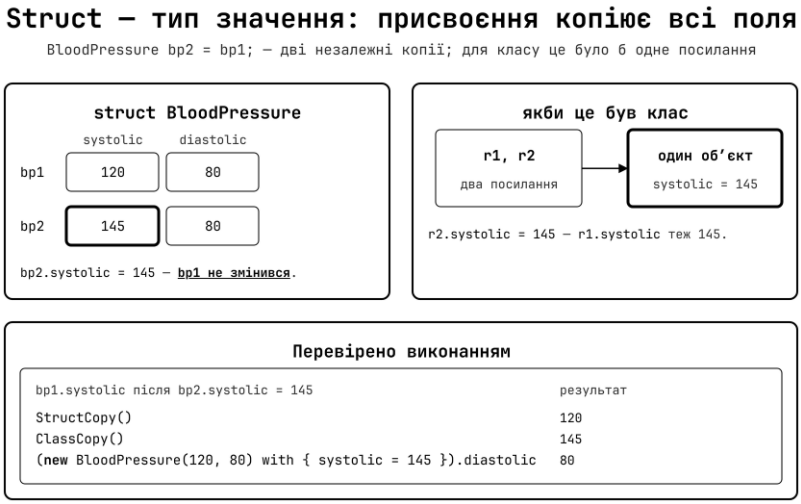
Тут об'єкт elevated отримує всі значення від initial, але з іншим систолічним тиском. Об'єкт crisis також базується на initial, але змінені обидва поля. При цьому initial лишається незмінним — оператор with ніколи не модифікує оригінал.

Struct як тип значення: copy semantics

Саме тут проявляється принципова відмінність структури від класу. Коли ми присвоюємо одну структуру іншій, C# **копіює всі поля** — виникає два повністю незалежних об'єкти. Зміна одного не впливає на інший:

```
C#
 1  using System;
 2
 3  BloodPressure bp1 = new BloodPressure { systolic = 120, diastolic = 80 };
 4  BloodPressure bp2 = bp1; // повна копія значень
 5
 6  bp2.systolic = 145; // змінюємо лише bp2
 7
 8  bp1.Print(); // Тиск: 120/80 мм рт. ст. — bp1 не змінився
 9  bp2.Print(); // Тиск: 145/80 мм рт. ст.
10
11  struct BloodPressure
12  {
13      public int systolic;
14      public int diastolic;
15
16      public void Print() => Console.WriteLine($"Тиск: {systolic}/{diastolic} мм
    рт. ст.");
17  }
```

Якби BloodPressure було класом, обидві змінні вказували б на один і той самий об'єкт у пам'яті — зміна bp2.systolic змінила б і bp1.systolic. Зі структурою цього не відбувається.



Struct — тип значення: копіювання створює незалежний об'єкт

Незмінна структура: readonly struct

Якщо структура представляє дані, які не повинні змінюватись після створення — наприклад, зафіксований результат вимірювання — її варто оголосити як readonly struct. Компілятор заборонить будь-які присвоювання полям поза конструктором, що унеможливлює випадкову мутацію.

```C#
using System;

BloodPressure bp = new BloodPressure(120, 80);
bp.Print();

// bp.systolic = 130; // помилка компіляції — readonly struct незмінна

readonly struct BloodPressure
{
    public readonly int systolic;
    public readonly int diastolic;

    public BloodPressure(int systolic, int diastolic)
```

```
14      {
15          this.systolic  = systolic;
16          this.diastolic = diastolic;
17      }
18
19      public void Print() => Console.WriteLine($"Тиск: {systolic}/{diastolic} мм
    рт. ст.");
20  }
```

readonly struct має перевагу з точки зору продуктивності: компілятор знає, що об'єкт незмінний, тому, коли структура передається як in-параметр або зберігається в readonly-полі, йому не потрібно робити «захисну копію» перед кожним викликом методу (для звичайної структури компілятор копіює її, щоб метод випадково не змінив значення, яке має лишатися незмінним).

Record struct (C# 10)

C# 10 додав синтаксис record struct — структура з автоматично згенерованими Equals, GetHashCode, ToString і підтримкою оператора with. Оголошується схоже на первинний конструктор:

```
C#
 1  using System;
 2
 3  BloodPressure bp1 = new BloodPressure(120, 80);
 4  BloodPressure bp2 = bp1 with { systolic = 145 };
 5  BloodPressure bp3 = new BloodPressure(120, 80);
 6
 7  bp1.Print();
 8  bp2.Print();
 9
10  Console.WriteLine($"bp1 == bp3: {(bp1 == bp3).ToString()}"); // True —
    порівняння за значенням
11
12  record struct BloodPressure(int systolic, int diastolic)
13  {
14      public void Print() => Console.WriteLine($"Тиск: {systolic}/{diastolic} мм
    рт. ст.");
15  }
```

На відміну від звичайної структури, record struct автоматично реалізує **порівняння за значенням**: bp1 == bp3 повертає true, бо всі поля збігаються. ToString() автоматично повертає BloodPressure { systolic = 120, diastolic = 80 } — зручно для відлагодження. record struct — оптимальний вибір для невеликих незмінних моделей даних, де потрібне структурне порівняння без написання шаблонного коду вручну.

3.5. Типи значень та типи посилань

У попередніх розділах ми розглядали різні типи даних C#: прості (int, double, bool), а також складені — структури та класи. Усі ці типи поділяються на дві фундаментальні категорії: **типи значень** (value types) і **типи посилань** (reference types). Розуміння цієї різниці є одним із ключових у мові C#, оскільки вона визначає поведінку при копіюванні, передачі у методи та роботі з пам'яттю.

Типи значень:

- Цілочисленні типи: byte, sbyte, short, ushort, int, uint, long, ulong
- Типи з плаваючою комою: float, double
- Тип decimal
- Тип bool
- Тип char
- Перерахування enum
- Структури struct

Типи посилань:

- Тип object
- Тип string
- Класи class
- Інтерфейси interface
- Делегати delegate

Організація пам'яті: стек і купа

Щоб зрозуміти різницю між цими двома категоріями, необхідно розібратися з тим, як .NET організовує пам'ять. Середовище виконання використовує два принципово різних регіони пам'яті: **стек** (stack) і **купу** (heap).

Стек — це область пам'яті, що працює за принципом «останній зайшов — перший вийшов». Кожен виклик методу виділяє у стеку окремий **кадр** (stack frame), де зберігаються локальні змінні та параметри цього методу. Коли метод завершується, його кадр автоматично звільняється. Стек дуже швидкий, але обмежений за розміром.

Купа (heap) — невпорядкований набір об'єктів різного розміру. Тут зберігаються дані об'єктів посилального типу. Час життя об'єктів у купі не прив'язаний до конкретного методу — вони живуть доти, доки

на них існують посилання. Коли посилань більше немає, **збирач сміття** (garbage collector) автоматично звільняє зайняту пам'ять.

Ось як виглядає розміщення при роботі з методом, де є локальні змінні:

```
C#
1  static void RegisterVisit(int patientId)
2  {
3      int roomNumber = 12;
4      int duration   = 30;
5      int totalCost  = duration * patientId;
6  }
```

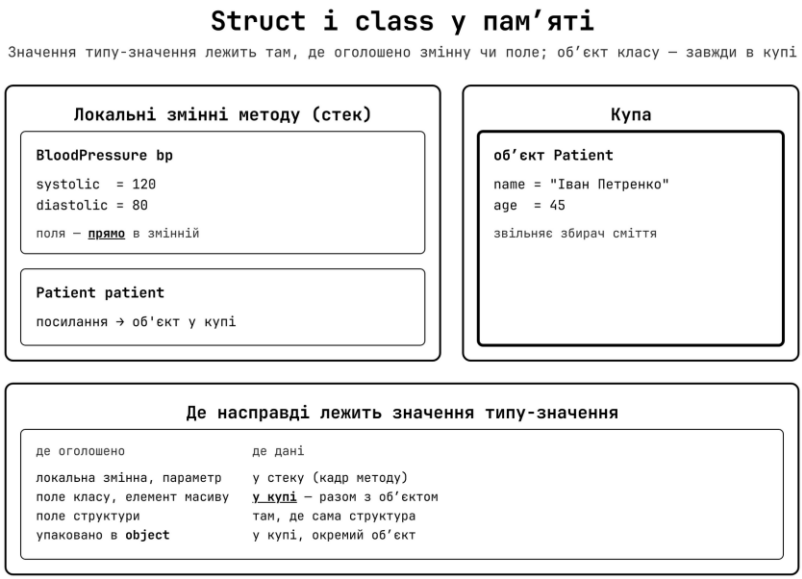
При виклику RegisterVisit у стеку виділяється кадр з трьома змінними: patientId, roomNumber, duration і totalCost — усі типу int, тобто типи значень. По завершенні методу кадр зникає, а пам'ять повторно використовується.

Struct у стеку, клас у купі

Ключова відмінність між структурою і класом у тому, де зберігаються їх дані. Коли оголошується **локальна** змінна типу структури, її поля розміщуються **безпосередньо у стеку**, у кадрі методу. Точніше: значення типу-значення зберігається там, де оголошено змінну. Структура — поле класу або елемент масиву — лежить у купі разом з цим об'єктом; поле однієї структури всередині іншої — там, де зовнішня структура. Коли оголошується змінна типу класу, у стеку зберігається лише **посилання** (адреса), а самі дані об'єкта розміщуються у **купі**:

```
C#
 1  using System;
 2
 3  BloodPressure bp      = new BloodPressure(120, 80); // struct — дані у стеку
 4  Patient       patient = new Patient("Іван Петренко", 45); // class — дані у
    купі
 5
 6  bp.Print();
 7  patient.Print();
 8
 9  struct BloodPressure
10  {
11      public int systolic;
12      public int diastolic;
13      public BloodPressure(int s, int d) { systolic = s; diastolic = d; }
14      public void Print() => Console.WriteLine($"Тиск: {systolic}/{diastolic}");
15  }
16
17  class Patient
```

```
{
    public string name;
    public int age;
    public Patient(string name, int age) { this.name = name; this.age = age; }
    public void Print() => Console.WriteLine($"Пацієнт: {name}, {age} років");
}
```



Розміщення struct та class у пам'яті: стек та купа

Коли об'єкт Patient більше не потрібний (на нього немає жодного посилання), збирач сміття автоматично звільняє пам'ять у купі. Зі стеком цього не потрібно — кадр методу зникає сам по собі.

Копіювання значень

Тип даних суттєво впливає на те, що саме відбувається при присвоєнні однієї змінної іншій.

При присвоєнні **типу значень** створюється повна незалежна копія даних. Подальші зміни однієї змінної не впливають на іншу:

```
C#
using System;

BloodPressure bp1 = new BloodPressure(120, 80);
```

```
BloodPressure bp2 = bp1; // копія значень

bp2.systolic = 145;

Console.WriteLine($"bp1: {bp1.systolic}/{bp1.diastolic}"); // 120/80 — не змінився
Console.WriteLine($"bp2: {bp2.systolic}/{bp2.diastolic}"); // 145/80

struct BloodPressure
{
    public int systolic;
    public int diastolic;
    public BloodPressure(int s, int d) { systolic = s; diastolic = d; }
}
```

При присвоєнні **типу посилань** копіюється лише посилання — обидві змінні вказують на той самий об'єкт у купі. Зміна через одну змінну відразу видна через іншу:

```
C#
using System;

Patient patient1 = new Patient("Іван Петренко", 45);
Patient patient2 = patient1; // копіюється лише посилання

patient2.name = "Марія Сидоренко"; // змінюємо через patient2

Console.WriteLine(patient1.name); // Марія Сидоренко — теж змінилось!
Console.WriteLine(patient2.name); // Марія Сидоренко

class Patient
{
    public string name;
    public int age;
    public Patient(string name, int age) { this.name = name; this.age = age; }
}
```

Це пояснює, чому в попередньому розділі зміна bp2.systolic не торкалась bp1 — там структура. А тут зміна patient2.name одразу відображається в patient1 — бо обидві змінні вказують на один і той самий об'єкт у купі.

Типи посилань всередині типів значень

Окремо варто розглянути ситуацію, коли структура містить поле посилального типу. Це тонкий момент, де легко допуститися помилки. Розглянемо структуру Visit (візит пацієнта), яка містить поле типу класу Diagnosis:

```
C#
using System;

```

```
Visit visit1 = new Visit("2026-05-21");
Visit visit2 = visit1; // копіюємо struct

visit2.diagnosis.title = "Ангіна"; // змінюємо через visit2

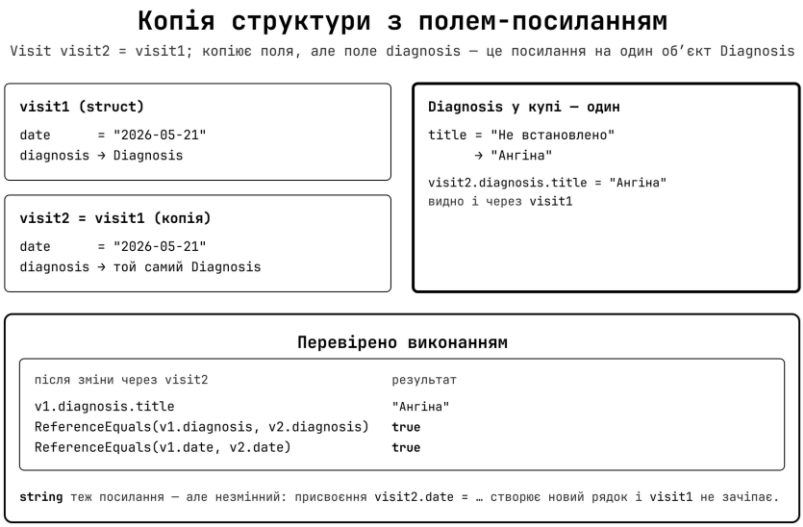
Console.WriteLine(visit1.diagnosis.title); // Ангіна — теж змінилось!
Console.WriteLine(visit2.diagnosis.title); // Ангіна

struct Visit
{
    public string date;
    public Diagnosis diagnosis;

    public Visit(string date)
    {
        this.date  = date;
        diagnosis  = new Diagnosis();
    }
}

class Diagnosis
{
    public string title = "Не встановлено";
}
```

Що тут відбувається? При присвоєнні visit2 = visit1 структура справді копіюється — але лише **поверхово** (shallow copy). Обидва поля — date (рядок) і diagnosis — посилального типу, тож копіюються лише посилання. Для рядка це непомітно: string незмінний, і присвоєння visit2.date = "…" створює новий рядок, не зачіпаючи visit1. А от поле diagnosis — це посилання на об'єкт у купі, і копіюється лише це посилання, а не сам об'єкт. Тому visit1.diagnosis і visit2.diagnosis вказують на той самий об'єкт у купі, і зміна через одну структуру впливає на іншу.



Дві структури з посиланням на один об'єкт у купі

Це типове джерело помилок при роботі зі структурами, що містять поля-класи. Якщо потрібна справжня незалежна копія, необхідно явно створювати новий об'єкт вкладеного класу — або взагалі переосмислити архітектуру і замінити клас структурою там, де це доречно.

Boxing та Unboxing

У C# всі типи — і значень, і посилань — наслідують від базового типу object. Це дозволяє покласти, наприклад, int у змінну типу object. Але оскільки int — тип значень, а object — тип посилань, відбувається **boxing** (упаковка): у купі виділяється новий об'єкт-обгортка, і значення копіюється в нього. Кожна упаковка створює **окремий** об'єкт: два object o1 = age; object o2 = age; — різні посилання (ReferenceEquals дає false).

Зворотна операція — **unboxing** (розпаковка): явне приведення object назад до конкретного типу значень, при якому значення копіюється з купи назад у стек.

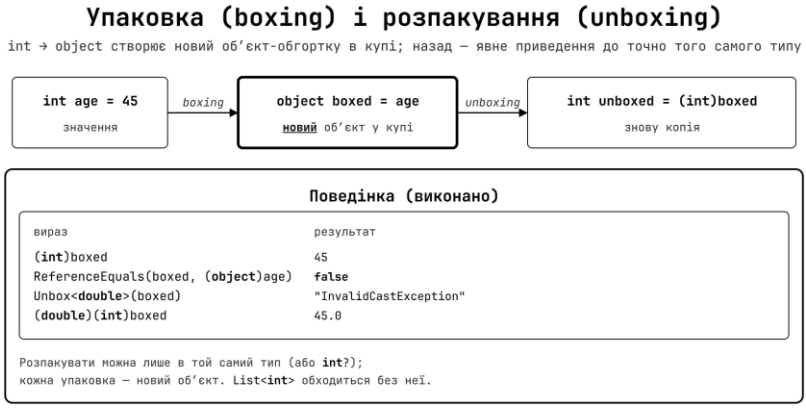
```C#
using System;

int age = 45;
```

```
 4  object boxed = age;        // boxing: int зі стека → обгортка в купі
 5
 6  int unboxed = (int)boxed;  // unboxing: назад зі купи до стека
 7
 8  Console.WriteLine(age.ToString());     // 45
 9  Console.WriteLine(boxed.ToString());   // 45
10  Console.WriteLine(unboxed.ToString()); // 45
```

На перший погляд — зручно. Але boxing і unboxing мають реальну ціну: кожен boxing — це виділення пам'яті в купі і копіювання даних, а кожен unboxing — ще одне копіювання і перевірка типу. У гарячих ділянках коду (цикли з тисячами ітерацій, масові колекції) це може суттєво знизити продуктивність.

Якщо тип виконання об'єкта не збігається з типом приведення — виникне виняток InvalidCastException. Наприклад, (double)boxedInt впаде, навіть якщо числово int і double сумісні — потрібно спочатку розпакувати в int, а потім перетворити.



Boxing та Unboxing — переміщення між стеком і купою

Саме для уникнення boxing у C# з'явились **узагальнення** (generics) — наприклад, List<int> зберігає int безпосередньо без упаковки, на відміну від старого ArrayList, який приймав object і боксував кожне значення. Детально generics розглядаються у наступних розділах.

Питання для самоконтролю — Розділ 3. Класи, структури та простір імен

3.1. Класи та об'єкти

1. Чим відрізняється **клас** від **об'єкта**? Наведіть аналогію з реального життя і поясніть, чому не можна поставити знак рівності між цими поняттями.
2. Що відбувається покроково при виконанні рядка Patient patient1 = new Patient();? Опишіть всі три дії, які виконує середовище виконання.
3. Що виведе наступний код? Поясніть чому.

```
C#
1  Patient a = new Patient();
2  a.Name = "Олена";
3  Patient b = a;
4  b.Name = "Марія";
5  Console.WriteLine(a.Name);
```

4. У чому різниця між полем і методом класу? Чому метод може звертатися до полів без додаткових параметрів?
5. Пояснить різницю між public і private модифікаторами доступу на рівні полів. Чому в реальних проєктах поля роблять приватними?
6. Що таке null для змінної типу класу? Що станеться, якщо викликати метод через null-посилання, і як цього уникнути?
7. Напишіть клас Book з полями Title (рядок), Author (рядок), Year (ціле число) і константою MaxTitleLength = 200. Додайте метод Display(), що виводить інформацію про книгу. Створіть масив з трьох книг і виведіть їх у циклі.
8. Де зберігаються значення полів об'єкта — у стеку чи в купі? Де зберігається змінна, що містить посилання на об'єкт? Чому ця різниця важлива для розуміння семантики присвоєння?

3.2. Конструктори, ініціалізатори та деконструктори

9. Що станеться, якщо у класі визначити конструктор з параметрами, але не визначити конструктор без параметрів? Чи можна тоді писати new Patient()?

10. Поясніть синтаксис : this(...) у конструкторі. Який порядок виконання, коли один конструктор делегує іншому?
11. Що виведе наступний код?

```C#
class Patient
{
    public string Name;
    public int Age;
    public Patient() : this("Невідомий") {}
    public Patient(string name) : this(name, 0) {}
    public Patient(string name, int age)
    {
        Name = name;
        Age  = age;
        Console.WriteLine($"Patient({name}, {age})");
    }
}
Patient p = new Patient();
```

12. Чим ініціалізатор об'єкта { field = value } відрізняється від конструктора? Що відбувається першим — конструктор чи ініціалізатор?
13. Поясніть, для чого потрібен статичний конструктор. Коли він виконується і скільки разів?
14. Порівняйте ланцюжок конструкторів через this(...) і єдиний конструктор з параметрами за замовчуванням. Коли один підхід кращий за інший?
15. Що таке деконструктор у C# (метод Deconstruct)? Чим він відрізняється від деструктора? Напишіть клас Point з координатами X і Y, реалізуйте деконструктор і продемонструйте його використання.
16. Для чого використовується private конструктор? Наведіть приклад патерну Singleton з поясненням, чому він потребує саме приватного конструктора.

3.3. Клас Program та метод Main. Програми верхнього рівня

17. Що таке точка входу програми в C#? Як CLR знаходить її при запуску?
18. Що робить компілятор, коли бачить програму верхнього рівня (без явного класу Program і методу Main)? Який еквівалентний код він генерує?

19. Визначте, що з наступного потрапить усередину методу Main, а що буде поза ним при top-level синтаксисі:

- оголошення локальної змінної string greeting = "Привіт"; - оголошення локальної функції void Say(string s) { ... } - оголошення класу class Helper { ... }

20. Яке обмеження існує на порядок розміщення класів і інструкцій у top-level програмі? Що станеться, якщо порушити цей порядок?
21. У яких ситуаціях доцільно писати явний class Program і static void Main(...), а не покладатися на top-level синтаксис?
22. Як оголосити точку входу, щоб можна було використовувати await прямо в Main? Що відбувається з top-level програмою, коли в ній вперше зустрічається await?
23. Чому метод Main завжди оголошується як static? Що це означає стосовно об'єктів?

3.4. Структури

24. Назвіть ключову відмінність між struct і class з точки зору семантики копіювання. Що відбудеться, якщо присвоїти одну структуру іншій?
25. Що виведе наступний код? Чому?

```
C#
1  BloodPressure bp1 = new BloodPressure { systolic = 120, diastolic = 80 };
2  BloodPressure bp2 = bp1;
3  bp2.systolic = 145;
4  Console.WriteLine(bp1.systolic);
```

26. Починаючи з якої версії C# дозволено ініціалізувати поля структури при оголошенні? Що обов'язково потрібно зробити, якщо такі значення визначені?
27. Поясніть оператор with для структур. Чи змінює він оригінальний об'єкт?
28. Що таке readonly struct? Яку гарантію вона надає і яка від цього перевага з точки зору продуктивності?
29. Порівняйте struct і record struct. Що record struct додає автоматично і коли його варто використовувати замість звичайної структури?

30. Чи може структура мати конструктор без параметрів? Чи була ця можливість завжди доступна? Що відбувається, якщо структура має параметризований конструктор, але не має конструктора без параметрів, і ми пишемо new BloodPressure()?
31. Спроєктуйте структуру Temperature для зберігання температури тіла (у градусах Цельсія) як readonly struct. Додайте конструктор з параметром і метод IsFever(), що повертає true якщо температура перевищує 37.5.

3.5. Типи значень та типи посилань

32. Назвіть щонайменше по чотири приклади типів значень і типів посилань у C#. До якої категорії належить string?
33. Поясніть різницю між стеком і купою: яка область пам'яті для чого призначена, хто звільняє пам'ять у кожній з них?
34. Що виведе цей код і чому?

```
C#
1  Patient patient1 = new Patient("Іван", 45);
2  Patient patient2 = patient1;
3  patient2.name = "Марія";
4  Console.WriteLine(patient1.name);
```

35. Що таке **поверхове копіювання** (shallow copy)? Наведіть сценарій зі структурою, що містить поле-клас, де поверхове копіювання призводить до несподіваного результату.
36. Поясніть механізм boxing і unboxing. Які витрати вони спричиняють і чому?
37. Що станеться при виконанні такого коду?

```
C#
1  int age = 45;
2  object boxed = age;
3  double wrong = (double)boxed;
```

Де виникне помилка — під час компіляції чи виконання? Як це виправити?

38. Чому List<int> є кращим рішенням ніж ArrayList для зберігання цілих чисел? Яка роль generics у цьому контексті?
39. Порівняйте поведінку при присвоєнні int b = a та Patient b = a. Поясніть, що саме копіюється у кожному випадку, і намалюйте (або опишіть словами) стан пам'яті після кожного присвоєння.

Розділ 4. Об'єктно-орієнтоване програмування

4.1. Успадкування

Успадкування (inheritance) — один із трьох фундаментальних принципів об'єктно-орієнтованого програмування поряд із інкапсуляцією та поліморфізмом. Воно дозволяє одному класу перейняти функціональність іншого: поля, властивості та методи базового класу стають доступними в похідному без повторного написання коду. Це не просто зручність — це архітектурний інструмент для моделювання реальних відносин між сутностями.

Уявімо клінічну систему, де є різні типи людей: пацієнти, лікарі, адміністратори. Всі вони мають спільні атрибути (ім'я, вік) і спільні дії (ідентифікація, виведення інформації). Замість того, щоб дублювати цей код у кожному класі окремо, ми виносимо спільне в базовий клас Person і успадковуємо від нього.

Базовий та похідний клас

Клас, від якого успадковують, називається **базовим класом** (base class, superclass, батьківський клас). Клас, який успадковує, — **похідним класом** (derived class, subclass, дочірній клас). Для позначення успадкування після назви похідного класу через двокрапку вказується базовий клас:

```
C#
1  class ПохіднийКлас : БазовийКлас
2  {
3      // нові члени похідного класу
4  }
```

Нехай у нас є базовий клас Person:

```
C#
 1  class Person
 2  {
 3      public string Name { get; set; }
 4      public int Age { get; set; }
 5
 6      public void Print()
 7      {
 8          Console.WriteLine($"{Name}, {Age} років");
```

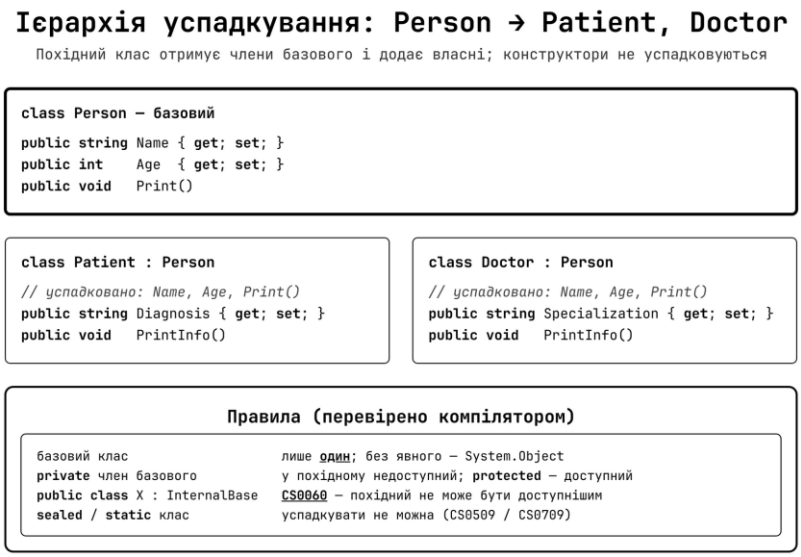
```
 9      }
10  }
```

Тоді Patient і Doctor — похідні класи від Person:

```
C#
 1  using System;
 2
 3  Person p = new Person { Name = "Невідомий", Age = 0 };
 4  p.Print();
 5
 6  Patient patient = new Patient { Name = "Іван Петренко", Age = 45, Diagnosis =
    "Гіпертонія" };
 7  patient.Print();      // успадкований метод Print()
 8  patient.PrintInfo();  // власний метод класу Patient
 9
10  Doctor doctor = new Doctor { Name = "Олена Коваль", Age = 38, Specialization =
    "Кардіологія" };
11  doctor.Print();
12  doctor.PrintInfo();
13
14  class Person
15  {
16      public string Name { get; set; } = "";
17      public int Age { get; set; }
18
19      public void Print()
20      {
21          Console.WriteLine($"{Name}, {Age} років");
22      }
23  }
24
25  class Patient : Person
26  {
27      public string Diagnosis { get; set; } = "";
28
29      public void PrintInfo()
30      {
31          Console.WriteLine($"Пацієнт: {Name}, {Age} р. | Діагноз: {Diagnosis}");
32      }
33  }
34
35  class Doctor : Person
36  {
37      public string Specialization { get; set; } = "";
38
39      public void PrintInfo()
40      {
41          Console.WriteLine($"Лікар: {Name}, {Age} р. | Спеціалізація:
    {Specialization}");
42      }
43  }
```

Клас Patient успадкував від Person властивості Name, Age та метод Print() — і не потрібно було їх писати заново. Додатково Patient

визначає власну властивість Diagnosis та метод PrintInfo(). Те саме для Doctor.



Ієрархія успадкування: Person → Patient, Doctor

Відношення is-a та has-a

Успадкування моделює відношення **is-a** («є»): об'єкт похідного класу є об'єктом базового класу. Patient is-a Person — пацієнт є людиною. Це означає, що там, де очікується Person, можна передати Patient:

```
C#
 1  using System;
 2
 3  Person p1 = new Patient { Name = "Іван Петренко", Age = 45, Diagnosis =
    "Гіпертонія" };
 4  Person p2 = new Doctor  { Name = "Олена Коваль",  Age = 38, Specialization =
    "Кардіологія" };
 5
 6  p1.Print(); // метод Person.Print()
 7  p2.Print(); // метод Person.Print()
 8
 9  class Person
10  {
11      public string Name { get; set; } = "";
12      public int Age { get; set; }
13      public void Print() => Console.WriteLine($"{Name}, {Age} років");
14  }
```

```
15
16  class Patient : Person
17  {
18      public string Diagnosis { get; set; } = "";
19  }
20
21  class Doctor : Person
22  {
23      public string Specialization { get; set; } = "";
24  }
```

Змінна типу Person може зберігати будь-який об'єкт з ієрархії — це основа поліморфізму, який детально розглядається в наступних розділах.

Відношення is-a слід відрізняти від **has-a** («має»): якщо одна сутність містить іншу як компонент, використовується **композиція**, а не успадкування. Наприклад, клас Appointment (прийом) має Patient і Doctor — це has-a, тому Appointment не має від них успадковувати. Чіткий вибір між is-a і has-a є ознакою правильно спроектованої об'єктної моделі.

Клас Object як базовий для всіх

За замовчуванням усі класи в C# неявно успадковують від базового класу System.Object (або просто object). Це означає, що навіть клас Person, для якого не вказано явного базового класу, насправді успадковує від Object. А Patient і Doctor перебувають у ланцюжку: Patient → Person → Object.

Саме тому будь-який об'єкт у C# завжди має такі методи:

- ToString() — рядкове представлення об'єкта
- Equals(object obj) — перевірка рівності
- GetHashCode() — хеш-код для колекцій
- GetType() — тип об'єкта під час виконання

```
C#
 1  using System;
 2
 3  Patient patient = new Patient { Name = "Іван Петренко", Age = 45 };
 4
 5  Console.WriteLine(patient.GetType().Name); // Patient
 6  Console.WriteLine(patient.ToString());     // Patient (за замовчуванням — назва
    типу)
 7
 8  class Person
 9  {
10      public string Name { get; set; } = "";
11      public int Age { get; set; }
12  }
```

```
13
14  class Patient : Person
15  {
16      public string Diagnosis { get; set; } = "";
17  }
```

Обмеження успадкування

C# накладає кілька важливих обмежень:

- **Одиночне успадкування** — клас може мати лише один безпосередній базовий клас. Множинне успадкування класів не підтримується (на відміну від інтерфейсів, які розглядаються в наступних розділах).
- **Рівні доступу** — похідний клас не може бути доступнішим за базовий: якщо базовий клас має модифікатор internal, похідний може бути internal (а вкладений — ще й private чи protected), але не public — інакше помилка CS0060. Якщо класи в різних збірках, похідний клас може успадковувати лише від public-класу.
- **Статичний клас** — від статичного класу неможливо успадковувати.
- **Запечатаний клас** — клас з модифікатором sealed не допускає спадкоємців.

```
C#
1  sealed class AdministratorAccount
2  {
3      // Від цього класу не можна успадковувати
4  }
```

Модифікатор sealed корисний там, де необхідно запобігти зміні поведінки класу через похідні класи — наприклад, для критичних класів безпеки або коли клас спроектований настільки специфічно, що розширення може порушити його коректність. У стандартній бібліотеці .NET клас String є sealed саме з міркувань безпеки та продуктивності.

Доступ до членів базового класу

Похідний клас успадковує члени базового, але рівні доступу залишаються в силі. Розглянемо ситуацію, коли Person має приватне поле:

```
C#
 1  using System;
```

```
2
3  Doctor doctor = new Doctor { Age = 38, Specialization = "Кардіологія" };
4  doctor.SetName("Олена Коваль");
5  doctor.PrintInfo();
6
7  class Person
8  {
9      private string _name = ""; // приватне поле — недоступне ззовні
10
11     public string Name
12     {
13         get { return _name; }
14         set { _name = value; }
15     }
16
17     public int Age { get; set; }
18
19     public void Print() => Console.WriteLine($"{Name}, {Age} років");
20 }
21
22 class Doctor : Person
23 {
24     public string Specialization { get; set; } = "";
25
26     public void SetName(string name)
27     {
28         // _name = name; // Помилка! _name — private, доступний лише в Person
29         Name = name;    // Правильно: через public-властивість
30     }
31
32     public void PrintInfo() => Console.WriteLine($"Лікар: {Name}, {Age} р. |
   {Specialization}");
33 }
```

Похідний клас може звертатися лише до членів базового класу з такими модифікаторами:

- public — доступний усюди
- protected — доступний у базовому та всіх похідних класах
- internal — доступний у межах однієї збірки
- protected internal — union: збірка або похідний клас
- private protected — intersection: та сама збірка і похідний клас

Члени з private залишаються виключно в тому класі, де оголошені.

Модифікатор protected

Модифікатор protected створений спеціально для ієрархій успадкування. Він дозволяє базовому класу відкрити доступ до своїх членів для всіх похідних класів, але закрити їх від зовнішнього світу. Це золота середина між public (надто відкрито) і private (повністю закрито):

```C#
using System;

Patient patient = new Patient("Іван Петренко", 45, "Гіпертонія");
patient.PrintInfo();

// patient.medicalRecordId — помилка: protected
// patient._internalCode  — помилка: private

class Person
{
    private string _internalCode;          // тільки Person
    protected string medicalRecordId;      // Person та похідні

    public string Name { get; set; }
    public int Age { get; set; }

    public Person(string name, int age)
    {
        Name = name;
        Age  = age;
        _internalCode   = $"INT-{age}";
        medicalRecordId = $"MR-{name[0]}{age}";
    }
}

class Patient : Person
{
    public string Diagnosis { get; set; }

    public Patient(string name, int age, string diagnosis) : base(name, age)
    {
        Diagnosis = diagnosis;
    }

    public void PrintInfo()
    {
        // medicalRecordId доступний — він protected
        Console.WriteLine($"[{medicalRecordId}] Пацієнт: {Name}, {Age} р. | {Diagnosis}");
        // _internalCode — НЕ доступний
    }
}
```

Поле medicalRecordId оголошене як protected: клас Patient може його читати і використовувати, але зовнішній код (patient.medicalRecordId) отримає помилку компіляції. Поле _internalCode залишається private — недоступне навіть для Patient.

Ключове слово base

Ключове слово base дозволяє явно звернутися до членів **безпосереднього** базового класу: викликати конструктор або метод. Це особливо важливо, коли похідний клас перевизначає метод базового і водночас хоче використати оригінальну реалізацію.

```C#
using System;

Doctor doctor = new Doctor("Олена Коваль", 38, "Кардіологія", "UA-12345");
doctor.Print();     // базовий метод
doctor.PrintInfo(); // розширений метод

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Person(string name, int age)
    {
        Name = name;
        Age  = age;
    }

    public void Print()
    {
        Console.WriteLine($"{Name}, {Age} років");
    }
}

class Doctor : Person
{
    public string Specialization { get; set; }
    public string LicenseNumber  { get; set; }

    public Doctor(string name, int age, string specialization, string licenseNumber)
        : base(name, age)       // викликаємо конструктор Person
    {
        Specialization = specialization;
        LicenseNumber  = licenseNumber;
    }

    public void PrintInfo()
    {
        base.Print();           // викликаємо метод Person.Print()
        Console.WriteLine($"  Спеціалізація: {Specialization} | Ліцензія: {LicenseNumber}");
    }
}
```

У конструкторі Doctor вираз : base(name, age) передає ім'я та вік конструктору Person — немає потреби дублювати код ініціалізації. У методі PrintInfo() виклик base.Print() виконує базову реалізацію, після чого виводить додаткову інформацію.

Конструктори у похідних класах

Конструктори **не передаються** похідному класу при успадкуванні — це принципова відмінність від методів і властивостей. Кожен клас повинен мати власні конструктори.

Якщо в базовому класі є лише конструктори з параметрами (і немає конструктора без параметрів), то кожен конструктор похідного класу зобов'язаний явно викликати один із конструкторів базового через base(...):

```C#
using System;

// Коректне визначення
Patient patient = new Patient("Марія Сидоренко", 32, "Бронхіт");
patient.PrintInfo();

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    // Є лише параметризований конструктор — без параметрів недоступний
    public Person(string name, int age)
    {
        Name = name;
        Age  = age;
    }
}

class Patient : Person
{
    public string Diagnosis { get; set; }

    // Обов'язково викликаємо base(name, age)
    public Patient(string name, int age, string diagnosis)
        : base(name, age)
    {
        Diagnosis = diagnosis;
    }

    public void PrintInfo() =>
        Console.WriteLine($"Пацієнт: {Name}, {Age} р. | Діагноз: {Diagnosis}");
}
```

Якщо Patient не викличе base(name, age), компілятор видасть помилку — він не знає, як ініціалізувати успадковані властивості Name і Age.

Якщо ж базовий клас **має** конструктор без параметрів, похідний клас може не викликати base() явно — компілятор підставить його неявно. У такому разі наступні два конструктори еквівалентні:

```C#
// Явний виклик
public Patient(string diagnosis) : base()
{
    Diagnosis = diagnosis;
}

```

```
 7  // Неявний виклик (компілятор додасть base() автоматично)
 8  public Patient(string diagnosis)
 9  {
10      Diagnosis = diagnosis;
11  }
```

Порядок виклику конструкторів

При створенні об'єкта похідного класу конструктори викликаються в строго визначеному порядку: **від найбазовішого до найпохіднішого**. Тіла конструкторів виконуються у зворотному порядку відносно делегування — спочатку відпрацьовують більш загальні класи, потім — специфічні.

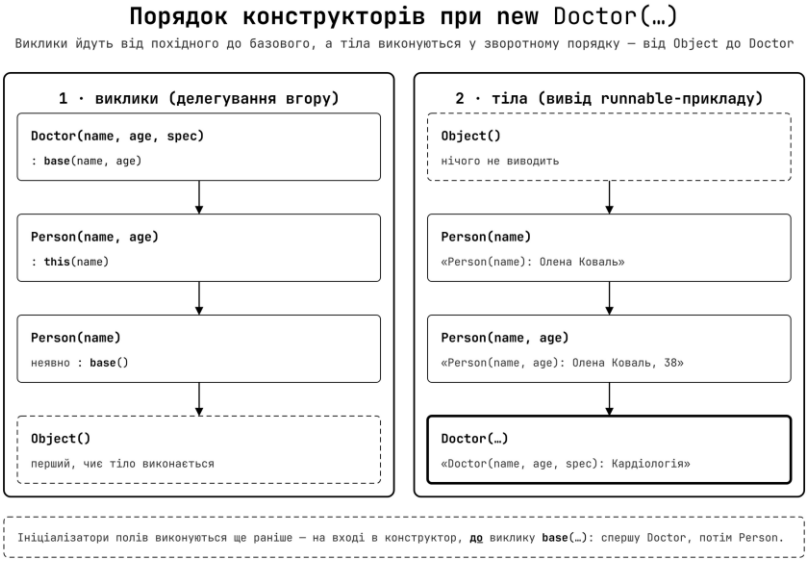
Розглянемо приклад з ланцюжком:

```
C#
 1  using System;
 2
 3  Doctor doctor = new Doctor("Олена Коваль", 38, "Кардіологія");
 4
 5  class Person
 6  {
 7      public string Name { get; set; }
 8      public int Age { get; set; }
 9
10      public Person(string name)
11      {
12          Name = name;
13          Console.WriteLine($"Person(name): {name}");
14      }
15
16      public Person(string name, int age) : this(name)
17      {
18          Age = age;
19          Console.WriteLine($"Person(name, age): {name}, {age}");
20      }
21  }
22
23  class Doctor : Person
24  {
25      public string Specialization { get; set; }
26
27      public Doctor(string name, int age, string specialization)
28          : base(name, age)
29      {
30          Specialization = specialization;
31          Console.WriteLine($"Doctor(name, age, spec): {specialization}");
32      }
33  }
```

Виконання розгортається так:

1. new Doctor(...) → викликається Doctor(name, age, spec), але спочатку делегує base(name, age)

2. Person(name, age) → не виконується одразу, делегує this(name)
3. Person(name) → делегує неявно Object()
4. **Виконується** Object() — найбазовіший конструктор
5. **Виконується** тіло Person(name) → виводить Person(name): Олена Коваль
6. **Виконується** тіло Person(name, age) → виводить Person(name, age): Олена Коваль, 38
7. **Виконується** тіло Doctor(...) → виводить Doctor(name, age, spec): Кардіологія



Порядок виклику конструкторів у ланцюжку успадкування

Цей порядок гарантує, що до моменту виконання конструктора похідного класу базовий клас вже повністю ініціалізований. Якби Doctor намагався звернутися до Name у своєму конструкторі — властивість вже була б встановлена конструктором Person.

Запечатаний клас: sealed

Модифікатор sealed повністю забороняє успадкування від класу. Це явна архітектурна декларація: «цей клас спроектований як кінцевий, розширення через успадкування не передбачено».

```
C#
 1  using System;
 2
 3  // Звичайне використання — працює
 4  CertifiedSurgeon surgeon = new CertifiedSurgeon("Андрій Мельник",
    "Кардіохірургія");
 5  surgeon.PrintInfo();
 6
 7  // class SpecialSurgeon : CertifiedSurgeon { }  // Помилка компіляції!
 8
 9  class Doctor
10  {
11      public string Name { get; set; }
12
13      public Doctor(string name) { Name = name; }
14
15      public virtual void PrintInfo() =>
16          Console.WriteLine($"Лікар: {Name}");
17  }
18
19  sealed class CertifiedSurgeon : Doctor
20  {
21      public string Specialization { get; set; }
22
23      public CertifiedSurgeon(string name, string specialization) : base(name)
24      {
25          Specialization = specialization;
26      }
27
28      public override void PrintInfo() =>
29          Console.WriteLine($"Сертифікований хірург: {Name} | {Specialization}");
30  }
```

sealed клас може сам успадковувати від інших класів (як CertifiedSurgeon від Doctor), але не дозволяє мати власних нащадків. У стандартній бібліотеці .NET таким чином оголошені String, StringBuilder та багато інших класів — їх не можна розширити через успадкування.

Окрім захисту архітектури, sealed дає компілятору можливість генерувати ефективніший код: якщо клас запечатаний, віртуальні виклики через змінну цього типу JIT-компілятор може замінити прямими (девіртуалізація), бо перевизначень гарантовано немає. Сам компілятор C# такої оптимізації не робить — її виконує середовище виконання.

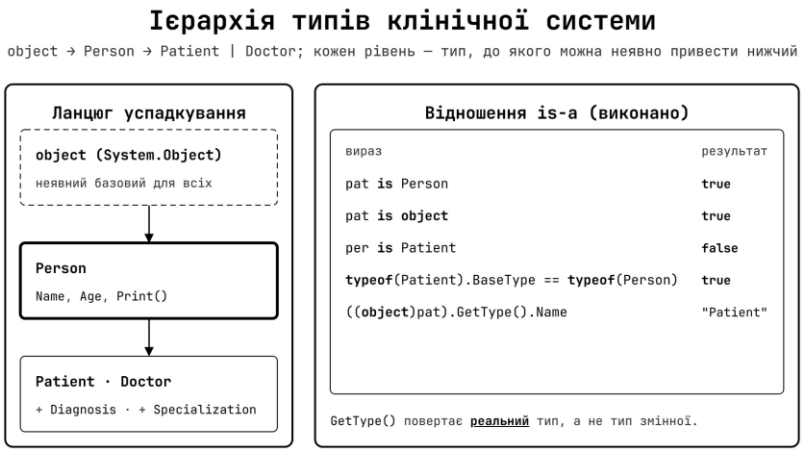
4.2. Перетворення типів

У попередньому розділі ми говорили про успадкування та відношення is-a між класами. З цим відношенням безпосередньо пов'язана можливість **перетворення типів** у ієрархії класів: об'єкт похідного класу в будь-який момент може бути представлений як об'єкт базового класу, і навпаки — за певних умов. Розуміння правил таких перетворень є обов'язковим для написання гнучкого коду, що працює з ієрархіями об'єктів.

Розглянемо клінічну ієрархію класів:

```
C#
 1  class Person
 2  {
 3      public string Name { get; set; }
 4      public int Age { get; set; }
 5
 6      public Person(string name, int age)
 7      {
 8          Name = name;
 9          Age  = age;
10      }
11
12      public void Print()
13      {
14          Console.WriteLine($"Особа: {Name}, {Age} р.");
15      }
16  }
17
18  class Patient : Person
19  {
20      public string Diagnosis { get; set; }
21
22      public Patient(string name, int age, string diagnosis) : base(name, age)
23      {
24          Diagnosis = diagnosis;
25      }
26  }
27
28  class Doctor : Person
29  {
30      public string Specialization { get; set; }
31
32      public Doctor(string name, int age, string specialization) : base(name,
    age)
33      {
34          Specialization = specialization;
35      }
36  }
```

Ланцюг успадкування: Object (неявно) → Person → Patient | Doctor. Базові типи знаходяться вгорі ієрархії, похідні — внизу.



Ієрархія типів у клінічній системі

Висхідні перетворення. Upcasting

Об'єкт похідного типу одночасно є об'єктом базового типу. Пацієнт (Patient) — це людина (Person), тому посилання на Patient можна зберегти у змінній типу Person. Таке перетворення від похідного до базового типу називається **висхідним** (upcasting) і відбувається **неявно** — без жодного додаткового синтаксису:

```C#
using System;

Patient patient = new Patient("Іван Петренко", 45, "Гіпертонія");
Person  person  = patient;   // upcasting: Patient → Person (неявно)

person.Print();              // Особа: Іван Петренко, 45 р.
Console.WriteLine(person.Name); // Іван Петренко

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age) { Name = name; Age = age; }
    public void Print() => Console.WriteLine($"Особа: {Name}, {Age} р.");
}

class Patient : Person
{
    public string Diagnosis { get; set; }
    public Patient(string name, int age, string diag) : base(name, age)
    { Diagnosis = diag; }
```

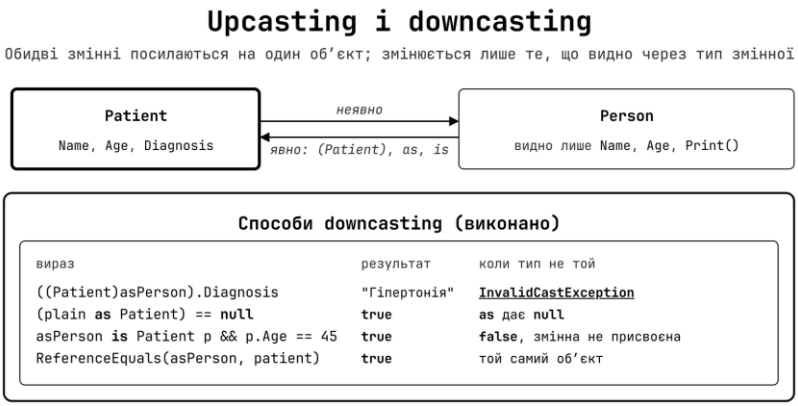
```
22  }
```

Змінні patient і person вказують на **один і той самий об'єкт** у пам'яті. Але через змінну person доступна лише та частина функціоналу, яку визначає тип Person — властивість Diagnosis буде недоступна.

Висхідне перетворення відбувається і під час присвоєння до типу object, оскільки він є базовим для всіх:

```
C#
 1  using System;
 2
 3  object obj1 = new Patient("Іван Петренко", 45, "Гіпертонія"); // Patient →
    object
 4  object obj2 = new Doctor("Олена Коваль", 38, "Кардіологія");  // Doctor →
    object
 5  object obj3 = new Person("Сергій Бойко", 52);                 // Person →
    object
 6
 7  Console.WriteLine(obj1.GetType().Name); // Patient
 8  Console.WriteLine(obj2.GetType().Name); // Doctor
 9  Console.WriteLine(obj3.GetType().Name); // Person
10
11  class Person
12  {
13      public string Name { get; set; }
14      public int Age { get; set; }
15      public Person(string name, int age) { Name = name; Age = age; }
16  }
17  class Patient : Person
18  {
19      public string Diagnosis { get; set; }
20      public Patient(string name, int age, string diag) : base(name, age) {
    Diagnosis = diag; }
21  }
22  class Doctor : Person
23  {
24      public string Specialization { get; set; }
25      public Doctor(string name, int age, string spec) : base(name, age) {
    Specialization = spec; }
26  }
```

Зверніть увагу: метод GetType() завжди повертає **реальний тип об'єкта**, незалежно від типу змінної, що його зберігає.



Upcasting та Downcasting у ієрархії класів

Низхідні перетворення. Downcasting

Якщо upcasting — це завжди безпечно і неявно, то зворотна операція — **низхідне перетворення** (downcasting) від базового до похідного типу — вимагає **явного вказання типу** і несе в собі ризик. Не кожна людина є пацієнтом, тому компілятор не може самостійно вирішити, чи допустиме таке перетворення:

```
C#
 1  using System;
 2
 3  Patient patient1 = new Patient("Іван Петренко", 45, "Гіпертонія");
 4  Person  person   = patient1;              // upcasting (неявне)
 5
 6  Patient patient2 = (Patient)person;       // downcasting (явне)
 7  Console.WriteLine(patient2.Diagnosis);    // Гіпертонія
 8
 9  class Person
10  {
11      public string Name { get; set; }
12      public int Age { get; set; }
13      public Person(string name, int age) { Name = name; Age = age; }
14  }
15  class Patient : Person
16  {
17      public string Diagnosis { get; set; }
18      public Patient(string name, int age, string diag) : base(name, age) {
    Diagnosis = diag; }
19  }
```

Якщо ж реальний тип об'єкта не відповідає типу, до якого відбувається приведення, виникає виняток InvalidCastException під час виконання програми:

```C#
using System;

try
{
    Person person = new Person("Сергій Бойко", 52); // звичайна Person, не Patient
    Patient patient = (Patient)person;              // InvalidCastException!
    Console.WriteLine(patient.Diagnosis);
}
catch (InvalidCastException ex)
{
    Console.WriteLine($"Помилка перетворення: {ex.Message}");
}

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age) { Name = name; Age = age; }
}
class Patient : Person
{
    public string Diagnosis { get; set; }
    public Patient(string name, int age, string diag) : base(name, age) { Diagnosis = diag; }
}
```

Підступність у тому, що компілятор не завжди може виявити некоректне перетворення — код скомпілюється, але впаде під час виконання. Тому для downcasting завжди потрібна додаткова перевірка.

Оператор as

Оператор as намагається виконати перетворення і у разі невдачі повертає null замість викидання винятку. Це безпечна альтернатива явному приведенню:

```C#
using System;

Person person1 = new Patient("Іван Петренко", 45, "Гіпертонія");
Person person2 = new Doctor("Олена Коваль", 38, "Кардіологія");

Patient? p = person1 as Patient;
Doctor?  d = person1 as Doctor;

if (p != null)
    Console.WriteLine($"Пацієнт: {p.Diagnosis}");  // Гіпертонія
```

```
11  else
12      Console.WriteLine("Не є пацієнтом");
13
14  if (d != null)
15      Console.WriteLine($"Лікар: {d.Specialization}");
16  else
17      Console.WriteLine("Не є лікарем");              // Не є лікарем
18
19  class Person
20  {
21      public string Name { get; set; }
22      public int Age { get; set; }
23      public Person(string name, int age) { Name = name; Age = age; }
24  }
25  class Patient : Person
26  {
27      public string Diagnosis { get; set; }
28      public Patient(string name, int age, string diag) : base(name, age) {
    Diagnosis = diag; }
29  }
30  class Doctor : Person
31  {
32      public string Specialization { get; set; }
33      public Doctor(string name, int age, string spec) : base(name, age) {
    Specialization = spec; }
34  }
```

Тип результату після as завжди nullable (Patient?, Doctor?) — тобто може містити або об'єкт, або null. Перевірка на null перед використанням є обов'язковою.

Оператор is

Оператор is перевіряє, чи є об'єкт представником певного типу, і повертає true або false. Починаючи з C# 7, він підтримує **перевірку з одночасним перетворенням** (pattern matching): якщо перевірка успішна, об'єкт автоматично приводиться до потрібного типу і зберігається у нову змінну:

```
C#
 1  using System;
 2
 3  Person person = new Patient("Марія Сидоренко", 32, "Бронхіт");
 4
 5  if (person is Patient patient)
 6  {
 7      // patient вже є типом Patient — не потрібне явне приведення
 8      Console.WriteLine($"Пацієнт: {patient.Name}, діагноз:
    {patient.Diagnosis}");
 9  }
10  else
11  {
12      Console.WriteLine("Не є пацієнтом");
13  }
```

```
14
15  // Без захоплення змінної — просто перевірка
16  if (person is Doctor)
17      Console.WriteLine("Це лікар");
18  else
19      Console.WriteLine("Це не лікар"); // виведе це
20
21  class Person
22  {
23      public string Name { get; set; }
24      public int Age { get; set; }
25      public Person(string name, int age) { Name = name; Age = age; }
26  }
27  class Patient : Person
28  {
29      public string Diagnosis { get; set; }
30      public Patient(string name, int age, string diag) : base(name, age) {
    Diagnosis = diag; }
31  }
32  class Doctor : Person
33  {
34      public string Specialization { get; set; }
35      public Doctor(string name, int age, string spec) : base(name, age) {
    Specialization = spec; }
36  }
```

Вираз person is Patient patient робить одразу дві речі: перевіряє тип і, якщо він відповідає, зберігає вже приведений об'єкт у змінну patient. Це лаконічніше і безпечніше, ніж окрема перевірка + явне приведення.

Pattern matching з switch

Оператор switch підтримує pattern matching по типах у case з C# 7, а у формі switch-виразу (p switch { … }) — з C# 8. Це зручно, коли потрібно обробити кілька різних типів з однієї ієрархії:

```
C#
 1  using System;
 2
 3  Person[] staff = {
 4      new Patient("Іван Петренко", 45, "Гіпертонія"),
 5      new Doctor("Олена Коваль", 38, "Кардіологія"),
 6      new Person("Адміністратор", 30),
 7  };
 8
 9  foreach (Person p in staff)
10  {
11      string info = p switch
12      {
13          Patient pat => $"[Пацієнт] {pat.Name} — {pat.Diagnosis}",
14          Doctor  doc => $"[Лікар]   {doc.Name} — {doc.Specialization}",
15          Person  per => $"[Особа]   {per.Name}",
16      };
17      Console.WriteLine(info);
18  }
```

```
19
20  class Person
21  {
22      public string Name { get; set; }
23      public int Age { get; set; }
24      public Person(string name, int age) { Name = name; Age = age; }
25  }
26  class Patient : Person
27  {
28      public string Diagnosis { get; set; }
29      public Patient(string name, int age, string diag) : base(name, age) {
    Diagnosis = diag; }
30  }
31  class Doctor : Person
32  {
33      public string Specialization { get; set; }
34      public Doctor(string name, int age, string spec) : base(name, age) {
    Specialization = spec; }
35  }
```

Важливий нюанс: гілки switch перевіряються **зверху донизу**, тому більш специфічні типи (Patient, Doctor) мають стояти **перед** більш загальним (Person). Якщо поставити Person per першим, він захопить усі об'єкти, і гілки Patient та Doctor стануть недосяжними — компілятор не дозволить такий код (помилка CS8510, перевірено на .NET 10).

Switch з pattern matching є найчистішим і найбезпечнішим способом роботи з поліморфними колекціями, оскільки компілятор перевіряє повноту гілок і видає попередження CS8509, якщо вираз може не збігтися з жодною гілкою. Зауважимо: для ієрархії класів компілятор не знає всіх можливих нащадків (їх можна додати будь-коли), тому повним switch-вираз вважається лише з завершальною гілкою для базового типу або _.

4.3. Віртуальні методи та властивості

При успадкуванні нерідко виникає необхідність змінити в похідному класі поведінку методу, успадкованого від базового. Стандартний метод неможливо перевизначити у нащадку — для цього базовий клас має явно **дозволити** перевизначення за допомогою ключового слова virtual. Такі методи та властивості називаються **віртуальними**.

У похідному класі метод, що замінює реалізацію базового, позначається ключовим словом override. Сигнатура перевизначеного методу — ім'я та параметри — повинна точно збігатися з сигнатурою базового віртуального методу.

Оголошення virtual та override

Розглянемо клінічну ієрархію: базовий клас Person з віртуальним методом Print(), та похідні класи Patient і Doctor, кожен з яких перевизначає цей метод по-своєму:

```C#
using System;

Person p  = new Person("Сергій Бойко", 52);
Patient pt = new Patient("Іван Петренко", 45, "Гіпертонія");
Doctor  dr = new Doctor("Олена Коваль", 38, "Кардіологія");

p.Print();   // Person.Print()
pt.Print();  // Patient.Print() — перевизначений
dr.Print();  // Doctor.Print()  — перевизначений

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Person(string name, int age) { Name = name; Age = age; }

    public virtual void Print()
    {
        Console.WriteLine($"Особа: {Name}, {Age} р.");
    }
}

class Patient : Person
{
    public string Diagnosis { get; set; }

    public Patient(string name, int age, string diagnosis) : base(name, age)
    {
        Diagnosis = diagnosis;
    }

    public override void Print()
    {
        Console.WriteLine($"Пацієнт: {Name}, {Age} р. | Діагноз: {Diagnosis}");
    }
}

class Doctor : Person
{
    public string Specialization { get; set; }

    public Doctor(string name, int age, string spec) : base(name, age)
    {
        Specialization = spec;
    }

    public override void Print()
    {
        Console.WriteLine($"Лікар: {Name}, {Age} р. | {Specialization}");
    }
```

```
52  }
```

Якщо похідний клас не перевизначає віртуальний метод — використовується реалізація базового класу. Тобто virtual лише **дозволяє** перевизначення, але не зобов'язує до нього.

При перевизначенні слід враховувати кілька обмежень:

— Модифікатор доступу перевизначеного методу повинен **збігатися** з модифікатором базового (public virtual → public override).

— Неможливо оголосити virtual або перевизначити static-метод.

— Неможливо перевизначити метод без virtual у базовому класі (для цього є приховування, яке розглядається в розділі 4.4).

Поліморфізм: виклик за реальним типом об'єкта

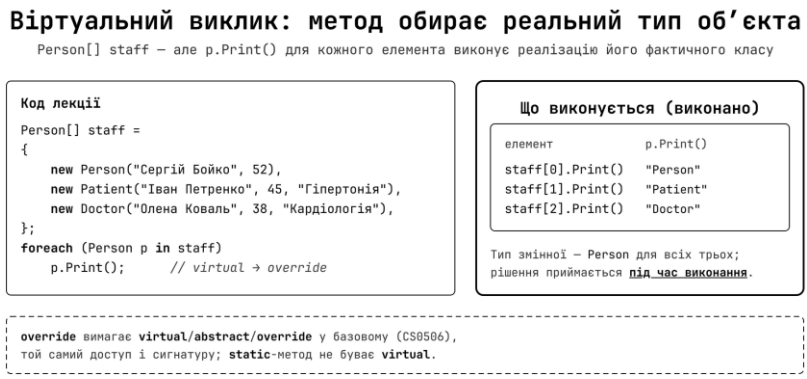
Справжня сила віртуальних методів розкривається через **поліморфізм** — здатність коду однаково звертатись до об'єктів різних типів через змінну базового класу, але при цьому кожен об'єкт виконує свою власну реалізацію методу.

Коли через змінну типу Person викликається метод Print(), середовище виконання визначає **реальний тип об'єкта** під час виконання програми і викликає відповідну реалізацію — Patient.Print() для пацієнта, Doctor.Print() для лікаря:

```
C#
 1  using System;
 2
 3  Person[] staff = {
 4      new Person("Адміністратор", 30),
 5      new Patient("Іван Петренко", 45, "Гіпертонія"),
 6      new Doctor("Олена Коваль", 38, "Кардіологія"),
 7      new Patient("Марія Сидоренко", 32, "Бронхіт"),
 8  };
 9
10  foreach (Person p in staff)
11  {
12      p.Print(); // кожен об'єкт виконує свою реалізацію
13  }
14
15  class Person
16  {
17      public string Name { get; set; }
18      public int Age { get; set; }
19      public Person(string name, int age) { Name = name; Age = age; }
20      public virtual void Print() =>
21          Console.WriteLine($"Особа: {Name}");
22  }
23
24  class Patient : Person
```

```
25  {
26      public string Diagnosis { get; set; }
27      public Patient(string name, int age, string diag) : base(name, age) {
    Diagnosis = diag; }
28      public override void Print() =>
29          Console.WriteLine($"Пацієнт: {Name} | {Diagnosis}");
30  }
31
32  class Doctor : Person
33  {
34      public string Specialization { get; set; }
35      public Doctor(string name, int age, string spec) : base(name, age) {
    Specialization = spec; }
36      public override void Print() =>
37          Console.WriteLine($"Лікар: {Name} | {Specialization}");
38  }
```

Масив оголошено як Person[], але кожен елемент зберігає об'єкт свого реального типу. Під час виклику p.Print() C# не дивиться на тип змінної — він шукає реалізацію методу у фактичному типі об'єкта. Цей механізм називається **пізнім зв'язуванням** (late binding або dynamic dispatch).



Virtual dispatch — виклик методу за реальним типом об'єкта

Поліморфізм є фундаментальним принципом ООП: він дозволяє писати код, який працює з абстракцією (Person), не знаючи конкретних типів — і при цьому кожен об'єкт поводиться відповідно до своєї природи.

Звернення до базового методу через base

Перевизначений метод у похідному класі може викликати реалізацію базового класу через ключове слово base. Це дозволяє **розширити** поведінку, а не повністю замінити її:

```
C#
 1  using System;
 2
 3  Doctor doctor = new Doctor("Олена Коваль", 38, "Кардіологія");
 4  doctor.Print();
 5
 6  class Person
 7  {
 8      public string Name { get; set; }
 9      public int Age { get; set; }
10      public Person(string name, int age) { Name = name; Age = age; }
11
12      public virtual void Print()
13      {
14          Console.WriteLine($"{Name}, {Age} p.");
15      }
16  }
17
18  class Doctor : Person
19  {
20      public string Specialization { get; set; }
21
22      public Doctor(string name, int age, string spec) : base(name, age)
23      {
24          Specialization = spec;
25      }
26
27      public override void Print()
28      {
29          base.Print();                                   // виводить: Олена
    Коваль, 38 p.
30          Console.WriteLine($"  Спеціалізація: {Specialization}"); // додає рядок
31      }
32  }
```

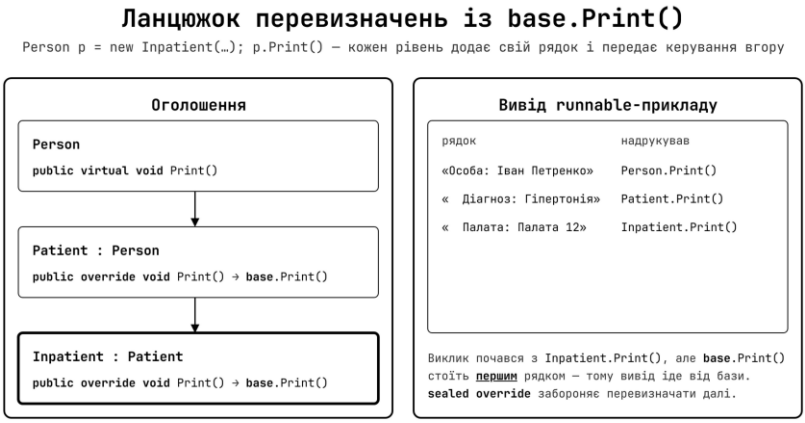
Виклик base.Print() запускає саме реалізацію класу Person, навіть якщо викликається через об'єкт Doctor. Це типовий патерн для поступового збагачення поведінки в ланцюжку успадкування.

Ланцюжок перевизначень

Перевизначення може тривати через кілька рівнів ієрархії. Кожен похідний клас може перевизначити метод далі — або знову використати base:

```
C#
 1  using System;
```

```
2
3  Person p   = new Inpatient("Іван Петренко", 45, "Гіпертонія", "Палата 12");
4  p.Print();
5
6  class Person
7  {
8      public string Name { get; set; }
9      public int Age { get; set; }
10     public Person(string name, int age) { Name = name; Age = age; }
11     public virtual void Print() => Console.WriteLine($"Особа: {Name}");
12 }
13
14 class Patient : Person
15 {
16     public string Diagnosis { get; set; }
17     public Patient(string name, int age, string diag) : base(name, age) {
   Diagnosis = diag; }
18
19     public override void Print()
20     {
21         base.Print();
22         Console.WriteLine($"  Діагноз: {Diagnosis}");
23     }
24 }
25
26 class Inpatient : Patient  // стаціонарний пацієнт
27 {
28     public string Room { get; set; }
29     public Inpatient(string name, int age, string diag, string room)
30         : base(name, age, diag) { Room = room; }
31
32     public override void Print()
33     {
34         base.Print();                          // викликає Patient.Print() →
   Person.Print()
35         Console.WriteLine($"  Палата: {Room}");
36     }
37 }
```



Ланцюжок перевизначень virtual → override

При виклику p.Print() через змінну типу Person, де реальний тип Inpatient, спочатку виконується Inpatient.Print(), яка через base.Print() викликає Patient.Print(), яка у свою чергу через base.Print() викликає Person.Print().

Перевизначення властивостей

Так само як і методи, властивості можна оголошувати virtual і перевизначати в нащадках за допомогою override. Це корисно, коли похідний клас має накладати додаткові обмеження або змінювати логіку доступу:

```
C#
 1  using System;
 2
 3  Person person = new Person("Сергій Бойко");
 4  person.Age = 5;
 5  Console.WriteLine($"Person.Age = {person.Age.ToString()}");   // 5
 6
 7  Doctor doctor = new Doctor("Олена Коваль", "Кардіологія");
 8  doctor.Age = 16;
 9  Console.WriteLine($"Doctor.Age = {doctor.Age.ToString()}");   // 18 (мін. для
    лікаря)
10  doctor.Age = 35;
11  Console.WriteLine($"Doctor.Age = {doctor.Age.ToString()}");   // 35
12
13  class Person
14  {
15      private int _age = 1;
16
```

```
    public virtual int Age
    {
        get => _age;
        set { if (value > 0 && value < 120) _age = value; }
    }

    public string Name { get; set; }
    public Person(string name) { Name = name; }
}

class Doctor : Person
{
    public string Specialization { get; set; }

    public Doctor(string name, string spec) : base(name)
    {
        Specialization = spec;
        base.Age = 25; // вік за замовчуванням для лікаря
    }

    public override int Age
    {
        get => base.Age;
        set
        {
            // Лікар повинен мати не менше 25 років (мінімум для спеціаліста)
            if (value >= 25 && value < 120) base.Age = value;
        }
    }
}
```

У класі Person властивість Age перевіряє лише загальний діапазон (1..119). У класі Doctor перевизначена властивість додає вимогу мінімального віку 25 років. При встановленні значення менше допустимого воно просто ігнорується.

Заборона подальшого перевизначення: sealed

Якщо перевизначений метод у похідному класі не повинен перевизначатися у його нащадках, його оголошують з модифікатором sealed. Цей модифікатор завжди використовується разом з override:

```
C#
using System;

ChiefDoctor chief = new ChiefDoctor("Андрій Мельник", "Хірургія", "Клінічна
лікарня №1");
chief.Print();

// Якби існував клас SeniorChiefDoctor : ChiefDoctor,
// він не зміг би перевизначити Print() — компілятор видасть помилку.

class Person
{
    public string Name { get; set; }
```

```
    public int Age { get; set; }
    public Person(string name) { Name = name; }
    public virtual void Print() => Console.WriteLine($"Особа: {Name}");
}

class Doctor : Person
{
    public string Specialization { get; set; }
    public Doctor(string name, string spec) : base(name) { Specialization =
spec; }

    public override void Print() =>
        Console.WriteLine($"Лікар: {Name} | {Specialization}");
}

class ChiefDoctor : Doctor
{
    public string Hospital { get; set; }

    public ChiefDoctor(string name, string spec, string hospital)
        : base(name, spec) { Hospital = hospital; }

    public override sealed void Print()  // sealed: подальше перевизначення
заборонено
    {
        base.Print();
        Console.WriteLine($"  Головний лікар: {Hospital}");
    }
}
```

sealed override означає: «я перевизначаю метод базового класу, але забороняю будь-якому нащадку перевизначати його далі». Це корисно, коли реалізація є критичною і не повинна бути змінена у підкласах — наприклад, метод, що реалізує специфічну логіку ліцензування або безпеки.

4.4. Приховування методів та властивостей

У попередньому розділі ми розглянули перевизначення — механізм заміни реалізації віртуального методу в похідному класі. Проте іноді виникає ситуація, коли метод у базовому класі не є virtual, а отже не може бути перевизначений. Або ж нас влаштовує поточна реалізація для базового типу, але в похідному класі потрібна інша поведінка. У таких випадках застосовується **приховування** (hiding / shadowing).

Приховування полягає у визначенні в похідному класі члена з тим самим іменем і набором параметрів, що й у базовому класі. Для явного позначення приховування використовується ключове слово new. Якщо new не вказати — компілятор видасть попередження, але код скомпілюється.

Приховування методів

Розглянемо базовий клас Person із звичайним (не virtual) методом Print(). Клас Doctor хоче вивести у Print() додаткову інформацію про спеціалізацію, але перевизначити невіртуальний метод неможливо — тому використовується new:

```C#
using System;

Person person = new Person("Сергій Бойко", 52);
person.Print();   // Person.Print()

Doctor doctor = new Doctor("Олена Коваль", 38, "Кардіологія");
doctor.Print();   // Doctor.Print() — прихований метод

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Person(string name, int age) { Name = name; Age = age; }

    public void Print()   // НЕ virtual
    {
        Console.WriteLine($"Особа: {Name}, {Age} р.");
    }
}

class Doctor : Person
{
    public string Specialization { get; set; }

    public Doctor(string name, int age, string spec) : base(name, age)
    {
        Specialization = spec;
    }

    public new void Print()   // приховуємо метод базового класу
    {
        Console.WriteLine($"Лікар: {Name}, {Age} р. | {Specialization}");
    }
}
```

Ключове слово new сигналізує компілятору та читачу коду: «це навмисне приховування, а не помилка». Без new код працює так само, але компілятор виведе попередження CS0108.

Виклик базового методу через base

Якщо у прихованому методі потрібно скористатися реалізацією базового класу, можна звернутися до неї через base:

```C#
using System;

Doctor doctor = new Doctor("Олена Коваль", 38, "Кардіологія");
doctor.Print();

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age) { Name = name; Age = age; }

    public void Print()
    {
        Console.WriteLine($"Особа: {Name}, {Age} р.");
    }
}

class Doctor : Person
{
    public string Specialization { get; set; }
    public Doctor(string name, int age, string spec) : base(name, age) { Specialization = spec; }

    public new void Print()
    {
        base.Print();                                       // Особа: Олена Коваль, 38 р.
        Console.WriteLine($"  Спеціалізація: {Specialization}");
    }
}
```

Ключова відмінність від override

Приховування і перевизначення виглядають схоже, але поводяться принципово по-різному. Якщо об'єкт Doctor зберігається у змінній типу Person, то:

- при **перевизначенні** (override) — виконається Doctor.Print()
- при **приховуванні** (new) — виконається Person.Print()

```C#
using System;

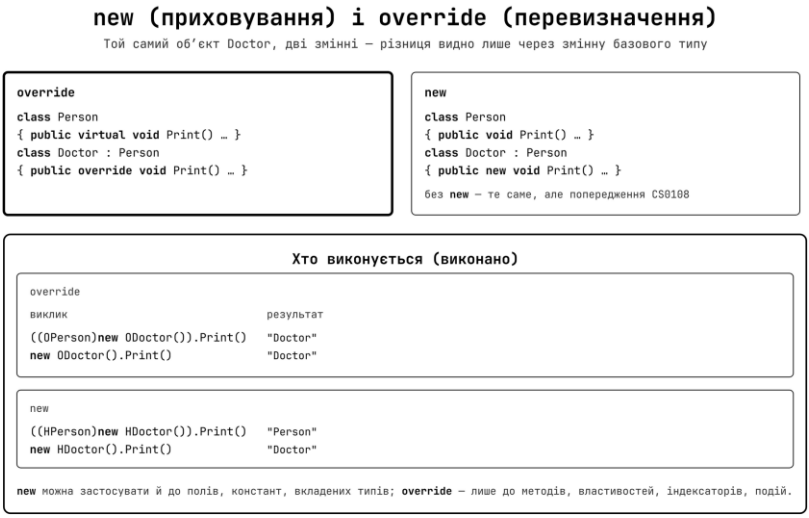
// Змінна типу Person, об'єкт Doctor
Person p = new Doctor("Олена Коваль", 38, "Кардіологія");
p.Print(); // яка реалізація?

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age) { Name = name; Age = age; }

    public void Print()   // не virtual — hiding
    {
        Console.WriteLine($"Person.Print: {Name}");
```

```
16      }
17  }
18
19  class Doctor : Person
20  {
21      public string Specialization { get; set; }
22      public Doctor(string name, int age, string spec) : base(name, age) {
    Specialization = spec; }
23
24      public new void Print()
25      {
26          Console.WriteLine($"Doctor.Print: {Name} | {Specialization}");
27      }
28  }
```

Результат: **Person.Print: Олена Коваль** — виконується метод класу Person, хоча реальний об'єкт є Doctor. Саме тому для поліморфної поведінки потрібен override, а не new.



new (приховування) vs override (перевизначення) — поведінка через змінну базового типу

Приховування властивостей

Так само як і методи, можна приховувати властивості. Це корисно, коли потрібно змінити логіку доступу у похідному класі без оголошення virtual у базовому:

```C#
using System;

Person person = new Person("Іван Петренко");
Console.WriteLine(person.Name); // Іван Петренко

Patient patient = new Patient("Марія Сидоренко", "MR-001");
Console.WriteLine(patient.Name); // [MR-001] Марія Сидоренко

class Person
{
    public string Name { get; set; }
    public Person(string name) { Name = name; }
}

class Patient : Person
{
    public string RecordId { get; set; }

    public Patient(string name, string recordId) : base(name)
    {
        RecordId = recordId;
    }

    // приховуємо властивість Name — додаємо номер картки пацієнта
    public new string Name
    {
        get => $"[{RecordId}] {base.Name}";
        set => base.Name = value;
    }
}
```

У блоці get звертаємось до base.Name, щоб отримати оригінальне ім'я і доповнити його. У блоці set передаємо значення безпосередньо властивості базового класу.

Приховування полів і констант

На відміну від override, new можна застосовувати не тільки до методів і властивостей, але й до **полів** та **констант**. Це дозволяє перевизначати статичні метадані класу в ієрархії:

```C#
using System;

Console.WriteLine(Person.MinAge.ToString());   // 0
Console.WriteLine(Person.TypeLabel);           // Особа

Console.WriteLine(Patient.MinAge.ToString());  // 0
Console.WriteLine(Patient.TypeLabel);          // Пацієнт

Console.WriteLine(Doctor.MinAge.ToString());   // 25
Console.WriteLine(Doctor.TypeLabel);           // Лікар

class Person
```

```
{
    public static readonly int MinAge = 0;
    public const string TypeLabel = "Особа";
}

class Patient : Person
{
    public new const string TypeLabel = "Пацієнт";
    // MinAge не приховується — успадковується зі значенням 0
}

class Doctor : Person
{
    public new static readonly int MinAge = 25;
    public new const string TypeLabel = "Лікар";
}
```

Кожен клас має свою версію констант і полів, незалежну від базового класу. Звернення через тип (Doctor.MinAge) повертає значення саме того класу, через який звертаємось.

Коли використовувати new замість override

Приховування є вузькоспеціалізованим інструментом. Типові сценарії:

- **Метод базового класу не є virtual** — якщо немає доступу до вихідного коду або змінювати базовий клас небажано.
- **Навмисна ізоляція поведінки** — потрібна різна реалізація залежно від типу змінної (не об'єкта), що є рідкісною але законною архітектурною потребою.
- **Статичні поля та константи** — єдиний спосіб «замінити» статичний член у похідному класі.

В усіх інших випадках, де потрібна поліморфна поведінка, слід надавати перевагу virtual + override. Приховування через new без чіткого розуміння наслідків є поширеним джерелом помилок.

4.5. Відмінність перевизначення та приховування методів

У попередніх розділах ми розглянули два способи змінити поведінку методу в похідному класі: **перевизначення** (virtual + override) і **приховування** (new). Зовні вони схожі — в обох випадках похідний клас має метод з тим самим іменем, що й базовий. Але поводяться вони принципово по-різному, і розуміння цієї різниці є критичним для написання коректного поліморфного коду.

Перевизначення: пізнє зв'язування

При перевизначенні через virtual + override рішення про те, яку реалізацію методу викликати, приймається **під час виконання програми** на основі реального типу об'єкта. Цей механізм називається **пізнім зв'язуванням** (late binding або dynamic dispatch).

```C#
using System;

Person p = new Doctor("Олена Коваль", 38, "Кардіологія");
p.Print(); // Doctor.Print() — незважаючи на тип змінної Person

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age) { Name = name; Age = age; }

    public virtual void Print()
    {
        Console.WriteLine($"Person.Print: {Name}");
    }
}

class Doctor : Person
{
    public string Specialization { get; set; }
    public Doctor(string name, int age, string spec) : base(name, age) { Specialization = spec; }

    public override void Print()
    {
        Console.WriteLine($"Doctor.Print: {Name} | {Specialization}");
    }
}
```

Змінна p оголошена як Person, але реальний об'єкт у пам'яті — Doctor. При виклику p.Print() середовище виконання перевіряє реальний тип об'єкта і знаходить реалізацію Doctor.Print().

Таблиця віртуальних методів (VMT)

Для реалізації пізнього зв'язування середовище виконання (CLR) під час завантаження типу формує для кожного класу **таблицю віртуальних методів** (Virtual Method Table, VMT). У цій таблиці зберігаються адреси реалізацій усіх віртуальних методів класу. Коли створюється об'єкт, у його заголовку зберігається посилання на опис свого реального класу, а через нього — на VMT. Компілятор C# лише генерує інструкцію callvirt і вказує, який віртуальний метод викликати; сама таблиця будується вже під час виконання.

При виклику p.Print():

8. Середовище виконання бере посилання на VMT з об'єкта — це VMT класу Doctor
9. У VMT Doctor знаходить адресу Doctor.Print()
10. Виконує Doctor.Print()

Саме тому тип змінної не має значення — вирішує VMT об'єкта. Варто врахувати, що через необхідність звернення до VMT виклик віртуального методу є трохи повільнішим порівняно зі звичайним. Проте у переважній більшості задач ця різниця непомітна.

Приховування: раннє зв'язування

При приховуванні через new рішення приймається **під час компіляції** на основі типу змінної. Це **раннє зв'язування** (early binding або static dispatch). Похідний клас не замінює метод у VMT — він просто визначає новий метод, який існує паралельно:

```C#
using System;

Person p = new Doctor("Олена Коваль", 38, "Кардіологія");
p.Print(); // Person.Print() — тип змінної вирішує!

Doctor d = new Doctor("Олена Коваль", 38, "Кардіологія");
d.Print(); // Doctor.Print() — тип змінної Doctor

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age) { Name = name; Age = age; }

    public void Print()   // не virtual
    {
        Console.WriteLine($"Person.Print: {Name}");
    }
}

class Doctor : Person
{
    public string Specialization { get; set; }
    public Doctor(string name, int age, string spec) : base(name, age) {
Specialization = spec; }

    public new void Print()
    {
        Console.WriteLine($"Doctor.Print: {Name} | {Specialization}");
    }
}
```

Той самий об'єкт Doctor — але залежно від типу змінної викликається різна реалізація. Через Person p — Person.Print(), через Doctor d — Doctor.Print(). Компілятор прив'язує виклик до методу ще при компіляції, дивлячись лише на тип змінної.

Порівняння на одному прикладі

Щоб побачити різницю повністю, розглянемо масив Person[], де зберігаються різні об'єкти:

```C#
using System;

Person[] staff = {
    new Person("Адміністратор", 30),
    new Doctor("Олена Коваль", 38, "Кардіологія"),
    new Patient("Іван Петренко", 45, "Гіпертонія"),
};

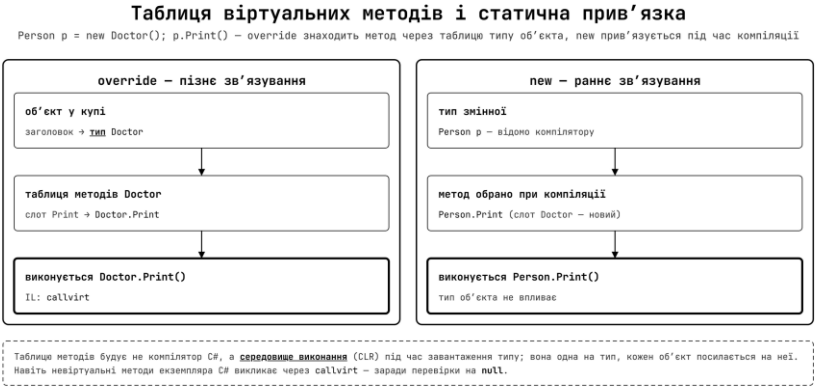
Console.WriteLine("=== override (virtual) ===");
foreach (Person p in staff)
    p.Print(); // кожен виконує свою реалізацію

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age) { Name = name; Age = age; }
    public virtual void Print() => Console.WriteLine($"Person: {Name}");
}

class Doctor : Person
{
    public string Specialization { get; set; }
    public Doctor(string name, int age, string spec) : base(name, age) { Specialization = spec; }
    public override void Print() => Console.WriteLine($"Doctor: {Name} | {Specialization}");
}

class Patient : Person
{
    public string Diagnosis { get; set; }
    public Patient(string name, int age, string diag) : base(name, age) { Diagnosis = diag; }
    public override void Print() => Console.WriteLine($"Patient: {Name} | {Diagnosis}");
}
```

Якби замість override використовувалось new — всі три виклики у циклі виводили б Person.Print(), оскільки змінна циклу p має тип Person.



VMT та статична прив'язка — як вирішується виклик методу

Підсумок

	override (перевизначення)	**new (приховування)**
Зв'язування	Пізнє (runtime)	Раннє (compile time)
Визначає виклик	Реальний тип об'єкта	Тип змінної
Поведінка	Поліморфна	Залежить від змінної
Потребує	virtual у базовому	Будь-який метод
VMT	Замінює запис	Не змінює VMT

> Загальне правило: якщо потрібна поліморфна поведінка — virtual + override. Якщо потрібно локально перевизначити невіртуальний метод без поліморфізму — new. Змішування цих підходів без чіткого розуміння є одним із найпоширеніших джерел прихованих помилок у C#.

4.6. Абстрактні класи та члени класів

Навіщо потрібні абстрактні класи

Класи зазвичай представляють конкретні сутності, з яких можна створювати об'єкти: Patient, Doctor, Appointment. Але іноді корисно мати клас, який описує загальний шаблон або контракт для цілої групи класів, і при цьому самостійно не має сенсу як окремий об'єкт.

Наприклад, «медичний персонал» — це абстракція. У клініці є конкретні лікарі, медсестри, хірурги, але «медичного персоналу взагалі» як конкретної людини не існує. Проте всі вони мають спільне: ім'я, роль, спосіб виконання своїх обов'язків. Для опису таких сутностей у C# призначені **абстрактні класи**.

Абстрактний клас оголошується з ключовим словом abstract. Його головна особливість: **неможливо створити екземпляр абстрактного класу безпосередньо**. Він існує лише як база для похідних класів.

Абстрактний клас з конкретними методами

Абстрактний клас може мати звичайні поля, властивості, конструктори та методи — і похідні класи успадкують їх без необхідності перевизначення:

```
C#
 1  using System;
 2
 3  MedicalStaff doctor = new Doctor("Олена Коваль", 38, "Кардіологія");
 4  MedicalStaff nurse  = new Nurse("Тетяна Мороз", 29, "Терапія");
 5
 6  doctor.PrintCard();
 7  nurse.PrintCard();
 8
 9  // new MedicalStaff("...", 0); // Помилка — абстрактний клас!
10
11  abstract class MedicalStaff
12  {
13      public string Name { get; set; }
14      public int Age { get; set; }
15
16      public MedicalStaff(string name, int age)
17      {
18          Name = name;
19          Age  = age;
20      }
21
22      // Звичайний метод — успадковується всіма похідними
23      public void PrintCard()
24      {
25          Console.WriteLine($"Співробітник: {Name}, {Age} р.");
26      }
```

```
}

class Doctor : MedicalStaff
{
    public string Specialization { get; set; }
    public Doctor(string name, int age, string spec) : base(name, age)
    { Specialization = spec; }
}

class Nurse : MedicalStaff
{
    public string Ward { get; set; }
    public Nurse(string name, int age, string ward) : base(name, age)
    { Ward = ward; }
}
```

Хоча MedicalStaff абстрактний, він може мати конструктор — але він викликається лише через base(...) у похідних класах. Звернення new MedicalStaff(...) безпосередньо заборонено компілятором.

Абстрактні методи

Абстрактний клас може визначати **абстрактні методи** — методи без реалізації, позначені ключовим словом abstract. Вони задають **контракт**: кожен неабстрактний похідний клас зобов'язаний реалізувати такий метод через override. Абстрактний метод не може мати тіла — лише сигнатуру:

C#
```
using System;

MedicalStaff[] staff = {
    new Doctor("Олена Коваль", 38, "Кардіологія"),
    new Nurse("Тетяна Мороз", 29, "Терапія"),
};

foreach (MedicalStaff m in staff)
{
    m.PrintCard();
    m.Examine();   // кожен виконує по-своєму
    Console.WriteLine();
}

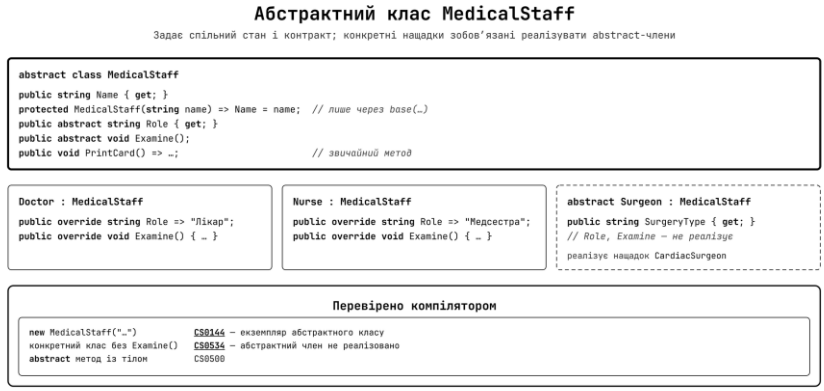
abstract class MedicalStaff
{
    public string Name { get; set; }
    public int Age { get; set; }

    public MedicalStaff(string name, int age) { Name = name; Age = age; }

    public void PrintCard() =>
        Console.WriteLine($"Співробітник: {Name}, {Age} р.");

    // Абстрактний метод — реалізація обов'язкова у похідних
```

```
26      public abstract void Examine();
27  }
28
29  class Doctor : MedicalStaff
30  {
31      public string Specialization { get; set; }
32      public Doctor(string name, int age, string spec) : base(name, age)
33      { Specialization = spec; }
34
35      public override void Examine()
36      {
37          Console.WriteLine($"Лікар {Name} проводить огляд за спеціалізацією:
    {Specialization}");
38      }
39  }
40
41  class Nurse : MedicalStaff
42  {
43      public string Ward { get; set; }
44      public Nurse(string name, int age, string ward) : base(name, age)
45      { Ward = ward; }
46
47      public override void Examine()
48      {
49          Console.WriteLine($"Медсестра {Name} вимірює показники у палаті:
    {Ward}");
50      }
51  }
```



Ієрархія абстрактного класу MedicalStaff

Абстрактні методи є частиною поліморфного інтерфейсу: виклик m.Examine() через змінну MedicalStaff завжди виконає реалізацію реального типу — так само, як і virtual + override. Різниця в тому, що abstract **не має реалізації за замовчуванням** і похідний клас **не може** відмовитись від реалізації (якщо сам не є абстрактним).

Абстрактні властивості

Крім методів, абстрактними можуть бути й **властивості**. Їх оголошення схоже на автовластивість, але без реального тіла — лише порожні блоки get та set:

```C#
using System;

MedicalStaff[] staff = {
    new Doctor("Олена Коваль", 38, "Кардіологія"),
    new Nurse("Тетяна Мороз", 29, "Терапія"),
};

foreach (MedicalStaff m in staff)
    Console.WriteLine($"{m.Name} — {m.Role}, ставка: {m.HourlyRate.ToString()} грн/год");

abstract class MedicalStaff
{
    public string Name { get; set; }
    public int Age { get; set; }

    public MedicalStaff(string name, int age) { Name = name; Age = age; }

    public abstract string Role { get; }           // абстрактна властивість
    public abstract decimal HourlyRate { get; }    // абстрактна властивість
}

class Doctor : MedicalStaff
{
    public string Specialization { get; set; }
    public Doctor(string name, int age, string spec) : base(name, age)
    { Specialization = spec; }

    public override string Role => "Лікар";
    public override decimal HourlyRate => 250m;
}

class Nurse : MedicalStaff
{
    public string Ward { get; set; }
    public Nurse(string name, int age, string ward) : base(name, age)
    { Ward = ward; }

    public override string Role => "Медсестра";
    public override decimal HourlyRate => 120m;
}
```

При перевизначенні абстрактної властивості у похідному класі її можна реалізувати як повноцінну властивість з полем або як автовластивість — залежно від потреб. У прикладі вище використані вирази-тіла (=>), що повертають константу.

Відмова від реалізації в проміжному класі

Якщо похідний клас не бажає або не може реалізувати всі абстрактні члени базового — він сам повинен бути оголошений як abstract. У такому разі обов'язок реалізації переходить до його нащадків:

```C#
using System;

MedicalStaff surgeon = new CardiacSurgeon("Андрій Мельник", 45);
surgeon.PrintCard();
surgeon.Examine();

abstract class MedicalStaff
{
    public string Name { get; set; }
    public int Age { get; set; }
    public MedicalStaff(string name, int age) { Name = name; Age = age; }
    public void PrintCard() => Console.WriteLine($"{Name}, {Age} p.");
    public abstract void Examine();
    public abstract string Role { get; }
}

// Хірург — ще абстрактний: не реалізує Examine() і Role
abstract class Surgeon : MedicalStaff
{
    public string SurgeryType { get; set; }
    public Surgeon(string name, int age, string surgeryType) : base(name, age)
    { SurgeryType = surgeryType; }
    // Examine() та Role — досі не реалізовані
}

// Конкретний клас — зобов'язаний реалізувати все
class CardiacSurgeon : Surgeon
{
    public CardiacSurgeon(string name, int age)
        : base(name, age, "Кардіохірургія") { }

    public override string Role => "Кардіохірург";

    public override void Examine()
    {
        Console.WriteLine($"{Name} проводить передопераційний огляд
({SurgeryType})");
    }
}
```

Клас Surgeon є проміжним абстрактним класом: він додає власну властивість SurgeryType, але не реалізує Examine() та Role. Клас CardiacSurgeon вже конкретний — і зобов'язаний реалізувати всі успадковані абстрактні члени.

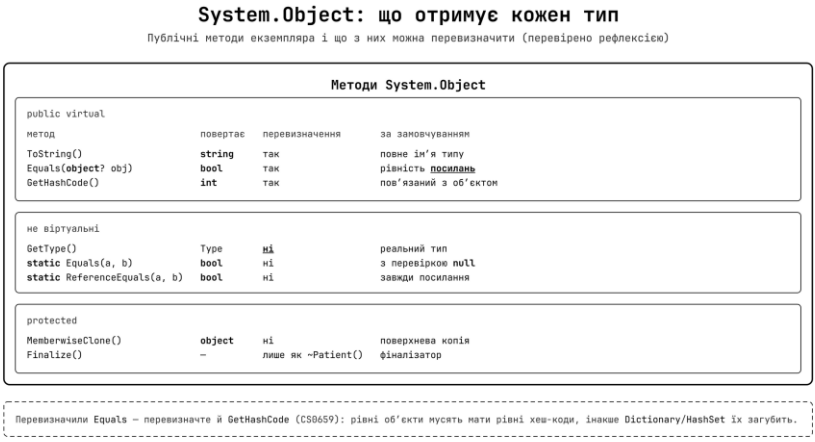
abstract vs virtual: коли що обирати

	abstract	**virtual**
Реалізація у базовому	Відсутня	Є (за замовчуванням)
Похідний зобов'язаний	Так (або бути abstract)	Ні (може не перевизначати)
Клас має бути abstract	Так	Ні
Поліморфізм	Так	Так

Правило вибору: якщо у базовому класі немає і не може бути розумної реалізації за замовчуванням — використовуйте abstract. Якщо базова реалізація має сенс, але похідні можуть її уточнити — використовуйте virtual.

4.7. Клас System.Object та його методи

Усі типи в .NET — як вбудовані (int, string, DateTime), так і ті, що ми створюємо самостійно — є похідними від класу System.Object. Для структур (int, DateTime) успадкування непряме: вони походять від System.ValueType, а той — від Object. Навіть якщо ми не вказуємо Object як базовий, компілятор додає це успадкування неявно. Це означає, що кожен об'єкт у C# гарантовано має чотири публічні методи екземпляра: ToString(), GetHashCode(), Equals() та GetType(). Перші три — віртуальні, їх можна перевизначати; GetType() — ні. Окрім них, Object має статичні Equals(a, b) і ReferenceEquals(a, b) та захищені MemberwiseClone() і Finalize().



Методи System.Object доступні у кожному класі

Метод ToString

Метод ToString() повертає рядкове представлення об'єкта. Для числових типів це їхнє значення у вигляді рядка. Для власних класів стандартна реалізація повертає повну назву типу з простором імен:

```
C#
 1  using System;
 2
 3  int age = 45;
 4  double temperature = 36.6;
 5
 6  Console.WriteLine(age.ToString());         // 45
 7  Console.WriteLine(temperature.ToString()); // 36,6
 8
 9  Patient patient = new Patient("Іван Петренко", 45, "Гіпертонія");
10  Console.WriteLine(patient.ToString()); // назва типу: Patient
11
12  class Patient
13  {
14      public string Name { get; set; }
15      public int Age { get; set; }
16      public string Diagnosis { get; set; }
17      public Patient(string name, int age, string diagnosis)
18      { Name = name; Age = age; Diagnosis = diagnosis; }
19  }
```

Ми можемо перевизначити ToString(), щоб повертати змістовний рядок замість назви типу:

```C#
using System;

Patient patient = new Patient("Іван Петренко", 45, "Гіпертонія");
Doctor  doctor  = new Doctor("Олена Коваль", 38, "Кардіологія");

Console.WriteLine(patient.ToString());
Console.WriteLine(doctor.ToString());

// Console.WriteLine автоматично викликає ToString()
Console.WriteLine(patient);
Console.WriteLine(doctor);

class Patient
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Diagnosis { get; set; }
    public Patient(string name, int age, string diagnosis)
    { Name = name; Age = age; Diagnosis = diagnosis; }

    public override string ToString()
    {
        return $"Пацієнт: {Name}, {Age} р. | {Diagnosis}";
    }
}

class Doctor
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Specialization { get; set; }
    public Doctor(string name, int age, string spec)
    { Name = name; Age = age; Specialization = spec; }

    public override string ToString()
    {
        return $"Лікар: {Name}, {Age} р. | {Specialization}";
    }
}
```

Зверніть увагу: Console.WriteLine(patient) автоматично викликає ToString() без явного звернення. Це справедливо для рядкової інтерполяції та конкатенації рядків — $"Пацієнт: {patient}" теж неявно викликає ToString().

Якщо клас частково заповнений (наприклад, відсутнє ім'я), можна повернути результат базової реалізації через base.ToString():

```C#
using System;

Patient named   = new Patient("Іван Петренко", 45);
Patient unnamed = new Patient("", 0);

Console.WriteLine(named.ToString());    // Іван Петренко, 45 р.
Console.WriteLine(unnamed.ToString());  // Patient (базова реалізація)
```

```
 8
 9  class Patient
10  {
11      public string Name { get; set; } = "";
12      public int Age { get; set; }
13      public Patient(string name, int age) { Name = name; Age = age; }
14
15      public override string? ToString()
16      {
17          if (string.IsNullOrEmpty(Name))
18              return base.ToString(); // повертає "Patient"
19          return $"{Name}, {Age} p.";
20      }
21  }
```

Метод GetHashCode

Метод GetHashCode() повертає ціле число — **хеш-код** об'єкта. Хеш-коди використовуються колекціями на основі хеш-таблиць (Dictionary<K,V>, HashSet<T>) для швидкого пошуку та зберігання об'єктів.

```
C#
 1  using System;
 2
 3  Patient p1 = new Patient("Іван Петренко", 45);
 4  Patient p2 = new Patient("Іван Петренко", 45);
 5  Patient p3 = new Patient("Марія Сидоренко", 32);
 6
 7  Console.WriteLine(p1.GetHashCode().ToString());
 8  Console.WriteLine(p2.GetHashCode().ToString()); // однаковий з p1
 9  Console.WriteLine(p3.GetHashCode().ToString()); // інший
10
11  class Patient
12  {
13      public string Name { get; set; }
14      public int Age { get; set; }
15      public Patient(string name, int age) { Name = name; Age = age; }
16
17      public override int GetHashCode()
18      {
19          // Комбінуємо хеш-коди всіх полів, що беруть участь у порівнянні
20          return HashCode.Combine(Name, Age);
21      }
22  }
```

HashCode.Combine(...) — зручний вбудований спосіб об'єднати хеш-коди кількох полів. Два об'єкти, які рівні за Equals, **зобов'язані** повертати однаковий хеш-код — це фундаментальний контракт, який забезпечує коректну роботу колекцій.

Контракт рівності: Equals і GetHashCode завжди разом

Методи Equals і GetHashCode утворюють **контракт рівності**: якщо два об'єкти рівні (Equals повертає true), вони повинні мати однаковий хеш-код. Порушення цього правила призводить до некоректної поведінки словників і множин — об'єкти, які логічно рівні, не знаходитимуться у колекції.

Тому: **якщо перевизначаєте Equals — завжди перевизначайте GetHashCode**, і навпаки.

```C#
using System;

Patient p1 = new Patient("Іван Петренко", "P-001");
Patient p2 = new Patient("Марія Сидоренко", "P-002");
Patient p3 = new Patient("Іван Петренко", "P-001");

Console.WriteLine(p1.Equals(p2).ToString()); // False
Console.WriteLine(p1.Equals(p3).ToString()); // True — однаковий recordId
Console.WriteLine((p1.GetHashCode() == p3.GetHashCode()).ToString()); // True

class Patient
{
    public string Name { get; set; }
    public string RecordId { get; set; }

    public Patient(string name, string recordId)
    { Name = name; RecordId = recordId; }

    public override bool Equals(object? obj)
    {
        if (obj is Patient other)
            return RecordId == other.RecordId; // рівність за унікальним ID картки
        return false;
    }

    public override int GetHashCode() => RecordId.GetHashCode();

    public override string ToString() => $"{Name} [{RecordId}]";
}
```

У клінічному контексті логічно вважати двох пацієнтів однаковими, якщо збігається їхній унікальний ідентифікатор картки (RecordId), — навіть якщо ім'я записане по-різному.

Метод GetType

Метод GetType() повертає об'єкт типу Type, який описує реальний тип об'єкта під час виконання. На відміну від інших методів Object,

GetType() **не можна перевизначити** — це гарантує надійність системи типів:

```
C#
 1  using System;
 2
 3  Person person  = new Patient("Іван Петренко", 45, "Гіпертонія");
 4  Patient patient = new Patient("Марія Сидоренко", 32, "Бронхіт");
 5
 6  Console.WriteLine(person.GetType().Name);   // Patient (реальний тип!)
 7  Console.WriteLine(patient.GetType().Name);  // Patient
 8
 9  // Порівняння типів через typeof
10  if (person.GetType() == typeof(Patient))
11      Console.WriteLine("Це Patient");
12
13  // Коротший спосіб — оператор is
14  if (person is Patient p)
15      Console.WriteLine($"Діагноз: {p.Diagnosis}");
16
17  class Person
18  {
19      public string Name { get; set; }
20      public int Age { get; set; }
21      public Person(string name, int age) { Name = name; Age = age; }
22  }
23
24  class Patient : Person
25  {
26      public string Diagnosis { get; set; }
27      public Patient(string name, int age, string diagnosis) : base(name, age)
28      { Diagnosis = diagnosis; }
29  }
```

GetType() завжди повертає реальний тип об'єкта, навіть якщо змінна оголошена як базовий тип. Це відрізняє його від оператора is, який також перевіряє сумісність з ієрархією (тобто patient is Person поверне true), тоді як GetType() == typeof(Person) поверне false для об'єкта Patient.

4.8. Узагальнення

Крім звичайних типів, фреймворк .NET підтримує **узагальнені типи** (generics) — класи, структури, інтерфейси та методи, які параметризуються типом. Generics є одним із найважливіших механізмів C# і широко використовуються у стандартній бібліотеці: List<T>, Dictionary<TKey, TValue>, Queue<T> та багато інших.

Проблема без узагальнень

Щоб зрозуміти навіщо потрібні generics, розглянемо задачу. У клінічній системі медична картка пацієнта може мати ідентифікатор різного типу: в одній системі це int, в іншій — string типу "P-001", у третій — Guid. Як написати один клас, що підтримує всі варіанти?

Перше рішення — використати object, оскільки це базовий тип для всіх:

```C#
using System;

MedicalRecord r1 = new MedicalRecord(1001, "Гіпертонія");
MedicalRecord r2 = new MedicalRecord("P-001", "Бронхіт");

int id1 = (int)r1.Id;         // розпакування — додаткові витрати
string id2 = (string)r2.Id;

Console.WriteLine($"Картка #{id1.ToString()}: {r1.Description}");
Console.WriteLine($"Картка {id2}: {r2.Description}");

// Небезпечно: компілятор не зловить помилку — лише runtime!
try
{
    string wrongId = (string)r1.Id; // InvalidCastException
}
catch (InvalidCastException)
{
    Console.WriteLine("Помилка: невірне приведення типу");
}

class MedicalRecord
{
    public object Id { get; }
    public string Description { get; }

    public MedicalRecord(object id, string description)
    {
        Id = id;
        Description = description;
    }
}
```

Два суттєві недоліки цього підходу:

11. **Boxing/unboxing** — при передачі int у object відбувається упаковка (boxing): значення копіюється в новий об'єкт-обгортку, виділений у купі (звідки береться значення — зі стека чи з поля іншого об'єкта — не важливо). Зворотна операція (розпакування) — теж копіювання. У гарячих ділянках коду це помітно знижує продуктивність.

12. **Відсутність типобезпеки** — компілятор не знає реального типу Id, тому не може попередити про неправильне приведення. Помилка виникне лише під час виконання як InvalidCastException.

Узагальнений клас

Generics вирішують обидві проблеми. Замість object використовується **параметр типу** T — своєрідний «слот», який заповнюється конкретним типом при використанні класу:

```C#
using System;

MedicalRecord<int>    r1 = new MedicalRecord<int>(1001, "Гіпертонія");
MedicalRecord<string> r2 = new MedicalRecord<string>("P-001", "Бронхіт");

int    id1 = r1.Id; // без розпакування
string id2 = r2.Id; // без приведення типів

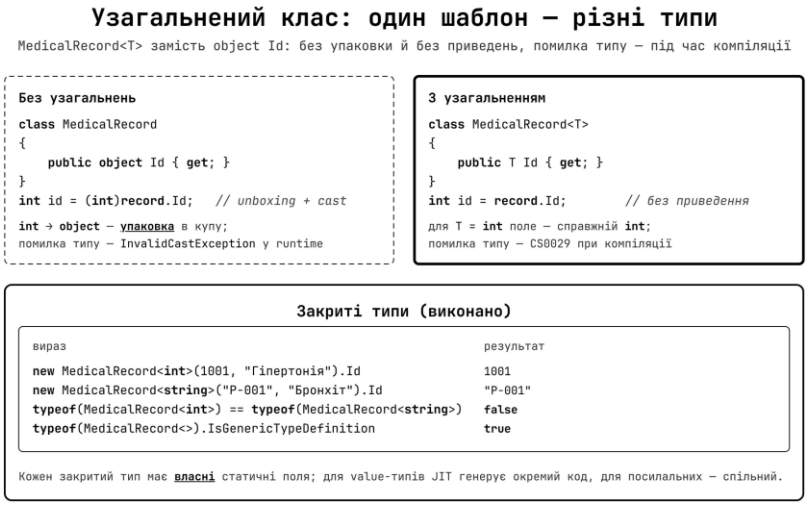
Console.WriteLine($"Картка #{id1.ToString()}: {r1.Description}");
Console.WriteLine($"Картка {id2}: {r2.Description}");

// MedicalRecord<int> r3 = new MedicalRecord<int>("abc", "..."); // помилка компіляції!

class MedicalRecord<T>
{
    public T Id { get; }
    public string Description { get; }

    public MedicalRecord(T id, string description)
    {
        Id = id;
        Description = description;
    }

    public override string ToString() => $"[{Id}] {Description}";
}
```



Узагальнений клас: один шаблон — різні типи

Кутові дужки <T> в оголошенні class MedicalRecord<T> вказують, що клас є узагальненим. Буква T — умовна назва параметра типу; замість неї може бути будь-який ідентифікатор. При створенні об'єкта конкретний тип вказується у кутових дужках: new MedicalRecord<int>(...). Компілятор замінює T на int і перевіряє всі операції з Id відповідно до цього типу — помилки типів виловлюються на етапі компіляції, а не виконання.

Конвенції іменування параметрів типу

У C# прийнято такі угоди щодо назв параметрів типу:

- T — загальний параметр типу (скорочення від *Type*)
- TKey, TValue — для пар ключ/значення (як у Dictionary<TKey, TValue>)
- TResult — для типу результату (як у Func<T, TResult>)
- TEntity, TModel — описові назви для конкретного контексту

Кілька параметрів типу

Клас може мати кілька параметрів типу одночасно. Наприклад, клас призначення Assignment пов'язує пацієнта та лікаря, причому обидва можуть мати свої типи ідентифікаторів:

```
C#
 1  using System;
 2
 3  Assignment<int, string> a1 = new Assignment<int, string>(
 4      1001, "P-001", "Первинний огляд");
 5
 6  Console.WriteLine(a1.ToString());
 7
 8  class Assignment<TPatientId, TDoctorId>
 9  {
10      public TPatientId PatientId { get; }
11      public TDoctorId  DoctorId  { get; }
12      public string     Purpose   { get; }
13
14      public Assignment(TPatientId patientId, TDoctorId doctorId, string purpose)
15      {
16          PatientId = patientId;
17          DoctorId  = doctorId;
18          Purpose   = purpose;
19      }
20
21      public override string ToString() =>
22          $"Призначення: пацієнт #{PatientId} → лікар {DoctorId} | {Purpose}";
23  }
```

Кожен параметр типу замінюється незалежно: TPatientId → int, TDoctorId → string.

Статичні поля узагальнених класів

Важливий нюанс: при типізації узагальненого класу різними типами для кожної комбінації типів створюється **окремий набір статичних членів**. Тобто MedicalRecord<int> і MedicalRecord<string> — це фактично два різних класи з власними статичними полями:

```
C#
 1  using System;
 2
 3  MedicalRecord<int>.Prefix    = "INT";
 4  MedicalRecord<string>.Prefix = "STR";
 5
 6  Console.WriteLine(MedicalRecord<int>.Prefix);    // INT
 7  Console.WriteLine(MedicalRecord<string>.Prefix); // STR — незалежне поле
 8
 9  class MedicalRecord<T>
10  {
11      public static string Prefix = "DEFAULT";
```

```
    public T Id { get; }
    public MedicalRecord(T id) { Id = id; }
}
```

Узагальнений клас як тип поля

Параметр типу може сам бути узагальненим класом. Наприклад, клас Clinic зберігає головного лікаря як MedicalRecord<T>:

```
C#
using System;

MedicalRecord<int> chiefRecord = new MedicalRecord<int>(42, "Головний лікар");
Clinic<MedicalRecord<int>> clinic = new Clinic<MedicalRecord<int>>("Клініка №1", chiefRecord);

Console.WriteLine(clinic.ToString());

class MedicalRecord<T>
{
    public T Id { get; }
    public string Description { get; }
    public MedicalRecord(T id, string desc) { Id = id; Description = desc; }
    public override string ToString() => $"[{Id}] {Description}";
}

class Clinic<TChief>
{
    public string Name  { get; }
    public TChief Chief { get; }
    public Clinic(string name, TChief chief) { Name = name; Chief = chief; }
    public override string ToString() => $"{Name}, керівник: {Chief}";
}
```

Узагальнені методи

Generics застосовуються не лише до класів, але й до окремих методів. Узагальнений метод оголошує власний параметр типу і може бути визначений у будь-якому класі — в тому числі у звичайному (не generic):

```
C#
using System;

int a = 10, b = 20;
Swap<int>(ref a, ref b);
Console.WriteLine($"a={a.ToString()} b={b.ToString()}"); // a=20 b=10

string s1 = "Кардіологія", s2 = "Неврологія";
Swap(ref s1, ref s2); // тип виводиться автоматично
Console.WriteLine($"s1={s1} s2={s2}");

MedicalRecord<int> r1 = new MedicalRecord<int>(1, "Гіпертонія");
MedicalRecord<int> r2 = new MedicalRecord<int>(2, "Бронхіт");
```

```
Swap(ref r1, ref r2);
Console.WriteLine($"r1={r1} r2={r2}");

void Swap<T>(ref T x, ref T y)
{
    T temp = x;
    x = y;
    y = temp;
}

class MedicalRecord<T>
{
    public T Id { get; }
    public string Description { get; }
    public MedicalRecord(T id, string desc) { Id = id; Description = desc; }
    public override string ToString() => $"[{Id}] {Description}";
}
```

Зверніть увагу на виклик Swap(ref s1, ref s2) без явного вказання типу — компілятор сам визначає T = string з типів переданих аргументів. Це називається **виведення типу** (type inference) і робить код лаконічнішим.

Generics у стандартній бібліотеці

Generics є основою колекцій .NET. Завдяки ним колекції зберігають елементи без boxing і з повною типобезпекою:

```
C#
using System;
using System.Collections.Generic;

List<string> diagnoses = new List<string>();
diagnoses.Add("Гіпертонія");
diagnoses.Add("Бронхіт");
diagnoses.Add("Діабет");

foreach (string d in diagnoses)
    Console.WriteLine(d);

Dictionary<int, string> patients = new Dictionary<int, string>();
patients[1001] = "Іван Петренко";
patients[1002] = "Марія Сидоренко";

Console.WriteLine(patients[1001]);
Console.WriteLine(patients[1002]);
```

List<string> зберігає рядки безпосередньо, без упаковки в object — на відміну від старого ArrayList. Dictionary<int, string> типізує і ключ, і значення, тому жодних явних приведень не потрібно. Детально колекції розглядаються у наступних розділах курсу.

4.9. Обмеження узагальнень

Узагальнені типи дозволяють використовувати будь-який тип як параметр. Але іноді всередині узагальненого класу чи методу необхідно звертатися до конкретних членів об'єкта — властивостей, методів. Компілятор не дозволить цього, якщо параметр T нічим не обмежений, бо у такому разі T може бути будь-яким типом, і немає гарантії, що потрібний член існує.

Обмеження узагальнень (generic constraints) через ключове слово where дозволяють вказати, яким типам відповідає параметр T. Це дає компілятору достатньо інформації для перевірки коректності звернень до членів.

Проблема без обмежень

Розглянемо клінічний приклад. У нас є клас Notification з властивістю Text, і ми хочемо написати узагальнений метод Send<T>:

```
C#
 1  using System;
 2
 3  // Без обмеження — компілятор не знає що таке T
 4  // і не дозволяє звертатися до message.Text
 5
 6  Notification sms   = new SmsNotification("Ваш прийом о 10:00");
 7  Notification email = new EmailNotification("Нагадування про аналізи");
 8
 9  // Send(sms);  // помилка: T не має властивості Text
10
11  class Notification
12  {
13      public string Text { get; }
14      public Notification(string text) { Text = text; }
15  }
16
17  class SmsNotification : Notification
18  {
19      public SmsNotification(string text) : base(text) {}
20  }
21
22  class EmailNotification : Notification
23  {
24      public EmailNotification(string text) : base(text) {}
25  }
```

Якщо написати void Send<T>(T message) { Console.WriteLine(message.Text); } — компілятор видасть помилку: T може бути чим завгодно, і не факт, що в нього є Text. Саме тут на допомогу приходять обмеження.

Обмеження методу: where T : ТипКласу

Обмеження методу вказується після списку параметрів через where:

```
C#
using System;

Send(new SmsNotification("Ваш прийом о 10:00"));
Send(new EmailNotification("Нагадування про аналізи"));
Send<Notification>(new Notification("Загальне повідомлення"));

// Send("просто рядок");  // помилка компіляції — string не є Notification

void Send<T>(T message) where T : Notification
{
    // T гарантовано є Notification або похідним — Text доступний
    Console.WriteLine($"Надсилається: {message.Text}");
}

class Notification
{
    public string Text { get; }
    public Notification(string text) { Text = text; }
}

class SmsNotification : Notification
{
    public SmsNotification(string text) : base(text) {}
}

class EmailNotification : Notification
{
    public EmailNotification(string text) : base(text) {}
}
```

Вираз where T : Notification каже компілятору: «T — це завжди Notification або похідний від нього клас». Тому звернення до message.Text стає безпечним і дозволеним. При виклику Send(new SmsNotification(...)) компілятор сам визначає T = SmsNotification — явно вказувати тип не обов'язково.

Обмеження класу: where T : ТипКласу

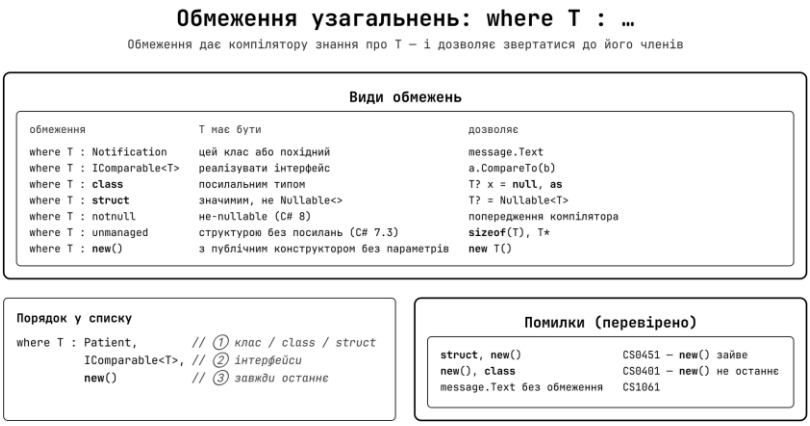
Так само обмеження застосовуються до узагальнених класів. Синтаксис:

```
class ІмʼяКласу<T> where T : ТипОбмеження
```

```
C#
using System;

NotificationSender<SmsNotification> smsSender = new();
smsSender.Send(new SmsNotification("Підтвердження запису"));
smsSender.Send(new SmsNotification("Нагадування о 09:00"));
```

```
 6
 7  NotificationSender<EmailNotification> emailSender = new();
 8  emailSender.Send(new EmailNotification("Результати аналізів готові"));
 9
10  class NotificationSender<T> where T : Notification
11  {
12      public void Send(T message)
13      {
14          Console.WriteLine($"[{typeof(T).Name}] {message.Text}");
15      }
16  }
17
18  class Notification
19  {
20      public string Text { get; }
21      public Notification(string text) { Text = text; }
22  }
23
24  class SmsNotification : Notification
25  {
26      public SmsNotification(string text) : base(text) {}
27  }
28
29  class EmailNotification : Notification
30  {
31      public EmailNotification(string text) : base(text) {}
32  }
```



Типи обмежень узагальнень у C#

Стандартні обмеження: class, struct, new()

Окрім конкретних класів та інтерфейсів, є вбудовані обмеження:

where T : class — T має бути reference type (клас, інтерфейс, рядок):

```C#
using System;

Repository<Patient> repo = new();
repo.Add(new Patient("Іван Петренко", 45));
repo.Add(new Patient("Марія Сидоренко", 32));
repo.PrintAll();

class Repository<T> where T : class
{
    private T[] _items = new T[10];
    private int _count = 0;

    public void Add(T item) => _items[_count++] = item;

    public void PrintAll()
    {
        for (int i = 0; i < _count; i++)
            Console.WriteLine(_items[i]!.ToString());
    }
}

class Patient
{
    public string Name { get; }
    public int Age { get; }
    public Patient(string name, int age) { Name = name; Age = age; }
    public override string ToString() => $"Пацієнт: {Name}, {Age} р.";
}
```

where T : struct — T має бути value type (int, double, struct і т.д.):

```C#
using System;

MeasurementLog<int>    log1 = new("Пульс");
MeasurementLog<double> log2 = new("Температура");

log1.Add(72); log1.Add(80); log1.PrintAll();
log2.Add(36.6); log2.Add(37.2); log2.PrintAll();

class MeasurementLog<T> where T : struct
{
    private string _label;
    private T[] _values = new T[100];
    private int _count = 0;

    public MeasurementLog(string label) { _label = label; }
    public void Add(T value) => _values[_count++] = value;

    public void PrintAll()
    {
        Console.Write($"{_label}: ");
        for (int i = 0; i < _count; i++)
            Console.Write(_values[i].ToString() + " ");
        Console.WriteLine();
    }
}
```

where T : new() — T має публічний конструктор без параметрів. Це дозволяє всередині узагальненого класу створювати екземпляри типу T через new T():

```C#
using System;

Factory<Patient> factory = new();
Patient p = factory.Create();
Console.WriteLine(p.ToString());

class Factory<T> where T : new()
{
    public T Create() => new T(); // можливо лише з обмеженням new()
}

class Patient
{
    public string Name { get; set; } = "Невідомий";
    public int Age { get; set; } = 0;
    public override string ToString() => $"Пацієнт: {Name}, {Age} р.";
}
```

Обмеження за інтерфейсом

Як обмеження можна вказати **інтерфейс** — тоді T гарантовано реалізує всі члени цього інтерфейсу. Це дозволяє звертатися до методів інтерфейсу всередині узагальненого класу:

```C#
using System;

Sorter<int> sorter = new();
int[] ages = { 45, 32, 58, 27, 41 };
sorter.BubbleSort(ages);
foreach (int age in ages)
    Console.Write(age.ToString() + " "); // 27 32 41 45 58
Console.WriteLine();

class Sorter<T> where T : IComparable<T>
{
    public void BubbleSort(T[] array)
    {
        for (int i = 0; i < array.Length - 1; i++)
            for (int j = 0; j < array.Length - 1 - i; j++)
                if (array[j].CompareTo(array[j+1]) > 0)
                {
                    T temp = array[j];
                    array[j] = array[j+1];
                    array[j+1] = temp;
                }
    }
}
```

IComparable<T> гарантує наявність методу CompareTo(), тому сортування працює для будь-якого типу, що його реалізує: int, string, DateTime, або власний клас.

Кілька обмежень одночасно

Для одного параметра можна задати кілька обмежень. Вони вказуються у строгому порядку: спочатку клас або class/struct, потім інтерфейси, в кінці new():

```C#
using System;

MedicalProtocol<Patient, Doctor> protocol = new(
    new Patient("Іван Петренко", 45),
    new Doctor("Олена Коваль", 38, "Кардіологія"),
    "Обстеження серцево-судинної системи"
);
protocol.Execute();

class MedicalProtocol<TPatient, TDoctor>
    where TPatient : Patient, new()
    where TDoctor  : Doctor
{
    private TPatient _patient;
    private TDoctor  _doctor;
    private string   _purpose;

    public MedicalProtocol(TPatient patient, TDoctor doctor, string purpose)
    {
        _patient = patient;
        _doctor  = doctor;
        _purpose = purpose;
    }

    public void Execute()
    {
        Console.WriteLine($"Протокол: {_purpose}");
        Console.WriteLine($"  Пацієнт: {_patient.Name}, {_patient.Age} р.");
        Console.WriteLine($"  Лікар:   {_doctor.Name},
{_doctor.Specialization}");
    }
}

class Patient
{
    public string Name { get; set; } = "";
    public int Age { get; set; }
    public Patient() {}
    public Patient(string name, int age) { Name = name; Age = age; }
}

class Doctor
{
    public string Name { get; set; } = "";
    public int Age { get; set; }
```

```
45      public string Specialization { get; set; } = "";
46      public Doctor(string name, int age, string spec)
47      { Name = name; Age = age; Specialization = spec; }
48  }
```

where TPatient : Patient, new() означає: TPatient повинен бути Patient або похідним класом **і** мати публічний конструктор без параметрів. Для різних параметрів типу обмеження задаються окремими рядками where.

Порядок та правила обмежень

Кілька правил, які слід пам'ятати:

- Одночасно можна вказати лише одне з: конкретний клас, class, struct (вони несумісні між собою)
- Інтерфейсів може бути кілька
- new() завжди стоїть останнім
- struct несумісне з new() — структури за визначенням мають конструктор без параметрів

```
C#
1  // Правильно:
2  void Process<T>(T item) where T : Notification, new() { }
3
4  // Правильно — кілька інтерфейсів:
5  void Compare<T>(T a, T b) where T : class, IComparable<T>, IEquatable<T> { }
6
7  // Помилка — не можна одночасно class і struct:
8  // void Wrong<T>() where T : class, struct { }
```

Обмеження notnull (C# 8) та unmanaged (C# 7.3)

Є ще два додаткових обмеження: notnull (C# 8) і unmanaged (з'явилося раніше, у C# 7.3).

where T : notnull — T не може бути nullable reference type (string?, Patient?) або Nullable<T> (int?, double?). Це корисно в nullable-aware context, де компілятор відстежує nullable-анотації:

```
C#
 1  using System;
 2
 3  // T : notnull — компілятор не дозволить передати nullable-тип
 4  Cache<string> cache = new();
 5  cache.Set("key1", "Петренко Іван");
 6  cache.Set("key2", "Коваль Марія");
 7
 8  Console.WriteLine(cache.Get("key1") ?? "(відсутньо)");
 9  Console.WriteLine(cache.Get("key3") ?? "(відсутньо)");
10
```

```
// cache.Set("key4", null);  // попередження/помилка: string? не відповідає notnull

class Cache<T> where T : notnull
{
    private readonly System.Collections.Generic.Dictionary<string, T> _store = new();

    public void Set(string key, T value) => _store[key] = value;

    public T? Get(string key) =>
        _store.TryGetValue(key, out T? val) ? val : default;
}
```

where T : unmanaged — T є **unmanaged-типом**: примітивний числовий тип (int, double, bool, char), enum, покажчик або структура, яка на всіх рівнях вкладеності складається виключно з таких типів (без полів-посилань). Часто такі типи неточно називають blittable, але це різні поняття: bool і char є unmanaged, проте не blittable — при передачі в нативний код їх представлення може змінюватися. Це обмеження дозволяє використовувати unsafe-операції (sizeof(T), поінтер T*, Span<T> зі стек-алокацією), що важливо при роботі з бінарними протоколами та медичним обладнанням (DICOM, HL7 Binary):

```
C#
using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// T : unmanaged — можна отримати розмір і серіалізувати без рефлексії
BinarySerializer<VitalMeasurement> serializer = new();
VitalMeasurement m = new() { PatientId = 101, Systolic = 140f, Diastolic = 90f, HeartRate = 78f };
byte[] bytes = serializer.ToBytes(m);

Console.WriteLine($"Розмір структури: {Unsafe.SizeOf<VitalMeasurement>()} байт");
Console.WriteLine($"Байтів серіалізовано: {bytes.Length}");
VitalMeasurement m2 = serializer.FromBytes(bytes);
Console.WriteLine($"Відновлено: пацієнт #{m2.PatientId}, АТ {m2.Systolic}/{m2.Diastolic}, ЧСС {m2.HeartRate}");

// Бінарна структура вимірювання — без посилань, blittable
[StructLayout(LayoutKind.Sequential, Pack = 1)]
struct VitalMeasurement
{
    public ushort PatientId;   // 2 байти
    public float  Systolic;    // 4 байти
    public float  Diastolic;   // 4 байти
    public float  HeartRate;   // 4 байти
}

class BinarySerializer<T> where T : unmanaged
{
```

```
27      // MemoryMarshal працює лише з unmanaged-типами — саме це гарантує
    обмеження
28      public byte[] ToBytes(T value)
29      {
30          byte[] result = new byte[Unsafe.SizeOf<T>()];
31          MemoryMarshal.Write(result, in value);
32          return result;
33      }
34
35      public T FromBytes(byte[] data) => MemoryMarshal.Read<T>(data);
36  }
```

Попередня версія цього прикладу використовувала unsafe-код із покажчиками (fixed, *(T*)ptr). Він теж коректний, але вимагає дозволу AllowUnsafeBlocks у проєкті й не виконувався б у браузерному раннері сайту, а ще ставив оголошення структури перед інструкціями верхнього рівня (помилка CS8803). MemoryMarshal з простору імен System.Runtime.InteropServices робить те саме без unsafe — і так само потребує обмеження unmanaged. Перевірено на .NET 10: розмір структури 14 байт, дані відновлюються без втрат.

Зведена таблиця обмежень

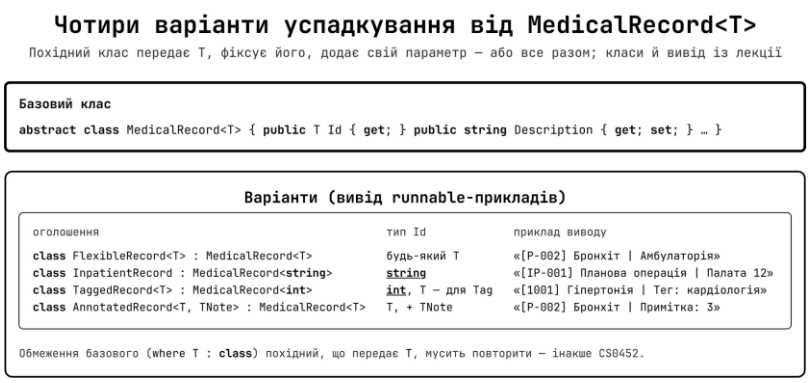
Обмеження	Що дозволяє T	Типовий сценарій
where T : ClassName	T — цей клас або похідний	Доступ до членів базового класу
where T : IInterface	T реалізує інтерфейс	Виклик методів контракту
where T : class	Reference type (клас, рядок)	Можна порівнювати з null
where T : struct	Value type (struct, int…)	T ніколи не null
where T : notnull	Не nullable (C# 8+)	Nullable-safe контейнери
where T : unmanaged	Unmanaged value type, без полів-посилань (C# 7.3+)	Unsafe-операції, бінарна серіалізація
where T : new()	Публічний конструктор без параметрів	new T() всередині узагальненого класу

4.10. Наслідування узагальнених типів

Узагальнені класи можуть успадковуватися один від одного — так само як і звичайні. При цьому є чотири принципово різних варіанти того, як похідний клас взаємодіє з параметром типу базового. Вибір між ними визначається тим, наскільки гнучким або конкретним має бути похідний клас.

Як базовий клас для всіх прикладів використаємо MedicalRecord<T>:

```
C#
 1  abstract class MedicalRecord<T>
 2  {
 3      public T Id { get; }
 4      public string Description { get; set; }
 5
 6      public MedicalRecord(T id, string description)
 7      {
 8          Id          = id;
 9          Description = description;
10      }
11
12      public override string ToString() => $"[{Id}] {Description}";
13  }
```



4 варіанти успадкування узагальнених класів

Варіант 1: передача параметра далі — Child<T> : Base<T>

Похідний клас залишається узагальненим і передає свій параметр T у базовий клас. Це найбільш гнучкий варіант — конкретний тип визначається лише при створенні екземпляра:

```C#
using System;

FlexibleRecord<int>    r1 = new FlexibleRecord<int>(1001, "Гіпертонія",
"Стаціонар");
FlexibleRecord<string> r2 = new FlexibleRecord<string>("P-002", "Бронхіт",
"Амбулаторія");

Console.WriteLine(r1.ToString());
Console.WriteLine(r2.ToString());

abstract class MedicalRecord<T>
{
    public T Id { get; }
    public string Description { get; }
    public MedicalRecord(T id, string desc) { Id = id; Description = desc; }
    public override string ToString() => $"[{Id}] {Description}";
}

class FlexibleRecord<T> : MedicalRecord<T>   // T передається далі
{
    public string Ward { get; }

    public FlexibleRecord(T id, string desc, string ward) : base(id, desc)
    {
        Ward = ward;
    }

    public override string ToString() => $"{base.ToString()} | {Ward}";
}
```

FlexibleRecord<int> і FlexibleRecord<string> — це два різних класи, кожен зі своїм типом Id. Похідний клас просто «пробрасує» параметр у базовий.

Варіант 2: фіксація типу — Child : Base<string>

Похідний клас є звичайним (неузагальненим) і фіксує конкретний тип для базового. Це доречно, коли для конкретної предметної ситуації тип ідентифікатора відомий заздалегідь:

```C#
using System;

InpatientRecord r1 = new InpatientRecord("IP-001", "Планова операція", 12);
InpatientRecord r2 = new InpatientRecord("IP-002", "Спостереження", 7);

Console.WriteLine(r1.ToString());
Console.WriteLine(r2.ToString());

// r1 — це також MedicalRecord<string>:
MedicalRecord<string> rec = r1;
Console.WriteLine(rec.Id);

abstract class MedicalRecord<T>
```

```
{
    public T Id { get; }
    public string Description { get; }
    public MedicalRecord(T id, string desc) { Id = id; Description = desc; }
    public override string ToString() => $"[{Id}] {Description}";
}

class InpatientRecord : MedicalRecord<string>   // T зафіксований як string
{
    public int RoomNumber { get; }

    public InpatientRecord(string id, string desc, int room) : base(id, desc)
    {
        RoomNumber = room;
    }

    public override string ToString() => $"{base.ToString()} | Палата
{RoomNumber.ToString()}";
}
```

InpatientRecord — звичайний клас без власних параметрів типу. Він завжди має Id типу string, і його можна зберігати у змінній MedicalRecord<string>.

Варіант 3: власний параметр при фіксованому базовому — Child<T> : Base<int>

Похідний клас є узагальненим з власним параметром T, але у базового клас тип зафіксований. Це дозволяє додавати нові generic-поля незалежно від базового:

C#
```
using System;

TaggedRecord<string> r1 = new TaggedRecord<string>(1001, "Гіпертонія",
"кардіологія");
TaggedRecord<string[]> r2 = new TaggedRecord<string[]>(1002, "Діабет",
    new[] { "ендокринологія", "дієтологія" });

Console.WriteLine(r1.ToString());
Console.WriteLine($"[{r2.Id}] {r2.Description} | Теги: {string.Join(", ",
r2.Tag)}");

abstract class MedicalRecord<T>
{
    public T Id { get; }
    public string Description { get; }
    public MedicalRecord(T id, string desc) { Id = id; Description = desc; }
    public override string ToString() => $"[{Id}] {Description}";
}

class TaggedRecord<T> : MedicalRecord<int>  // int фіксовано у базовому, T —
своє
{
    public T Tag { get; }
```

```
21
22      public TaggedRecord(int id, string desc, T tag) : base(id, desc)
23      {
24          Tag = tag;
25      }
26
27      public override string ToString() => $"{base.ToString()} | Тег: {Tag}";
28  }
```

TaggedRecord<T> успадковує MedicalRecord<int> — тому Id завжди int. Але сам клас залишається параметричним за T, що використовується для поля Tag.

Варіант 4: розширення базового параметра — Child<T, TExtra> : Base<T>

Похідний клас передає базовому параметр T і одночасно додає власний новий параметр. Це найбільш потужний варіант для побудови складних ієрархій:

```
C#
 1  using System;
 2
 3  AnnotatedRecord<int, string> r1 =
 4      new AnnotatedRecord<int, string>(1001, "Гіпертонія", "Лікар: Коваль О.В.");
 5
 6  AnnotatedRecord<string, int> r2 =
 7      new AnnotatedRecord<string, int>("P-002", "Бронхіт", 3);
 8
 9  Console.WriteLine(r1.ToString());
10  Console.WriteLine(r2.ToString());
11
12  abstract class MedicalRecord<T>
13  {
14      public T Id { get; }
15      public string Description { get; }
16      public MedicalRecord(T id, string desc) { Id = id; Description = desc; }
17      public override string ToString() => $"[{Id}] {Description}";
18  }
19
20  class AnnotatedRecord<T, TNote> : MedicalRecord<T>  // T передається, TNote —
    нове
21  {
22      public TNote Note { get; }
23
24      public AnnotatedRecord(T id, string desc, TNote note) : base(id, desc)
25      {
26          Note = note;
27      }
28
29      public override string ToString() => $"{base.ToString()} | Примітка:
    {Note}";
30  }
```

AnnotatedRecord<T, TNote> повністю гнучкий: T визначає тип Id (як у базовому), а TNote — тип додаткової анотації. Обидва параметри незалежні.

Успадкування обмежень

Якщо базовий клас встановлює обмеження на параметр типу, похідний клас зобов'язаний підтримати або посилити це обмеження для того самого параметра:

```
C#
 1  using System;
 2
 3  FlexibleRecord<Patient> r = new FlexibleRecord<Patient>(
 4      new Patient("Іван Петренко"), "Огляд", "Терапія");
 5  Console.WriteLine(r.ToString());
 6
 7  // FlexibleRecord<int> wrong = new(...);  // помилка: int — не клас
 8
 9  abstract class MedicalRecord<T> where T : class   // обмеження на рівні
    базового
10  {
11      public T Id { get; }
12      public string Description { get; }
13      public MedicalRecord(T id, string desc) { Id = id; Description = desc; }
14      public override string ToString() => $"[{Id}] {Description}";
15  }
16
17  class FlexibleRecord<T> : MedicalRecord<T> where T : class  // обмеження
    повторюється
18  {
19      public string Ward { get; }
20      public FlexibleRecord(T id, string desc, string ward) : base(id, desc)
21      { Ward = ward; }
22      public override string ToString() => $"{base.ToString()} | {Ward}";
23  }
24
25  class Patient
26  {
27      public string Name { get; }
28      public Patient(string name) { Name = name; }
29      public override string ToString() => $"Пацієнт: {Name}";
30  }
```

Правило просте: якщо базовий клас має where T : class, то і похідний повинен вказати where T : class або конкретніше обмеження (наприклад, where T : Patient). Це гарантує, що компілятор зможе перевіряти обмеження на всіх рівнях ієрархії.

Підсумок: коли що використовувати

Варіант	Синтаксис	Коли застосовувати
Передача T	Child<T> : Base<T>	Похідний залишається гнучким
Фіксація типу	Child : Base<string>	Тип відомий — клас стає конкретним
Свій T, фіксований базовий	Child<T> : Base<int>	Додаємо гнучкість у нових полях
Розширення параметрів	Child<T, K> : Base<T>	Максимальна гнучкість

Питання для самоконтролю — Розділ 4. Об'єктно-орієнтоване програмування

4.1. Успадкування

1. Що таке відношення **is-a**? Чим воно відрізняється від відношення **has-a**? Наведіть приклад неправильного застосування успадкування там, де слід використовувати композицію.
2. Поясніть синтаксис : base(...) у конструкторі похідного класу. Що буде, якщо базовий клас має лише параметризований конструктор, а похідний не викличе base(...)?
3. Що виведе цей код і в якому порядку?

```
C#
 1  class A
 2  {
 3      public A(string s) { Console.WriteLine($"A({s})"); }
 4  }
 5  class B : A
 6  {
 7      public B() : base("від B") { Console.WriteLine("B()"); }
 8  }
 9  class C : B
10  {
11      public C() { Console.WriteLine("C()"); }
12  }
13  new C();
```

4. Чим відрізняються модифікатори protected та private з точки зору ієрархії успадкування? Коли доцільно використовувати protected?
5. Що таке sealed клас? Чому у стандартній бібліотеці .NET клас String є sealed? Наведіть приклад ситуації, де sealed архітектурно виправданий.
6. Чи передаються конструктори при успадкуванні? Що станеться, якщо базовий клас не має конструктора без параметрів і похідний клас не визначає власний конструктор?
7. Спроєктуйте невелику ієрархію для транспортних засобів: базовий клас Vehicle з властивостями Make і Speed, та два похідних — Car (з NumberOfDoors) і Truck (з LoadCapacity). Всі класи мають конструктори, що використовують base(...).
8. Від якого класу неявно успадковують усі класи в C#? Які методи це надає кожному об'єкту?

4.2. Перетворення типів

9. Поясніть різницю між **upcasting** і **downcasting**. Яке з них безпечне і виконується неявно, а яке несе ризик і вимагає явного синтаксису?
10. Що виведе цей код?

```
C#
1  Patient patient = new Patient("Іван", 45, "Грип");
2  Person  person  = patient;
3  Console.WriteLine(person.GetType().Name);
```

Поясніть: чи є person і patient різними об'єктами у пам'яті?

11. Яку помилку спричинить наступний код і коли саме — під час компіляції чи виконання?

```
C#
1  Person person = new Person("Сергій", 52);
2  Patient patient = (Patient)person;
```

12. У чому перевага оператора as перед явним приведенням (Patient)person? Яке значення повертає as у разі невдачі?
13. Що робить вираз person is Patient patient (C# 7+)? Чим він зручніший за комбінацію окремої перевірки типу і явного приведення?

14. У масиві Person[] зберігаються об'єкти типів Patient, Doctor і Person. Напишіть цикл з switch (pattern matching), що обробляє кожен тип окремо. Чому порядок гілок switch важливий?
15. Порівняйте GetType() == typeof(Patient) та obj is Patient. В якому випадку кожен підхід поверне true, а в якому false?
16. Вас попросили реалізувати метод PrintDetails(Person p), що виводить специфічну для кожного типу інформацію — Diagnosis для Patient, Specialization для Doctor. Реалізуйте його двома способами: через is-перевірку та через switch з pattern matching. Порівняйте читабельність.

4.3. Віртуальні методи та властивості

17. Що означає ключове слово virtual у оголошенні методу? Що буде, якщо похідний клас не перевизначає віртуальний метод?
18. Що таке **поліморфізм** у контексті віртуальних методів? Поясніть, чому наступний цикл виводить різний текст для кожного елемента масиву:

```
C#
1  Person[] staff = { new Person("А",30), new Patient("Б",45,"Грип"), new
   Doctor("В",38,"Кардіо") };
2  foreach (Person p in staff)
3      p.Print();
```

19. Що таке **пізнє зв'язування** (late binding / dynamic dispatch)? Коли саме — під час компіляції чи виконання — вирішується, яка реалізація методу буде викликана?
20. Що виведе цей код?

```
C#
 1  class Person  { public virtual void Print() => Console.WriteLine("Person"); }
 2  class Patient : Person { public override void Print() =>
    Console.WriteLine("Patient"); }
 3  class Inpatient : Patient
 4  {
 5      public override void Print()
 6      {
 7          base.Print();
 8          Console.WriteLine("Inpatient");
 9      }
10  }
11  Person p = new Inpatient();
12  p.Print();
```

21. Як можна перевизначити властивість у похідному класі? Чи зобов'язані збігатися accessor'и (get/set) у базовому та похідному класі?
22. Що означає sealed override? Яку проблему він вирішує і коли виправданий?
23. Спроєктуйте клас Shape з віртуальним методом Area(), що повертає double. Реалізуйте похідні класи Circle (з радіусом) і Rectangle (з шириною і висотою). Напишіть метод, що приймає Shape[] і виводить площу кожної фігури.
24. Порівняйте два підходи до зміни поведінки в похідному класі: виклик base.Print() перед власним кодом (розширення) та повна заміна реалізації без виклику base. Коли доцільний кожен підхід?

4.4. Приховування методів та властивостей

25. Що таке **приховування методу** в C#? Яке ключове слово використовується і що станеться, якщо його не вказати?
26. Що виведе наступний код?

```
C#
1  class A { public void Print() => Console.WriteLine("A"); }
2  class B : A { public new void Print() => Console.WriteLine("B"); }
3  A obj = new B();
4  obj.Print();
```

Поясніть результат.

27. Коли використання new (приховування) є єдиним або найкращим варіантом замість override (перевизначення)?
28. Чим синтаксично відрізняється приховування властивості від перевизначення? Як у прихованій властивості звернутися до оригінальної базової реалізації?
29. Чи можна приховати const або static readonly поле у похідному класі? Який результат дасть наступний код?

```
C#
1  class Base { public const string Label = "Base"; }
2  class Child : Base { public new const string Label = "Child"; }
3  Console.WriteLine(Base.Label);
4  Console.WriteLine(Child.Label);
```

30. Поясніть ситуацію: є бібліотека з класом Logger, у якого метод Write(string msg) не є virtual. Ви хочете додати до нього форматування і створили MyLogger : Logger. Як реалізувати власну версію Write? Які обмеження вашого рішення?
31. Знайдіть потенційну проблему в наступному коді:

```
C#
1  class Animal { public void Speak() => Console.WriteLine("..."); }
2  class Dog : Animal { public new void Speak() => Console.WriteLine("Гав!"); }
3
4  Animal[] animals = { new Dog(), new Dog() };
5  foreach (Animal a in animals)
6      a.Speak();
```

Що виведе цей код і чому це може бути несподіваним?

4.5. Відмінність перевизначення та приховування методів

32. Сформулюйте у двох реченнях ключову відмінність між override і new з точки зору зв'язування (binding). Що вирішує виклик у кожному випадку?
33. Що таке **таблиця віртуальних методів (VMT)**? Як вона формується для класу і як використовується при виклику методу?
34. Заповніть таблицю: що виведуть такі виклики?

```
C#
 1  // override варіант:
 2  class A { public virtual void F() => Console.WriteLine("A"); }
 3  class B : A { public override void F() => Console.WriteLine("B"); }
 4  A x = new B(); x.F();   // ?
 5  B y = new B(); y.F();   // ?
 6
 7  // new варіант:
 8  class C { public void F() => Console.WriteLine("C"); }
 9  class D : C { public new void F() => Console.WriteLine("D"); }
10  C p = new D(); p.F();   // ?
11  D q = new D(); q.F();   // ?
```

35. Чому виклик віртуального методу дещо повільніший, ніж невіртуального? За яких умов ця різниця суттєва?
36. Якщо потрібно, щоб колекція List<Person> при переборі викликала правильну реалізацію Print() для кожного об'єкту, яку техніку слід використовувати — override чи new? Обґрунтуйте.
37. Розробник написав такий код і дивується, чому Print() у Doctor не викликається:

```
C#
1  class Person  { public void Print() => Console.WriteLine("Person"); }
2  class Doctor : Person { public new void Print() => Console.WriteLine("Doctor");
   }
3
4  Person[] team = { new Doctor("Коваль",38,"Кардіо") };
5  foreach (Person p in team) p.Print();
```

Поясніть, у чому помилка і як її виправити.

38. Порівняйте у вигляді таблиці override та new за критеріями: зв'язування, що визначає виклик (тип змінної / тип об'єкта), чи потрібен virtual у базовому, VMT-поведінка.

4.6. Абстрактні класи та члени класів

39. Чому не можна створити екземпляр абстрактного класу? Яку роль він відіграє в ієрархії?
40. Порівняйте abstract метод і virtual метод за наявністю реалізації у базовому класі та зобов'язаннями похідних класів.
41. Що виведе цей код або яку помилку він спричинить?

```
C#
 1  abstract class Shape
 2  {
 3      public abstract double Area();
 4      public void Describe() => Console.WriteLine($"Площа: {Area().ToString()}");
 5  }
 6  class Circle : Shape
 7  {
 8      double _r;
 9      public Circle(double r) { _r = r; }
10      public override double Area() => Math.PI * _r * _r;
11  }
12  Shape s = new Circle(5);
13  s.Describe();
```

42. Що буде, якщо абстрактний клас успадковує інший абстрактний клас і не реалізовує всі його абстрактні методи? Хто тоді зобов'язаний реалізувати їх?
43. Як оголосити абстрактну **властивість**? Чим її синтаксис відрізняється від оголошення абстрактного методу?
44. У чому перевага абстрактного класу порівняно зі звичайним класом, у якого virtual-методи мають «заглушку» (порожню реалізацію або throw new NotImplementedException())?
45. Спроєктуйте ієрархію для системи сповіщень: абстрактний клас Notifier з абстрактним методом Send(string message) і

конкретним методом Log(string message). Реалізуйте два похідних класи — SmsNotifier і EmailNotifier. Напишіть метод, що приймає Notifier і надсилає повідомлення.

46. Чому не можна оголосити абстрактний метод у неабстрактному класі? Поясніть логіку цього обмеження.

4.7. Клас System.Object та його методи

47. Які чотири методи гарантовано є в кожному об'єкті C#? Від чого це походить?
48. Що виведе виклик Console.WriteLine(patient), якщо клас Patient не перевизначає ToString()? Що зміниться після перевизначення?
49. Поясніть **контракт рівності**: яке правило між Equals і GetHashCode обов'язково дотримуватися і чому його порушення призведе до помилок у словниках і множинах?
50. Реалізуйте Equals і GetHashCode для класу Patient, де рівність визначається збігом поля RecordId (рядок). Що повертає ваш Equals, коли передається об'єкт іншого типу?
51. Чим відрізняється person.GetType() == typeof(Patient) від person is Patient? Наведіть приклад, коли вони дають різний результат.
52. Чому метод GetType() не можна перевизначити? Яку гарантію це забезпечує?
53. Як перевизначити ToString() так, щоб при порожньому імені поверталась базова реалізація (base.ToString()), а при заповненому — змістовний рядок? Напишіть повну реалізацію.
54. У вас є Dictionary<Patient, string>. Поясніть, чому без правильного перевизначення Equals і GetHashCode два об'єкти Patient з однаковим RecordId будуть вважатися різними ключами.

4.8. Узагальнення

55. Які дві ключові проблеми вирішують узагальнення (generics) порівняно з використанням object як типу загального призначення?
56. Що таке **параметр типу** і **аргумент типу**? Поясніть різницю на прикладі class Box<T> та Box<int> b = new Box<int>().
57. Що виведе наступний код?

```C#
class Container<T>
{
    public static int Count = 0;
    public Container() { Count++; }
}
new Container<int>();
new Container<int>();
new Container<string>();
Console.WriteLine(Container<int>.Count);
Console.WriteLine(Container<string>.Count);
```

58. Що таке **виведення типу** (type inference)? Як компілятор визначає T у виклику Swap(ref a, ref b) без явного вказання <int>?
59. Напишіть узагальнений метод FindFirst<T>(T[] array, T target), що повертає індекс першого входження елемента або -1, якщо не знайдено. Чому для порівняння елементів не можна використати == без обмеження типу?
60. Чому List<string> не можна присвоїти змінній List<object>, навіть якщо string є нащадком object? Як ця проблема пов'язана з поняттям **інваріантності** generics у C#?
61. Порівняйте List<int> і старий ArrayList з точки зору boxing/unboxing та типобезпеки. Чому перший кращий?
62. Спроєктуйте узагальнений клас Pair<T1, T2>, що зберігає два значення різних типів. Додайте метод Swap(), що повертає новий Pair<T2, T1> з переставленими значеннями. Продемонструйте використання.

4.9. Обмеження узагальнень

63. Навіщо потрібні обмеження (where) у узагальнених класах і методах? Що дозволяє зробити обмеження, що неможливо без нього?
64. Поясніть різницю між where T : class і where T : struct. Що гарантує кожне з них?
65. Що дозволяє обмеження where T : new()? Чому без нього написати new T() у тілі узагальненого класу є помилкою компіляції?
66. Чому where T : struct і where T : new() є несумісними? Поясніть логіку.
67. Що виведе цей код?

```C#
class Printer<T> where T : IComparable<T>
{
    public void PrintMax(T a, T b)
    {
        Console.WriteLine(a.CompareTo(b) >= 0 ? a : b);
    }
}
new Printer<int>().PrintMax(3, 7);
new Printer<string>().PrintMax("яблуко", "банан");
```

68. У чому різниця між where T : SomeClass і where T : ISomeInterface? Яка перевага обмеження за інтерфейсом?
69. Напишіть узагальнений метод Max<T>(T a, T b) where T : IComparable<T>, що повертає більше з двох значень. Поясніть, чому без обмеження цей метод написати неможливо.
70. Ви проєктуєте Repository<T>, де T повинен бути reference type і мати конструктор без параметрів. Запишіть правильне оголошення класу з обмеженнями і поясніть кожне.

4.10. Наслідування узагальнених типів

71. Опишіть чотири варіанти відношення між параметром типу базового і похідного узагальнених класів. Наведіть синтаксис кожного.
72. Коли доцільно фіксувати тип параметра у похідному класі (Child : Base<string>)? Що при цьому відбувається з гнучкістю похідного класу?
73. Що виведе цей код?

```C#
class Base<T>
{
    public T Value { get; }
    public Base(T v) { Value = v; }
    public override string ToString() => $"[{Value}]";
}
class Child : Base<int>
{
    public string Tag { get; }
    public Child(int v, string tag) : base(v) { Tag = tag; }
    public override string ToString() => $"{base.ToString()} — {Tag}";
}
Child c = new Child(42, "тест");
Base<int> b = c;
Console.WriteLine(c);
Console.WriteLine(b);
```

74. Якщо базовий клас Base<T> має обмеження where T : class, чи може похідний клас Child<T> : Base<T> прибрати це обмеження? Чому?
75. Ви маєте AnnotatedRecord<T, TNote> : MedicalRecord<T>. Що означає такий підпис і яку гнучкість він надає?
76. У чому різниця між Child<T> : Base<T> і Child<T> : Base<int>? Де в кожному варіанті визначається тип Id базового класу?
77. Спроєктуйте мінімальну ієрархію: абстрактний Repository<T> where T : class з методами Add(T item) і GetAll(), та конкретний PatientRepository : Repository<Patient>. Реалізуйте зберігання через масив фіксованого розміру.
78. Порівняйте варіанти FlexibleRecord<T> : MedicalRecord<T> (передача параметра) та InpatientRecord : MedicalRecord<string> (фіксація типу). Для яких задач підходить перший варіант, а для яких другий?

Розділ 5. Обробка винятків

5.1. Конструкція try..catch..finally

У реальних програмах завжди є ситуації, які важко або неможливо повністю передбачити під час написання коду. Файл може не існувати, мережа може обірватися, користувач може ввести не число, а текст. Такі позаштатні події під час виконання програми називаються **винятками** (exceptions).

У C# передбачено спеціальний механізм обробки винятків — конструкція try...catch...finally. Вона дозволяє виявити аварійну ситуацію, обробити її або зробити необхідне завершення ресурсів — і продовжити виконання програми замість аварійного завершення.

Що таке виняток

Виняток — це об'єкт, який середовище виконання CLR (Common Language Runtime) автоматично створює, коли відбувається помилка, і «викидає» вгору по стеку викликів у пошуках обробника. Якщо обробника немає — програма аварійно завершується.

Кожен виняток є екземпляром класу, що успадковується від System.Exception. Наприклад:

- DivideByZeroException — ділення цілого числа на нуль
- FormatException — некоректний формат рядка при конвертації
- NullReferenceException — звернення до null-посилання
- IndexOutOfRangeException — вихід за межі масиву

Структура try...catch...finally

Загальна форма конструкції:

```
C#
 1  try
 2  {
 3      // код, в якому може виникнути виняток
 4  }
 5  catch
 6  {
 7      // виконується, якщо виняток виник
 8  }
 9  finally
10  {
11      // виконується завжди — і при винятку, і без нього
```

```
12  }
```

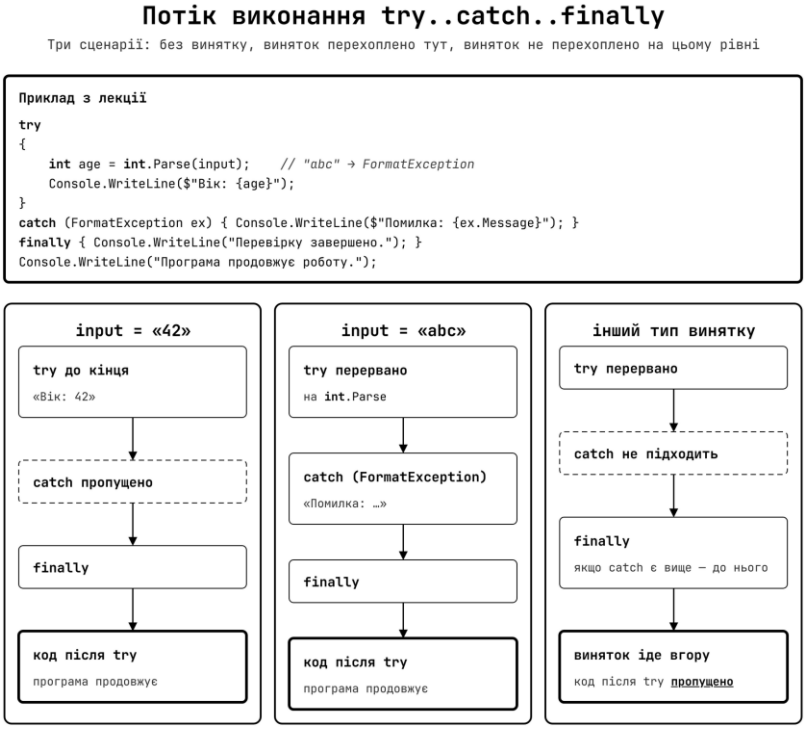


Схема виконання try..catch..finally

Логіка роботи:

1. Виконуються інструкції у блоці try.
2. Якщо жодного винятку не виникло — після try виконується finally, блок catch пропускається.
3. Якщо виняток виник — виконання try зупиняється на місці помилки. CLR шукає відповідний блок catch.
4. Якщо catch знайдено — він виконується, потім finally.
5. Якщо відповідного catch на цьому рівні немає — виняток іде вгору по стеку викликів. Якщо обробник знайдеться вище, finally цього рівня виконається під час розкрутки стека (детально — у 5.6). Якщо обробника немає ніде, програма

аварійно завершується; чи встигне виконатися finally у такому разі, стандарт не гарантує — це залежить від середовища виконання (у консольному застосунку на .NET 10 він виконується вже після повідомлення Unhandled exception, але покладатися на це не можна).

Перший приклад: парсинг даних пацієнта

Уявімо, що реєстратура вводить вік пацієнта як рядок із зовнішньої форми. Якщо рядок містить нецифрові символи, метод int.Parse() кине FormatException:

```
C#
 1  using System;
 2
 3  string input = "abc"; // некоректні дані з форми
 4
 5  try
 6  {
 7      int age = int.Parse(input);
 8      Console.WriteLine($"Вік пацієнта: {age.ToString()} р.");
 9  }
10  catch
11  {
12      Console.WriteLine("Помилка: введений вік має некоректний формат.");
13  }
14  finally
15  {
16      Console.WriteLine("Обробку запису завершено.");
17  }
18
19  Console.WriteLine("Програма продовжує роботу.");
```

Без конструкції try...catch цей код завершився б аварійно з FormatException. З обробником — програма перехоплює помилку, виводить повідомлення і продовжує роботу далі.

Зверніть увагу: рядок "Програма продовжує роботу." виводиться завжди — виняток перехоплено, а не проігноровано.

Поширення винятку по стеку викликів

Виняток не обов'язково обробляється там, де виник. Якщо в методі немає try...catch, виняток «спливає» вгору — до того методу, який викликав поточний. CLR послідовно перевіряє кожен рівень стека, поки не знайде відповідний catch.

```
C#
 1  using System;
```

```
 2
 3  RegisterPatient("Іван Петренко", "42");   // коректно
 4  RegisterPatient("Марія Сидоренко", "???"); // некоректний вік
 5
 6  void RegisterPatient(string name, string ageInput)
 7  {
 8      try
 9      {
10          int age = ParseAge(ageInput);  // виняток кинуто тут...
11          Console.WriteLine($"Зареєстровано: {name}, {age.ToString()} р.");
12      }
13      catch
14      {
15          // ...але перехоплено тут, двома рівнями вище
16          Console.WriteLine($"Помилка реєстрації пацієнта «{name}»: некоректний
    вік.");
17      }
18  }
19
20  int ParseAge(string s)
21  {
22      return int.Parse(s); // FormatException якщо s — не число
23  }
```

Якби у ParseAge не було catch, виняток піднявся б до RegisterPatient. Якби і там не було — до точки виклику, і далі. Це дозволяє централізувати обробку помилок у потрібному місці ієрархії.

Варіант: тільки try...catch (без finally)

Блок finally не є обов'язковим. Конструкція try...catch без finally — найпоширеніший варіант:

```
C#
 1  using System;
 2
 3  string[] inputs = { "120", "ab", "98", "" };
 4
 5  foreach (string raw in inputs)
 6  {
 7      try
 8      {
 9          int pressure = int.Parse(raw);
10          Console.WriteLine($"Тиск: {pressure.ToString()} мм рт.ст.");
11      }
12      catch
13      {
14          Console.WriteLine($"«{raw}» — некоректне значення тиску, запис
    пропущено.");
15      }
16  }
```

Цикл продовжується навіть після помилки: кожен запис обробляється незалежно. Некоректні дані пропускаються, але програма не зупиняється.

Варіант: тільки try...finally (без catch)

finally можна використовувати без catch — якщо ловити виняток не потрібно, але важливо гарантувати виконання завершального коду (наприклад, закрити файл або звільнити ресурс):

```
C#
 1  using System;
 2
 3  Console.WriteLine("Початок прийому пацієнта.");
 4
 5  try
 6  {
 7      ProcessRecord("P-001");
 8      Console.WriteLine("Запис оброблено успішно.");
 9  }
10  finally
11  {
12      // виконається завжди — навіть якщо кинуто виняток
13      Console.WriteLine("Сесію прийому закрито.");
14  }
15
16  void ProcessRecord(string id)
17  {
18      Console.WriteLine($"Обробка картки {id}...");
19      // якби тут виник виняток — finally все одно спрацює
20  }
```

Важливо: конструкція try...finally без catch не перехоплює виняток — вона лише гарантує виконання блоку finally перед тим, як виняток продовжить поширюватися вгору. Якщо у жодного виклику не буде catch, програма все одно аварійно завершиться.

Умовні конструкції замість try...catch

Частину потенційних помилок можна уникнути, перевіривши умову заздалегідь, — і це **ефективніший підхід**, ніж catch. Механізм винятків в CLR витрачає значно більше ресурсів, ніж звичайна умовна перевірка, тому не варто використовувати try...catch там, де можна перевірити коректність даних до виконання операції.

Наприклад, для парсингу існують TryParse-методи:

```
C#
 1  using System;
 2
```

```
 3  string[] ages = { "34", "ab", "61", "0", "xyz" };
 4
 5  foreach (string raw in ages)
 6  {
 7      if (int.TryParse(raw, out int age))
 8      {
 9          Console.WriteLine($"Вік пацієнта: {age.ToString()} р.");
10      }
11      else
12      {
13          Console.WriteLine($"«{raw}» — некоректний вік, запис пропущено.");
14      }
15  }
```

Метод int.TryParse(string, out int) повертає true, якщо конвертація успішна, і false — якщо ні. При успіху змінна age містить результат. Жодного винятку не виникає — обидва сценарії опрацьовуються нормальним потоком виконання.

Порівняйте: try...catch з int.Parse — це «обробити аварію після зіткнення». TryParse — це «перевірити умови безпеки перед стартом». Другий підхід і семантично коректніший, і продуктивніший.

Коли використовувати try...catch, а коли умовні перевірки

Ситуація	Рекомендований підхід
Парсинг числа з рядка	TryParse + умова
Перевірка на null	if (x != null) або ?.
Робота з файлом / мережею	try...catch
Звернення до бази даних	try...catch
Зовнішній код, що може кинути виняток	try...catch
Гарантоване звільнення ресурсу	try...finally або using

Правило просте: якщо помилку можна передбачити і перевірити — перевіряйте. Якщо помилка залежить від зовнішніх умов (файлова система, мережа, зовнішній сервіс) — обробляйте через try...catch.

Підсумок

— try — блок захищеного коду, де може виникнути виняток

— catch — обробник винятку; виконується лише якщо виняток виник

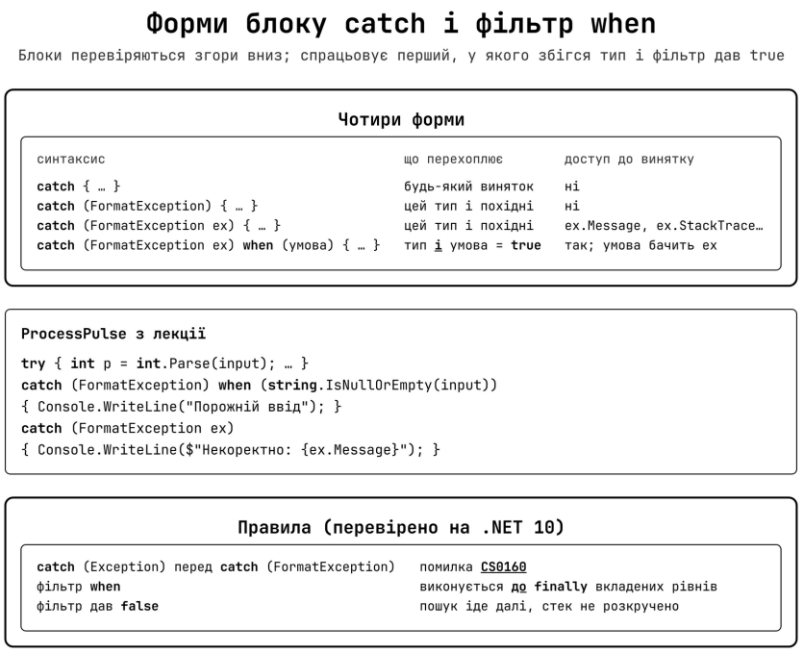
— finally — гарантовано виконується завжди, незалежно від винятку

— Виняток поширюється вгору по стеку до першого відповідного catch

— Для передбачуваних помилок краще використовувати умовні перевірки (TryParse, null-перевірки) — це ефективніше, ніж catch

5.2. Блок catch та фільтри винятків

Блок catch є серцем механізму обробки винятків. Він визначає, які саме винятки перехоплюються і яку інформацію про помилку надає розробнику. У C# є три форми блоку catch, а також можливість задати додатковий фільтр через when.



Форми блоку catch та фільтр when

Форма 1: catch без параметрів

Найпростіший варіант — catch без вказання типу. Такий блок перехоплює **будь-який** виняток, незалежно від його типу:

```
C#
1  catch
2  {
3      // виконується при будь-якому винятку
4  }
```

Це зручно на початку або коли тип помилки не важливий. Недолік: немає доступу до інформації про виняток — невідомо, що саме сталося.

```
C#
 1  using System;
 2
 3  string[] inputs = { "36.6", "abc", "37.2" };
 4
 5  foreach (string raw in inputs)
 6  {
 7      try
 8      {
 9          double temp = double.Parse(raw);
10          Console.WriteLine($"Температура: {temp.ToString()} °C");
11      }
12      catch
13      {
14          Console.WriteLine($"Помилка читання: «{raw}» — значення пропущено.");
15      }
16  }
```

Форма 2: catch з типом винятку

Якщо вказати тип у дужках, блок catch перехопить **лише** цей тип (або його похідні). Винятки інших типів через цей блок не пройдуть:

```
C#
1  catch (FormatException)
2  {
3      // обробляє лише FormatException
4  }
```

```
C#
 1  using System;
 2
 3  string input = "не_число";
 4
 5  try
 6  {
 7      int age = int.Parse(input);
 8      Console.WriteLine($"Вік: {age.ToString()} р.");
 9  }
```

```
10  catch (FormatException)
11  {
12      Console.WriteLine("Вік введено у некоректному форматі.");
13  }
14  catch (OverflowException)
15  {
16      Console.WriteLine("Число виходить за допустимий діапазон.");
17  }
```

Тут два блоки catch — кожен обробляє свій тип. Якщо рядок некоректний — спрацьовує FormatException. Якщо число занадто велике — OverflowException. Якщо виникне інший тип помилки — жоден блок не перехопить його.

Важливо: блоки catch перевіряються **зверху вниз** у порядку оголошення. Спрацьовує перший відповідний.

Форма 3: catch з типом і змінною

Найінформативніша форма: дає доступ до об'єкта винятку через іменовану змінну. Змінна містить усю доступну інформацію про помилку — передусім властивість Message:

```
C#
1  catch (FormatException ex)
2  {
3      Console.WriteLine(ex.Message); // деталі помилки
4  }
```

```
C#
 1  using System;
 2
 3  RegisterPatient("Іван Петренко", "45");
 4  RegisterPatient("Марія Сидоренко", "??");
 5  RegisterPatient("Олег Бойко", "999999999999");
 6
 7  void RegisterPatient(string name, string ageInput)
 8  {
 9      try
10      {
11          int age = int.Parse(ageInput);
12          Console.WriteLine($"Зареєстровано: {name}, {age.ToString()} р.");
13      }
14      catch (FormatException ex)
15      {
16          Console.WriteLine($"[{name}] Некоректний формат віку: {ex.Message}");
17      }
18      catch (OverflowException ex)
19      {
20          Console.WriteLine($"[{name}] Вік поза допустимим діапазоном:
    {ex.Message}");
21      }
22  }
```

Властивість ex.Message містить зрозумілий опис помилки від CLR. Це особливо корисно для логування: у виробничих системах повідомлення про помилку зазвичай записується у журнал, а не виводиться користувачу напряму.

Фільтри винятків: when

Оператор when дозволяє додати **умову** до блоку catch. Блок спрацює лише якщо умова повертає true:

```
C#
1  catch (ExceptionType ex) when (умова)
2  {
3      // виконується лише якщо умова == true
4  }
```

Якщо умова false — CLR продовжує шукати наступний відповідний catch, навіть якщо тип збігається.

Практичний приклад: в медичній системі ввід пульсу обробляється по-різному залежно від того, чи рядок порожній, чи містить некоректні символи:

```
C#
 1  using System;
 2
 3  ProcessPulse("");        // порожній рядок
 4  ProcessPulse("abc");     // некоректні символи
 5  ProcessPulse("72");      // коректне значення
 6
 7  void ProcessPulse(string input)
 8  {
 9      try
10      {
11          if (string.IsNullOrEmpty(input))
12              throw new FormatException("рядок порожній");
13
14          int pulse = int.Parse(input);
15          Console.WriteLine($"Пульс: {pulse.ToString()} уд/хв");
16      }
17      catch (FormatException ex) when (string.IsNullOrEmpty(input))
18      {
19          Console.WriteLine($"Пульс не введено (порожній рядок): {ex.Message}");
20      }
21      catch (FormatException ex)
22      {
23          Console.WriteLine($"Некоректне значення пульсу «{input}»:
    {ex.Message}");
24      }
25  }
```

Обидва catch обробляють FormatException, але:

— перший — лише якщо input порожній (when (string.IsNullOrEmpty(input)))

— другий — у всіх інших випадках FormatException

Це дозволяє розрізнити різні смислові сценарії для одного типу винятку — без вкладених if у тілі catch.

Антипатерн: «ковтання» винятків

Один із найпоширеніших і найнебезпечніших патернів — блок catch, що не робить нічого або просто виводить повідомлення, але не сигналізує про помилку далі:

```
C#
 1  // ПОГАНО — помилка «ковтається», програма продовжує роботу з пошкодженим
    станом:
 2  try
 3  {
 4      int age = int.Parse(input);
 5      SaveToDatabase(age);
 6  }
 7  catch (Exception)
 8  {
 9      // нічого не робимо
10  }
```

У медичній системі це особливо небезпечно: якщо SaveToDatabase кинула виняток (зв'язок з БД обірвався), а помилка «ковтнута» — дані пацієнта мовчки не зберігаються. Система вважає, що все добре, але картка так і не записана.

```
C#
 1  using System;
 2
 3  // Симуляція: критична помилка "ковтається"
 4  string patientData = "P-1001";
 5
 6  Console.WriteLine("=== Антипатерн: ковтання помилки ===");
 7  try
 8  {
 9      SaveRecord(patientData); // кинуть Exception
10  }
11  catch (Exception)
12  {
13      // нічого — помилка зникла
14  }
15  Console.WriteLine("Програма вважає, що дані збережено. Насправді — ні.");
16
17  Console.WriteLine("\n=== Правильно: логувати і перекидати або обробляти явно
    ===");
18  try
19  {
```

```
20      SaveWithLogging(patientData); // логує і перекидає
21  }
22  catch (InvalidOperationException ex)
23  {
24      // верхній рівень знає, що робити: повідомити користувача, повторити спробу
    тощо
25      Console.WriteLine($"[ВЕРХНІЙ РІВЕНЬ] Запис не збережено: {ex.Message}");
26  }
27
28  void SaveWithLogging(string data)
29  {
30      try
31      {
32          SaveRecord(data);
33      }
34      catch (InvalidOperationException ex)
35      {
36          Console.WriteLine($"[ПОМИЛКА ЗБЕРЕЖЕННЯ] {ex.Message}");
37          // у реальному коді: logger.Error(ex)
38          throw; // перекидаємо, щоб вище знали про помилку
39      }
40  }
41
42  void SaveRecord(string data) =>
43      throw new InvalidOperationException($"З'єднання з БД втрачено під час
    запису {data}.");
```

Зверніть увагу: throw; має сенс лише тоді, коли вище є хтось, хто перехопить виняток. Якщо перекинути його з верхнього рівня програми, вона аварійно завершиться — тому в прикладі перекидання винесено в метод SaveWithLogging, а остаточна обробка — на верхньому рівні.

Правила:

— **Ніколи не залишайте catch порожнім** у виробничому коді — принаймні логуйте виняток

— **catch (Exception)** — тільки на верхньому рівні (main loop, middleware), і лише якщо ви справді знаєте, що з ним робити

— Якщо не знаєте, як обробити виняток — використовуйте throw;, щоб він «спливав» вище

Порядок блоків catch

Кілька блоків catch перевіряються у порядку оголошення. Є одне важливе правило: **загальніший тип має йти після конкретнішого**. Наприклад, Exception (базовий тип для всіх) повинен бути останнім, інакше він перехопить усе раніше:

```
C#
 1  using System;
 2
```

```
 3  string[] records = { "120", "abc", "0", "37" };
 4
 5  foreach (string r in records)
 6  {
 7      try
 8      {
 9          int val = int.Parse(r);
10          if (val == 0) throw new DivideByZeroException();
11          Console.WriteLine($"Значення: {val.ToString()}");
12      }
13      catch (FormatException ex)
14      {
15          Console.WriteLine($"Формат: {ex.Message}");
16      }
17      catch (DivideByZeroException)
18      {
19          Console.WriteLine("Значення не може бути нулем.");
20      }
21      catch (Exception ex)
22      {
23          // загальний — завжди останнім
24          Console.WriteLine($"Невідома помилка: {ex.Message}");
25      }
26  }
```

Якщо поставити catch (Exception ex) першим — він перехопить усе, і специфічні блоки ніколи не виконаються. Компілятор цього не допустить: такий порядок — **помилка компіляції** CS0160 («попередній catch уже перехоплює всі винятки цього або базового типу»), а не попередження (перевірено на .NET 10).

Підсумок

Форма	Синтаксис	Що перехоплює
Без параметрів	catch { }	Будь-який виняток
З типом	catch (FormatException) { }	Лише вказаний тип
З типом і змінною	catch (FormatException ex) { }	Лише вказаний тип + доступ до ex.Message
З фільтром	catch (FormatException ex) when (умова) { }	Тип збігається **і** умова true

> Блоки перевіряються зверху вниз. Загальніший тип (Exception) завжди ставте останнім.

5.3. Типи винятків. Клас Exception

Кожен виняток у C# — це об'єкт. Усі класи винятків успадковуються від базового класу System.Exception. Це означає, що кожен виняток гарантовано містить набір стандартних властивостей з інформацією про помилку, а catch (Exception ex) здатен перехопити будь-який виняток у системі.

Клас Exception та його властивості

System.Exception визначає кілька ключових властивостей:

Властивість	Тип	Що містить
Message	string	Текстовий опис помилки
StackTrace	string	Стек викликів на момент винятку
InnerException	Exception?	Виняток, що спричинив поточний (якщо є)
TargetSite	MethodBase?	Метод, у якому виник виняток
Source	string?	Назва збірки або об'єкта-джерела

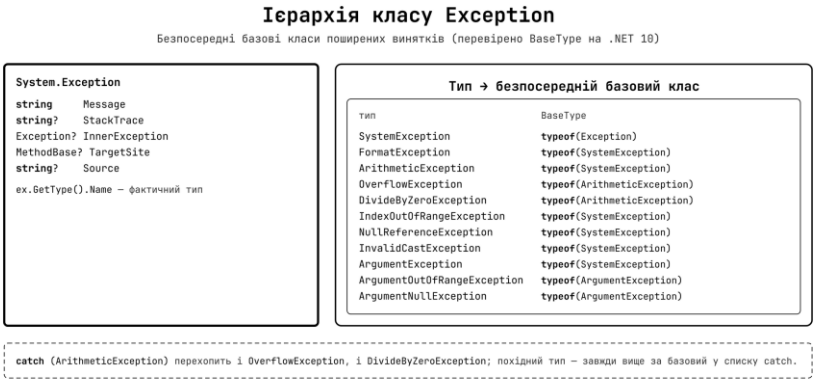
```C#
using System;

try
{
    string input = "не_число";
    int age = int.Parse(input);
    Console.WriteLine($"Вік: {age.ToString()} р.");
}
catch (Exception ex)
{
    Console.WriteLine($"Тип:      {ex.GetType().Name}");
    Console.WriteLine($"Повідомлення: {ex.Message}");
    Console.WriteLine($"Метод:    {ex.TargetSite}");
}
```

Властивість ex.GetType().Name показує реальний тип винятку — тут це буде FormatException, хоча перехоплюємо через базовий

Exception. Це корисно для логування: ми отримуємо і конкретний тип, і загальні властивості.

Ієрархія типів винятків



Ієрархія класу Exception та поширені похідні типи

У .NET існує багато спеціалізованих класів винятків. Кожен відповідає за конкретну категорію помилок. Найпоширеніші:

Тип	**Коли виникає**
FormatException	Некоректний формат рядка при Parse
OverflowException	Переповнення числового типу
DivideByZeroException	Ділення цілого числа на нуль
IndexOutOfRangeException	Вихід за межі масиву або рядка
NullReferenceException	Звернення до null-посилання
InvalidCastException	Неприпустиме явне перетворення типів
ArgumentException	Некоректний аргумент методу
ArgumentOutOfRangeException	Аргумент поза допустимим діапазоном

Усі вони є похідними від Exception, тому catch (Exception ex) перехопить кожен із них. Але похідними **не безпосередньо**: між ними і Exception є проміжні класи. Більшість системних винятків успадковується від SystemException; OverflowException і DivideByZeroException — від ArithmeticException; ArgumentOutOfRangeException і ArgumentNullException — від ArgumentException (перевірено властивістю Type.BaseType на .NET 10). Тому, наприклад, catch (ArithmeticException) перехопить і переповнення, і ділення на нуль.

Кілька блоків catch для різних типів

Якщо у блоці try можуть виникнути різні типи помилок, варто обробляти їх окремо — це дозволяє давати більш точні повідомлення та вживати відповідних заходів:

```
C#
 1  using System;
 2
 3  ProcessRecord("P-001", "45");    // коректно
 4  ProcessRecord("P-002", "abc");   // FormatException
 5  ProcessRecord("P-003", "");      // FormatException (порожній рядок)
 6
 7  void ProcessRecord(string id, string ageInput)
 8  {
 9      try
10      {
11          int age = int.Parse(ageInput);
12
13          string[] allowedIds = { "P-001", "P-003" };
14          int idx = Array.IndexOf(allowedIds, id);
15
16          if (allowedIds[idx] != id)   // IndexOutOfRangeException якщо idx == -1
17              throw new ArgumentException($"ID {id} не знайдено");
18
19          Console.WriteLine($"[{id}] Вік: {age.ToString()} р. — запис
    збережено.");
20      }
21      catch (FormatException ex)
22      {
23          Console.WriteLine($"[{id}] Некоректний вік: {ex.Message}");
24      }
25      catch (IndexOutOfRangeException)
26      {
27          Console.WriteLine($"[{id}] Запис не знайдено у системі.");
28      }
29      catch (Exception ex)
30      {
31          Console.WriteLine($"[{id}] Непередбачена помилка: {ex.Message}");
32      }
33  }
```

CLR перевіряє блоки catch зверху вниз і виконує **перший**, тип якого відповідає типу винятку. Після цього решта блоків ігнорується.

Один виняток зупиняє виконання try

Важливо розуміти: як тільки в try виникає виняток, всі рядки після нього **не виконуються**. CLR негайно передає керування відповідному catch:

```
C#
using System;

string[] names   = { "Іван Петренко", "Марія Сидоренко" };
string[] ageStrs = { "45", "abc" };

try
{
    for (int i = 0; i < names.Length; i++)
    {
        int age = int.Parse(ageStrs[i]);         // FormatException на i==1
        Console.WriteLine($"{names[i]}: {age.ToString()} р.");
        // рядок нижче не виконається після винятку
    }
    Console.WriteLine("Всі записи оброблено.");  // пропускається
}
catch (FormatException ex)
{
    Console.WriteLine($"Помилка: {ex.Message}");
}

Console.WriteLine("Програма завершила роботу.");
```

Рядок "Всі записи оброблено." ніколи не виведеться, якщо виняток виник до кінця циклу. Рядок після всього блоку try...catch — "Програма завершила роботу." — виведеться завжди, оскільки виняток було перехоплено.

Необроблений виняток аварійно завершує програму

Якщо тип винятку не відповідає жодному catch — виняток не перехоплюється. CLR шукає обробник вище по стеку, і якщо ніхто не перехопив — програма завершується аварійно:

```
C#
using System;

try
{
    object obj = "P-001";
    int id = (int)obj;           // InvalidCastException
    Console.WriteLine(id.ToString());
```

```
}
catch (FormatException)
{
    Console.WriteLine("FormatException — не перехоплено InvalidCastException");
}
catch (DivideByZeroException)
{
    Console.WriteLine("DivideByZeroException — також не підходить");
}
catch (Exception ex)
{
    // Exception — базовий тип, перехопить усе що не піймали вище
    Console.WriteLine($"Перехоплено базовим Exception: {ex.GetType().Name}");
    Console.WriteLine(ex.Message);
}
```

Блок catch (Exception ex) тут виступає «страхувальною сіткою» — він обробляє будь-який виняток, якого не перехопили конкретніші блоки.

Правила порядку блоків catch

6. **Конкретніший тип — раніше**: FormatException перед Exception
7. **Загальний Exception — завжди останнім**: інакше він перехопить усе, і специфічні блоки стануть недосяжним кодом — компілятор видасть помилку
8. **Похідний клас — перед базовим**: якщо ArgumentOutOfRangeException (похідний від ArgumentException) стоїть після ArgumentException — він ніколи не спрацює

```
C#
using System;

string input = "9999999999999";

try
{
    int age = int.Parse(input);
    Console.WriteLine($"Вік: {age.ToString()} р.");
}
catch (OverflowException ex)
{
    // конкретніший — перший
    Console.WriteLine($"Переповнення: {ex.Message}");
}
catch (FormatException ex)
{
    Console.WriteLine($"Формат: {ex.Message}");
}
catch (Exception ex)
```

```
20  {
21      // загальний — останній
22      Console.WriteLine($"Інша помилка: {ex.Message}");
23  }
```

Підсумок

- System.Exception — базовий клас для всіх винятків, містить Message, StackTrace, InnerException, TargetSite
- Кожен виняток — це об'єкт конкретного класу в ієрархії Exception
- Блоки catch перевіряються зверху вниз; спрацьовує перший відповідний
- Загальний catch (Exception ex) — страхувальна сітка, завжди останній
- Виняток зупиняє виконання try одразу; код після місця помилки не виконується

5.4. Генерація винятку та оператор throw

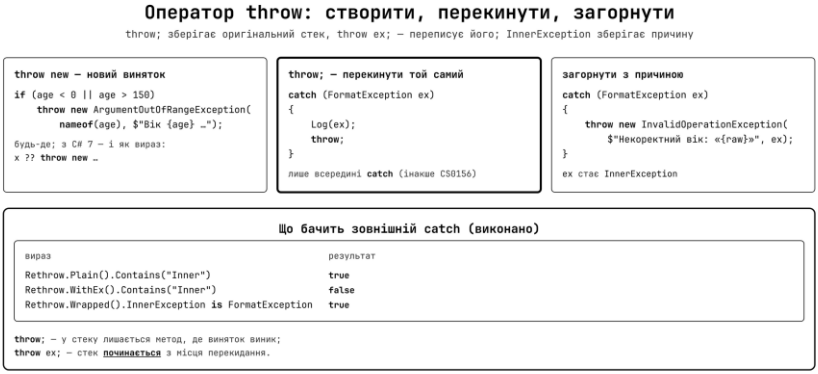
До цього моменту ми розглядали лише **перехоплення** винятків, які генерує сама система. Але в C# будь-який метод може **генерувати виняток вручну** — оператором throw. Це ключовий інструмент для впровадження бізнес-правил і валідації даних.

Навіщо генерувати виняток вручну

Іноді дані технічно коректні (рядок не порожній, число у діапазоні), але порушують бізнес-правило. Наприклад:

- вік пацієнта не може бути від'ємним
- діагноз не може бути порожнім рядком
- ідентифікатор картки повинен починатися з певного префікса

У таких випадках система не знає, що дані некоректні — це знаємо лише ми. Оператор throw дозволяє сигналізувати про помилку стандартним механізмом винятків.



Дві форми оператора throw та ланцюжок винятків

Форма 1: throw new — генерація нового винятку

```
C#
1  throw new ExceptionType("повідомлення про помилку");
```

Після throw вказується **новий об'єкт** будь-якого класу винятку. Рядок у конструкторі стає значенням властивості Message.

```
C#
 1  using System;
 2
 3  RegisterPatient("Іван Петренко", 45);
 4  RegisterPatient("", 30);
 5  RegisterPatient("Марія Сидоренко", -5);
 6
 7  void RegisterPatient(string name, int age)
 8  {
 9      if (string.IsNullOrWhiteSpace(name))
10          throw new ArgumentException("Ім'я пацієнта не може бути порожнім.");
11
12      if (age < 0 || age > 150)
13          throw new ArgumentOutOfRangeException(nameof(age),
14              $"Вік {age.ToString()} є неприпустимим значенням.");
15
16      Console.WriteLine($"Зареєстровано: {name}, {age.ToString()} р.");
17  }
```

Цей код компілюється, але виконати його до кінця не вдасться: вже на другому виклику RegisterPatient("", 30) виняток ArgumentException ніхто не перехоплює, і програма аварійно завершується (Unhandled exception), а третій виклик так і не відбудеться. Загорнемо виклики в try...catch:

```
C#
 1  using System;
 2
 3  TryRegister("Іван Петренко", 45);
 4  TryRegister("", 30);
 5  TryRegister("Марія Сидоренко", -5);
 6
 7  void TryRegister(string name, int age)
 8  {
 9      try
10      {
11          RegisterPatient(name, age);
12      }
13      catch (ArgumentException ex)
14      {
15          Console.WriteLine($"Помилка реєстрації: {ex.Message}");
16      }
17  }
18
19  void RegisterPatient(string name, int age)
20  {
21      if (string.IsNullOrWhiteSpace(name))
22          throw new ArgumentException("Ім'я пацієнта не може бути порожнім.");
23
24      if (age < 0 || age > 150)
25          throw new ArgumentOutOfRangeException(nameof(age),
26              $"Вік {age.ToString()} є неприпустимим значенням.");
27
28      Console.WriteLine($"Зареєстровано: {name}, {age.ToString()} р.");
29  }
```

Зверніть увагу: throw у методі RegisterPatient не потребує try...catch всередині нього. Виняток «спливає» вгору до місця виклику, де і перехоплюється.

Throw у методах: валідація аргументів

Найпоширеніший патерн — перевіряти аргументи на початку методу і кидати ArgumentException або ArgumentNullException при порушенні:

```
C#
 1  using System;
 2
 3  SetBloodPressure("П-001", 120, 80);
 4  SetBloodPressure("П-002", 300, 80);   // некоректний тиск
 5  SetBloodPressure("П-003", 120, -10);  // некоректне значення
 6
 7  void SetBloodPressure(string patientId, int systolic, int diastolic)
 8  {
 9      try
10      {
11          ValidatePressure(systolic, diastolic);
12          Console.WriteLine($"[{patientId}] Тиск:
    {systolic.ToString()}/{diastolic.ToString()} мм рт.ст.");
13      }
14      catch (ArgumentOutOfRangeException ex)
```

```
    {
        Console.WriteLine($"[{patientId}] {ex.Message}");
    }
}

void ValidatePressure(int systolic, int diastolic)
{
    if (systolic < 60 || systolic > 250)
        throw new ArgumentOutOfRangeException(nameof(systolic),
            $"Систолічний тиск {systolic.ToString()} поза нормою (60–250).");
    if (diastolic < 40 || diastolic > 150)
        throw new ArgumentOutOfRangeException(nameof(diastolic),
            $"Діастолічний тиск {diastolic.ToString()} поза нормою (40–150).");
}
```

Форма 2: throw без аргументів — перекидання винятку

Оператор throw без об'єкта можна використовувати **лише всередині блоку catch**. Він повторно кидає той самий виняток, що перехоплено — без змін і зі збереженням оригінального стека викликів:

```
C#
using System;

try
{
    ProcessMedicalRecord("P-001", "not-a-number");
}
catch (Exception ex)
{
    Console.WriteLine($"[Зовнішній catch] {ex.GetType().Name}: {ex.Message}");
}

void ProcessMedicalRecord(string id, string ageInput)
{
    try
    {
        int age = int.Parse(ageInput);
        Console.WriteLine($"[{id}] Вік: {age.ToString()} р.");
    }
    catch (FormatException ex)
    {
        Console.WriteLine($"[{id}] Логування помилки: {ex.Message}");
        throw;   // перекидаємо далі — зовнішній catch отримає той самий об'єкт
    }
}
```

Внутрішній catch логує помилку, але не пригнічує її — throw; передає виняток зовнішньому блоку. **Стек викликів залишається незмінним**: зовнішній обробник бачить оригінальне місце виникнення помилки.

throw як вираз і ThrowIfNull

З C# 7 throw може бути не лише інструкцією, а й **виразом** — у тернарному операторі, після ?? та в тілі-виразі =>. Це дозволяє компактно перевіряти аргументи:

```
C#
1  public Patient(string name) =>
2      Name = name ?? throw new ArgumentNullException(nameof(name));
```

Для найчастіших перевірок .NET має готові статичні методи: ArgumentNullException.ThrowIfNull(name) (.NET 6), ArgumentException.ThrowIfNullOrWhiteSpace(name) (.NET 8), ArgumentOutOfRangeException.ThrowIfNegative(age) (.NET 8). Вони самі підставляють ім'я параметра в повідомлення.

throw; vs throw ex; — важлива різниця

```
C#
 1  // Правильно — зберігає оригінальний StackTrace:
 2  catch (Exception ex)
 3  {
 4      Log(ex);
 5      throw;
 6  }
 7
 8  // Небезпечно — скидає StackTrace до цього рядка:
 9  catch (Exception ex)
10  {
11      Log(ex);
12      throw ex;   // StackTrace тепер вказує на цей рядок, а не на першопричину
13  }
```

throw ex; замінює стек викликів на поточне місце, що ускладнює діагностику — у логах не буде видно, де насправді сталася помилка. Завжди використовуйте throw; для перекидання.

Ланцюжок винятків через InnerException

Іноді доцільно «загорнути» низькорівневий виняток у вищорівневий — зі збереженням оригіналу як InnerException. Це дозволяє надати більш змістовний контекст, не втрачаючи першопричину:

```
C#
 1  using System;
 2
 3  LoadPatientRecord("P-001", "45");
 4  LoadPatientRecord("P-002", "??");
```

```
 5
 6  void LoadPatientRecord(string id, string rawAge)
 7  {
 8      try
 9      {
10          int age = ParseAge(rawAge);
11          Console.WriteLine($"[{id}] Вік: {age.ToString()} р. — завантажено.");
12      }
13      catch (Exception ex)
14      {
15          Console.WriteLine($"[{id}] Помилка: {ex.Message}");
16          if (ex.InnerException != null)
17              Console.WriteLine($"       Причина: {ex.InnerException.Message}");
18      }
19  }
20
21  int ParseAge(string raw)
22  {
23      try
24      {
25          return int.Parse(raw);
26      }
27      catch (FormatException ex)
28      {
29          // загортаємо у більш змістовний виняток, зберігаючи оригінал
30          throw new InvalidOperationException(
31              $"Не вдалося розпізнати вік зі значення «{raw}».", ex);
32      }
33  }
```

Конструктор більшості класів винятків приймає другим параметром innerException. Завдяки цьому зовнішній код може перевірити ex.InnerException і отримати повний ланцюжок причин.

Підсумок

Форма	Де використовується	Що робить
throw new Ex("msg")	Будь-де	Генерує новий виняток
throw new Ex("msg", innerEx)	Будь-де	Генерує виняток із збереженням причини
throw;	Лише у catch	Перекидає поточний виняток, зберігаючи StackTrace
throw ex;	Будь-де, де є змінна винятку (зазвичай у catch)	Кидає той самий об'єкт, але StackTrace **починається заново** з цього місця — уникайте

> throw у методі — це не помилка, а норма. Метод має «кидати» виняток, якщо не може виконати свою роботу коректно. Обробку помилки слід залишати тому рівню, який знає, що з нею робити.

5.5. Створення класів винятків

Вбудованих типів винятків (.NET надає їх понад 200) часто достатньо для технічних помилок. Але для **доменних помилок** — тих, що описують порушення бізнес-правил конкретної системи, — краще створювати власні класи винятків. Це дозволяє:

- точно вказувати catch на потрібну категорію помилок
- передавати у виняток додаткову доменну інформацію (ID пацієнта, некоректне значення)
- будувати ієрархії винятків під свою предметну область

Мінімальний власний клас винятку

Будь-який клас, що успадковується від Exception, є повноцінним типом винятку. Мінімальна реалізація — конструктор із рядком повідомлення:

```
C#
 1  using System;
 2
 3  try
 4  {
 5      RegisterPatient("Tom", 17);
 6  }
 7  catch (PatientException ex)
 8  {
 9      Console.WriteLine($"Помилка пацієнта: {ex.Message}");
10  }
11
12  void RegisterPatient(string name, int age)
13  {
14      if (age < 0 || age > 150)
15          throw new PatientException($"Вік {age.ToString()} є неприпустимим для
    пацієнта.");
16      Console.WriteLine($"Зареєстровано: {name}, {age.ToString()} р.");
17  }
18
19  class PatientException : Exception
20  {
21      public PatientException(string message) : base(message) { }
22  }
```

Конструктор передає рядок у базовий Exception через base(message). Тепер у catch можна точно вказати PatientException — і він не перехопить FormatException чи NullReferenceException.

Власні властивості у класі винятку

Справжня сила власних винятків — можливість зберігати доменні дані прямо в об'єкті помилки. Наприклад, некоректне значення, ID пацієнта, назву поля:

```C#
using System;

try
{
    RegisterPatient("Марія Сидоренко", -5);
}
catch (PatientAgeException ex)
{
    Console.WriteLine($"Помилка: {ex.Message}");
    Console.WriteLine($"Некоректний вік: {ex.InvalidAge.ToString()} р.");
}

void RegisterPatient(string name, int age)
{
    if (age < 0 || age > 150)
        throw new PatientAgeException(
            $"Вік {age.ToString()} виходить за допустимий діапазон (0–150).", age);
    Console.WriteLine($"Зареєстровано: {name}, {age.ToString()} р.");
}

class PatientAgeException : Exception
{
    public int InvalidAge { get; }

    public PatientAgeException(string message, int invalidAge) : base(message)
    {
        InvalidAge = invalidAge;
    }
}
```

Властивість InvalidAge дозволяє обробнику отримати конкретне значення, що спричинило помилку — без парсингу рядка Message. Це зручно для логування та відображення деталей у UI.

Вибір базового класу

Власний виняток не обов'язково успадковувати від кореневого Exception. Якщо є більш підходящий вбудований тип — краще

успадкувати від нього. Тоді catch (ArgumentException) також перехопить ваш клас:

```C#
using System;

// catch (ArgumentException) перехопить PatientValidationException теж
try
{
    SetDiagnosis("П-001", "");
}
catch (PatientValidationException ex)
{
    Console.WriteLine($"Валідація: {ex.Message} (поле: {ex.FieldName})");
}
catch (ArgumentException ex)
{
    Console.WriteLine($"Аргумент: {ex.Message}");
}

void SetDiagnosis(string patientId, string diagnosis)
{
    if (string.IsNullOrWhiteSpace(diagnosis))
        throw new PatientValidationException(
            "Діагноз не може бути порожнім.", nameof(diagnosis));
}

// Успадковуємо від ArgumentException — семантично коректно:
// це помилка некоректного аргументу в доменному контексті
class PatientValidationException : ArgumentException
{
    public string FieldName { get; }

    public PatientValidationException(string message, string fieldName)
        : base(message)
    {
        FieldName = fieldName;
    }
}
```

Ієрархія власних винятків

Для складніших систем будують цілі ієрархії. Базовий доменний виняток об'єднує всі помилки предметної області, а конкретні типи уточнюють причину:



Ієрархія власних класів винятків клінічної системи

```C#
using System;

ProcessRecord("П-001", "Іван Петренко", 45, "J18.9");
ProcessRecord("П-002", "", 30, "J18.9");
ProcessRecord("П-003", "Олег Бойко", -3, "J18.9");
ProcessRecord("П-004", "Марія Коваль", 28, "");

void ProcessRecord(string id, string name, int age, string diagnosisCode)
{
    try
    {
        ValidateRecord(id, name, age, diagnosisCode);
        Console.WriteLine($"[{id}] {name}, {age.ToString()} р., діагноз: {diagnosisCode} — збережено.");
    }
    catch (PatientAgeException ex)
    {
        Console.WriteLine($"[{id}] Вік: {ex.Message} (значення: {ex.InvalidAge.ToString()})");
    }
    catch (DiagnosisException ex)
```

```
    {
        Console.WriteLine($"[{id}] Діагноз: {ex.Message} (код:
{ex.DiagnosisCode})");
    }
    catch (MedicalException ex)
    {
        // перехоплює будь-який MedicalException, що не потрапив вище
        Console.WriteLine($"[{id}] Медична помилка: {ex.Message}");
    }
}

void ValidateRecord(string id, string name, int age, string diagnosisCode)
{
    if (string.IsNullOrWhiteSpace(name))
        throw new MedicalException("Ім'я пацієнта не може бути порожнім.", id);

    if (age < 0 || age > 150)
        throw new PatientAgeException(
            $"Неприпустимий вік для пацієнта.", id, age);

    if (string.IsNullOrWhiteSpace(diagnosisCode))
        throw new DiagnosisException(
            "Код діагнозу не може бути порожнім.", id, diagnosisCode ?? "");
}

// Базовий доменний виняток
class MedicalException : Exception
{
    public string PatientId { get; }

    public MedicalException(string message, string patientId) : base(message)
    {
        PatientId = patientId;
    }
}

// Конкретний: помилка віку
class PatientAgeException : MedicalException
{
    public int InvalidAge { get; }

    public PatientAgeException(string message, string patientId, int
invalidAge)
        : base(message, patientId)
    {
        InvalidAge = invalidAge;
    }
}

// Конкретний: помилка діагнозу
class DiagnosisException : MedicalException
{
    public string DiagnosisCode { get; }

    public DiagnosisException(string message, string patientId, string
diagnosisCode)
        : base(message, patientId)
    {
        DiagnosisCode = diagnosisCode;
    }
```

```
77  }
```

Ієрархія дає гнучкість: можна перехопити конкретний тип або всі помилки предметної області через базовий MedicalException.

Де оголошувати власні класи винятків

Кілька практичних правил:

- Ім'я завжди завершується словом Exception: PatientAgeException, DiagnosisException
- Клас розташовують поруч із тим, де він використовується (або в окремому файлі для великих проєктів)
- Якщо виняток вказує на некоректний аргумент — успадковуйте від ArgumentException
- Якщо виняток описує стан об'єкта, що не дозволяє операцію — від InvalidOperationException
- Якщо помилка специфічна для вашої предметної області — від Exception або власного базового

Підсумок

- Власний клас винятку — це звичайний клас, що успадковує Exception (або його похідний)
- Мінімальна реалізація: конструктор з string message, що передає його в base(message); настанови .NET рекомендують три стандартні конструктори — (), (string message) і (string message, Exception inner), щоб власний виняток можна було використовувати для загортання причини
- Додаткові властивості зберігають доменний контекст: ID, некоректне значення, назву поля
- Ієрархії дозволяють перехоплювати як конкретні типи, так і всю категорію помилок
- Ім'я класу завжди закінчується на Exception

5.6. Пошук блоку catch при обробці винятків

Виняток, що виник у методі, не зобов'язаний оброблятися там само. Якщо у поточному методі немає відповідного catch, CLR **піднімається вгору по стеку викликів** — до того методу, який викликав поточний,

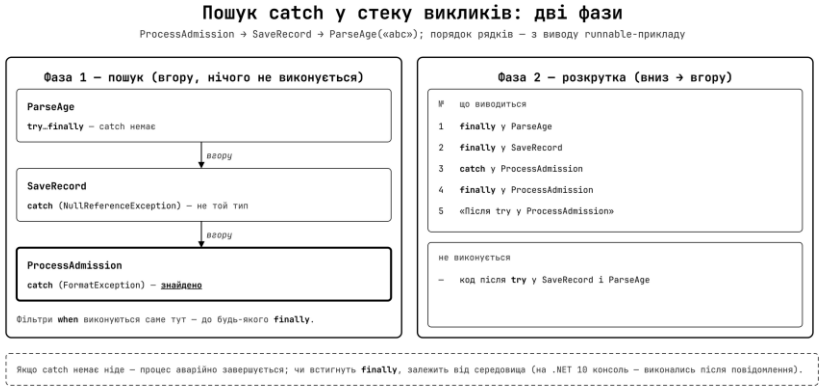
і так далі. Це дозволяє централізувати обробку помилок на потрібному рівні, не захаращуючи кожен метод блоками try...catch.

Як CLR шукає обробник: дві фази

Коли виникає виняток, CLR виконує **дві окремі фази**:

Фаза 1 — пошук: CLR рухається вгору по стеку від місця винятку, перевіряючи кожен рівень на наявність відповідного catch. Поки обробник не знайдено — жодний catch і жодний finally **не виконується**. Виняток становлять лише фільтри when: щоб вирішити, чи підходить блок, CLR виконує умову фільтра саме під час цієї фази. Тому код у when виконується **раніше** за блоки finally вкладених рівнів — перевірено на .NET 10: порядок виводу «filter → inner finally → catch». Це іноді використовують для логування винятку з повним станом стека до його розкрутки (catch (Exception ex) when (Log(ex)), де Log повертає false).

Фаза 2 — розкрутка стека: Щойно відповідний catch знайдено, CLR повертається від місця помилки **назад донизу**, виконуючи по дорозі всі блоки finally. Потім виконується знайдений catch.



Пошук catch у стеку викликів та порядок виконання finally

Покроковий приклад

Розглянемо ланцюжок методів клінічної системи: ProcessAdmission → SaveRecord → ParseAge. Виняток виникає у найглибшому методі:

```
C#
 1  using System;
```

```
 2
 3  ProcessAdmission("П-001", "Іван Петренко", "abc");
 4
 5  void ProcessAdmission(string id, string name, string ageInput)
 6  {
 7      try
 8      {
 9          SaveRecord(id, name, ageInput);
10          Console.WriteLine($"[{id}] Госпіталізацію завершено.");
11      }
12      catch (FormatException ex)
13      {
14          // CLR знайшов catch тут — на 3-му рівні
15          Console.WriteLine($"[{id}] Помилка даних: {ex.Message}");
16      }
17      finally
18      {
19          Console.WriteLine($"[{id}] finally у ProcessAdmission — виконується 3-
    м");
20      }
21      Console.WriteLine($"[{id}] Після try у ProcessAdmission.");
22  }
23
24  void SaveRecord(string id, string name, string ageInput)
25  {
26      try
27      {
28          int age = ParseAge(ageInput);   // виняток виникне тут
29          Console.WriteLine($"[{id}] {name}, {age.ToString()} р. — збережено.");
30      }
31      catch (NullReferenceException ex)
32      {
33          // не підходить для FormatException — пропускається
34          Console.WriteLine($"[{id}] NullRef: {ex.Message}");
35      }
36      finally
37      {
38          Console.WriteLine($"[{id}] finally у SaveRecord — виконується 2-м");
39      }
40      Console.WriteLine($"[{id}] Після try у SaveRecord."); // НЕ виконається
41  }
42
43  int ParseAge(string raw)
44  {
45      // try без catch — finally є, але catch немає
46      try
47      {
48          return int.Parse(raw);   // FormatException тут
49      }
50      finally
51      {
52          Console.WriteLine("finally у ParseAge — виконується 1-м");
53      }
54      // сюди не дійде — НЕ виконається
55  }
```

Порядок виведення показує обидві фази чітко:

9. finally у ParseAge — перший на шляху розкрутки

10. finally у SaveRecord — другий
11. catch у ProcessAdmission — виконується після всіх finally
12. finally у ProcessAdmission — останній
13. "Після try у ProcessAdmission." — виконується, бо виняток перехоплено

Що не виконується

Коли виняток не перехоплено на певному рівні, **увесь код після блоку try...catch на цьому рівні пропускається**. У прикладі вище:

— "Після try у SaveRecord." — не виводиться, хоча SaveRecord має finally

— Код у ParseAge після int.Parse(raw) — не досягається взагалі

Це принципово важливо: виняток негайно перериває нормальне виконання і передає керування механізму пошуку обробника.

Якщо catch не знайдено ніде

Якщо CLR пройде весь стек і не знайде жодного відповідного catch — програма аварійно завершується. При цьому блоки finally **можуть не виконатися** (залежить від середовища та типу завершення). Саме тому завжди варто мати «страхувальний» catch (Exception ex) на верхньому рівні програми:

```
C#
 1  using System;
 2
 3  // Верхній рівень — перехоплює все непередбачене
 4  try
 5  {
 6      RunClinicSystem();
 7  }
 8  catch (Exception ex)
 9  {
10      Console.WriteLine($"Критична помилка системи: {ex.GetType().Name}");
11      Console.WriteLine(ex.Message);
12  }
13
14  void RunClinicSystem()
15  {
16      // Симулюємо необроблений виняток у глибині системи
17      ProcessPatient("П-999", null!);
18  }
19
20  void ProcessPatient(string id, string name)
21  {
22      // NullReferenceException — не перехоплюється ніде нижче
23      Console.WriteLine($"Довжина імені: {name.Length.ToString()}");
```

```
24  }
```

Практичний патерн: логування + перекидання

Поширений підхід — логувати помилку на проміжному рівні через finally або catch + throw, а фактично обробляти її вище:

```
C#
 1  using System;
 2
 3  try
 4  {
 5      AdmitPatient("П-101", "Олена Коваль", "сорок п'ять");
 6  }
 7  catch (Exception ex)
 8  {
 9      Console.WriteLine($"[UI] Не вдалося прийняти пацієнта: {ex.Message}");
10  }
11
12  void AdmitPatient(string id, string name, string ageInput)
13  {
14      try
15      {
16          int age = int.Parse(ageInput);
17          Console.WriteLine($"[{id}] {name}, {age.ToString()} р. прийнято.");
18      }
19      catch (FormatException ex)
20      {
21          // логуємо деталь на цьому рівні
22          Console.WriteLine($"[Лог] ParseError у AdmitPatient: «{ageInput}» →
    {ex.Message}");
23          throw; // перекидаємо вище — без зміни StackTrace
24      }
25  }
```

throw; (без аргументу) зберігає оригінальний об'єкт і стек — верхній рівень отримає точне місце першопричини. Якщо написати throw ex; — стек скинеться до місця перекидання, що ускладнює діагностику.

Підсумок: алгоритм CLR при винятку

14. Виняток виникає у методі N
15. **Фаза пошуку:** CLR перевіряє рівні стека знизу вгору: N → N-1 → ... → точка входу
16. На кожному рівні перевіряє: чи є catch, чий тип відповідає винятку і чий when-фільтр (якщо є) дає true
17. Якщо знайдено → **фаза розкрутки:** виконує finally від рівня N до рівня знахідки, потім сам catch

18. Якщо не знайдено ніде → аварійне завершення програми
19. Код після try...catch на рівнях без обробника **не виконується**; код після try...catch на рівні з обробником — виконується нормально

Питання для самоконтролю — Розділ 5. Обробка винятків

5.1. Конструкція try..catch..finally

1. Що таке виняток (exception) у C#? Від якого базового класу успадковуються всі класи винятків? Хто і коли створює об'єкт винятку?
2. Опишіть порядок виконання блоків try, catch, finally у двох сценаріях: коли виняток виникає і коли не виникає.
3. Що виведе наступний код?

```C#
try
{
    Console.WriteLine("try");
    int x = int.Parse("abc");
    Console.WriteLine("після Parse");
}
catch
{
    Console.WriteLine("catch");
}
finally
{
    Console.WriteLine("finally");
}
Console.WriteLine("після блоку");
```

4. Чи виконається finally, якщо у catch теж виник виняток? Чи виконається finally, якщо у try вжито return?
5. Поясніть механізм **поширення винятку по стеку викликів**. Якщо метод A викликає B, який викликає C, а виняток виникає в C — де він може бути перехоплений?
6. Порівняйте два підходи: try { int age = int.Parse(s); } і if (int.TryParse(s, out int age)). Коли перший підхід кращий, коли другий? Чому не варто використовувати try...catch для передбачуваних сценаріїв?

7. Для чого може використовуватись try...finally без блоку catch? Наведіть приклад, де це потрібно, і поясніть: чи перехоплює така конструкція виняток?
8. Спроєктуйте метод ParseAndRegister(string name, string ageStr), який парсить вік, реєструє пацієнта і гарантує виведення рядка "Операцію завершено" у будь-якому разі — навіть при помилці. Використайте try...catch...finally.

5.2. Блок catch та фільтри винятків

9. Чим відрізняються три форми блоку catch? Запишіть синтаксис кожної і поясніть, що перехоплює кожна.
10. Що виведе наступний код?

```C#
try
{
    string s = null;
    Console.WriteLine(s.Length);
}
catch (FormatException)
{
    Console.WriteLine("FormatException");
}
catch (NullReferenceException ex)
{
    Console.WriteLine($"NullRef: {ex.Message}");
}
catch (Exception ex)
{
    Console.WriteLine($"Exception: {ex.Message}");
}
```

11. Якщо поставити catch (Exception ex) першим серед кількох блоків catch, що станеться? Чи виникне помилка компіляції?
12. Поясніть оператор when у блоці catch. Що відбувається, якщо тип винятку збігається, але умова when повертає false?
13. Напишіть метод ProcessInput(string input), що обробляє FormatException по-різному: окремо якщо рядок порожній (через when) і окремо для всіх інших некоректних значень.
14. Поясніть антипатерн «ковтання винятку» — порожній або беззмістовний блок catch. Чому він небезпечний у медичній або фінансовій системі?
15. Розробник написав catch (ArgumentException) перед catch (ArgumentOutOfRangeException). Яка проблема? Як виправити?

16. Є три блоки catch: FormatException, OverflowException, Exception. Для яких вхідних значень int.Parse(s) спрацює кожен з них? Наведіть конкретні приклади рядків-вхідних даних.

5.3. Типи винятків. Клас Exception

17. Перелічіть і коротко опишіть п'ять властивостей класу Exception. Яка з них найчастіше використовується у повідомленнях користувачу?
18. Що виведе ex.GetType().Name, якщо ex перехоплено через catch (Exception ex), а реальний виняток є FormatException?
19. Що таке InnerException? Наведіть сценарій, де воно корисне для діагностики помилки.
20. Що виведе цей код і в якому порядку?

```
C#
 1  string input = "999999999999999";
 2  try
 3  {
 4      int val = int.Parse(input);
 5  }
 6  catch (OverflowException ex)
 7  {
 8      Console.WriteLine($"Overflow: {ex.Message}");
 9  }
10  catch (FormatException ex)
11  {
12      Console.WriteLine($"Format: {ex.Message}");
13  }
14  catch (Exception ex)
15  {
16      Console.WriteLine($"Other: {ex.Message}");
17  }
```

21. Поясніть правило: «конкретніший тип — раніше». Чому ArgumentOutOfRangeException має стояти перед ArgumentException у ланцюжку catch?
22. Що означає «виняток зупиняє виконання try»? Напишіть приклад, де у try є три рядки коду, і виняток виникає на другому. Що відбувається з третім рядком?
23. Як catch (Exception ex) виступає «страхувальним мережем»? Чи слід його завжди додавати? В яких ситуаціях він обов'язковий, а в яких зайвий?
24. Для яких ситуацій призначені такі типи винятків: NullReferenceException, IndexOutOfRangeException,

InvalidCastException, ArgumentNullException? Наведіть по одному рядку коду, що спровокує кожен.

5.4. Генерація винятку та оператор throw

25. Навіщо генерувати виняток вручну через throw? Наведіть приклад бізнес-правила, порушення якого технічно не є помилкою, але потребує throw.
26. Чим відрізняються throw; (без аргументу) і throw ex; (з аргументом) у блоці catch? Яку помилку допускають при використанні throw ex;?
27. Що виведе цей код?

```C#
void Process(string name)
{
    try
    {
        Validate(name);
        Console.WriteLine("OK");
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine($"Caught: {ex.Message}");
    }
}
void Validate(string name)
{
    if (string.IsNullOrEmpty(name))
        throw new ArgumentException("Ім'я порожнє");
}
Process("");
Process("Іван");
```

28. Що таке **ланцюжок винятків** (InnerException)? Напишіть метод, що перехоплює FormatException і загортає його у InvalidOperationException зі збереженням оригіналу.
29. Яке призначення оператора nameof у throw new ArgumentOutOfRangeException(nameof(age), ...)? Що повертає nameof(age) і чому це краще, ніж рядок "age"?
30. Напишіть метод SetAge(int age), що кидає ArgumentOutOfRangeException якщо вік менший за 0 або більший за 150. Загорніть виклик у блок try...catch і виведіть повідомлення про помилку разом з некоректним значенням.
31. Розробник написав такий код:

```C#
catch (Exception ex)
{
    Log(ex);
    throw ex;
}
```

Знайдіть помилку і поясніть наслідки. Як виправити?

32. Де правильніше розміщувати throw для валідації аргументів: на початку методу чи в середині? Чому так зручніше для читача коду і для відлагодження?

5.5. Створення класів винятків

33. Чому варто створювати власні класи винятків для доменних помилок, якщо у .NET вже є понад 200 вбудованих типів?
34. Яка мінімальна реалізація необхідна для власного класу винятку? Що обов'язково вказати і навіщо?
35. Від якого класу краще успадковувати власний виняток, якщо він описує некоректний аргумент методу? А якщо стан об'єкта не дозволяє виконати операцію?
36. Що виведе цей код?

```C#
class AppException : Exception
{
    public string Code { get; }
    public AppException(string message, string code) : base(message)
    { Code = code; }
}
try
{
    throw new AppException("Недостатньо прав", "AUTH_ERR");
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
    if (ex is AppException app)
        Console.WriteLine($"Код: {app.Code}");
}
```

37. Поясніть перевагу зберігання доменних даних у власному виняткові (наприклад, поле InvalidAge) порівняно з формуванням рядку Message.
38. Спроєктуйте ієрархію винятків для інтернет-магазину: базовий ShopException (з полем OrderId) та два конкретних — OutOfStockException (з ProductName) і PaymentException (з

Amount). Напишіть метод обробки замовлення, що кидає обидва типи і ловить їх окремо.

39. Яка конвенція щодо іменування власних класів винятків? Наведіть три правильні і три неправильні назви.
40. Розробник вирішив не створювати власний клас, а покласти всі деталі у рядок Message. Порівняйте цей підхід із власним класом з властивостями. Де кожен з них кращий?

5.6. Пошук блоку catch при обробці винятків

41. Опишіть дві фази роботи CLR при виникненні винятку: фазу **пошуку** і фазу **розкрутки стека**. Що відбувається у кожній фазі і в якому порядку виконуються finally-блоки?
42. Дано ланцюжок викликів Main → A → B → C. Виняток виникає у C. У якому порядку CLR перевіряє рівні стека у фазі пошуку?
43. Що виведе цей код?

```C#
void A()
{
    try { B(); }
    catch (FormatException ex) { Console.WriteLine($"A catch: {ex.Message}"); }
    finally { Console.WriteLine("A finally"); }
}
void B()
{
    try { C(); }
    catch (NullReferenceException) { Console.WriteLine("B catch NullRef"); }
    finally { Console.WriteLine("B finally"); }
}
void C()
{
    try { int.Parse("abc"); }
    finally { Console.WriteLine("C finally"); }
}
A();
```

44. Виняток виник у методі на 3-му рівні стека. catch знайдено на 1-му рівні. Які finally-блоки виконаються і в якому порядку?
45. Якщо CLR не знаходить відповідного catch на жодному рівні стека, що відбувається? Чи гарантовано при цьому виконання finally-блоків?
46. Поясніть патерн «логування + перекидання»: catch (Exception ex) { Log(ex); throw; }. Чому це краще, ніж повністю ковтати

виняток або перехоплювати лише для логування і не перекидати?

47. Зверніть увагу: у прикладі нижче рядок після try...catch у SaveRecord **не виконується**, хоча у методі є finally. Поясніть чому.

```C#
void SaveRecord(string id, string input)
{
    try { int age = int.Parse(input); Console.WriteLine("Збережено"); }
    catch (NullReferenceException) { Console.WriteLine("NullRef"); }
    finally { Console.WriteLine("SaveRecord finally"); }
    Console.WriteLine("Після try у SaveRecord"); // коли виконується?
}
SaveRecord("P1", "abc");
```

48. Спроєктуйте трирівневий ланцюжок: UI() → Service() → Repository(). Repository кидає InvalidOperationException. Service логує помилку через finally і перекидає. UI перехоплює і відображає повідомлення користувачу. Напишіть повний код і поясніть порядок виконання кожного блоку.

Розділ 6. Делегати, події та лямбди

6.1. Делегати

Делегати — один із фундаментальних механізмів C#, який дозволяє зберігати посилання на методи і передавати методи як значення. Якщо змінна типу int зберігає число, а змінна типу string — рядок, то змінна-делегат зберігає посилання на метод. Через цю змінну метод можна викликати, передати в інший метод як аргумент або зберегти для виклику пізніше. Делегати є основою для подій, лямбда-виразів, функцій вищого порядку та патерну зворотного виклику (callback). Розуміння делегатів — це розуміння того, як у C# методи стають об'єктами першого класу.

Визначення делегата

Для оголошення делегата використовується ключове слово delegate, після якого вказується тип значення, що повертається, ім'я делегата та список параметрів. По суті, це оголошення нового типу, який описує сигнатуру методів, на які делегат може вказувати:

```
C#
1  delegate void Notify(string message);
```

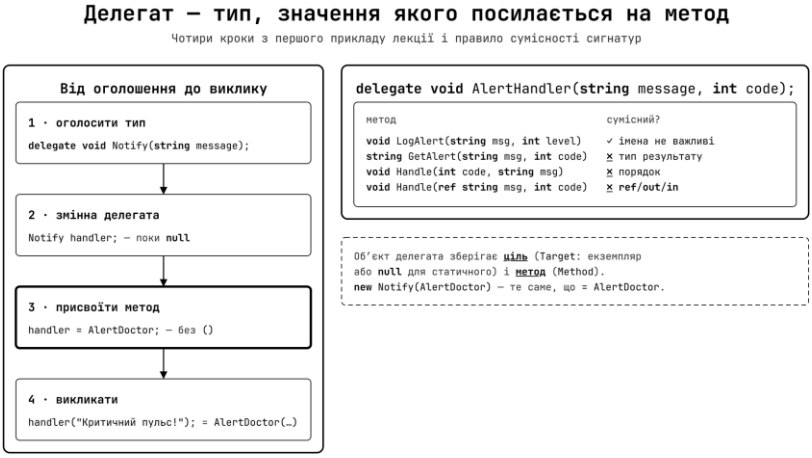
Делегат Notify відповідає будь-якому методу, який приймає один параметр string і нічого не повертає (void). Жоден інший набір параметрів або тип, що повертається, не підійде — компілятор перевіряє це статично.

Розглянемо застосування цього делегата:

```
C#
1  using System;
2
3  Notify handler;                  // 2. Створюємо змінну делегата
4  handler = AlertDoctor;           // 3. Присвоюємо адресу методу
5  handler("Критичний пульс!");     // 4. Викликаємо метод через делегат
6
7  void AlertDoctor(string msg) => Console.WriteLine($"[ЛІКАР] {msg}");
8  delegate void Notify(string message); // 1. Оголошуємо делегат
```

Насамперед необхідно визначити сам делегат — оголосити новий тип. Потім оголошується змінна цього типу. Далі делегату передається адреса певного методу — у нашому випадку методу AlertDoctor.

Зверніть увагу: цей метод має той самий тип, що повертається, і той самий набір параметрів, що і делегат. Нарешті, через змінну делегата викликається метод. Виклик делегата виглядає так само, як звичайний виклик методу.



Делегат як посилання на метод

При цьому делегати не обмежені методами того класу, де визначена змінна делегата. Це можуть бути також методи інших класів і структур:

```C#
using System;

Notify n1 = ClinicLogger.Print;
Notify n2 = new DoctorNotifier().Alert;

n1("Аналізи готові");   // виклик статичного методу
n2("Пацієнт Петренко потребує огляду"); // виклик екземплярного методу

delegate void Notify(string message);

class ClinicLogger
{
    public static void Print(string msg) =>
        Console.WriteLine($"[LOG] {msg}");
}

class DoctorNotifier
{
    public void Alert(string msg) =>
        Console.WriteLine($"[DR] {msg}");
```

```
21  }
```

Місце визначення делегата

Якщо ми визначаємо делегат у програмах верхнього рівня (top-level program), яку за замовчуванням представляє файл Program.cs (інструкції верхнього рівня з'явились у C# 9, а шаблон проєкту використовує їх з .NET 6 / C# 10), то, як і інші типи, делегат визначається в кінці коду (як у прикладах вище). Але делегат можна визначати і всередині класу:

```
C#
 1  class Program
 2  {
 3      delegate void Notify(string message); // делегат всередині класу
 4
 5      static void Main()
 6      {
 7          Notify handler;
 8          handler = AlertDoctor;
 9          handler("Критичний пульс!");
10
11          void AlertDoctor(string msg) => Console.WriteLine($"[ЛІКАР] {msg}");
12      }
13  }
```

Або поза класом:

```
C#
 1  delegate void Notify(string message); // делегат поза класом
 2
 3  class Program
 4  {
 5      static void Main()
 6      {
 7          Notify handler;
 8          handler = AlertDoctor;
 9          handler("Критичний пульс!");
10
11          void AlertDoctor(string msg) => Console.WriteLine($"[ЛІКАР] {msg}");
12      }
13  }
```

Параметри та результат делегата

Делегат повністю успадковує сигнатуру методу: параметри, їх типи і тип значення, що повертається. Розглянемо визначення та застосування делегата, який приймає параметри і повертає результат. Делегат можна перепризначати між методами, що мають однакову сигнатуру:

```C#
using System;

Calculate calc = Add; // делегат вказує на метод Add
double r1 = calc(10.5, 4.5); // фактично Add(10.5, 4.5)
Console.WriteLine($"Сума: {r1.ToString("F1")}");

calc = Subtract; // тепер делегат вказує на Subtract
double r2 = calc(10.5, 4.5);
Console.WriteLine($"Різниця: {r2.ToString("F1")}");

calc = Multiply;
double r3 = calc(10.5, 4.5);
Console.WriteLine($"Добуток: {r3.ToString("F1")}");

double Add(double x, double y)      => x + y;
double Subtract(double x, double y) => x - y;
double Multiply(double x, double y) => x * y;

delegate double Calculate(double a, double b);
```

Делегат Calculate повертає double і приймає два параметри double. Тому йому відповідає будь-який метод з таким самим підписом — Add, Subtract, Multiply. Оскільки делегат приймає два параметри, при виклику необхідно передати їх значення.

Існує також альтернативний синтаксис ініціалізації через конструктор делегата — обидва варіанти рівноцінні:

```C#
Calculate calc1 = Add;
Calculate calc2 = new Calculate(Add); // рівноцінно першому
```

Відповідність методів делегату

Як зазначено вище, методи відповідають делегату, якщо вони мають **той самий тип, що повертається**, і **той самий набір параметрів**. При цьому до уваги також беруться модифікатори ref, in та out. Наприклад, нехай у нас є делегат:

```C#
delegate void AlertHandler(string message, int code);
```

Цьому делегату відповідає такий метод:

```C#
void LogAlert(string msg, int level) { } // ОК — той самий підпис
```

А такі методи не відповідають:

```
C#
1  string GetAlert(string msg, int code) { return msg; } // інший тип повернення
2  void Handle(int code, string msg)     { }              // інший порядок
   параметрів
3  void Handle(ref string msg, int code) { }              // модифікатор ref
4  void Handle(out string msg, int code) { msg = ""; }    // модифікатор out
```

Метод з іншим типом повернення не підходить. Метод з іншим порядком параметрів — теж, навіть якщо типи самих параметрів збігаються. Наявність модифікаторів ref або out також робить метод несумісним із делегатом без цих модифікаторів.

Додавання методів у делегат

У прикладах вище змінна делегата вказувала на один метод. Насправді делегат може вказувати на **безліч методів**, які мають ту ж сигнатуру і тип, що повертається. Усі методи у делегаті потрапляють у спеціальний список — **список виклику** (invocation list). При виклику делегата всі методи цього списку послідовно викликаються. Для додавання методів до делегата застосовується операція +=:

```
C#
1  using System;
2
3  AlertHandler alert = LogToConsole;
4  alert += NotifyDoctor; // тепер alert вказує на два методи
5  alert("Критичні показники пацієнта Бойка!"); // викликаються обидва
6
7  void LogToConsole(string msg) => Console.WriteLine($"[КОНСОЛЬ] {msg}");
8  void NotifyDoctor(string msg) => Console.WriteLine($"[ЛІКАР] Увага: {msg}");
9  delegate void AlertHandler(string message);
```

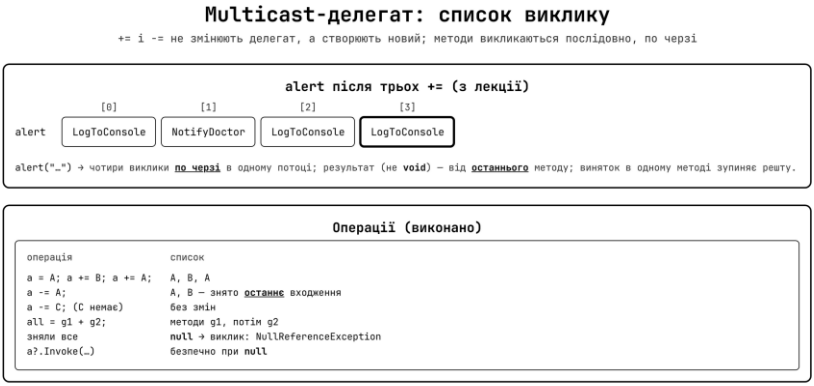
У цьому випадку до списку виклику делегата alert додаються два методи. При виклику alert викликаються відразу обидва.

Однак варто зазначити, що в реальності відбувається створення нового об'єкта делегата, який отримує методи старої копії і новий метод, і цей новий об'єкт присвоюється змінній alert.

При додаванні делегатів слід враховувати, що один і той самий метод можна додати кілька разів. У цьому випадку у списку виклику делегата буде кілька посилань на той самий метод, і при виклику делегата цей метод буде викликатися стільки разів, скільки він був доданий:

```
C#
 1  using System;
 2
 3  AlertHandler alert = LogToConsole;
```

```
 4  alert += NotifyDoctor;
 5  alert += LogToConsole; // LogToConsole додано вдруге
 6  alert += LogToConsole; // і втретє
 7  alert("Тиск 180/110 — критично!");
 8
 9  void LogToConsole(string msg) => Console.WriteLine($"[LOG] {msg}");
10  void NotifyDoctor(string msg) => Console.WriteLine($"[DR]  {msg}");
11  delegate void AlertHandler(string message);
```



Список виклику multicast-делегата

Подібним чином ми можемо видаляти методи з делегата за допомогою операції -=:

```
C#
 1  using System;
 2
 3  AlertHandler? alert = LogToConsole;
 4  alert += NotifyDoctor;
 5  alert("Температура 39.5°C"); // викликаються обидва методи
 6
 7  alert -= NotifyDoctor; // видаляємо NotifyDoctor
 8  if (alert != null) alert("Температура 38.0°C"); // викликається лише
    LogToConsole
 9
10  void LogToConsole(string msg) => Console.WriteLine($"[LOG] {msg}");
11  void NotifyDoctor(string msg) => Console.WriteLine($"[DR]  {msg}");
12  delegate void AlertHandler(string message);
```

При видаленні методів з делегата фактично також створюється новий делегат, який у списку виклику містить на один метод менше.

Варто відзначити: при видаленні може скластися ситуація, що в делегаті не залишиться жодного методу — тоді змінна матиме значення null. Тому в даному випадку змінна оголошена як AlertHandler?

(nullable), тобто тип, який може представляти як делегат, так і значення null. Крім того, перед другим викликом ми перевіряємо змінну на null.

При видаленні слід враховувати: якщо делегат містить кілька посилань на один і той самий метод, то операція -= починає пошук із кінця списку виклику і видаляє лише перше знайдене входження. Якщо такого методу у списку немає — операція -= не має жодного ефекту.

Об'єднання делегатів

Делегати можна поєднувати в інші делегати за допомогою оператора +. Отримаємо новий делегат, чий список виклику містить усі методи обох:

```
C#
 1  using System;
 2
 3  AlertHandler group1 = LogToConsole;
 4  group1 += NotifyDoctor;
 5
 6  AlertHandler group2 = SendSMS;
 7
 8  AlertHandler all = group1 + group2; // об'єднуємо делегати
 9  all("Пацієнт Сидоренко: пульс 145 уд/хв"); // викликаються всі методи з group1
    і group2
10
11  void LogToConsole(string msg) => Console.WriteLine($"[LOG]  {msg}");
12  void NotifyDoctor(string msg) => Console.WriteLine($"[DR]   {msg}");
13  void SendSMS(string msg)      => Console.WriteLine($"[SMS]  {msg}");
14  delegate void AlertHandler(string message);
```

Об'єднання делегатів означає, що до списку виклику делегата all потраплять усі методи із делегатів group1 та group2. При виклику all всі ці методи викликаються **послідовно**, один за одним, у порядку списку виклику (спочатку методи group1, потім group2) і в тому самому потоці — «одночасності» тут немає. Якщо один із методів кине виняток, решта методів списку вже не буде викликана; якщо потрібно викликати всі незалежно, обходять GetInvocationList() вручну з try/catch для кожного елемента.

Виклик делегата

У прикладах вище делегат викликався як звичайний метод. Якщо делегат приймає параметри — при виклику передаються необхідні значення:

```
C#
 1  using System;
 2
```

```
AlertHandler alert = LogToConsole;
alert("Критичний показник!");

Calculate calc = Add;
int n = calc(120, 80);
Console.WriteLine($"Результат: {n.ToString()}");

void LogToConsole(string msg) => Console.WriteLine($"[LOG] {msg}");
int Add(int x, int y) => x + y;

delegate void AlertHandler(string message);
delegate int Calculate(int x, int y);
```

Інший спосіб виклику делегата — метод Invoke():

```
C#
using System;

AlertHandler alert = LogToConsole;
alert.Invoke("Критичний показник!"); // еквівалентно alert(...)

Calculate calc = Add;
int n = calc.Invoke(120, 80);
Console.WriteLine($"Результат: {n.ToString()}");

void LogToConsole(string msg) => Console.WriteLine($"[LOG] {msg}");
int Add(int x, int y) => x + y;

delegate void AlertHandler(string message);
delegate int Calculate(int x, int y);
```

Якщо делегат приймає параметри — методу Invoke передаються їх значення.

Слід враховувати: якщо делегат порожній, тобто у його списку виклику немає жодного методу (делегат дорівнює null), то при виклику такого делегата виникне виняток:

```
C#
AlertHandler? alert;
// alert("Помилка!"); // NullReferenceException – делегат дорівнює null

Calculate? calc = Add;
calc -= Add; // делегат calc порожній
// int n = calc(10, 5); // NullReferenceException

delegate void AlertHandler(string message);
delegate int Calculate(int x, int y);
int Add(int x, int y) => x + y;
```

Тому при виклику делегата краще завжди перевіряти, чи він не дорівнює null. Або використовувати метод Invoke з оператором умовного null ?., який безпечно не виконає виклик якщо делегат null:

```C#
using System;

AlertHandler? alert = null;
alert?.Invoke("Безпечний виклик — нічого не станеться");

Calculate? calc = Add;
calc -= Add; // список виклику порожній
int? result = calc?.Invoke(10, 5); // result буде null, винятку немає
Console.WriteLine($"result = {result?.ToString() ?? "null"}");

int Add(int x, int y) => x + y;
delegate void AlertHandler(string message);
delegate int Calculate(int x, int y);
```

Якщо делегат повертає деяке значення і у його списку виклику кілька методів — повертається значення **останнього методу** зі списку. Наприклад:

```C#
using System;

Calculate calc = Method1;
calc += Method2;
calc += Method3;
Console.WriteLine($"Повернуто: {calc(10, 3).ToString()}"); // Add(10,3) = 13

int Method1(int x, int y) { Console.WriteLine($"Method1: {(x*y).ToString()}");
    return x * y; }
int Method2(int x, int y) { Console.WriteLine($"Method2: {(x-y).ToString()}");
    return x - y; }
int Method3(int x, int y) { Console.WriteLine($"Method3: {(x+y).ToString()}");
    return x + y; }

delegate int Calculate(int x, int y);
```

Узагальнені делегати

Делегати, як і інші типи, можуть бути узагальненими. Це дозволяє описати делегат, що відповідає методам із різними конкретними типами параметрів і повернення:

```C#
using System;

Transform<double, double> weightConv  = KgToLb;
Transform<int, int>       tempConv    = CelsiusToFahrenheit;
Transform<int, string>    pulseFormat = FormatPulse;

double result1 = weightConv(70.0);
Console.WriteLine($"70 кг = {result1.ToString("F1")} фунтів");

int result2 = tempConv(37);
Console.WriteLine($"37°C = {result2.ToString()}°F");
```

```
12
13  Console.WriteLine(pulseFormat(72));
14
15  double KgToLb(double kg) => kg * 2.20462;
16  int CelsiusToFahrenheit(int celsius) => celsius * 9 / 5 + 32;
17  string FormatPulse(int pulse) => $"{pulse.ToString()} уд/хв";
18
19  delegate T Transform<K, T>(K value);
```

Тут делегат Transform типізується двома параметрами типів. Параметр K представляє тип вхідного параметра, а параметр T — тип значення, що повертається. Таким чином, цьому делегату відповідає метод, який приймає параметр будь-якого типу та повертає значення будь-якого типу. Делегату Transform<double, double> відповідає метод, що приймає і повертає double, а делегату Transform<int, string> — метод, що приймає int і повертає string.

Делегати як параметри методів

Делегати можуть бути параметрами методів. Завдяки цьому один метод може отримувати інші методи як дії — параметри. Це і є функції вищого порядку:

C#
```
 1  using System;
 2
 3  CalculateVitals(120, 80, AddPressure);
 4  CalculateVitals(120, 80, SubtractPressure);
 5  CalculateVitals(120, 80, MultiplyPressure);
 6
 7  void CalculateVitals(int systolic, int diastolic, Operation op)
 8  {
 9      Console.WriteLine($"Результат: {op(systolic, diastolic).ToString()}");
10  }
11
12  int AddPressure(int x, int y)      => x + y;
13  int SubtractPressure(int x, int y) => x - y;
14  int MultiplyPressure(int x, int y) => x * y;
15
16  delegate int Operation(int x, int y);
```

Тут метод CalculateVitals як параметри приймає два числа і деяку дію у вигляді делегата Operation. Всередині методу викликаємо делегат, передаючи йому числа з перших двох параметрів. При виклику методу CalculateVitals ми можемо передати як третій параметр будь-який метод, що відповідає делегату Operation.

Повернення делегатів із методів

Також делегати можна повертати з методів. Тобто ми можемо повертати з методу якусь дію у вигляді іншого методу:

```
C#
 1  using System;
 2
 3  Operation op1 = SelectOperation(OperationType.Add);
 4  Console.WriteLine($"Результат: {op1(120, 80).ToString()}");
 5
 6  Operation op2 = SelectOperation(OperationType.Subtract);
 7  Console.WriteLine($"Результат: {op2(120, 80).ToString()}");
 8
 9  Operation op3 = SelectOperation(OperationType.Multiply);
10  Console.WriteLine($"Результат: {op3(120, 80).ToString()}");
11
12  Operation SelectOperation(OperationType opType)
13  {
14      switch (opType)
15      {
16          case OperationType.Add:      return Add;
17          case OperationType.Subtract: return Subtract;
18          default:                     return Multiply;
19      }
20  }
21
22  int Add(int x, int y)      => x + y;
23  int Subtract(int x, int y) => x - y;
24  int Multiply(int x, int y) => x * y;
25
26  enum OperationType { Add, Subtract, Multiply }
27  delegate int Operation(int x, int y);
```

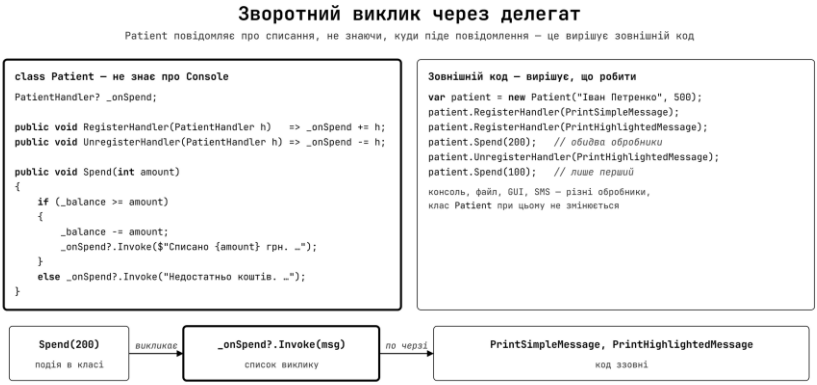
В даному випадку метод SelectOperation як параметр приймає значення перерахування OperationType. Залежно від значення параметра повертається певний метод. Оскільки тип методу, що повертається — делегат Operation, то метод повинен повернути метод, який відповідає цьому делегату. Тобто, якщо параметр дорівнює OperationType.Add, повертається метод Add.

При виклику SelectOperation ми отримуємо необхідну дію у змінну op1. І при виклику змінної op1 фактично буде викликатися отриманий з SelectOperation метод.

Застосування делегатів

Наведені вище приклади, можливо, не показують справжньої сили делегатів, оскільки потрібні нам методи ми могли б викликати і без будь-яких делегатів. Однак найбільш сильна сторона делегатів полягає в тому, що вони дозволяють **делегувати виконання певного коду**

ззовні. На момент написання класу ми можемо не знати, що саме буде виконуватися — ми просто викликаємо делегат. А який метод безпосередньо виконуватиметься, вирішуватиметься потім, при використанні класу.



Патерн зворотного виклику через делегат

Розглянемо детальний приклад. Нехай у нас є клас, який описує пацієнта в системі:

```C#
public class Patient
{
    int _balance; // умовний баланс страхових коштів
    public Patient(int balance) => _balance = balance;
    public void AddFunds(int amount) => _balance += amount;
    public void Spend(int amount)
    {
        if (_balance >= amount) _balance -= amount;
    }
}
```

Припустимо, нам треба повідомляти про кожне списання страхових коштів пацієнта. Якщо клас використовується лише в консольній програмі того самого проекту, де він створений, можна написати просто:

```C#
public void Spend(int amount)
{
    if (_balance >= amount)
    {
        _balance -= amount;
```

```
6           Console.WriteLine($"Списано {amount} грн. зі страхового рахунку.");
7       }
8   }
```

Але якщо наш клас планується використовувати в інших проектах — у графічному додатку Windows Forms або WPF, у мобільному додатку, у веб-API — рядок повідомлення Console.WriteLine(...) не матиме жодного сенсу. Більш того, якщо клас Patient використовуватиметься іншими розробниками у вигляді окремої бібліотеки, ці розробники захочуть повідомляти про списання коштів якимось іншим чином — про який ми навіть не здогадуємося на момент написання класу.

Тому жорстко вшитий Console.WriteLine — не найкраще рішення. Делегати дозволяють делегувати визначення дії із класу у зовнішній код, який використовуватиме цей клас. Змінимо клас, застосувавши делегати:

```
C#
 1  using System;
 2
 3  // Створюємо пацієнта зі страховим рахунком
 4  Patient patient = new Patient(500);
 5  // Передаємо обробник — консольний вивід
 6  patient.RegisterHandler(PrintMessage);
 7  // Двічі намагаємось списати кошти
 8  patient.Spend(200);
 9  patient.Spend(400);
10
11  void PrintMessage(string message) => Console.WriteLine(message);
12
13  // Оголошуємо делегат — тип для обробника подій
14  public delegate void PatientHandler(string message);
15
16  public class Patient
17  {
18      int _balance;
19      // Змінна делегата — зберігає посилання на обробник
20      PatientHandler? _onSpend;
21
22      public Patient(int balance) => _balance = balance;
23
24      // Метод для реєстрації обробника
25      public void RegisterHandler(PatientHandler handler)
26      {
27          _onSpend = handler;
28      }
29
30      public void AddFunds(int amount) => _balance += amount;
31
32      public void Spend(int amount)
33      {
34          if (_balance >= amount)
35          {
```

```
36              _balance -= amount;
37              // Викликаємо делегат — що саме відбудеться, вирішить зовнішній код
38              _onSpend?.Invoke($"Списано {amount.ToString()} грн. Залишок:
    {_balance.ToString()} грн.");
39          }
40          else
41          {
42              _onSpend?.Invoke($"Недостатньо коштів. Баланс:
    {_balance.ToString()} грн.");
43          }
44      }
45  }
```

Для делегування дії тут визначено делегат PatientHandler. Цей делегат відповідає будь-яким методам, які мають тип void та приймають параметр типу string:

```
C#
1  public delegate void PatientHandler(string message);
```

У класі Patient визначається змінна _onSpend, що представляє цей делегат. Далі визначається спеціальний метод RegisterHandler, через який передається реальна дія — конкретний метод ззовні:

```
C#
1  public void RegisterHandler(PatientHandler handler)
2  {
3      _onSpend = handler;
4  }
```

Виклик делегата відбувається у методі Spend. Залежно від того, чи відбулося списання, передаються різні повідомлення. Класу Patient не важливо, що саме відбудеться — він лише надсилає повідомлення через делегат.

Таким чином, ми створили механізм зворотного виклику для класу Patient. Тут ми виводимо повідомлення на консоль. Але зовнішній код міг би записати повідомлення у файл, надіслати на email, показати у графічному вікні — будь-який спосіб обробки, незалежно від класу Patient.

Додавання та видалення методів у делегаті

Хоча у прикладі наш делегат приймав адресу одного методу, насправді він може вказувати відразу на кілька. Крім того, за потреби ми можемо видалити посилання на певні методи, щоб вони не викликалися при виклику делегата. Змінимо клас Patient: метод RegisterHandler тепер використовуватиме +=, а новий метод UnregisterHandler — -=:

C#

```
using System;

Patient patient = new Patient(500);

// Реєструємо два обробники
patient.RegisterHandler(PrintSimpleMessage);
patient.RegisterHandler(PrintHighlightedMessage);

patient.Spend(200);
patient.Spend(400);

// Видаляємо другий обробник
patient.UnregisterHandler(PrintHighlightedMessage);
// Тепер спрацьовує лише перший
patient.Spend(100);

void PrintSimpleMessage(string message) => Console.WriteLine(message);
void PrintHighlightedMessage(string message)
{
    Console.WriteLine($"*** {message} ***");
}

public delegate void PatientHandler(string message);

public class Patient
{
    int _balance;
    PatientHandler? _onSpend;

    public Patient(int balance) => _balance = balance;

    // Реєструємо обробник — додаємо до списку
    public void RegisterHandler(PatientHandler handler)
    {
        _onSpend += handler;
    }

    // Скасування реєстрації обробника — видаляємо зі списку
    public void UnregisterHandler(PatientHandler handler)
    {
        _onSpend -= handler;
    }

    public void AddFunds(int amount) => _balance += amount;

    public void Spend(int amount)
    {
        if (_balance >= amount)
        {
            _balance -= amount;
            _onSpend?.Invoke($"Списано {amount.ToString()} грн. Залишок:
{_balance.ToString()} грн.");
        }
        else
        {
            _onSpend?.Invoke($"Недостатньо коштів. Баланс:
{_balance.ToString()} грн.");
        }
    }
```

```
58  }
```

У методі RegisterHandler делегати _onSpend і handler об'єднуються в один, який присвоюється змінній _onSpend. У методі UnregisterHandler зі змінної _onSpend видаляється делегат handler.

Анонімні методи

З делегатами тісно пов'язані анонімні методи. Анонімні методи використовуються для створення екземплярів делегатів без оголошення окремого іменованого методу.

Визначення анонімних методів починається з ключового слова delegate, після якого у дужках йде список параметрів та тіло методу у фігурних дужках:

```
1  delegate(параметри)
2  {
3      // інструкції
4  }
```

Наприклад:

```
C#
 1  using System;
 2
 3  PatientHandler handler = delegate(string msg)
 4  {
 5      Console.WriteLine($"[ОБРОБНИК] {msg}");
 6  };
 7
 8  handler("Пацієнт Коваль: списано 150 грн.");
 9
10  delegate void PatientHandler(string message);
```

Анонімний метод не може існувати сам по собі — він використовується для ініціалізації екземпляра делегата. У даному випадку змінна handler є анонімним методом, і через цю змінну делегата можна викликати цей анонімний метод.

Інший приклад анонімних методів — передача як аргумент для параметра, який представляє делегат:

```
C#
 1  using System;
 2
 3  ShowAlert("Пульс 145 уд/хв — критично!", delegate(string mes)
 4  {
 5      Console.WriteLine($"[СПОВІЩЕННЯ] {mes}");
 6  });
 7
 8  void ShowAlert(string message, PatientHandler handler)
```

```
 9  {
10      handler(message);
11  }
12
13  delegate void PatientHandler(string message);
```

Якщо анонімному методу не потрібні параметри — дужки з параметрами опускаються. При цьому навіть якщо делегат приймає кілька параметрів, в анонімному методі можна їх не вказувати:

```
C#
 1  using System;
 2
 3  PatientHandler handler = delegate
 4  {
 5      Console.WriteLine("Отримано сповіщення");
 6  };
 7
 8  handler("будь-яке повідомлення"); // параметр ігнорується
 9
10  delegate void PatientHandler(string message);
```

Тобто якщо анонімний метод містить параметри — вони обов'язково повинні відповідати параметрам делегата. Або анонімний метод взагалі може не містити жодних параметрів, тоді він відповідає будь-якому делегату з тим самим типом значення, що повертається. При цьому параметри анонімного методу не можуть бути опущені, якщо один або декілька параметрів визначено модифікатором out.

Так само, як і звичайні методи, анонімні можуть повертати результат:

```
C#
 1  using System;
 2
 3  Calculate calc = delegate(int x, int y)
 4  {
 5      return x + y;
 6  };
 7
 8  int result = calc(120, 80);
 9  Console.WriteLine($"Результат: {result.ToString()}");
10
11  delegate int Calculate(int x, int y);
```

При цьому анонімний метод має доступ до всіх змінних, визначених у зовнішньому коді (замикання):

```
C#
 1  using System;
 2
 3  string wardName  = "Кардіологія";
```

```
int    alertCount = 0;

PatientHandler handler = delegate(string msg)
{
    alertCount++;
    Console.WriteLine($"[{wardName}] #{alertCount.ToString()}: {msg}");
};

handler("Пульс 145");
handler("Тиск 180/100");
handler("Температура 39.2");
Console.WriteLine($"Всього сповіщень: {alertCount.ToString()}");

delegate void PatientHandler(string message);
```

Анонімний метод захоплює змінні wardName і alertCount із зовнішнього контексту — це замикання (closure). Зміна alertCount всередині анонімного методу відображається і в зовнішньому коді. Анонімні методи зазвичай використовують тоді, коли потрібно визначити одноразову дію, яка не має багато інструкцій та ніде більше не використовується. Зокрема, їх часто застосовують для обробки подій, які будуть розглянуті далі. Анонімні методи є попередниками лямбда-виразів — більш компактного сучасного способу запису того самого, який розглядатиметься у наступних розділах.

6.2. Лямбди

Лямбда-вираз — це компактний синтаксис для визначення анонімних методів. Там, де раніше потрібно було писати ключове слово delegate, список параметрів у дужках і тіло у фігурних дужках, лямбда дозволяє записати все в один рядок. З погляду типів даних лямбда-вираз представляє делегат — тобто посилання на метод із певною сигнатурою. Лямбди особливо корисні, коли метод короткий, застосовується в одному місці і немає сенсу виносити його у самостійну функцію.

Лямбда-оператор => розбиває вираз на дві частини: ліворуч — список параметрів, праворуч — тіло:

```
(список_параметрів) => тіло
```

Якщо тіло є одним виразом — фігурні дужки не потрібні. Якщо тіло містить кілька інструкцій — воно оформляється як звичайний блок у фігурних дужках.

Базовий синтаксис

Розглянемо найпростішу лямбду без параметрів, яка виводить повідомлення:

```
C#
using System;

PatientHandler greet = () => Console.WriteLine("Пацієнт зареєстрований у системі");
greet();
greet();

delegate void PatientHandler();
```

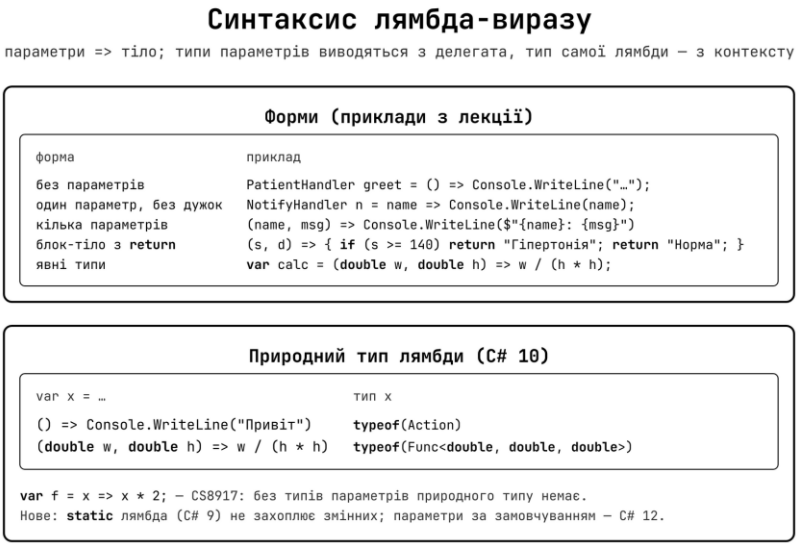
Змінна greet представляє делегат PatientHandler — метод без параметрів і без повернення. Як значення надається лямбда-вираз: ліворуч від => порожні дужки (немає параметрів), праворуч — єдиний вираз.

Якщо лямбда виконує кілька дій — вони поміщаються у фігурні дужки:

```
C#
using System;

PatientHandler register = () =>
{
    Console.WriteLine("Пацієнт зареєстрований у системі");
    Console.WriteLine("Очікуйте виклику лікаря");
};
register();

delegate void PatientHandler();
```



Синтаксис лямбда-виразу

Починаючи з C# 10 можна застосовувати неявну типізацію через var. Компілятор сам виводить тип делегата з контексту:

```
C#
1  using System;
2
3  var greet = () => Console.WriteLine("Пацієнт зареєстрований");
4  greet();
```

При неявній типізації компілятор зіставляє лямбда-вираз із вбудованим делегатом. У цьому прикладі greet буде типу Action — стандартного делегата без параметрів і без повернення.

Параметри лямбди

При визначенні списку параметрів типи даних зазвичай можна не вказувати — компілятор виводить їх з типу делегата:

```
C#
1  using System;
2
3  AlertHandler alert = (name, msg) => Console.WriteLine($"[{name}]: {msg}");
4  alert("Кардіологія", "Пацієнт Петренко: пульс 145");
```

```
5  alert("ICU", "Критичний стан — негайна допомога");
6
7  delegate void AlertHandler(string department, string message);
```

Компілятор бачить, що alert є типом AlertHandler, де обидва параметри — string, тому name і msg автоматично отримують тип string.

Якщо ж застосовується неявна типізація через var, компілятор не може вивести типи параметрів — тоді їх потрібно вказати явно:

```
C#
1  using System;
2
3  var alert = (string department, string message) =>
4      Console.WriteLine($"[{department}]: {message}");
5  alert("Неврологія", "Пацієнт Сидоренко: скарги на головний біль");
```

Якщо лямбда має рівно один параметр і його тип можна вивести — дужки навколо параметра можна опустити:

```
C#
1  using System;
2
3  NotifyHandler notify = patientName => Console.WriteLine($"Виклик:
   {patientName}");
4  notify("Марія Коваль");
5  notify("Олег Бойко");
6
7  delegate void NotifyHandler(string patientName);
```

Повернення результату

Лямбда-вираз може повертати результат. Якщо тіло є єдиним виразом — значення повертається автоматично, без return:

```
C#
 1  using System;
 2
 3  CalculateHandler bmi = (weight, height) => weight / (height * height);
 4  double result = bmi(70.0, 1.75);
 5  Console.WriteLine($"IMT: {result.ToString("F1")}");
 6
 7  ScoreHandler classify = score => score >= 60 ? "задовільно" : "критично";
 8  Console.WriteLine($"Оцінка стану: {classify(72)}");
 9  Console.WriteLine($"Оцінка стану: {classify(45)}");
10
11  delegate double CalculateHandler(double weight, double height);
12  delegate string ScoreHandler(int score);
```

Якщо лямбда містить кілька виразів — потрібен явний return, як у звичайному методі:

```C#
using System;

BloodPressureHandler classify = (systolic, diastolic) =>
{
    if (systolic > 140 || diastolic > 90) return "Гіпертонія";
    if (systolic < 90 || diastolic < 60) return "Гіпотонія";
    return "Норма";
};

Console.WriteLine($"150/95: {classify(150, 95)}");
Console.WriteLine($"85/55: {classify(85, 55)}");
Console.WriteLine($"120/80: {classify(120, 80)}");

delegate string BloodPressureHandler(int systolic, int diastolic);
```

Додавання та видалення дій у лямбда-виразі

Оскільки лямбда-вираз представляє делегат, до змінної-лямбди можна додавати інші методи та лямбди через +=, а також видаляти через -=:

```C#
using System;

AlertChain logAlert = () => Console.WriteLine("[LOG] Сигнал тривоги");
AlertChain notifyDoc = () => Console.WriteLine("[DR] Виклик лікаря");
AlertChain sendSMS = () => Console.WriteLine("[SMS] Повідомлення відправлено");

AlertChain alert = logAlert;
alert += notifyDoc;
alert += sendSMS;
alert();

Console.WriteLine("--- видаляємо SMS ---");
alert -= sendSMS;
alert?.Invoke();

delegate void AlertChain();
```

При виклику alert() послідовно виконуються всі методи зі списку виклику. Після видалення sendSMS залишаються лише два перших.

Лямбда-вираз як аргумент методу

Найпрактичніше застосування лямбд — передача їх як аргументи методам. Це дозволяє параметризувати поведінку: метод визначає загальну структуру, а конкретна дія передається ззовні:

```C#
using System;

double[] glucoseReadings = { 3.5, 7.2, 5.1, 8.8, 4.9, 6.3, 2.9 };

```

```
double highCount = CountMatching(glucoseReadings, x => x > 6.1);
double lowCount  = CountMatching(glucoseReadings, x => x < 3.9);

Console.WriteLine($"Вище норми (>6.1): {highCount.ToString("F0")} результатів");
Console.WriteLine($"Нижче норми (<3.9): {lowCount.ToString("F0")} результатів");

double CountMatching(double[] values, IsMatch condition)
{
    int count = 0;
    foreach (var v in values)
        if (condition(v)) count++;
    return count;
}

delegate bool IsMatch(double value);
```

Метод CountMatching не знає, яка конкретно умова застосовується — він лише викликає делегат condition для кожного елемента. Умова передається зовні через лямбду: x => x > 6.1 або x => x < 3.9. Такий підхід дозволяє повторно використовувати метод для будь-яких критеріїв фільтрації без зміни самого методу.

Лямбда-вираз як результат методу

Метод може повертати лямбда-вираз. Тип, що повертається, — делегат, якому відповідає лямбда:

```
C#
using System;

BloodAnalysis analyze = SelectAnalysis(AnalysisType.Glucose);
Console.WriteLine($"Глюкоза 7.5: {analyze(7.5)}");

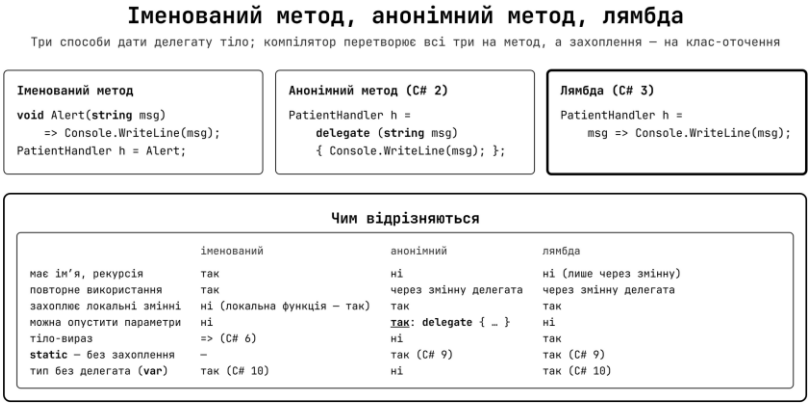
analyze = SelectAnalysis(AnalysisType.Hemoglobin);
Console.WriteLine($"Гемоглобін 105: {analyze(105.0)}");

analyze = SelectAnalysis(AnalysisType.Cholesterol);
Console.WriteLine($"Холестерин 6.2: {analyze(6.2)}");

BloodAnalysis SelectAnalysis(AnalysisType type)
{
    switch (type)
    {
        case AnalysisType.Glucose:
            return value => value < 3.9 ? "Низький" : value > 6.1 ? "Високий" : "Норма";
        case AnalysisType.Hemoglobin:
            return value => value < 120 ? "Низький" : value > 170 ? "Високий" : "Норма";
        default:
            return value => value < 3.0 ? "Низький" : value > 5.2 ? "Високий" : "Норма";
    }
```

```
23  }
24
25  enum AnalysisType { Glucose, Hemoglobin, Cholesterol }
26  delegate string BloodAnalysis(double value);
```

Метод SelectAnalysis отримує тип аналізу і повертає відповідну лямбду з потрібними порогами норми. Зовнішній код отримує готову функцію-класифікатор і може викликати її для будь-якого значення. Такий патерн часто називають «фабрикою функцій» — метод конструює і повертає поведінку, а не лише дані.



Лямбда vs анонімний метод vs іменований метод

6.3. Події

Події сигналізують системі про те, що сталася певна дія. Якщо делегат — це лише посилання на метод, то подія — це спеціальна конструкція, яка обгортає делегат і накладає обмеження на доступ до нього. Зовнішній код може лише підписатися на подію (+=) або відписатися від неї (-=), але не може викликати подію безпосередньо або замінити весь список обробників через звичайне присвоєння. Це важливе архітектурне рішення: клас контролює, коли і чому виникає подія, а зовнішній код лише реагує на неї.

Навіщо потрібні події

Розглянемо клас Patient, який описує пацієнта медичної системи із страховим рахунком:

```C#
class Patient
{
    public int Balance { get; private set; }
    public string Name { get; }
    public Patient(string name, int balance) { Name = name; Balance = balance; }

    public void AddFunds(int amount) => Balance += amount;

    public void Spend(int amount)
    {
        if (Balance >= amount) Balance -= amount;
    }
}
```

Якщо ми захочемо повідомляти про кожне списання страхових коштів, найпростіший варіант — додати Console.WriteLine прямо в метод Spend:

```C#
public void Spend(int amount)
{
    if (Balance >= amount)
    {
        Balance -= amount;
        Console.WriteLine($"Списано {amount} грн. зі страхового рахунку пацієнта {Name}.");
    }
}
```

Але цей підхід має серйозні обмеження. На момент написання класу ми можемо точно не знати, яку саме дію потрібно виконати після списання. У консольному застосунку — це виведення рядка. У графічному WPF-додатку — спливне вікно. У веб-API — запис у лог або відправка HTTP-запиту. Якщо клас Patient планується як бібліотека для різних проектів — жорстко вшитий Console.WriteLine не підходить: інші розробники захочуть реагувати на списання по-своєму, і ми навіть не знаємо заздалегідь як саме. Саме для вирішення цієї проблеми існують події.

Визначення та виклик події

Подія оголошується в класі за допомогою ключового слова event, після якого вказується тип делегата, що представляє подію:

```C#
delegate void PatientHandler(string message);
event PatientHandler Notify;
```

Спочатку визначається делегат PatientHandler, що приймає один параметр string. Потім за допомогою event визначається подія Notify, представлена цим делегатом. Назва події може бути довільною, але вона завжди прив'язана до конкретного делегата.

Оголошену подію можна викликати всередині класу як метод:

```
C#
1  Notify("Сталося списання");
```

Оскільки Notify представляє делегат PatientHandler, що приймає рядок, — при виклику передається рядок. Якщо жоден обробник не зареєстрований, подія дорівнює null, тому при виклику краще завжди перевіряти:

```
C#
1  Notify?.Invoke("Сталося списання");
```

Метод Invoke із оператором умовного null ?. не виконає виклик, якщо подія null, — виняток не виникне.

Поєднаємо все разом — визначимо подію і виклики у повному класі:

```
C#
 1  using System;
 2
 3  Patient p = new Patient("Іван Петренко", 500);
 4  // обробник ще не встановлено — виклики події не дають ефекту
 5  p.AddFunds(100);
 6  p.Spend(200);
 7  Console.WriteLine($"Баланс: {p.Balance.ToString()} грн.");
 8
 9  class Patient
10  {
11      public delegate void PatientHandler(string message);
12      public event PatientHandler? Notify;        // 1. Визначення події
13
14      public string Name { get; }
15      public int Balance { get; private set; }
16
17      public Patient(string name, int balance) { Name = name; Balance = balance;
    }
18
19      public void AddFunds(int amount)
20      {
21          Balance += amount;
22          Notify?.Invoke($"На рахунок {Name} зараховано: {amount.ToString()}
    грн.");  // 2. Виклик
23      }
24
25      public void Spend(int amount)
26      {
```

```
27          if (Balance >= amount)
28          {
29              Balance -= amount;
30              Notify?.Invoke($"З рахунку {Name} списано: {amount.ToString()}
    грн.");  // 3. Виклик
31          }
32          else
33          {
34              Notify?.Invoke($"Недостатньо коштів. Баланс {Name}:
    {Balance.ToString()} грн.");
35          }
36      }
37  }
```

Поки жоден обробник не зареєстрований, виклики Notify?.Invoke(...) нічого не роблять — подія null.

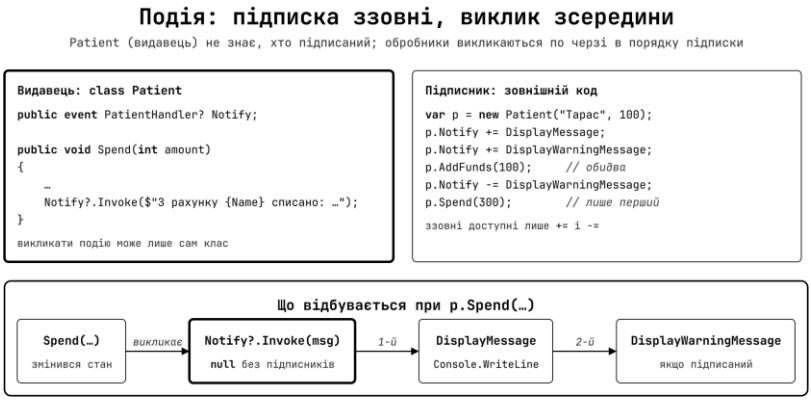
Додавання обробника події

Обробник події — це метод, який виконується під час виклику події. Його список параметрів і тип повернення мають відповідати делегату події. Обробник додається через +=:

```
C#
 1  using System;
 2
 3  Patient p = new Patient("Марія Коваль", 500);
 4  p.Notify += DisplayMessage;         // підписуємось на подію
 5
 6  p.AddFunds(200);
 7  Console.WriteLine($"Баланс: {p.Balance.ToString()} грн.");
 8  p.Spend(300);
 9  Console.WriteLine($"Баланс: {p.Balance.ToString()} грн.");
10  p.Spend(600);
11  Console.WriteLine($"Баланс: {p.Balance.ToString()} грн.");
12
13  void DisplayMessage(string message) => Console.WriteLine(message);
14
15  class Patient
16  {
17      public delegate void PatientHandler(string message);
18      public event PatientHandler? Notify;
19      public string Name { get; }
20      public int Balance { get; private set; }
21      public Patient(string name, int balance) { Name = name; Balance = balance;
    }
22      public void AddFunds(int amount)
23      {
24          Balance += amount;
25          Notify?.Invoke($"На рахунок {Name} зараховано: {amount.ToString()}
    грн.");
26      }
27      public void Spend(int amount)
28      {
29          if (Balance >= amount)
30          {
```

```
            Balance -= amount;
            Notify?.Invoke($"З рахунку {Name} списано: {amount.ToString()}
грн.");
        }
        else
        {
            Notify?.Invoke($"Недостатньо коштів. Баланс {Name}:
{Balance.ToString()} грн.");
        }
    }
}
```

Метод DisplayMessage відповідає делегату PatientHandler — приймає string, нічого не повертає. При виклику Notify?.Invoke(...) тепер буде виконуватися цей метод. Клас Patient нічого не знає про Console.WriteLine — він лише надсилає повідомлення через подію. Зовнішній код вирішує, що з ним робити.



Потік події: реєстрація обробника та виклик

Додавання та видалення обробників

Для однієї події можна зареєструвати кілька обробників — і в будь-який момент видалити будь-який із них через -=:

```
C#
using System;

Patient p = new Patient("Олег Бойко", 800);

p.Notify += DisplayMessage;         // реєструємо перший обробник
p.Notify += DisplayWarningMessage;  // реєструємо другий

p.AddFunds(100);                    // спрацюють обидва
```

```
Console.WriteLine("---");
p.Notify -= DisplayWarningMessage;  // видаляємо другий
p.Spend(300);                       // спрацює лише перший

void DisplayMessage(string message) => Console.WriteLine(message);
void DisplayWarningMessage(string message)
{
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.WriteLine($"[!] {message}");
    Console.ResetColor();
}

class Patient
{
    public delegate void PatientHandler(string message);
    public event PatientHandler? Notify;
    public string Name { get; }
    public int Balance { get; private set; }
    public Patient(string name, int balance) { Name = name; Balance = balance; }
    public void AddFunds(int amount)
    {
        Balance += amount;
        Notify?.Invoke($"На рахунок {Name} зараховано: {amount.ToString()} грн.");
    }
    public void Spend(int amount)
    {
        if (Balance >= amount)
        {
            Balance -= amount;
            Notify?.Invoke($"З рахунку {Name} списано: {amount.ToString()} грн.");
        }
        else
        {
            Notify?.Invoke($"Недостатньо коштів. Баланс: {Balance.ToString()} грн.");
        }
    }
}
```

Як обробники можна використовувати не лише іменовані методи, а й анонімні методи та лямбда-вирази:

```
C#
using System;

Patient p = new Patient("Тетяна Руденко", 500);

// обробник через делегат
p.Notify += new Patient.PatientHandler(msg => Console.WriteLine($"[ДЕЛЕГАТ] {msg}"));
// обробник через анонімний метод
p.Notify += delegate(string msg) { Console.WriteLine($"[АНОНІМНИЙ] {msg}"); };
// обробник через лямбду
p.Notify += msg => Console.WriteLine($"[ЛЯМБДА] {msg}");

```

```
p.Spend(200);

class Patient
{
    public delegate void PatientHandler(string message);
    public event PatientHandler? Notify;
    public string Name { get; }
    public int Balance { get; private set; }
    public Patient(string name, int balance) { Name = name; Balance = balance; }
    public void Spend(int amount)
    {
        if (Balance >= amount) { Balance -= amount; Notify?.Invoke($"Списано: {amount.ToString()} грн."); }
        else Notify?.Invoke($"Недостатньо коштів. Баланс: {Balance.ToString()} грн.");
    }
}
```

Управління обробниками (аксесори add/remove)

За допомогою спеціальних аксесорів add і remove можна керувати логікою підписки та відписки. Це корисно тоді, коли потрібно, наприклад, записати у лог хто підписався, або обмежити кількість обробників:

```
C#
using System;

Patient p = new Patient("Василь Мороз", 600);

p.Notify += DisplayMessage;
p.Spend(100);
p.Notify -= DisplayMessage;
p.Spend(50);

void DisplayMessage(string message) => Console.WriteLine(message);

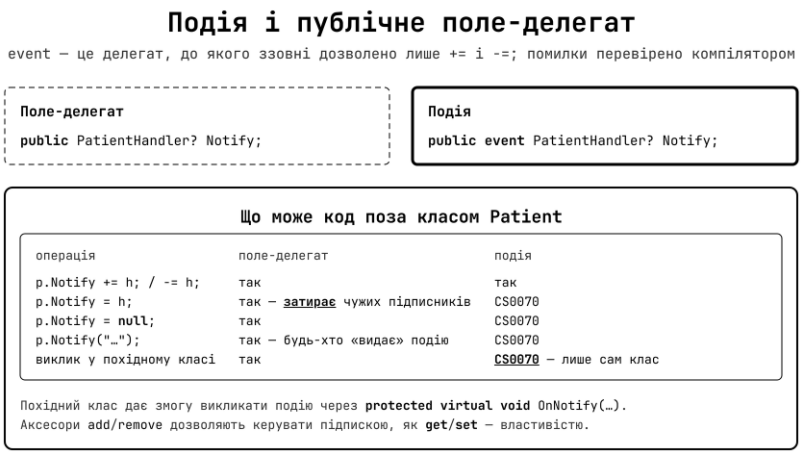
class Patient
{
    public delegate void PatientHandler(string message);

    PatientHandler? _notify;        // приватна змінна делегата

    public event PatientHandler Notify
    {
        add
        {
            _notify += value;
            Console.WriteLine($"[ПІДПИСКА] Обробник '{value.Method.Name}' зареєстровано");
        }
        remove
        {
            _notify -= value;
            Console.WriteLine($"[ВІДПИСКА] Обробник '{value.Method.Name}' видалено");
```

```
29          }
30      }
31
32      public string Name { get; }
33      public int Balance { get; private set; }
34      public Patient(string name, int balance) { Name = name; Balance = balance;
    }
35
36      public void Spend(int amount)
37      {
38          if (Balance >= amount)
39          {
40              Balance -= amount;
41              _notify?.Invoke($"З рахунку {Name} списано: {amount.ToString()}
    грн.");
42          }
43          else
44          {
45              _notify?.Invoke($"Недостатньо коштів. Баланс: {Balance.ToString()}
    грн.");
46          }
47      }
48  }
```

Визначення події тепер розбивається на дві частини. Спочатку оголошується приватна змінна делегата _notify, якою клас викликає обробники зсередини. Потім визначаються аксесори: add виконується при операції +=, а remove — при -=. Усередині аксесора обробник, що додається або видаляється, доступний через ключове слово value. Зовнішній код працює з Notify (публічна подія), але реальний список обробників зберігається у _notify.



Подія vs делегат: ключові відмінності

Передача даних події

Нерідко обробнику події потрібно отримати детальну інформацію про те, що саме сталося — не просто рядок, а структурований об'єкт з кількома полями. Для цього визначається спеціальний клас аргументів події. Додамо клас PatientEventArgs:

```
C#
 1  using System;
 2
 3  Patient p = new Patient("Надія Литвин", 700);
 4  p.Notify += DisplayTransactionInfo;
 5
 6  p.AddFunds(150);
 7  p.Spend(400);
 8  p.Spend(600);
 9
10  void DisplayTransactionInfo(Patient sender, PatientEventArgs e)
11  {
12      Console.WriteLine($"Пацієнт: {sender.Name}");
13      Console.WriteLine($"Операція: {e.Message}");
14      Console.WriteLine($"Сума: {e.Amount.ToString()} грн. | Поточний баланс:
    {sender.Balance.ToString()} грн.");
15      Console.WriteLine("---");
16  }
17
18  class PatientEventArgs
19  {
20      public string Message { get; }
21      public int Amount { get; }
22      public PatientEventArgs(string message, int amount)
```

```
23      {
24          Message = message;
25          Amount  = amount;
26      }
27  }
28
29  class Patient
30  {
31      public delegate void PatientHandler(Patient sender, PatientEventArgs e);
32      public event PatientHandler? Notify;
33
34      public string Name { get; }
35      public int Balance { get; private set; }
36
37      public Patient(string name, int balance) { Name = name; Balance = balance;
    }
38
39      public void AddFunds(int amount)
40      {
41          Balance += amount;
42          Notify?.Invoke(this, new PatientEventArgs($"На рахунок зараховано
    {amount.ToString()} грн.", amount));
43      }
44
45      public void Spend(int amount)
46      {
47          if (Balance >= amount)
48          {
49              Balance -= amount;
50              Notify?.Invoke(this, new PatientEventArgs($"Списано
    {amount.ToString()} грн. зі страхового рахунку", amount));
51          }
52          else
53          {
54              Notify?.Invoke(this, new PatientEventArgs("Недостатньо коштів на
    страховому рахунку", amount));
55          }
56      }
57  }
```

Делегат PatientHandler тепер приймає два параметри: перший — об'єкт Patient, що є джерелом події (відправник), другий — PatientEventArgs із деталями операції. Перший параметр this передає посилання на сам об'єкт Patient, тому обробник може звернутися до будь-якого стану пацієнта — наприклад, прочитати поточний баланс. Другий параметр містить повідомлення і суму операції.

Такий патерн — (sender, eventArgs) — є стандартним у .NET і використовується у бібліотечних подіях: Button.Click (WinForms/WPF), System.Timers.Timer.Elapsed, FileSystemWatcher.Changed, Process.Exited — усі вони слідують цій же конвенції. У реальних проектах PatientEventArgs зазвичай успадковують від System.EventArgs, а делегат замінюють вбудованим EventHandler<T>, що буде розглянуто пізніше.

Антипатерн: витік пам'яті через незвільнені підписки

Підписка на подію через += створює посилання від **видавця** (publisher) до **підписника** (subscriber). Якщо видавець — довготривалий об'єкт (наприклад, статичний клас або синглтон), а підписник — об'єкт із коротким циклом життя, сміттєзбирач (GC) **не зможе видалити** підписника, доки він залишається в списку обробників події.

Це — **витік пам'яті через події**: об'єкти, які мали б зібратися GC, залишаються живими через приховане посилання з події.

```
C#
 1  using System;
 2
 3  // Видавець — довгоживучий (симулюємо статичним полем)
 4  var hospitalSystem = new HospitalMonitor();
 5
 6  Console.WriteLine("=== Реєструємо 3 монітори пацієнтів ===");
 7  for (int i = 1; i <= 3; i++)
 8  {
 9      var monitor = new PatientMonitor($"Monitor-{i}");
10      monitor.Subscribe(hospitalSystem);
11      // monitor виходить зі scope — але GC НЕ збере його!
12  }
13
14  GC.Collect();
15  GC.WaitForPendingFinalizers();
16
17  Console.WriteLine($"\nПісля GC.Collect() — живих підписників:
    {hospitalSystem.SubscriberCount}");
18  Console.WriteLine("(Очікували 0, але всі 3 залишились — витік пам'яті!)");
19
20  Console.WriteLine("\n=== Сигнал тривоги — усі підписники ще реагують ===");
21  hospitalSystem.RaiseAlert("Критичний показник у палаті 5");
22
23  // Клас без відписки — витік пам'яті
24  class PatientMonitor
25  {
26      public string Name { get; }
27      public PatientMonitor(string name) { Name = name;
    Console.WriteLine($"[{name}] Створено"); }
28
29      public void Subscribe(HospitalMonitor monitor) =>
30          monitor.AlertRaised += OnAlert; // підписуємось, але ніколи не
    відписуємось!
31
32      void OnAlert(string message) =>
33          Console.WriteLine($"  [{Name}] Отримав сигнал: {message}");
34
35      ~PatientMonitor() => Console.WriteLine($"[{Name}] Фіналізовано (GC
    зібрав)");
36  }
37
38  class HospitalMonitor
39  {
40      private event Action<string>? _alertRaised;
41
```

```
    public event Action<string> AlertRaised
    {
        add    { _alertRaised += value; }
        remove { _alertRaised -= value; }
    }

    public int SubscriberCount =>
        _alertRaised?.GetInvocationList().Length ?? 0;

    public void RaiseAlert(string message) => _alertRaised?.Invoke(message);
}
```

```
C#
using System;

// Правильний патерн: завжди відписуватися або реалізовувати IDisposable
var hospitalSystem = new HospitalMonitor();

Console.WriteLine("=== Правильно: відписка через IDisposable ===");
var monitor1 = new PatientMonitorSafe("Monitor-1", hospitalSystem);
var monitor2 = new PatientMonitorSafe("Monitor-2", hospitalSystem);

Console.WriteLine($"Підписників: {hospitalSystem.SubscriberCount}"); // 2

hospitalSystem.RaiseAlert("Тест");

monitor1.Dispose(); // відписка
Console.WriteLine($"\nПісля monitor1.Dispose():
{hospitalSystem.SubscriberCount} підписник");

monitor2.Dispose();
Console.WriteLine($"Після monitor2.Dispose(): {hospitalSystem.SubscriberCount}
підписників");

GC.Collect();
GC.WaitForPendingFinalizers();
Console.WriteLine("GC не має що збирати — об'єкти коректно звільнені");

// Правильний клас: реалізує IDisposable для відписки
class PatientMonitorSafe : IDisposable
{
    private readonly HospitalMonitor _source;
    public string Name { get; }

    public PatientMonitorSafe(string name, HospitalMonitor source)
    {
        Name = name;
        _source = source;
        _source.AlertRaised += OnAlert; // підписуємось
        Console.WriteLine($"[{name}] Підписано");
    }

    void OnAlert(string message) =>
        Console.WriteLine($"  [{Name}] Сигнал: {message}");

    public void Dispose()
    {
        _source.AlertRaised -= OnAlert; // ОБОВ'ЯЗКОВО відписуємось
        Console.WriteLine($"[{Name}] Відписано");
```

```
45      }
46
47      ~PatientMonitorSafe() => Console.WriteLine($"[{Name}] GC зібрав");
48  }
49
50  class HospitalMonitor
51  {
52      private event Action<string>? _alertRaised;
53      public event Action<string> AlertRaised
54      {
55          add    { _alertRaised += value; }
56          remove { _alertRaised -= value; }
57      }
58      public int SubscriberCount => _alertRaised?.GetInvocationList().Length ??
    0;
59      public void RaiseAlert(string message) => _alertRaised?.Invoke(message);
60  }
```

Правила безпечної роботи з подіями:

Сценарій	Рекомендація
Підписник живе стільки ж, скільки видавець	Відписка необов'язкова, але бажана для ясності
Підписник має коротший цикл життя	Реалізуйте IDisposable, відписуйтесь у Dispose()
Підписка через лямбду	Збережіть лямбду у поле, щоб можна було передати в -=
Відписатися від усіх одразу	У класі-видавці зробіть метод ClearSubscriptions()

Правило просте: **кожному += має відповідати -=** у відповідному місці (Dispose, Close або явна очистка). Деструктор (фіналізатор) для цього не підходить: поки видавець тримає посилання на підписника через подію, збирач сміття вважає підписника живим, і його фіналізатор просто не буде викликаний — відписка в ньому ніколи не відбудеться.

6.4. Коваріантність та контраваріантність делегатів

Коваріантність і контраваріантність описують, як делегати взаємодіють зі спадкуванням типів. Вони дозволяють присвоювати делегату метод, сигнатура якого не збігається з делегатом точно, але є сумісною через ієрархію класів. Це підвищує гнучкість коду: замість того щоб

оголошувати окремий делегат для кожного типу ієрархії, можна використати один делегат для роботи з базовим або похідним типом.

— **Коваріантність** стосується типу, що **повертається**: метод може повертати більш похідний (конкретніший) тип, ніж оголошено в делегаті.

— **Контраваріантність** стосується **параметрів**: метод може приймати більш загальний (базовий) тип, ніж оголошено в делегаті.

Розглянемо ці поняття на прикладі ієрархії класів медичних сповіщень:

```
C#
 1  using System;
 2
 3  // просто перевіряємо ієрархію
 4  Notification n = new EmailNotification("Результати аналізів готові");
 5  n.Print();
 6  n = new SmsNotification("Прийом завтра о 10:00");
 7  n.Print();
 8
 9  class Notification
10  {
11      public string Text { get; }
12      public Notification(string text) => Text = text;
13      public virtual void Print() => Console.WriteLine($"Сповіщення: {Text}");
14  }
15
16  class EmailNotification : Notification
17  {
18      public EmailNotification(string text) : base(text) { }
19      public override void Print() => Console.WriteLine($"Email: {Text}");
20  }
21
22  class SmsNotification : Notification
23  {
24      public SmsNotification(string text) : base(text) { }
25      public override void Print() => Console.WriteLine($"SMS: {Text}");
26  }
```

Клас Notification — базовий для всіх типів сповіщень. EmailNotification і SmsNotification — похідні класи, кожен з яких перевизначає метод Print.

Коваріантність

Коваріантність дозволяє передати делегату метод, тип якого, що **повертається**, є **похідним** від типу, що повертається делегатом. Якщо делегат оголошує повернення Notification, то метод може повертати EmailNotification:

```C#
using System;

// делегат повертає базовий тип Notification
// метод повертає похідний тип EmailNotification — коваріантність
NotificationBuilder builder = CreateEmail;

Notification result = builder("Аналізи пацієнта Петренка готові");
result.Print(); // Email: ...

EmailNotification CreateEmail(string text) => new EmailNotification(text);

class Notification
{
    public string Text { get; }
    public Notification(string text) => Text = text;
    public virtual void Print() => Console.WriteLine($"Сповіщення: {Text}");
}
class EmailNotification : Notification
{
    public EmailNotification(string text) : base(text) { }
    public override void Print() => Console.WriteLine($"Email: {Text}");
}

// делегат повертає базовий тип Notification
delegate Notification NotificationBuilder(string text);
```

Компілятор дозволяє це, бо будь-який об'єкт EmailNotification є водночас об'єктом Notification — відносини «є» (is-a). Якщо делегат очікує Notification, метод, що повертає EmailNotification, завжди задовольняє цю вимогу.

Контраваріантність

Контраваріантність дозволяє передати делегату метод, тип **параметра** якого є **базовим** по відношенню до типу параметра делегата. Якщо делегат оголошує параметр EmailNotification, то метод може приймати Notification:

```C#
using System;

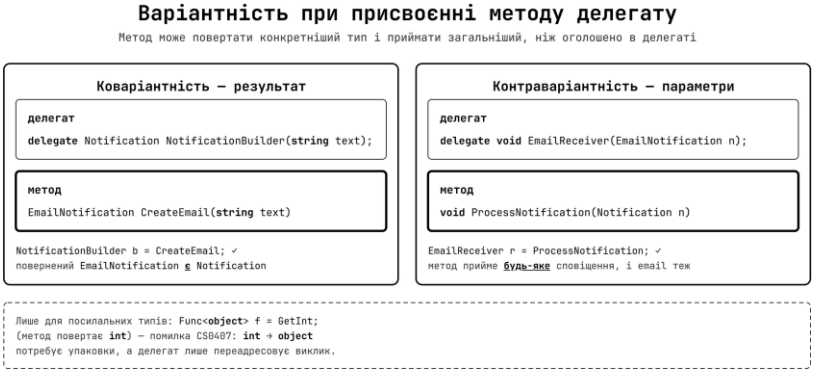
// метод приймає базовий тип Notification — контраваріантність
EmailReceiver receiver = ProcessNotification;
receiver(new EmailNotification("Прийом лікаря підтверджено"));

void ProcessNotification(Notification n) => n.Print();

class Notification
{
    public string Text { get; }
    public Notification(string text) => Text = text;
    public virtual void Print() => Console.WriteLine($"Сповіщення: {Text}");
}
```

```
15  class EmailNotification : Notification
16  {
17      public EmailNotification(string text) : base(text) { }
18      public override void Print() => Console.WriteLine($"Email: {Text}");
19  }
20
21  // делегат приймає похідний тип EmailNotification
22  delegate void EmailReceiver(EmailNotification notification);
```

На перший погляд це може здатися суперечливим: делегат оголошує EmailNotification, а метод приймає Notification. Але логіка правильна: при виклику receiver(...) ми завжди передаємо EmailNotification. Будь-який EmailNotification є Notification, тому метод ProcessNotification(Notification n) коректно обробить його. Метод з ширшим типом параметра — більш гнучкий, і все, що він може зробити з Notification, він зможе зробити і з EmailNotification.



Коваріантність і контраваріантність делегатів

Коваріантність та контраваріантність в узагальнених делегатах

Узагальнені делегати також підтримують коваріантність і контраваріантність — через ключові слова out і in у параметрах типу.

Коваріантний узагальнений делегат (out)

Ключове слово out у параметрі типу означає, що цей тип використовується лише як **тип, що повертається**. Завдяки цьому делегат із більш конкретним типом можна присвоїти змінній делегата з більш загальним типом:

```C#
using System;

// повертає EmailNotification — більш конкретний тип
NotificationBuilder<EmailNotification> emailBuilder = text => new
EmailNotification(text);

// завдяки out — можна присвоїти делегату з базовим типом
NotificationBuilder<Notification> generalBuilder = emailBuilder; //
ковариантність

Notification n = generalBuilder("Результати МРТ");
n.Print(); // Email: Результати МРТ

class Notification
{
    public string Text { get; }
    public Notification(string text) => Text = text;
    public virtual void Print() => Console.WriteLine($"Сповіщення: {Text}");
}
class EmailNotification : Notification
{
    public EmailNotification(string text) : base(text) { }
    public override void Print() => Console.WriteLine($"Email: {Text}");
}

delegate T NotificationBuilder<out T>(string text);
```

Без out компілятор заборонив би таке присвоєння, навіть попри те, що EmailNotification є Notification.

Контраваріантний узагальнений делегат (in)

Ключове слово in означає, що параметр типу використовується лише як **тип параметра** делегата. Завдяки цьому делегат із більш загальним типом можна присвоїти змінній делегата з більш конкретним типом:

```C#
using System;

// приймає базовий тип Notification
NotificationReceiver<Notification> generalReceiver = n => n.Print();

// завдяки in — можна присвоїти делегату з похідним типом
NotificationReceiver<EmailNotification> emailReceiver = generalReceiver; //
контраваріантність

generalReceiver(new Notification("Загальне повідомлення"));     // Сповіщення:
...
generalReceiver(new EmailNotification("Результати аналізів"));  // Email: ...
emailReceiver(new EmailNotification("Прийом підтверджено"));    // Email: ...

class Notification
{
    public string Text { get; }
    public Notification(string text) => Text = text;
```

```
    public virtual void Print() => Console.WriteLine($"Сповіщення: {Text}");
}
class EmailNotification : Notification
{
    public EmailNotification(string text) : base(text) { }
    public override void Print() => Console.WriteLine($"Email: {Text}");
}

delegate void NotificationReceiver<in T>(T notification);
```

Як і у випадку з узагальненими інтерфейсами: параметр коваріантного типу (out) застосовується лише до типу, що повертається, а параметр контраваріантного типу (in) — лише до параметрів делегата.

Поєднання коваріантності та контраваріантності

Один узагальнений делегат може одночасно використовувати обидва оператори — in для параметра і out для типу, що повертається:

C#
```
using System;

// конвертер: з будь-якого Notification створює EmailNotification
NotificationConverter<Notification, EmailNotification> toEmail =
    n => new EmailNotification($"[Email] {n.Text}");

// контраваріантність по M: SmsNotification → Notification (ширший тип)
// коваріантність по E:     EmailNotification → Notification (ширший тип)
NotificationConverter<SmsNotification, Notification> converter = toEmail;

Notification result = converter(new SmsNotification("Аналіз крові"));
result.Print(); // Email: [Email] Аналіз крові

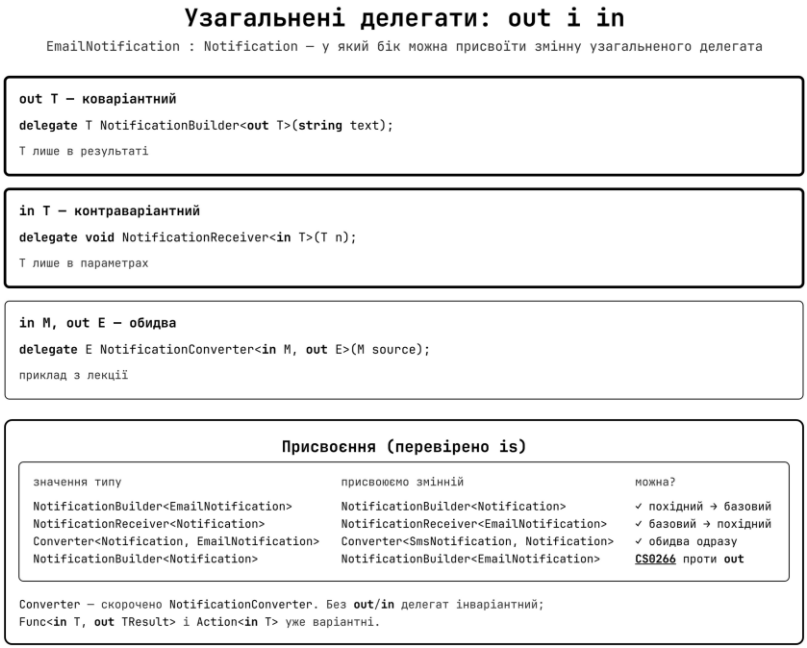
class Notification
{
    public string Text { get; }
    public Notification(string text) => Text = text;
    public virtual void Print() => Console.WriteLine($"Сповіщення: {Text}");
}
class EmailNotification : Notification
{
    public EmailNotification(string text) : base(text) { }
    public override void Print() => Console.WriteLine($"Email: {Text}");
}
class SmsNotification : Notification
{
    public SmsNotification(string text) : base(text) { }
    public override void Print() => Console.WriteLine($"SMS: {Text}");
}

// конвертер: приймає тип M, повертає тип E
delegate E NotificationConverter<in M, out E>(M source);
```

Тут делегат converter очікує: взяти SmsNotification і повернути Notification. Ми присвоїли йому toEmail, який бере будь-який Notification і повертає EmailNotification. Контраваріантність по M означає: SmsNotification → Notification (параметр стає ширшим). Коваріантність по E означає: EmailNotification → Notification (тип повернення стає ширшим). Обидві заміни безпечні, і компілятор це перевіряє статично.

Важливе обмеження: варіантність працює лише для **посилальних типів**. Метод int GetCount() не можна присвоїти змінній Func<object> (помилка CS0407), хоча int неявно перетворюється на object: таке перетворення вимагає упаковки (створення нового об'єкта в купі), а делегат лише переадресовує виклик і не може вставити код перетворення. Для string GetName() присвоєння Func<object> працює, бо string — посилальний тип і посилання передається як є (перевірено на .NET 10).

Якщо узагальнити: **коваріантність** — від більш похідного до більш загального типу (EmailNotification → Notification), **контраваріантність** — від більш загального до більш похідного (Notification → EmailNotification).



Коваріантність та контраваріантність: напрямки заміни типів

6.5. Делегати Action, Predicate та Func

У .NET є кілька вбудованих узагальнених делегатів, які покривають переважну більшість сценаріїв використання делегатів без потреби оголошувати власні типи. Найбільш вживані з них — Action, Predicate та Func. Їх наявність у стандартній бібліотеці дозволяє писати компактний код: замість delegate void AlertHandler(string msg) достатньо написати Action<string>, і компілятор розуміє те ж саме.

Action

Делегат Action представляє дію, яка нічого **не повертає** (void). Він має кілька перевантажених версій залежно від кількості параметрів: від Action (без параметрів) до Action<T1, T2, …, T16> (до 16 параметрів):

```
C#
1  public delegate void Action()
```

```
2  public delegate void Action<in T>(T obj)
3  public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2)
4  // ... до Action<T1..T16>
```

Action зазвичай передається як параметр методу і описує реакцію на якусь подію. Наприклад, метод обробки показника пацієнта може приймати Action як параметр і виконати потрібну дію залежно від контексту:

```
C#
 1  using System;
 2
 3  ProcessVitals("Іван Петренко", 145, LogAlert);
 4  ProcessVitals("Марія Коваль",  72,  LogNormal);
 5
 6  void ProcessVitals(string patient, int pulse, Action<string, int> report)
 7      => report(patient, pulse);
 8
 9  void LogAlert(string name, int pulse)
10      => Console.WriteLine($"[УВАГА] {name}: пульс {pulse.ToString()} уд/хв —
    критично!");
11
12  void LogNormal(string name, int pulse)
13      => Console.WriteLine($"[OK] {name}: пульс {pulse.ToString()} уд/хв —
    норма");
```

Метод ProcessVitals не містить жодної логіки виведення — він лише викликає дію report, яку передає зовнішній код. Це дає гнучкість: в одному місці передаємо LogAlert, в іншому — LogNormal, або будь-яку лямбду.

Action без параметрів зручний для реєстрації простих зворотних викликів:

```
C#
 1  using System;
 2
 3  RunProcedure("Забір крові",   () => Console.WriteLine("  → Зразок взято,
    відправлено до лабораторії"));
 4  RunProcedure("ЕКГ",           () => Console.WriteLine("  → ЕКГ записано,
    роздруковано"));
 5  RunProcedure("МРТ",           () => Console.WriteLine("  → МРТ виконано, знімки
    збережено"));
 6
 7  void RunProcedure(string name, Action onComplete)
 8  {
 9      Console.WriteLine($"Процедура: {name}");
10      onComplete();
11  }
```

Predicate

Делегат Predicate<T> приймає один параметр типу T і повертає bool. Він описує умову — відповідає або не відповідає об'єкт певному критерію:

```
C#
1  delegate bool Predicate<in T>(T obj);
```

Predicate використовується для фільтрації, перевірки, пошуку. Наприклад, перевіримо кілька показників пацієнта на відповідність нормі:

```
C#
 1  using System;
 2
 3  Predicate<double> isGlucoseNormal    = g => g >= 3.9 && g <= 6.1;
 4  Predicate<int>    isPulseNormal      = p => p >= 60  && p <= 100;
 5  Predicate<double> isCholesterolHigh  = c => c > 5.2;
 6
 7  Console.WriteLine($"Глюкоза 5.4: {(isGlucoseNormal(5.4) ? "норма" :
    "відхилення")}");
 8  Console.WriteLine($"Глюкоза 7.8: {(isGlucoseNormal(7.8) ? "норма" :
    "відхилення")}");
 9  Console.WriteLine($"Пульс 88: {(isPulseNormal(88) ? "норма" : "відхилення")}");
10  Console.WriteLine($"Холестерин 6.1: {(isCholesterolHigh(6.1) ? "підвищений" :
    "норма")}");
```

Predicate<T> можна передавати у методи як параметр фільтрації:

```
C#
 1  using System;
 2
 3  double[] glucoseTests = { 3.5, 5.1, 7.2, 4.8, 8.9, 3.1, 6.5 };
 4
 5  int aboveNorm = CountIf(glucoseTests, g => g > 6.1);
 6  int belowNorm = CountIf(glucoseTests, g => g < 3.9);
 7
 8  Console.WriteLine($"Вище норми: {aboveNorm.ToString()} результатів");
 9  Console.WriteLine($"Нижче норми: {belowNorm.ToString()} результатів");
10
11  int CountIf(double[] values, Predicate<double> condition)
12  {
13      int count = 0;
14      foreach (var v in values)
15          if (condition(v)) count++;
16      return count;
17  }
```

Func

Делегат Func повертає значення і може приймати параметри. Останній параметр типу завжди є **типом повернення** (позначається TResult):

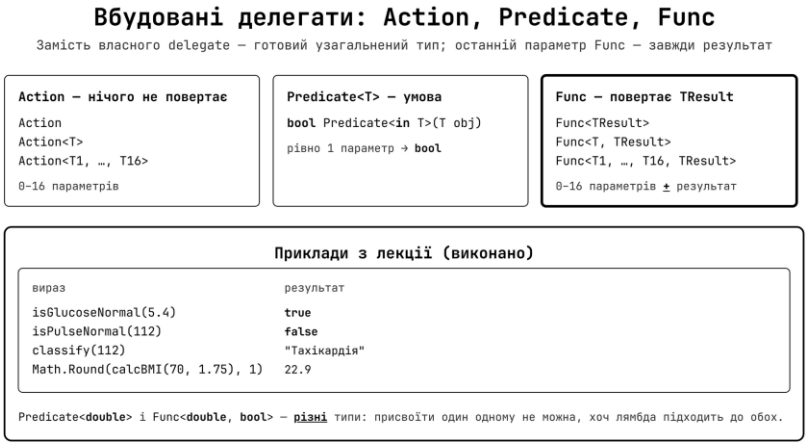
```
C#
1  Func<TResult>                          // без параметрів, повертає TResult
2  Func<T, TResult>                       // один параметр T, повертає TResult
3  Func<T1, T2, TResult>                  // два параметри, повертає TResult
4  // ... до Func<T1..T16, TResult>
```

Func часто використовується для обчислень і перетворень. Наприклад, набір функцій обробки медичних показників:

```
C#
1  using System;
2
3  Func<double, double, double> calcBMI  = (weight, height) => weight / (height *
   height);
4  Func<int, string>            classify = pulse => pulse > 100 ? "Тахікардія" :
   pulse < 60 ? "Брадикардія" : "Норма";
5  Func<double, double>         kgToLb   = kg => kg * 2.20462;
6
7  Console.WriteLine($"ІМТ 70 кг / 1.75 м: {calcBMI(70, 1.75).ToString("F1")}");
8  Console.WriteLine($"Пульс 110: {classify(110)}");
9  Console.WriteLine($"70 кг у фунтах: {kgToLb(70).ToString("F1")}");
```

Func зручний також як параметр методу — для стратегій обчислення:

```
C#
 1  using System;
 2
 3  double dose1 = CalcDose("Аспірін",    70.0, weight => weight * 10.0);
 4  double dose2 = CalcDose("Ібупрофен",  60.0, weight => weight * 5.0);
 5  double dose3 = CalcDose("Парацетамол",80.0, weight => weight * 15.0);
 6
 7  Console.WriteLine($"Аспірін:     {dose1.ToString("F0")} мг");
 8  Console.WriteLine($"Ібупрофен:   {dose2.ToString("F0")} мг");
 9  Console.WriteLine($"Парацетамол: {dose3.ToString("F0")} мг");
10
11  double CalcDose(string drug, double weightKg, Func<double, double> formula)
12  {
13      double dose = formula(weightKg);
14      return dose;
15  }
```



Вбудовані делегати: Action, Predicate, Func

Замикання

Замикання (closure) — це функція, яка «запам'ятовує» своє лексичне оточення навіть тоді, коли виконується поза областю видимості, де вона була створена. Якщо звичайна функція звертається лише до своїх параметрів і локальних змінних, то замикання також має доступ до змінних зовнішньої функції, у якій воно визначене.

Технічно замикання складається з трьох компонентів:

— **зовнішня функція** — визначає область видимості і змінні (лексичне оточення)

— **лексичне оточення** — змінні та параметри зовнішньої функції

— **вкладена функція** — захоплює і використовує лексичне оточення

У C# замикання реалізуються через локальні функції та лямбда-вирази.

Замикання через локальні функції

C#

```
using System;

var counter = CreateAlertCounter("ICU");
counter(); // ICU: сповіщення #1
counter(); // ICU: сповіщення #2
counter(); // ICU: сповіщення #3
```

```
 7
 8  Action CreateAlertCounter(string ward)
 9  {
10      int count = 0;                          // лексичне оточення
11      void Increment()                        // локальна функція
12      {
13          count++;                            // захоплює count із зовнішньої
    функції
14          Console.WriteLine($"{ward}: сповіщення #{count.ToString()}");
15      }
16      return Increment;                       // повертаємо функцію разом з
    оточенням
17  }
```

Тут CreateAlertCounter повертає локальну функцію Increment. Після виходу з CreateAlertCounter змінна count більше не існує «звичайним чином» на стеку — але замикання зберігає її в купі (heap). Кожен виклик counter() читає і змінює саме цю захоплену змінну, тому лічильник зростає між викликами.

Замикання через лямбда-вирази

За допомогою лямбд можна скоротити визначення замикання:

```
C#
 1  using System;
 2
 3  var createCounter = (string ward) =>
 4  {
 5      int count = 0;
 6      Action increment = () =>
 7      {
 8          count++;
 9          Console.WriteLine($"{ward}: сповіщення #{count.ToString()}");
10      };
11      return increment;
12  };
13
14  var icuCounter  = createCounter("ICU");
15  var cardCounter = createCounter("Кардіологія");
16
17  icuCounter();   // ICU: сповіщення #1
18  icuCounter();   // ICU: сповіщення #2
19  cardCounter();  // Кардіологія: сповіщення #1
20  icuCounter();   // ICU: сповіщення #3
```

Кожен виклик createCounter створює **нове** замикання з власним незалежним count. icuCounter і cardCounter — два окремих замикання, кожне з яких зберігає свій власний лічильник.

Застосування параметрів у замиканнях

До лексичного оточення замикання належать не лише локальні змінні, а й **параметри зовнішньої функції**. Це дозволяє створювати «фабрики функцій» — методи, що повертають налаштовані дії:

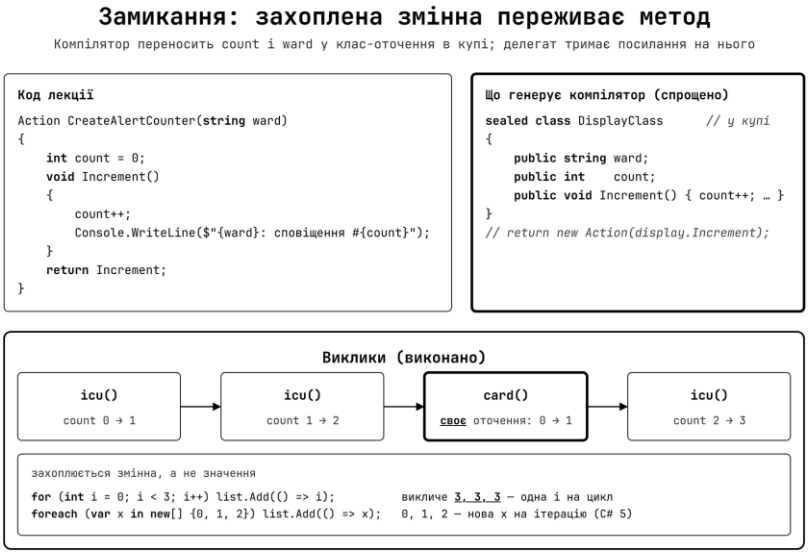
```C#
using System;

var glucoseCheck  = CreateRangeCheck(3.9, 6.1);
var cholesterolCheck = CreateRangeCheck(3.0, 5.2);

Console.WriteLine($"Глюкоза 5.4: {glucoseCheck(5.4)}");
Console.WriteLine($"Глюкоза 7.8: {glucoseCheck(7.8)}");
Console.WriteLine($"Холестерин 4.1: {cholesterolCheck(4.1)}");
Console.WriteLine($"Холестерин 6.0: {cholesterolCheck(6.0)}");

Func<double, string> CreateRangeCheck(double low, double high)
{
    // параметри low і high захоплюються замиканням
    return value =>
        value < low  ? $"Нижче норми (норма {low.ToString("F1")}–{high.ToString("F1")})" :
        value > high ? $"Вище норми (норма {low.ToString("F1")}–{high.ToString("F1")})" :
                       "Норма";
}
```

Метод CreateRangeCheck приймає два порогові значення і повертає лямбду, яка їх «запам'ятовує». glucoseCheck і cholesterolCheck — це два різних замикання, кожне з яких містить свої пороги. Замість передавати low і high при кожному виклику — вони «вбудовані» у функцію один раз при її створенні.



Механізм замикання: захоплення лексичного оточення

Пастка: замикання в циклі

Замикання захоплює **змінну**, а не її значення на момент створення лямбди. Найчастіше це проявляється в циклах. Змінна циклу for оголошена один раз на весь цикл, тому всі лямбди, створені всередині, захоплюють ту саму змінну і після завершення циклу бачать її останнє значення:

```C#
using System;
using System.Collections.Generic;

var forChecks = new List<Func<string>>();
for (int i = 0; i < 3; i++)
    forChecks.Add(() => $"палата {i}");

var foreachChecks = new List<Func<string>>();
foreach (var ward in new[] { 0, 1, 2 })
    foreachChecks.Add(() => $"палата {ward}");

foreach (var f in forChecks)     Console.WriteLine($"for:     {f()}");     // палата 3 — тричі
foreach (var f in foreachChecks) Console.WriteLine($"foreach: {f()}");     // палата 0, 1, 2
```

Цикл foreach з C# 5 створює нову змінну на кожну ітерацію, тому там пастки немає. Для for виправлення просте — скопіювати значення в локальну змінну всередині тіла: int copy = i; forChecks.Add(() => $"палата {copy}");. Перевірено на .NET 10: for дає «палата 3» тричі, foreach — 0, 1, 2.

Скорочений запис через каррінг (ланцюг лямбд):

```
C#
1  using System;
2
3  var check = (double low, double high) => (double value) =>
4      value < low ? "Нижче" : value > high ? "Вище" : "Норма";
5
6  var hemoglobin = check(120, 170);
7  Console.WriteLine($"Гемоглобін 105: {hemoglobin(105)}");
8  Console.WriteLine($"Гемоглобін 145: {hemoglobin(145)}");
9  Console.WriteLine($"Гемоглобін 185: {hemoglobin(185)}");
```

Замикання — потужний механізм, але він несе відповідальність: захоплена змінна живе у пам'яті доти, доки живе замикання. Якщо замикання зберігається довго (наприклад, підписане на подію, що ніколи не відписується), захоплені великі об'єкти не звільняться збирачем сміття. При роботі з замиканнями слід усвідомлювати, які саме змінні захоплюються і який час вони мають жити.

Питання для самоконтролю — Розділ 6. Делегати, події та лямбди

6.1. Делегати

1. Що таке делегат у C#? Чим він принципово відрізняється від звичайної змінної типу int або string? Яке місце займають делегати в концепції «методи як об'єкти першого класу»?
2. Оголошено делегат delegate int Calculate(int x, int y). Які з наведених методів відповідають цьому делегату, а які — ні? Поясніть причину для кожного:

```
C#
1  int Add(int a, int b) => a + b;
2  double Add(int a, int b) => a + b;
3  int Add(int a, int b, int c) => a + b + c;
4  int Multiply(ref int a, int b) => a * b;
```

3. Що виведе наступний код? Поясніть, яке значення повертається, якщо делегат з кількома методами у списку виклику повертає значення:

```
C#
1  Calculate calc = M1;
2  calc += M2;
3  calc += M3;
4  Console.WriteLine(calc(10, 2));
5
6  int M1(int x, int y) { Console.WriteLine("M1"); return x + y; }
7  int M2(int x, int y) { Console.WriteLine("M2"); return x - y; }
8  int M3(int x, int y) { Console.WriteLine("M3"); return x * y; }
9  delegate int Calculate(int x, int y);
```

4. Порівняйте два синтаксиси ініціалізації делегата: Calculate calc = Add та Calculate calc = new Calculate(Add). Чи є між ними функціональна різниця? В яких ситуаціях кожен з них зручніший?
5. Поясніть патерн «зворотного виклику» (callback) через делегати. Чому жорстко вшитий Console.WriteLine у метод бізнес-класу — це погане рішення, і як делегати вирішують цю проблему? Наведіть конкретний приклад.
6. Напишіть метод ApplyOperation(double[] values, Transform op), де Transform — делегат delegate double Transform(double x), який застосовує операцію до кожного елемента масиву і повертає новий масив результатів. Викличте метод тричі з різними операціями.
7. Що відбудеться при виклику делегата, якщо його список виклику порожній (делегат дорівнює null)? Як безпечно викликати делегат, не отримуючи NullReferenceException? Покажіть два різних способи.
8. Що таке анонімний метод? Чим він відрізняється від іменованого методу і де його доречно використовувати? Чи може анонімний метод не вказувати параметри, навіть якщо делегат їх приймає?

6.2. Лямбди

9. Що таке лямбда-вираз у C#? Назвіть основні складові синтаксису: лямбда-оператор, список параметрів, тіло. Яким типом даних є лямбда-вираз з точки зору системи типів C#?

10. Порівняйте три способи записати одну й ту саму логіку: іменований метод, анонімний метод і лямбда-вираз. Напишіть кожен варіант для делегата delegate bool IsMatch(double value) з умовою value > 6.1.
11. Що виведе наступний код? Поясніть механізм автоматичного виведення типів параметрів лямбди:

```
C#
1  AlertHandler alert = (name, msg) => Console.WriteLine($"[{name}]: {msg}");
2  alert("ICU", "Критичний стан");
3
4  delegate void AlertHandler(string department, string message);
```

12. Коли в лямбді можна опустити дужки навколо параметра? Коли потрібен явний return, а коли він автоматичний? Наведіть приклади обох варіантів.
13. Чим лямбда-вираз, що повертає результат, відрізняється від лямбди без повернення? Напишіть лямбда-вираз для класифікації показника пульсу: якщо менше 60 — «Брадикардія», більше 100 — «Тахікардія», інакше — «Норма».
14. Що виведе наступний код? Поясніть, як і чому спрацьовує видалення лямбди через -=:

```
C#
1  AlertChain log = () => Console.WriteLine("LOG");
2  AlertChain sms = () => Console.WriteLine("SMS");
3  AlertChain all = log;
4  all += sms;
5  all -= sms;
6  all?.Invoke();
7  delegate void AlertChain();
```

15. Поясніть концепцію «фабрики функцій»: метод повертає лямбда-вираз як результат. Яким є тип такого методу? Напишіть метод CreateThresholdCheck(double min, double max), який повертає лямбду, що перевіряє, чи потрапляє значення в межі норми.
16. Коли краще використовувати лямбда-вираз замість іменованого методу? Чи є ситуації, коли іменований метод переважніший? Аргументуйте свою відповідь.

6.3. Події

17. Що таке подія у C#? Чим подія принципово відрізняється від звичайного делегата? Які обмеження накладає ключове слово event на зовнішній код?
18. У чому проблема жорстко вшитого Console.WriteLine у бізнес-клас? Як події вирішують цю проблему? Поясніть ролі «видавця» (publisher) і «підписника» (subscriber) у патерні «видавець-підписник».
19. Знайдіть помилку у наступному коді та поясніть, чому вона виникає:

```C#
class Button
{
    public event Action Clicked;
}
class Program
{
    static void Main()
    {
        Button btn = new Button();
        btn.Clicked = () => Console.WriteLine("Натиснуто!"); // рядок 9
    }
}
```

20. Поясніть, навіщо при виклику події використовувати Notify?.Invoke(...) замість просто Notify(...). Що відбудеться в обох випадках, якщо жоден обробник не зареєстрований?
21. Що таке аксесори add і remove для подій? В яких ситуаціях їх варто визначати явно? Напишіть приклад події з аксесорами, що реєструють назву методу-обробника в консолі при підписці.
22. Спроєктуйте клас MedicalDevice з подією StatusChanged, яка передає обробнику два аргументи: попередній і новий статус пристрою. Визначте відповідний делегат і клас аргументів події.
23. Поясніть антипатерн «витік пам'яті через події». За яких умов він виникає? Як його уникнути? Що означає правило «кожному += має відповідати -=»?
24. Порівняйте три способи реєстрації обробника події: через іменований метод, анонімний метод і лямбду. Яка особливість виникає при спробі видалити анонімний обробник або лямбду через -=?

6.4. Коваріантність та контраваріантність делегатів

25. Що таке коваріантність і контраваріантність делегатів? Поясніть простими словами: яка відмінність між ними і чого кожна стосується (параметри чи тип повернення)?
26. Є ієрархія: EmailNotification успадковує Notification. Оголошено делегат delegate Notification Build(string text). Чи можна присвоїти йому метод EmailNotification CreateEmail(string text)? Чому так або ні? Яке це явище?
27. Поясніть логіку контраваріантності: делегат оголошує параметр EmailNotification, а метод приймає Notification. Чому компілятор вважає таке присвоєння безпечним? Обгрунтуйте через принцип «is-a».
28. Що виведе наступний код? Поясніть, чому присвоєння у рядку 3 є допустимим або недопустимим:

```
C#
1  // Ієрархія: Cat : Animal
2  delegate Animal GetAnimal();
3  GetAnimal factory = () => new Cat(); // рядок 3
4  Animal a = factory();
```

29. Навіщо в узагальнених делегатах потрібні ключові слова out і in? Без них компілятор відмовляє у присвоєнні навіть для сумісних типів — чому?
30. Оголошено delegate T Transform<out T>(string input). Поясніть, що означає out у цьому контексті. Яке присвоєння тепер стає дозволеним: Transform<string> у Transform<object> чи навпаки?
31. Спроєктуйте узагальнений делегат delegate R Converter<in T, out R>(T input) для конвертації між медичними одиницями. Покажіть, як завдяки in і out можна безпечно присвоїти Converter<Animal, Cat> у Converter<Cat, Animal> — або навпаки, залежно від ієрархії.
32. Порівняйте коваріантність і контраваріантність: що з ними «спільне», а що «протилежне»? В якому напрямі можливе присвоєння для кожної з форм? Наведіть приклад ієрархії і покажіть обидва напрями.

6.5. Делегати Action, Predicate та Func

33. Навіщо у .NET існують вбудовані делегати Action, Predicate та Func? Чим вони зручніші порівняно з оголошенням власного делегата щоразу?

34. В чому різниця між Action<T> і Func<T, bool> і Predicate<T>? Чи є функціональна різниця між Predicate<double> і Func<double, bool>? Коли кожен з них доречніший?
35. Напишіть метод Filter(int[] source, Predicate<int> condition), який повертає масив елементів, що задовольняють умові. Викличте його з лямбдою для вибору парних чисел, а потім для чисел більших за 50.
36. Що виведе наступний код? Поясніть механізм замикання:

```C#
string ward = "Кардіологія";
int count = 0;
Action<string> log = msg =>
{
    count++;
    Console.WriteLine($"[{ward}] #{count}: {msg}");
};
log("Пульс 145");
log("Тиск 180");
Console.WriteLine($"Всього: {count}");
```

37. Поясніть, що таке замикання (closure). Які компоненти входять до замикання? Чому після виходу з зовнішньої функції захоплені змінні продовжують існувати?
38. Напишіть «фабрику предикатів» — метод CreateRangeFilter(double min, double max), що повертає Predicate<double> і перевіряє, чи потрапляє значення в заданий діапазон. Покажіть, що два різних виклики фабрики дають два незалежних предикати з різними порогами.
39. Поясніть потенційну небезпеку замикань: яким чином захоплення великих об'єктів або довгоживучих об'єктів може призвести до проблем з пам'яттю? Назвіть принаймні одну ситуацію, де це стає реальною проблемою.
40. Порівняйте Func<double, double, double> і власний делегат delegate double Calculate(double a, double b). Коли варто оголошувати власний делегат, а коли досить вбудованого Func? Наведіть аргументи на користь кожного підходу.

Розділ 7. Інтерфейси

7.1. Визначення інтерфейсів

Інтерфейс — це посилальний тип, який задає контракт: перелік методів, властивостей, подій та індексаторів, які зобов'язаний реалізувати будь-який клас або структура, що приймає цей інтерфейс. Сам інтерфейс не містить стану об'єкта і, як правило, не містить реалізації — він лише описує, що клас повинен вміти робити, але не як саме. Завдяки цьому один інтерфейс можуть реалізовувати абсолютно різні класи: і Patient, і Doctor, і MedicalDevice — якщо кожен із них відповідає заданому контракту. Саме це робить інтерфейси головним інструментом поліморфізму та слабкого зв'язування в об'єктно-орієнтованому програмуванні.

Важливо розуміти принципову відмінність від успадкування: клас може успадковувати лише один базовий клас, проте реалізовувати будь-яку кількість інтерфейсів. Це дає можливість описувати різні ролі одного об'єкта: пацієнт може водночас реалізовувати IDiagnosable (піддається діагностиці), IBillable (підлягає оплаті) та INotifiable (отримує сповіщення). Інтерфейси не замінюють успадкування — вони доповнюють його там, де потрібно виразити поведінку, а не походження.

Визначення інтерфейсу

Для визначення інтерфейсу використовується ключове слово interface. За угодою іменування в C# назви інтерфейсів починаються з великої літери I — наприклад, IComparable, IEnumerable, IDiagnosable. Префікс I — це угода іменування .NET (історично — відгомін угорської нотації, від якої .NET в іншому відмовився): вона не є обов'язковою вимогою мови, але є загальноприйнятим стилем у всій екосистемі .NET, якого слід дотримуватись для читабельності коду.

Інтерфейс може визначати такі сутності:

— **Методи** — підписи без реалізації (або з реалізацією за замовчуванням починаючи з C# 8.0)

— **Властивості** — оголошення з аксесорами get та/або set, без реалізації

— **Індексатори** — доступ до об'єкта за індексом

— **Події** — оголошення подій через event

— **Статичні поля, константи та статичні конструктори** — починаючи з C# 8.0

— **Статичні абстрактні члени** (static abstract) — починаючи з C# 11; їх реалізує сам тип, а не об'єкт, що дає змогу описувати в інтерфейсі, наприклад, оператори для узагальненої математики

Проте інтерфейси **не можуть** визначати нестатичні змінні (поля екземпляра), конструктори екземпляра та деструктори (статичний конструктор, навпаки, дозволений з C# 8 — він ініціалізує статичні поля інтерфейсу; перевірено на .NET 10). Заборона на нестатичні поля зумовлена природою інтерфейсу: він описує поведінку, а не стан. Зберігати стан — обов'язок класу, який реалізує інтерфейс. Розглянемо клінічний інтерфейс, що охоплює всі дозволені категорії членів:

```
C#
 1  interface IDiagnosable
 2  {
 3      // константа
 4      const int MaxSeverityLevel = 5;
 5      // статична змінна (C# 8.0+)
 6      static int DiagnosisCount = 0;
 7      // метод без реалізації
 8      void RunDiagnostics();
 9      // властивість без реалізації
10      string DiagnosisCode { get; set; }
11      // делегат для події
12      delegate void DiagnosisHandler(string message);
13      // подія
14      event DiagnosisHandler OnDiagnosed;
15  }
```

У цьому прикладі інтерфейс IDiagnosable описує об'єкт медичної системи, який піддається діагностиці. Він не знає, чи це пацієнт, чи лабораторний зразок, чи медичний прилад — він лише фіксує, що будь-який такий об'єкт повинен мати метод запуску діагностики, властивість з кодом діагнозу і подію про завершення діагностики. Конкретна логіка залишається за реалізацією.

Методи та властивості інтерфейсу не мають реалізації — у цьому вони зближуються з абстрактними методами та властивостями абстрактних класів. Метод RunDiagnostics не приймає параметрів, нічого не повертає і не має тіла. Те саме стосується властивості DiagnosisCode: на перший погляд вона схожа на автоматичну властивість, але насправді це лише декларація в інтерфейсі — реалізація буде в класі.

Анатомія інтерфейсу: що може і що не може містити

Інтерфейс описує поведінку, а не стан; усе з верхньої таблиці компілюється на .NET 10

Може містити

член	приклад	з версії
метод	`void RunDiagnostics();`	C# 1
властивість	`string DiagnosisCode { get; set; }`	C# 1
індексатор	`string this[int i] { get; }`	C# 1
подія	`event DiagnosisHandler OnDiagnosed;`	C# 1
вкладений тип	`delegate void DiagnosisHandler(…);`	C# 8
метод з тілом	`void Log() => …;`	C# 8
константа	`const int MaxSeverityLevel = 5;`	C# 8
статичне поле	`static int DiagnosisCount;`	C# 8
статичний конструктор	`static IDiagnosable() { … }`	C# 8
`private / protected` член	`private void Helper() { … }`	C# 8
`static abstract` член	`static abstract int Max { get; }`	C# 11

Не може містити

член	приклад
поле екземпляра	`int _severity;`
конструктор екземпляра	`IDiagnosable() { }`
деструктор	`~IDiagnosable() { }`
автовластивість зі станом	`int X { get; set; } = 1;`

Члени без модифікатора — **public**;
сам інтерфейс без модифікатора — **internal**.

Анатомія інтерфейсу: що може і що не може містити

Модифікатори доступу

Якщо члени інтерфейсу — методи та властивості — не мають явних модифікаторів доступу, то фактично за замовчуванням вони мають доступ public. Це принципово відрізняється від класів і структур, де члени за замовчуванням мають private. Таке рішення закономірне: мета інтерфейсу — визначити відкритий контракт для зовнішнього світу, тому всі оголошені в ньому методи та властивості апріорі публічні. Клас, який реалізує інтерфейс, не може зменшити рівень доступу реалізованого члена — він зобов'язаний залишити його public.

Це правило стосується також констант і статичних змінних. Якщо в класах і структурах вони за замовчуванням мають private, то в інтерфейсах — public. Тому до них можна звертатися безпосередньо через ім'я інтерфейсу:

```
C#
 1  using System;
 2
 3  Console.WriteLine($"Максимальний рівень тяжкості:
    {IDiagnosable.MaxSeverityLevel.ToString()}");
 4  Console.WriteLine($"Кількість діагнозів:
    {IDiagnosable.DiagnosisCount.ToString()}");
 5
 6  interface IDiagnosable
 7  {
 8      const int MaxSeverityLevel = 5;
 9      static int DiagnosisCount = 0;
10      void RunDiagnostics();
11      string DiagnosisCode { get; set; }
12  }
```

Починаючи з C# 8.0, у членів інтерфейсу можна явно вказувати модифікатори доступу. Це дає більш точний контроль над видимістю окремих членів. Наприклад, приватний статичний член може бути допоміжним для реалізації default-методів, тоді як публічні члени формують зовнішній контракт:

```
C#
1  interface IDiagnosable
2  {
3      public const int MaxSeverityLevel = 5;
4      private static int _internalCounter = 0;
5      public void RunDiagnostics();
6      protected internal string DiagnosisCode { get; set; }
7      public delegate void DiagnosisHandler(string message);
8      public event DiagnosisHandler OnDiagnosed;
9  }
```

Як і класи, інтерфейси за замовчуванням мають рівень доступу internal — тобто такий інтерфейс доступний лише в межах поточного проекту (збірки). Щоб зробити інтерфейс доступним із зовнішніх збірок, наприклад якщо він є частиною бібліотеки, використовується модифікатор public:

```
C#
1  public interface IDiagnosable
2  {
3      void RunDiagnostics();
4  }
```

У реальних проектах публічні інтерфейси розміщують в окремій збірці-контракті, яку підключають і клієнтський код, і реалізація. Це дозволяє змінювати реалізацію, не змінюючи публічний контракт.

Антипатерн: «Бог-інтерфейс» — надмірно широкий контракт

Найпоширеніша помилка при проектуванні інтерфейсів — об'єднати в одному інтерфейсі надто багато непов'язаних між собою обов'язків. Такий інтерфейс називають **«Бог-інтерфейс»** (God Interface): він «знає все» і «вміє все», але саме через це стає антипатерном.

Розглянемо типовий приклад. Розробник вирішив описати всі можливі дії над пацієнтом в одному інтерфейсі:

```C#
// АНТИПАТЕРН: надто широкий контракт в одному інтерфейсі
interface IPatient
{
    // діагностика
    void RunDiagnostics();
    string GetDiagnosisCode();
    // лікування
    void PrescribeMedication(string med);
    void AssignToWard(string ward);
    // адміністрування
    void Register();
    void Discharge();
    void GenerateBill();
    // сповіщення
    void SendReminder(string message);
    void NotifyEmergencyContact();
}
```

Проблема виникає, коли цей інтерфейс намагаються реалізувати різні класи. Амбулаторний пацієнт не госпіталізується і не потребує AssignToWard, але **зобов'язаний** реалізувати цей метод — адже він є частиною контракту. Єдине, що може зробити розробник, — кинути виняток, перетворюючи помилку проектування на помилку часу виконання:

```C#
class OutpatientRecord : IPatient
{
    public void RunDiagnostics() { /* ... */ }
    public string GetDiagnosisCode() => "J00";
    public void PrescribeMedication(string med) { /* ... */ }

    // Змушені реалізовувати методи, що не стосуються амбулаторного пацієнта:
    public void AssignToWard(string ward)
        => throw new NotSupportedException("Амбулаторний пацієнт не госпіталізується!");
    public void NotifyEmergencyContact()
        => throw new NotSupportedException("Не реалізовано для амбулаторних!");

    public void Register() { /* ... */ }
    public void Discharge() { /* ... */ }
    public void GenerateBill() { /* ... */ }
```

```
    public void SendReminder(string message) { /* ... */ }
}
```

NotSupportedException у реалізації інтерфейсного методу — чіткий сигнал порушення **принципу розділення інтерфейсів** (ISP — Interface Segregation Principle, детально у розділі 20): клієнти не повинні залежати від методів, які вони не використовують. Компілятор C# не може захистити від помилки, яку можна виявити лише у runtime.

Рішення — розбити «Бог-інтерфейс» на вузькоспеціалізовані інтерфейси. Множинна реалізація інтерфейсів (розділ 7.2) дозволяє кожному класу зібрати рівно той набір обов'язків, який він справді виконує; успадкування інтерфейсів (розділ 7.4) дає змогу ще й об'єднати часто вживаний набір під одним іменем:

C#

```
using System;

var inpatient  = new InpatientRecord("Петренко Іван");
var outpatient = new OutpatientRecord("Коваль Марія");

Console.WriteLine("=== Стаціонар ===");
inpatient.RunDiagnostics();
inpatient.AssignToWard("кардіологія");
inpatient.GenerateBill();
inpatient.NotifyEmergencyContact();

Console.WriteLine("\n=== Амбулаторія ===");
outpatient.RunDiagnostics();
outpatient.GenerateBill();
// outpatient.AssignToWard(...) — помилка компіляції: OutpatientRecord не
реалізує ITreatable

// Кожен інтерфейс — одна відповідальність
interface IDiagnosable
{
    void RunDiagnostics();
    string GetDiagnosisCode();
}

interface ITreatable
{
    void PrescribeMedication(string med);
    void AssignToWard(string ward);
}

interface IAdministrable
{
    void Register();
    void Discharge();
    void GenerateBill();
}

interface INotifiable
{
```

```
    void SendReminder(string message);
    void NotifyEmergencyContact();
}

// Стаціонарний пацієнт потребує всіх ролей
class InpatientRecord : IDiagnosable, ITreatable, IAdministrable, INotifiable
{
    public string Name { get; }
    public InpatientRecord(string name) => Name = name;

    public void RunDiagnostics()          => Console.WriteLine($"{Name}: МРТ, ЕКГ");
    public string GetDiagnosisCode()      => "I10.9";
    public void PrescribeMedication(string med) => Console.WriteLine($"{Name}: призначено {med}");
    public void AssignToWard(string ward) => Console.WriteLine($"{Name}: переведено → {ward}");
    public void Register()                => Console.WriteLine($"{Name}: госпіталізовано");
    public void Discharge()               => Console.WriteLine($"{Name}: виписано");
    public void GenerateBill()            => Console.WriteLine($"{Name}: рахунок сформовано");
    public void SendReminder(string msg)  => Console.WriteLine($"{Name}: нагадування — {msg}");
    public void NotifyEmergencyContact()  => Console.WriteLine($"{Name}: повідомлено родичів");
}

// Амбулаторний пацієнт — тільки потрібні ролі; компілятор не дасть викликати AssignToWard
class OutpatientRecord : IDiagnosable, IAdministrable
{
    public string Name { get; }
    public OutpatientRecord(string name) => Name = name;

    public void RunDiagnostics()     => Console.WriteLine($"{Name}: аналізи крові");
    public string GetDiagnosisCode() => "J00";
    public void Register()           => Console.WriteLine($"{Name}: записано на прийом");
    public void Discharge()          => Console.WriteLine($"{Name}: консультацію завершено");
    public void GenerateBill()       => Console.WriteLine($"{Name}: рахунок — 350 грн");
}
```

Тепер OutpatientRecord реалізує лише ті інтерфейси, що дійсно стосуються амбулаторного пацієнта. Помилка виявляється на **етапі компіляції**, а не у runtime. Кожен інтерфейс можна підмінити окремим тест-заглушенням (mock), не зачіпаючи інших.

	Бог-інтерфейс	Вузькі інтерфейси
Відповідальність	Один клас знає і вміє все	Кожен інтерфейс — одна здібність
Непотрібні методи	Клас реалізує методи, що не стосуються його	Клас реалізує лише релевантні інтерфейси
Де виявляється помилка	NotSupportedException у runtime	Помилка компіляції
Тестування	Великий mock із багатьма методами	Мінімальний mock із потрібними методами
Вплив змін	Зміна будь-якого методу зачіпає всіх реалізаторів	Зміна вузького інтерфейсу зачіпає лише залежних від нього

Реалізація за замовчуванням

Починаючи з C# 8.0, інтерфейси підтримують реалізацію методів і властивостей за замовчуванням. Це означає, що в інтерфейсі можна визначити повноцінний метод із тілом, як у звичайному класі. Такий метод буде автоматично доступний усім класам, що реалізують інтерфейс, — якщо клас не перевизначить його, буде використана реалізація інтерфейсу. Це зручно для поступового розширення вже опублікованих інтерфейсів без порушення зворотної сумісності: новий метод можна додати з реалізацією за замовчуванням, і всі вже існуючі реалізатори продовжать компілюватись без змін.

```C#
using System;

IDiagnosable p = new Patient("Марія Коваль");
p.RunDiagnostics();
p.LogDiagnostics();

interface IDiagnosable
{
    void RunDiagnostics();
    // реалізація методу за замовчуванням
    void LogDiagnostics()
    {
        Console.WriteLine("[LOG] Діагностика завершена — результат записано");
    }
}

class Patient : IDiagnosable
```

```
{
    public string Name { get; }
    public Patient(string name) => Name = name;
    public void RunDiagnostics() =>
        Console.WriteLine($"Діагностика пацієнта {Name}");
    // LogDiagnostics не перевизначаємо — використовується default
}
```

З реалізацією властивостей за замовчуванням в інтерфейсах ситуація складніша: оскільки інтерфейси не можуть визначати нестатичні змінні, властивість інтерфейсу не може маніпулювати станом екземпляра. Тим не менш реалізацію за замовчуванням для властивості визначити можна — наприклад, для властивості тільки для читання, що повертає фіксоване значення:

```C#
using System;

IDiagnosable sample = new LabSample("LAB-2024-001");
sample.RunDiagnostics();
Console.WriteLine($"Рівень тяжкості за замовчуванням:
{sample.DefaultSeverity.ToString()}");

interface IDiagnosable
{
    void RunDiagnostics() => Console.WriteLine("Стандартна діагностика");
    // властивість тільки для читання з реалізацією за замовчуванням
    int DefaultSeverity { get { return 1; } }
}

class LabSample : IDiagnosable
{
    public string SampleId { get; }
    public LabSample(string id) => SampleId = id;
    // RunDiagnostics і DefaultSeverity — використовуємо default
}
```

Варто зазначити: якщо інтерфейс має приватні методи або властивості (з модифікатором private), вони обов'язково повинні мати реалізацію за замовчуванням — адже до них не може звернутися реалізуючий клас. Те саме стосується статичних методів, які теж мають бути реалізовані в самому інтерфейсі:

```C#
using System;

Console.WriteLine($"Максимум: {IDiagnosable.MaxSeverityLevel.ToString()}");
IDiagnosable.MaxSeverityLevel = 7;
Console.WriteLine($"Оновлено: {IDiagnosable.MaxSeverityLevel.ToString()}");
double score = IDiagnosable.CalcRiskScore(145, 98);
Console.WriteLine($"Ризик-скор: {score.ToString("F2")}");

interface IDiagnosable
```

```
{
    public const int MinSeverityLevel = 0;
    private static int _maxSeverity = 5;

    static int MaxSeverityLevel
    {
        get => _maxSeverity;
        set { if (value > 0) _maxSeverity = value; }
    }

    // статичний допоміжний метод із реалізацією в інтерфейсі
    static double CalcRiskScore(int systolic, int diastolic)
        => systolic * 0.6 + diastolic * 0.4;
}
```

Статичний метод CalcRiskScore визначений і реалізований прямо в інтерфейсі. Він не належить жодному конкретному об'єкту і не успадковується класами-реалізаторами — це суто утилітарний метод інтерфейсу, доступний через ім'я інтерфейсу.

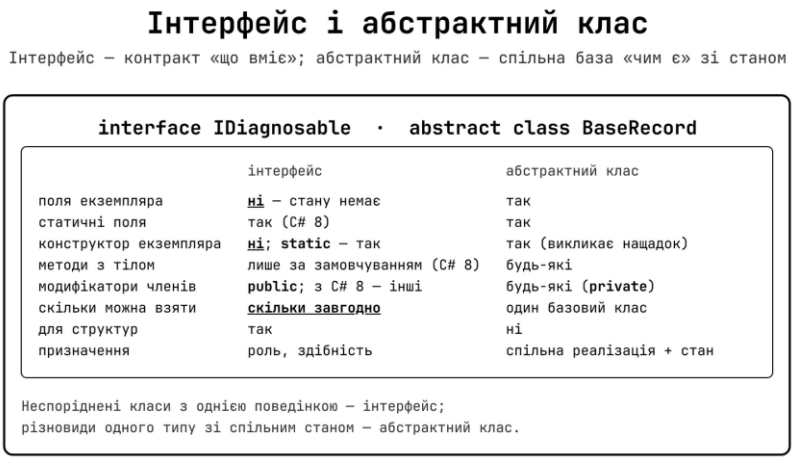
Інтерфейс vs Абстрактний клас

Студенти часто запитують: коли варто використовувати інтерфейс, а коли абстрактний клас? Обидва інструменти дозволяють описати «незавершений» тип, який конкретизується у нащадках, але їх призначення принципово різне.

Абстрактний клас — це неповна реалізація: він може містити поля, конструктори, реалізовані методи та абстрактні методи. Він описує спільну базу для споріднених класів і передає їм стан та частину поведінки. Наприклад, AbstractMedicalRecord може зберігати PatientId, дату створення і реалізовувати метод Format(), лишивши абстрактним метод Validate().

Інтерфейс — це чистий контракт поведінки: він не зберігає стану і не нав'язує реалізацію (крім default-методів). Він описує, що об'єкт вміє робити, незалежно від того, що він є. Клас може реалізовувати кілька інтерфейсів одночасно, чого не можна зробити з абстрактними класами. Наприклад, Patient може одночасно реалізовувати IDiagnosable, IBillable та INotifiable.

Практичне правило: якщо різні, не споріднені класи мають розділяти одну поведінку — використовуй інтерфейс. Якщо класи є різновидами одного типу і мають спільний стан або базову реалізацію — використовуй абстрактний клас.



Інтерфейс vs Абстрактний клас

Визначення інтерфейсу у проекті

У Visual Studio є спеціальний шаблон для швидкого додавання нового інтерфейсу в окремому файлі. Для цього можна натиснути праву кнопку миші на проекті в Solution Explorer і у контекстному меню вибрати **Додати → Новий елемент**, а потім у діалоговому вікні — пункт **Інтерфейс**. Visual Studio автоматично створить файл із заготовкою interface IMyInterface { } і відповідним namespace.

Водночас можна додати звичайний файл класу (.cs) і вручну написати ключове слово interface замість class — компілятор не робить різниці, в якому файлі визначено інтерфейс. За угодою, кожен інтерфейс розміщують в окремому файлі, назва якого збігається з іменем інтерфейсу, наприклад IDiagnosable.cs. Це полегшує навігацію в проекті та є стандартом у більшості C# кодових баз.

7.2. Застосування інтерфейсів

Інтерфейс являє собою певний опис типу — набір компонентів, який повинен мати тип даних, що реалізує цей інтерфейс. Принципова відмінність від класу: ми не можемо створювати об'єкти інтерфейсу безпосередньо через конструктор. Інтерфейс — це не клас, у нього немає конструктора і немає тіла для розміщення стану:

```
C#
1  IDiagnosable d = new IDiagnosable(); // ! Помилка — інтерфейс не можна
   інстанціювати
2
3  interface IDiagnosable
4  {
5      void RunDiagnostics();
6  }
```

Натомість інтерфейс призначений для реалізації у класах і структурах. Змінна типу інтерфейсу може зберігати посилання на будь-який об'єкт, клас якого реалізує цей інтерфейс. Саме це і є ключовою перевагою: код, що працює з IDiagnosable, не знає — і не повинен знати — чи це Patient, LabSample, чи MedicalDevice. Він знає лише, що об'єкт гарантовано має метод RunDiagnostics().

Реалізація інтерфейсу у класі та структурі

Для застосування інтерфейсу після імені класу або структури через двокрапку вказується ім'я інтерфейсу — так само, як при успадкуванні. Клас зобов'язаний реалізувати **всі** методи та властивості інтерфейсу, якщо вони не мають реалізації за замовчуванням. Невиконання цієї умови — помилка компіляції.

Якщо методи та властивості інтерфейсу оголошені без модифікатора доступу, вони вважаються public. При реалізації в класі або структурі до них можна застосовувати виключно модифікатор public — зменшити рівень доступу неможливо, адже інтерфейс є публічним контрактом.

Інтерфейс може реалізовувати не лише клас, а й структура. Для структури це особливо важливо: вона не може успадковувати клас, але може реалізовувати будь-яку кількість інтерфейсів:

```
C#
 1  using System;
 2
 3  Patient p      = new Patient("Марія Коваль");
 4  LabSample s    = new LabSample("LAB-2024-099");
 5  p.RunDiagnostics();
 6  s.RunDiagnostics();
 7
 8  // реалізація інтерфейсу в класі
 9  class Patient : IDiagnosable
10  {
11      public string Name { get; }
12      public Patient(string name) => Name = name;
13
14      public void RunDiagnostics()
15          => Console.WriteLine($"Діагностика пацієнта {Name} розпочата");
16  }
```

```
17
18  // реалізація інтерфейсу в структурі
19  struct LabSample : IDiagnosable
20  {
21      public string SampleId { get; }
22      public LabSample(string id) => SampleId = id;
23
24      public void RunDiagnostics()
25          => Console.WriteLine($"Аналіз зразка {SampleId} виконано");
26  }
27
28  interface IDiagnosable
29  {
30      void RunDiagnostics();
31  }
```

Інтерфейс як параметр методу — поліморфізм

Найпрактичніше застосування інтерфейсів — передача як параметра методу. Метод, що приймає IDiagnosable, може працювати з будь-яким об'єктом, що реалізує цей інтерфейс: класом, структурою, або будь-яким майбутнім типом, який ще не написаний. На момент написання такого методу достатньо знати лише контракт — що у переданого об'єкта є метод RunDiagnostics().

Це і є поліморфізм через інтерфейс: один метод обробляє різні типи однаково, через спільний контракт:

```
C#
 1  using System;
 2
 3  Patient   p = new Patient("Іван Петренко");
 4  LabSample s = new LabSample("LAB-2024-007");
 5
 6  ProcessDiagnostics(p);
 7  ProcessDiagnostics(s);
 8
 9  void ProcessDiagnostics(IDiagnosable target)
10  {
11      Console.WriteLine("--- Початок діагностики ---");
12      target.RunDiagnostics();
13      Console.WriteLine("--- Завершено ---");
14  }
15
16  interface IDiagnosable
17  {
18      void RunDiagnostics();
19  }
20
21  class Patient : IDiagnosable
22  {
23      public string Name { get; }
24      public Patient(string name) => Name = name;
25      public void RunDiagnostics()
26          => Console.WriteLine($"Пацієнт {Name}: клінічний огляд");
27  }
```

```
28
29  struct LabSample : IDiagnosable
30  {
31      public string SampleId { get; }
32      public LabSample(string id) => SampleId = id;
33      public void RunDiagnostics()
34          => Console.WriteLine($"Зразок {SampleId}: хімічний аналіз");
35  }
```

Метод ProcessDiagnostics не має жодної залежності від Patient чи LabSample. Інтерфейс — це контракт, що певний тип обов'язково реалізує деякий функціонал. Якщо завтра з'явиться новий тип MedicalDevice : IDiagnosable, метод ProcessDiagnostics одразу зможе з ним працювати без жодних змін.

Реалізація за замовчуванням

Починаючи з C# 8.0, інтерфейси підтримують реалізацію методів і властивостей за замовчуванням. Це вирішує важливу практичну проблему: якщо опублікований інтерфейс використовують десятки класів у різних бібліотеках, додавання нового абстрактного методу зламає всі ці класи — вони перестануть компілюватись. Замість цього новий метод можна додати з реалізацією за замовчуванням: усі наявні реалізатори продовжать працювати, а ті, кому потрібна специфічна поведінка, перевизначать метод самостійно.

```
C#
 1  using System;
 2
 3  IDiagnosable p1 = new Patient("Олег Бойко");
 4  IDiagnosable p2 = new LabSample("LAB-2024-042");
 5
 6  p1.RunDiagnostics();
 7  p1.LogResult();   // використовує default-реалізацію
 8
 9  p2.RunDiagnostics();
10  p2.LogResult();   // LabSample перевизначив LogResult
11
12  interface IDiagnosable
13  {
14      void RunDiagnostics();
15      // реалізація за замовчуванням — додана пізніше, не ламає наявні класи
16      void LogResult()
17          => Console.WriteLine("[LOG] Результат діагностики збережено в
    системі");
18  }
19
20  class Patient : IDiagnosable
21  {
22      public string Name { get; }
23      public Patient(string name) => Name = name;
24      public void RunDiagnostics()
```

```
25          => Console.WriteLine($"Пацієнт {Name}: огляд завершено");
26      // LogResult не перевизначаємо — береться з інтерфейсу
27  }
28
29  class LabSample : IDiagnosable
30  {
31      public string SampleId { get; }
32      public LabSample(string id) => SampleId = id;
33      public void RunDiagnostics()
34          => Console.WriteLine($"Зразок {SampleId}: аналіз завершено");
35      // власна реалізація LogResult
36      public void LogResult()
37          => Console.WriteLine($"[LAB-LOG] Зразок {SampleId} — результат
    відправлено до лабораторії");
38  }
```

Варто зазначити важливий нюанс: якщо змінна оголошена типом конкретного класу (а не інтерфейсу), виклик default-методу через неї неможливий, якщо клас його не перевизначив. Default-реалізація доступна лише через змінну типу інтерфейсу:

```
C#
 1  using System;
 2
 3  IDiagnosable asInterface = new Patient("Тетяна Руденко");
 4  asInterface.LogResult(); // OK — default реалізація через інтерфейс
 5
 6  Patient asConcrete = new Patient("Тетяна Руденко");
 7  // asConcrete.LogResult(); // ! Помилка — клас Patient не визначив LogResult
 8
 9  interface IDiagnosable
10  {
11      void LogResult() => Console.WriteLine("[LOG] Збережено");
12  }
13
14  class Patient : IDiagnosable
15  {
16      public string Name { get; }
17      public Patient(string name) => Name = name;
18  }
```

Множинна реалізація інтерфейсів

В C# клас може успадковувати лише один базовий клас, але реалізовувати будь-яку кількість інтерфейсів. Це дозволяє описати різні ролі одного об'єкта незалежно одна від одної. Реальний клінічний об'єкт — наприклад, пацієнт — одночасно підлягає діагностиці, виставленню рахунків і отриманню сповіщень. Кожна ця роль виражається своїм інтерфейсом, і клас Patient реалізує їх усі:

```
C#
1  class Patient : IDiagnosable, IBillable, INotifiable
2  {
```

```
3      // ...
4  }
```

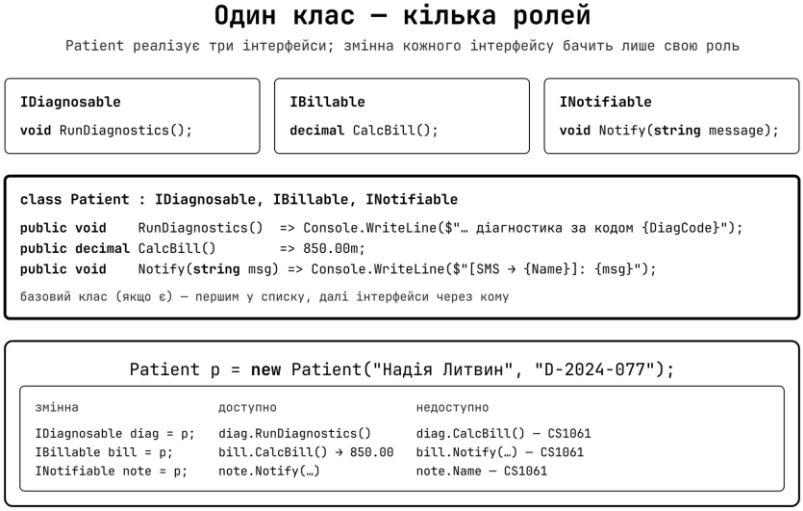
Усі реалізовані інтерфейси перераховуються через кому. Якщо клас одночасно успадковує базовий клас і реалізує інтерфейси, базовий клас вказується першим:

```
C#
1  class Patient : BaseRecord, IDiagnosable, IBillable, INotifiable
2  {
3      // ...
4  }
```

Розглянемо повний клінічний приклад із трьома інтерфейсами:

```
C#
 1  using System;
 2
 3  Patient p = new Patient("Надія Литвин", "D-2024-077");
 4
 5  // через різні інтерфейси — різні ролі одного об'єкта
 6  IDiagnosable  diag  = p;
 7  IBillable     bill  = p;
 8  INotifiable   note  = p;
 9
10  diag.RunDiagnostics();
11  Console.WriteLine($"Рахунок: {bill.CalcBill().ToString()} грн.");
12  note.Notify("Результати готові — зверніться до лікаря");
13
14  interface IDiagnosable
15  {
16      void RunDiagnostics();
17  }
18  interface IBillable
19  {
20      decimal CalcBill();
21  }
22  interface INotifiable
23  {
24      void Notify(string message);
25  }
26
27  class Patient : IDiagnosable, IBillable, INotifiable
28  {
29      public string Name       { get; }
30      public string DiagCode   { get; }
31
32      public Patient(string name, string diagCode)
33      {
34          Name     = name;
35          DiagCode = diagCode;
36      }
37
38      public void RunDiagnostics()
39          => Console.WriteLine($"Пацієнт {Name}: діагностика за кодом
   {DiagCode}");
```

```
40
41      public decimal CalcBill()
42          => 850.00m;
43
44      public void Notify(string message)
45          => Console.WriteLine($"[SMS → {Name}]: {message}");
46  }
```



Клас реалізує кілька інтерфейсів

Інтерфейси у перетвореннях типів

Все сказане щодо перетворення типів між класами стосується і інтерфейсів. Оскільки клас Patient реалізує інтерфейс IDiagnosable, змінна типу IDiagnosable може зберігати посилання на об'єкт типу Patient. Таке перетворення від конкретного класу до інтерфейсу є **розширювальним** (widening) і відбувається автоматично — воно завжди безпечне, бо будь-який Patient гарантовано є IDiagnosable:

```
C#
1  using System;
2
3  Patient p = new Patient("Василь Мороз", "D-001");
4
5  // автоматичне розширювальне перетворення — завжди ОК
6  IDiagnosable diag = p;
7  diag.RunDiagnostics();
8
```

```
// через інтерфейс доступні тільки члени інтерфейсу
// diag.Name — ! Помилка: IDiagnosable не має властивості Name

interface IDiagnosable
{
    void RunDiagnostics();
}

class Patient : IDiagnosable
{
    public string Name    { get; }
    public string DiagCode { get; }
    public Patient(string name, string code) { Name = name; DiagCode = code; }
    public void RunDiagnostics()
        => Console.WriteLine($"Пацієнт {Name} [{DiagCode}]: діагностику
розпочато");
}
```

Зворотне перетворення — від інтерфейсу до конкретного класу — є **звужувальним** (narrowing). Воно не відбувається автоматично, тому що не кожен IDiagnosable є Patient: інтерфейс можуть реалізовувати й інші класи. Для звужувального перетворення потрібне явне приведення або перевірка через is/as:

```
C#
using System;

IDiagnosable diag = new Patient("Олена Сидоренко", "D-055");

// is — перевірка типу + безпечне приведення
if (diag is Patient pat)
{
    Console.WriteLine($"Ім'я: {pat.Name}");
    Console.WriteLine($"Код: {pat.DiagCode}");
}

// as — повертає null якщо тип не відповідає (не кидає виняток)
Patient? asPat = diag as Patient;
if (asPat != null)
    Console.WriteLine($"Через as: {asPat.Name}");

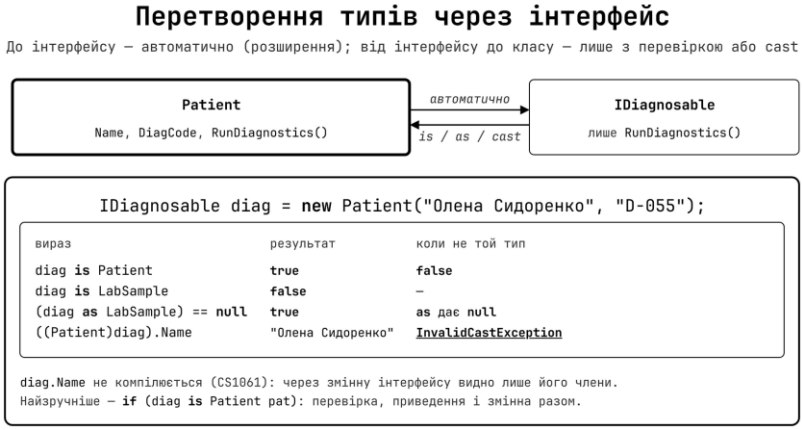
// явне приведення — кине InvalidCastException якщо тип не відповідає
Patient castPat = (Patient)diag;
Console.WriteLine($"Явне: {castPat.Name}");

interface IDiagnosable
{
    void RunDiagnostics();
}

class Patient : IDiagnosable
{
    public string Name    { get; }
    public string DiagCode { get; }
    public Patient(string name, string code) { Name = name; DiagCode = code; }
    public void RunDiagnostics()
        => Console.WriteLine($"Діагностика: {Name}");
```

```
33  }
```

На практиці перевага — за is із pattern matching: він одночасно перевіряє тип і виконує приведення в одному виразі, не кидаючи винятків. Оператор as зручний, коли потрібно лише перевірити і не виконувати тіло if одразу. Явне приведення через (T) застосовується лише тоді, коли тип відомий з абсолютною впевненістю.



Перетворення типів через інтерфейс

7.3. Явна реалізація інтерфейсів

Окрім звичайної (неявної) реалізації інтерфейсу, де метод просто оголошується у класі з модифікатором public, існує **явна реалізація** — коли перед ім'ям методу або властивості вказується назва конкретного інтерфейсу. При явній реалізації модифікатор доступу не вказується взагалі — метод автоматично є закритим для прямого доступу через змінну класу і доступний виключно через змінну типу інтерфейсу.

Явна реалізація вирішує дві головні проблеми: по-перше, вона дозволяє розмежувати методи з однаковими підписами, якщо клас реалізує кілька інтерфейсів, що мають однойменні члени. По-друге, вона потрібна, коли члени інтерфейсу мають модифікатор доступу, відмінний від public — такі члени неможливо реалізувати неявно.

Синтаксис явної реалізації виглядає так: замість public void RunDiagnostics() пишемо void IDiagnosable.RunDiagnostics() — без public, з префіксом інтерфейсу:

```C#
using System;

// доступ ТІЛЬКИ через змінну інтерфейсу
IDiagnosable p = new Patient("Марія Коваль");
p.RunDiagnostics();

class Patient : IDiagnosable
{
    public string Name { get; }
    public Patient(string name) => Name = name;

    // явна реалізація — без public, з префіксом інтерфейсу
    void IDiagnosable.RunDiagnostics()
        => Console.WriteLine($"[IDiagnosable] Діагностика пацієнта {Name}");
}

interface IDiagnosable
{
    void RunDiagnostics();
}
```

Доступ до явно реалізованих членів

Принципова особливість явної реалізації — явно реалізований член є частиною інтерфейсу класу, а не самого класу. Тому звернутися до нього безпосередньо через об'єкт класу неможливо: компілятор не знайде такого методу. Метод доступний лише через змінну типу відповідного інтерфейсу.

Це обмеження відіграє роль захисту інкапсуляції: зовнішній код може викликати RunDiagnostics лише у тому контексті, де він знає що об'єкт є IDiagnosable, — тобто свідомо, через контракт. Це запобігає випадковому виклику методу через «неправильний» контекст.

Для доступу до явно реалізованого члена через об'єкт класу потрібне явне приведення — безпечне через is або небезпечне через (IInterface):

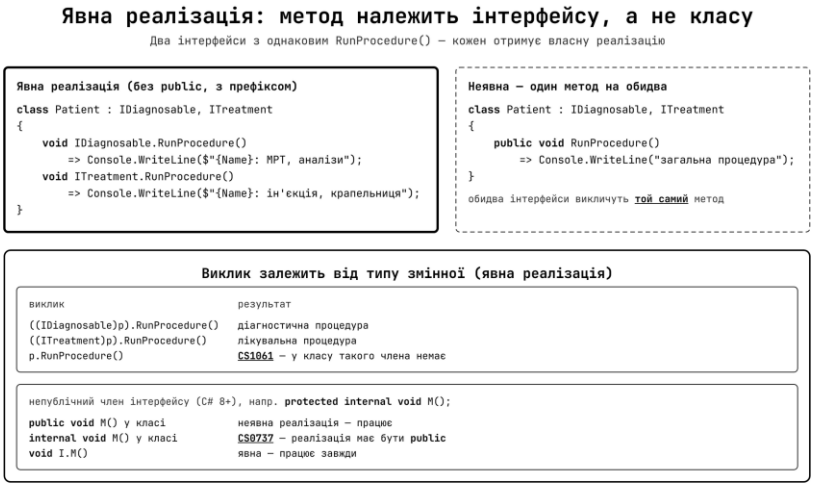
```C#
using System;

Patient p = new Patient("Іван Петренко");

// p.RunDiagnostics(); // ! Помилка — клас Patient не має публічного методу
RunDiagnostics

// варіант 1: небезпечне приведення (кине виняток якщо тип не збігається)
((IDiagnosable)p).RunDiagnostics();
```

```
 9
10  // варіант 2: безпечне приведення через is (рекомендовано)
11  if (p is IDiagnosable diag) diag.RunDiagnostics();
12
13  // варіант 3: змінна типу інтерфейсу одразу
14  IDiagnosable d = new Patient("Олена Сидоренко");
15  d.RunDiagnostics();
16
17  class Patient : IDiagnosable
18  {
19      public string Name { get; }
20      public Patient(string name) => Name = name;
21      void IDiagnosable.RunDiagnostics()
22          => Console.WriteLine($"Діагностика: {Name}");
23  }
24
25  interface IDiagnosable
26  {
27      void RunDiagnostics();
28  }
```



Явна vs неявна реалізація: диспетчеризація через тип змінної

Конфлікт імен при множинній реалізації

Найбільш поширена ситуація, коли без явної реалізації не обійтись, — клас реалізує кілька інтерфейсів, які мають методи з однаковою сигнатурою, але різним змістом. У клінічній системі такий конфлікт виникає природно: наприклад, IDiagnosable і ITreatment обидва можуть мати метод RunProcedure() — але перший запускає діагностичну процедуру, а другий — лікувальну.

Якщо клас визначає один метод RunProcedure() без явної реалізації, він стає спільним для обох інтерфейсів — і виклик через IDiagnosable, і виклик через ITreatment дасть той самий результат. Це може бути прийнятним, але частіше потрібна різна поведінка для різних ролей:

```
C#
 1  using System;
 2
 3  // без явної реалізації — один спільний метод
 4  Patient p = new Patient("Василь Мороз");
 5  ((IDiagnosable)p).RunProcedure();  // загальна процедура
 6  ((ITreatment)p).RunProcedure();    // загальна процедура — той самий метод
 7
 8  class Patient : IDiagnosable, ITreatment
 9  {
10      public string Name { get; }
11      public Patient(string name) => Name = name;
12
13      public void RunProcedure()
14          => Console.WriteLine($"{Name}: загальна процедура (однакова для обох
    інтерфейсів)");
15  }
16
17  interface IDiagnosable { void RunProcedure(); }
18  interface ITreatment   { void RunProcedure(); }
```

Щоб розмежувати реалізацію — застосовуємо явну реалізацію для кожного інтерфейсу окремо. Тоді кожен інтерфейс отримує власну незалежну логіку:

```
C#
 1  using System;
 2
 3  Patient p = new Patient("Тетяна Руденко");
 4  ((IDiagnosable)p).RunProcedure();
 5  ((ITreatment)p).RunProcedure();
 6
 7  class Patient : IDiagnosable, ITreatment
 8  {
 9      public string Name { get; }
10      public Patient(string name) => Name = name;
11
12      void IDiagnosable.RunProcedure()
13          => Console.WriteLine($"{Name}: діагностична процедура — МРТ, аналізи");
14
15      void ITreatment.RunProcedure()
16          => Console.WriteLine($"{Name}: лікувальна процедура — ін'єкція,
    крапельниця");
17  }
18
19  interface IDiagnosable { void RunProcedure(); }
20  interface ITreatment   { void RunProcedure(); }
```

Модифікатори доступу та явна реалізація

Члени інтерфейсу можуть мати різні модифікатори доступу — починаючи з C# 8.0. Якщо член інтерфейсу оголошений не як public, а, наприклад, як protected internal, у реалізації є два шляхи. Неявна реалізація можлива, але реалізуючий метод класу **має бути public**: public void UpdateStatus(string s) реалізує protected internal void UpdateStatus(string s) без помилок. Спроба зробити метод класу internal дає помилку CS0737 («член не може реалізувати член інтерфейсу, бо не є публічним»). Другий шлях — явна реалізація: вона працює завжди і не робить член видимим через тип класу. Обидва варіанти перевірено на .NET 10.

Логіка така: неявна реалізація означає, що метод одночасно є членом і класу, і інтерфейсу, тому мова вимагає для нього найширшого рівня доступу — public. Якщо ж ми не хочемо робити такий член публічним у класі (адже інтерфейс навмисно обмежив його видимість), правильний вибір — явна реалізація: метод належить виключно інтерфейсу і доступний лише через нього.

Особливо це стосується подій і властивостей з нестандартними модифікаторами — їх явна реалізація вимагає повного запису аксесорів:

```C#
using System;

IMonitorable mon = new Patient("Надія Литвин");
mon.StatusChanged += () => Console.WriteLine($"[ALERT] Статус пацієнта змінився");
mon.UpdateStatus("Критичний стан");

interface IMonitorable
{
    protected internal void UpdateStatus(string status);
    protected internal string CurrentStatus { get; }
    delegate void StatusHandler();
    protected internal event StatusHandler StatusChanged;
}

class Patient : IMonitorable
{
    string _status = "Норма";
    IMonitorable.StatusHandler? _statusChanged;

    // явна реалізація події з аксесорами
    event IMonitorable.StatusHandler IMonitorable.StatusChanged
    {
        add    => _statusChanged += value;
        remove => _statusChanged -= value;
    }

```

```
    // явна реалізація властивості
    string IMonitorable.CurrentStatus => _status;

    public string Name { get; }
    public Patient(string name) => Name = name;

    // явна реалізація методу
    void IMonitorable.UpdateStatus(string status)
    {
        _status = status;
        Console.WriteLine($"{Name}: статус оновлено → {_status}");
        _statusChanged?.Invoke();
    }
}
```

Важливо: до явно реалізованих методів, властивостей і подій можна звертатися виключно через змінну інтерфейсу, але не через змінну класу. Це означає, що Patient p = new Patient(...) не дасть доступу до UpdateStatus, CurrentStatus чи StatusChanged — лише IMonitorable m = new Patient(...) відкриє ці члени.

Реалізація інтерфейсів у базових та похідних класах

Якщо клас реалізує інтерфейс, він зобов'язаний реалізувати всі методи і властивості, що не мають реалізації за замовчуванням. Однак є виняток: клас може оголосити такі методи **абстрактними**, делегуючи відповідальність за реалізацію своїм похідним класам. У цьому випадку сам клас має бути абстрактним:

```
C#
using System;

IDiagnosable r1 = new ClinicalRecord("Олег Бойко");
IDiagnosable r2 = new LabRecord("Марина Шевченко");
r1.RunDiagnostics();
r2.RunDiagnostics();

interface IDiagnosable
{
    void RunDiagnostics();
}

// абстрактний клас реалізує інтерфейс, але відкладає реалізацію
abstract class MedicalRecord : IDiagnosable
{
    public string PatientName { get; }
    protected MedicalRecord(string name) => PatientName = name;

    public abstract void RunDiagnostics(); // нехай похідні класи вирішують
}

class ClinicalRecord : MedicalRecord
{
```

```
24      public ClinicalRecord(string name) : base(name) { }
25      public override void RunDiagnostics()
26          => Console.WriteLine($"{PatientName}: клінічний огляд — анамнез,
    симптоми");
27  }
28
29  class LabRecord : MedicalRecord
30  {
31      public LabRecord(string name) : base(name) { }
32      public override void RunDiagnostics()
33          => Console.WriteLine($"{PatientName}: лабораторна діагностика — кров,
    сеча, біопсія");
34  }
```

Інша важлива ситуація: похідний клас може «успадкувати» реалізацію інтерфейсу від базового класу, навіть якщо сам базовий клас не оголошував реалізацію інтерфейсу явно. Якщо у базовому класі є публічний метод з відповідною сигнатурою, і похідний клас оголошує реалізацію інтерфейсу — компілятор автоматично прив'яже існуючий метод:

```
C#
 1  using System;
 2
 3  IDiagnosable sr = new SpecialRecord();
 4  sr.RunDiagnostics(); // викликається метод з BaseRecord
 5
 6  interface IDiagnosable
 7  {
 8      void RunDiagnostics();
 9  }
10
11  class BaseRecord
12  {
13      // публічний метод — компілятор підхопить його як реалізацію IDiagnosable
14      public void RunDiagnostics()
15          => Console.WriteLine("BaseRecord: стандартна діагностика");
16  }
17
18  // HeroRecord успадковує метод і реалізує IDiagnosable через нього
19  class SpecialRecord : BaseRecord, IDiagnosable { }
```

Якщо клас одночасно успадковує клас і реалізує інтерфейс, ім'я базового класу завжди вказується першим, а інтерфейси — після нього:

```
C#
1  class SpecialRecord : BaseRecord, IDiagnosable, ITreatment { }
```

Зміна реалізації інтерфейсів у похідних класах

Може скластися ситуація, що базовий клас вже реалізував інтерфейс, але в похідному класі потрібно змінити цю реалізацію. Для цього

існують чотири підходи, які відрізняються за механікою диспетчеризації (тобто якій саме реалізації відповідає виклик через різні типи змінних).

Варіант 1 — перевизначення через override

Найправильніший і найпередбачуваніший варіант. Базовий клас оголошує метод як virtual (або abstract), а похідний — перевизначає через override. При цьому поліморфізм працює повністю: незалежно від того, чи тримаємо ми об'єкт у змінній базового класу, чи інтерфейсу, — завжди викликається реалізація з фактичного типу об'єкта:

```C#
using System;

BaseRecord  r1 = new SpecialRecord();
IDiagnosable r2 = new SpecialRecord();
SpecialRecord r3 = new SpecialRecord();

r1.RunDiagnostics(); // SpecialRecord — поліморфізм
r2.RunDiagnostics(); // SpecialRecord — поліморфізм
r3.RunDiagnostics(); // SpecialRecord

interface IDiagnosable { void RunDiagnostics(); }

class BaseRecord : IDiagnosable
{
    public virtual void RunDiagnostics()
        => Console.WriteLine("BaseRecord: стандартна діагностика");
}

class SpecialRecord : BaseRecord
{
    public override void RunDiagnostics()
        => Console.WriteLine("SpecialRecord: розширена діагностика");
}
```

Варіант 2 — приховування через new

Похідний клас визначає метод із new, не перевизначаючи базовий. Такий метод «приховує» базовий, але лише для тих змінних, тип яких є похідним класом. Через змінну базового класу або інтерфейсу буде видно реалізацію з **базового** класу — адже саме там інтерфейс і був реалізований:

```C#
using System;

BaseRecord   r1 = new SpecialRecord();
IDiagnosable r2 = new SpecialRecord();
SpecialRecord r3 = new SpecialRecord();

```

```
 7  r1.RunDiagnostics(); // BaseRecord — раннє зв'язування
 8  r2.RunDiagnostics(); // BaseRecord — інтерфейс реалізований у BaseRecord
 9  r3.RunDiagnostics(); // SpecialRecord — прямий тип
10
11  interface IDiagnosable { void RunDiagnostics(); }
12
13  class BaseRecord : IDiagnosable
14  {
15      public void RunDiagnostics()
16          => Console.WriteLine("BaseRecord: стандартна діагностика");
17  }
18
19  class SpecialRecord : BaseRecord
20  {
21      public new void RunDiagnostics()
22          => Console.WriteLine("SpecialRecord: спеціальна діагностика");
23  }
```

Варіант 3 — повторна реалізація інтерфейсу разом із new

Якщо похідний клас явно вказує інтерфейс у своєму оголошенні (: BaseRecord, IDiagnosable), це означає **повторну реалізацію** інтерфейсу. Тепер компілятор прив'яже інтерфейс до нового методу new у похідному класі, а не до базового. Через змінну інтерфейсу буде видно реалізацію з **похідного** класу, але через змінну базового — як і раніше базова:

```
C#
 1  using System;
 2
 3  BaseRecord   r1 = new SpecialRecord();
 4  IDiagnosable r2 = new SpecialRecord();
 5  SpecialRecord r3 = new SpecialRecord();
 6
 7  r1.RunDiagnostics(); // BaseRecord — раннє зв'язування за типом змінної
 8  r2.RunDiagnostics(); // SpecialRecord — інтерфейс тепер прив'язаний до
    SpecialRecord
 9  r3.RunDiagnostics(); // SpecialRecord
10
11  interface IDiagnosable { void RunDiagnostics(); }
12
13  class BaseRecord : IDiagnosable
14  {
15      public void RunDiagnostics()
16          => Console.WriteLine("BaseRecord: стандартна діагностика");
17  }
18
19  // повторна реалізація — IDiagnosable вказаний явно у SpecialRecord
20  class SpecialRecord : BaseRecord, IDiagnosable
21  {
22      public new void RunDiagnostics()
23          => Console.WriteLine("SpecialRecord: спеціальна діагностика");
24  }
```

Варіант 4 — явна реалізація інтерфейсу в похідному класі

Найбільш гнучкий варіант: похідний клас одночасно має new-метод (для доступу через власний тип) і явну реалізацію інтерфейсу (для доступу через змінну інтерфейсу). Це дає три різних результати залежно від типу змінної:

```C#
using System;

BaseRecord   r1 = new SpecialRecord();
IDiagnosable r2 = new SpecialRecord();
SpecialRecord r3 = new SpecialRecord();

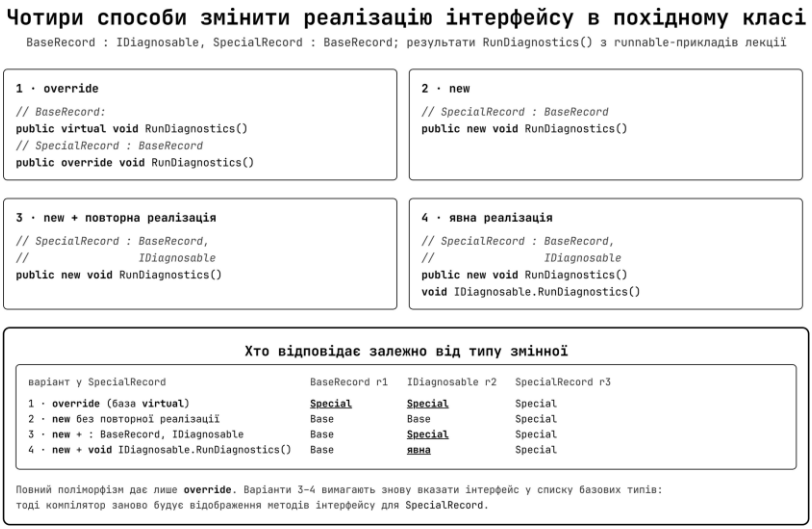
r1.RunDiagnostics(); // BaseRecord — раннє зв'язування
r2.RunDiagnostics(); // явна IDiagnosable — найвищий пріоритет
r3.RunDiagnostics(); // SpecialRecord — новий public метод

interface IDiagnosable { void RunDiagnostics(); }

class BaseRecord : IDiagnosable
{
    public void RunDiagnostics()
        => Console.WriteLine("BaseRecord: стандартна діагностика");
}

class SpecialRecord : BaseRecord, IDiagnosable
{
    public new void RunDiagnostics()
        => Console.WriteLine("SpecialRecord: спеціальна діагностика");

    // явна реалізація — виключно для змінної типу IDiagnosable
    void IDiagnosable.RunDiagnostics()
        => Console.WriteLine("SpecialRecord (явна IDiagnosable): протокол діагностики");
}
```



4 варіанти зміни реалізації інтерфейсу у похідному класі

Вибір між варіантами залежить від задачі: якщо потрібен повноцінний поліморфізм — override. Якщо потрібно лише «перекрити» метод для прямого використання — new. Якщо потрібно перезв'язати інтерфейс із новою реалізацією — повторна реалізація (new + інтерфейс у заголовку). Якщо потрібна абсолютно незалежна логіка для інтерфейсного контексту — явна реалізація.

7.4. Успадкування інтерфейсів

Інтерфейси не лише описують контракти — вони можуть розширювати один одного. Механізм **успадкування інтерфейсів** (interface inheritance) дозволяє будувати ієрархії контрактів: від загального до конкретного. Клас, який реалізує похідний інтерфейс, зобов'язаний виконати контракти всіх інтерфейсів у ланцюжку.

Це дає змогу проектувати системи поступово: спочатку визначати базові здібності об'єкта, потім нашаровувати на них більш спеціалізовані вимоги — не змінюючи вже написаний код.

Базовий синтаксис

Синтаксис успадкування інтерфейсів ідентичний синтаксису успадкування класів — через двокрапку:

```
C#
1  interface IExaminable
2  {
3      void Examine();       // провести огляд пацієнта
4  }
5
6  interface IDiagnosable : IExaminable
7  {
8      void Diagnose();      // поставити діагноз
9  }
```

Тут IDiagnosable розширює IExaminable. Це означає, що IDiagnosable включає в себе всі члени IExaminable плюс власні.

Будь-який клас чи структура, що реалізує IDiagnosable, зобов'язана реалізувати **обидва** методи — і Examine(), і Diagnose():

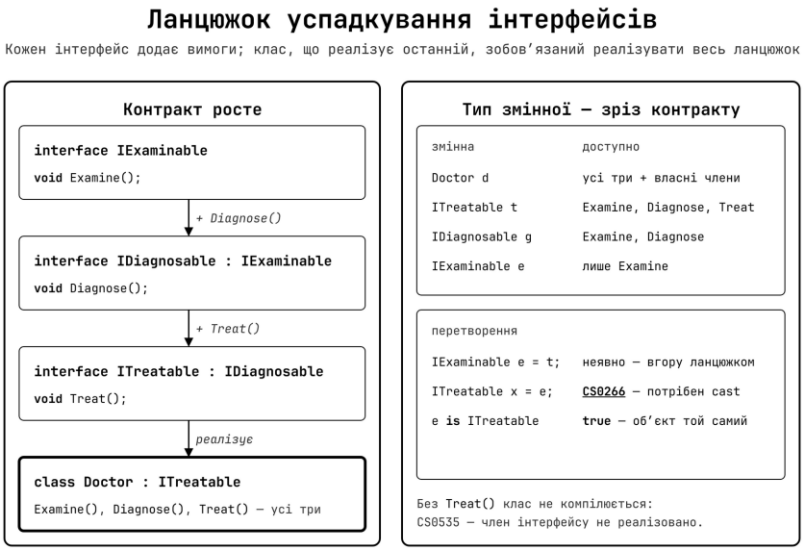
```
C#
 1  using System;
 2
 3  interface IExaminable
 4  {
 5      void Examine();
 6  }
 7
 8  interface IDiagnosable : IExaminable
 9  {
10      void Diagnose();
11  }
12
13  class GeneralPractitioner : IDiagnosable
14  {
15      public void Examine()
16      {
17          Console.WriteLine("Лікар проводить загальний огляд пацієнта.");
18      }
19
20      public void Diagnose()
21      {
22          Console.WriteLine("Лікар ставить діагноз на основі огляду.");
23      }
24  }
25
26  class Program
27  {
28      static void Main()
29      {
30          GeneralPractitioner gp = new GeneralPractitioner();
31
32          // Використання через тип IDiagnosable — бачимо обидва методи
33          IDiagnosable diagnosable = gp;
34          diagnosable.Examine();
```

```
35          diagnosable.Diagnose();
36
37          Console.WriteLine("---");
38
39          // Використання через тип IExaminable — бачимо лише Examine
40          IExaminable examinable = gp;
41          examinable.Examine();
42      }
43  }
```

Зверніть увагу: змінна типу IExaminable дає доступ лише до Examine(), хоч і містить той самий об'єкт. Тип змінної визначає, який «зріз» контракту видно у конкретному місці коду.

Ланцюжок успадкування

Інтерфейси можуть утворювати ланцюжки довільної довжини. Кожен новий інтерфейс у ланцюжку додає нові вимоги до контракту:



Ланцюжок успадкування інтерфейсів

Клас Doctor, що реалізує ITreatable, зобов'язаний надати реалізацію всіх трьох методів — Examine(), Diagnose() і Treat(). Компілятор перевіряє це під час збірки: якщо хоча б один метод не реалізовано — отримаємо помилку компіляції.

C#

```
using System;

interface IExaminable
{
    void Examine();
}

interface IDiagnosable : IExaminable
{
    void Diagnose();
}

interface ITreatable : IDiagnosable
{
    void Treat();
}

class Doctor : ITreatable
{
    private string _name;

    public Doctor(string name)
    {
        _name = name;
    }

    public void Examine()
    {
        Console.WriteLine($"Доктор {_name} проводить огляд.");
    }

    public void Diagnose()
    {
        Console.WriteLine($"Доктор {_name} ставить діагноз.");
    }

    public void Treat()
    {
        Console.WriteLine($"Доктор {_name} призначає лікування.");
    }
}

class Program
{
    static void Main()
    {
        Doctor doctor = new Doctor("Ковальчук");

        // Повний контракт через ITreatable
        ITreatable treatable = doctor;
        treatable.Examine();
        treatable.Diagnose();
        treatable.Treat();

        Console.WriteLine("---");

        // Частковий контракт через IExaminable
        IExaminable examinable = doctor;
        examinable.Examine();
```

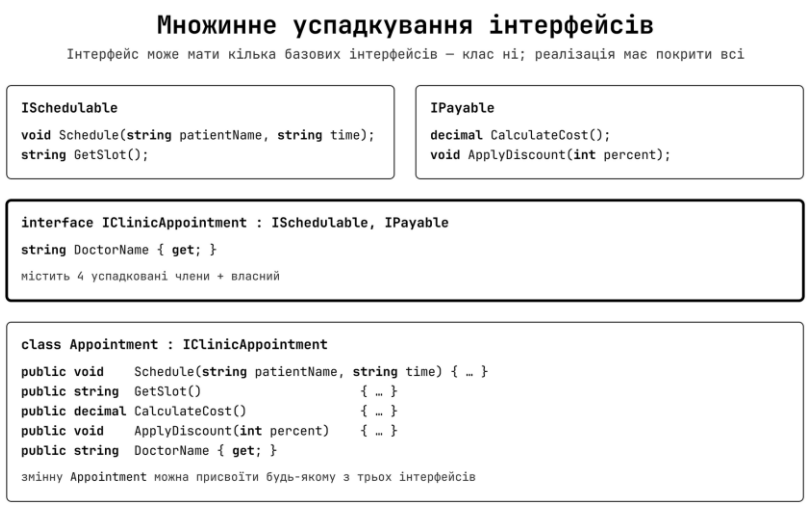
```
60          // examinable.Diagnose(); // помилка — IExaminable не знає про Diagnose
61      }
62  }
```

Множинне успадкування інтерфейсів

На відміну від класів, **інтерфейс може успадковувати відразу кілька батьківських інтерфейсів**. Це одна з ключових відмінностей інтерфейсів від класів у C# — множинне успадкування реалізації заборонене для класів, але множинне успадкування контракту — цілком допустиме для інтерфейсів.

Синтаксис: після двокрапки перераховуємо всі базові інтерфейси через кому:

```
C#
1  interface IClinicAppointment : ISchedulable, IPayable
2  {
3      // ...
4  }
```



Множинне успадкування інтерфейсів

Клас Appointment, що реалізує IClinicAppointment, зобов'язаний виконати контракти обох батьківських інтерфейсів — ISchedulable і IPayable — а також власні члени IClinicAppointment:

C#

```
using System;

interface ISchedulable
{
    void Schedule(string patientName, string time);
    string GetSlot();
}

interface IPayable
{
    decimal CalculateCost();
    void ApplyDiscount(int percent);
}

interface IClinicAppointment : ISchedulable, IPayable
{
    string DoctorName { get; }
}

class Appointment : IClinicAppointment
{
    private string _patient;
    private string _time;
    private decimal _baseCost;
    private int _discountPercent;

    public string DoctorName { get; }

    public Appointment(string doctor, decimal baseCost)
    {
        DoctorName = doctor;
        _baseCost = baseCost;
        _discountPercent = 0;
        _patient = "";
        _time = "";
    }

    public void Schedule(string patientName, string time)
    {
        _patient = patientName;
        _time = time;
        Console.WriteLine($"Записано: {_patient} до {DoctorName} на {_time}.");
    }

    public string GetSlot()
    {
        return $"{_time} ({_patient})";
    }

    public decimal CalculateCost()
    {
        decimal discount = _baseCost * _discountPercent / 100m;
        return _baseCost - discount;
    }

    public void ApplyDiscount(int percent)
    {
        _discountPercent = percent;
        Console.WriteLine($"Знижка {percent}% застосована.");
```

```
60      }
61  }
62
63  class Program
64  {
65      static void Main()
66      {
67          Appointment appt = new Appointment("Ковальчук", 800m);
68
69          appt.Schedule("Петренко І.", "14:30");
70          appt.ApplyDiscount(10);
71
72          Console.WriteLine($"Вартість прийому: {appt.CalculateCost().ToString()}
    грн");
73          Console.WriteLine($"Слот: {appt.GetSlot()}");
74          Console.WriteLine($"Лікар: {appt.DoctorName}");
75      }
76  }
```

Ключове слово new — приховування членів базового інтерфейсу

Коли базовий інтерфейс має метод із **реалізацією за замовчуванням** (default interface implementation), похідний інтерфейс може її перевизначити за допомогою ключового слова new. Це приховування (hiding), а не перевизначення у звичному розумінні.

```
C#
 1  using System;
 2
 3  interface IExaminable
 4  {
 5      // Реалізація за замовчуванням
 6      void Examine() => Console.WriteLine("Стандартний огляд: вимірювання
    температури і тиску.");
 7  }
 8
 9  interface IDiagnosable : IExaminable
10  {
11      // Приховуємо Examine з IExaminable і даємо нову реалізацію за
    замовчуванням
12      new void Examine() => Console.WriteLine("Розширений огляд: аналізи + УЗД +
    огляд.");
13  }
14
15  class Patient { }  // Порожній клас — не реалізує жодного інтерфейсу
16
17  class Specialist : IDiagnosable { }  // Реалізації за замовчуванням не
    вимагають тіла
18
19  class Program
20  {
21      static void Main()
22      {
23          Specialist s = new Specialist();
24
25          // Тип IDiagnosable — викликається прихована версія
26          IDiagnosable d = s;
```

```
        d.Examine();

        // Тип IExaminable — викликається оригінальна версія
        IExaminable e = s;
        e.Examine();
    }
}
```

Поведінка залежить від **типу змінної**, а не від фактичного об'єкта. Якщо змінна оголошена як IDiagnosable — буде викликана його версія Examine(). Якщо IExaminable — оригінальна. Це принципово відрізняється від поліморфізму через virtual/override, де завжди викликається версія фактичного об'єкта.

Клас може повністю взяти контроль і реалізувати Examine() самостійно — тоді обидва типи повертатимуть ту саму реалізацію:

```
C#
using System;

interface IExaminable
{
    void Examine() => Console.WriteLine("Стандартний огляд.");
}

interface IDiagnosable : IExaminable
{
    new void Examine() => Console.WriteLine("Розширений огляд.");
}

class Cardiologist : IDiagnosable
{
    // Власна реалізація перекриває обидві
    public void Examine() => Console.WriteLine("Кардіологічний огляд: ЕКГ + ехокардіографія.");
}

class Program
{
    static void Main()
    {
        Cardiologist c = new Cardiologist();

        IDiagnosable d = c;
        d.Examine();  // Кардіологічний огляд

        IExaminable e = c;
        e.Examine();  // Кардіологічний огляд — однакова відповідь
    }
}
```

Модифікатори sealed та abstract для інтерфейсів

Інтерфейси мають власні правила щодо модифікаторів доступу та структури:

sealed — заборонений для самого інтерфейсу. Модифікатор sealed у класах забороняє успадкування. Для інтерфейсів такого механізму не існує — будь-який інтерфейс можна успадкувати. Це відповідає природі інтерфейсу як відкритого контракту. Проте з C# 8 sealed можна поставити перед **членом** інтерфейсу з реалізацією за замовчуванням: такий член не може бути перевизначений ні похідним інтерфейсом, ні класом.

abstract — надлишковий для інтерфейсів. Інтерфейс вже за визначенням є абстрактним: він описує контракт без (обов'язкової) реалізації. Тому ключове слово abstract на рівні інтерфейсу не має сенсу і не допускається компілятором. На рівні **члена** воно з C# 8 дозволене і навіть корисне: похідний інтерфейс може знову зробити абстрактним член, для якого базовий інтерфейс мав реалізацію за замовчуванням (abstract void IExaminable.Examine();), — тоді кожен клас мусить реалізувати його сам. Обидва прийоми перевірено на .NET 10.

Правила рівня доступу

При успадкуванні інтерфейсів діє таке саме правило, що й при успадкуванні класів: **похідний інтерфейс не може бути менш обмеженим, ніж базовий**.

Припустимо, що базовий інтерфейс є public — тоді похідний може бути або public, або internal:

```
C#
 1  public interface IExaminable
 2  {
 3      void Examine();
 4  }
 5
 6  // Коректно: internal є більш суворим, ніж public
 7  internal interface IDiagnosable : IExaminable
 8  {
 9      void Diagnose();
10  }
```

Але не навпаки. Якщо базовий є internal, то похідний **не може бути public** — це порушує принцип інкапсуляції типів:

```
C#
 1  internal interface IExaminable
 2  {
 3      void Examine();
 4  }
 5
 6  // ПОМИЛКА: IRunAction може бути лише internal,
 7  // бо базовий IExaminable є internal
 8  public interface IDiagnosable : IExaminable
```

```
 9  {
10      void Diagnose();
11  }
```

Компілятор поверне помилку: неможливо оголосити public тип, що успадковує internal тип, оскільки internal тип недоступний за межами збірки.

Підсумок

Успадкування інтерфейсів — потужний інструмент для побудови гнучких та розширюваних систем. Основні правила:

- Інтерфейс успадковує члени всіх своїх базових інтерфейсів.
- Клас, що реалізує похідний інтерфейс, зобов'язаний реалізувати весь ланцюжок.
- Інтерфейс може успадковувати **кілька** базових інтерфейсів одночасно.
- sealed і abstract на рівні інтерфейсу не допускаються.
- new у похідному інтерфейсі приховує (а не перевизначає) метод базового.
- Доступ похідного інтерфейсу не може бути менш суворим, ніж базового.

7.5. Інтерфейси в узагальненнях

Інтерфейси та узагальнення (generics) тісно взаємодіють між собою у C#. З одного боку, інтерфейси виступають як **обмеження (constraints)** для узагальнених типів — вони гарантують, що параметр типу T реалізує певний контракт. З іншого боку, самі інтерфейси можуть бути **узагальненими** — тобто мати власний параметр типу T, яким їхні члени типізуються під час реалізації.

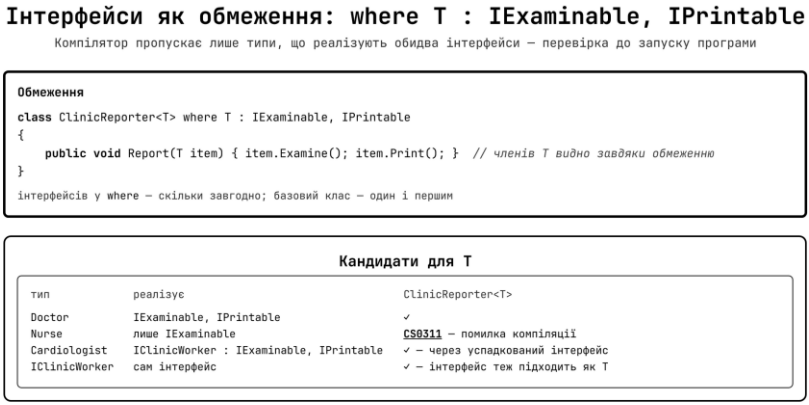
Інтерфейси як обмеження узагальнень

Коли ви оголошуєте узагальнений клас або метод, ви можете обмежити допустимі типи для параметра T за допомогою ключового слова where. Інтерфейс у ролі обмеження означає: «T повинен реалізовувати цей інтерфейс». Компілятор перевіряє це під час збірки — і якщо спробувати підставити тип, що не відповідає обмеженню, виникне помилка до запуску програми.

Ключова перевага перед обмеженнями за класом: **обмежень-інтерфейсів можна вказати скільки завгодно**, тоді як базовий клас у where може бути лише один.

```
C#
1  // можна вказати кілька інтерфейсів:
2  class ClinicReporter<T> where T : IExaminable, IPrintable { }
3
4  // але базовий клас — тільки один (і він йде першим):
5  class ClinicReporter<T> where T : BaseMedical, IExaminable, IPrintable { }
```

Розглянемо практичний приклад. У клінічній системі є два інтерфейси: IExaminable описує об'єкти, щодо яких можна провести огляд, а IPrintable — об'єкти, що підтримують виведення звіту. Клас ClinicReporter<T> приймає лише ті типи, що реалізують обидва:



Обмеження узагальнень

```
C#
 1  using System;
 2
 3  interface IExaminable
 4  {
 5      string Summary { get; }
 6      void Examine();
 7  }
 8
 9  interface IPrintable
10  {
11      void Print();
12  }
13
14  // Doctor реалізує обидва інтерфейси — допустимий тип для T
```

```
class Doctor : IExaminable, IPrintable
{
    public string Name { get; }
    public string Summary => $"Лікар: {Name}";

    public Doctor(string name) { Name = name; }

    public void Examine() =>
        Console.WriteLine($"{Name} проводить огляд пацієнта.");

    public void Print() =>
        Console.WriteLine($"[Звіт] {Summary}");
}

// Nurse реалізує лише IExaminable — НЕ допустимий для ClinicReporter<T>
class Nurse : IExaminable
{
    public string Name { get; }
    public string Summary => $"Медсестра: {Name}";

    public Nurse(string name) { Name = name; }

    public void Examine() =>
        Console.WriteLine($"{Name} вимірює температуру.");
}

// Клас приймає лише T, що реалізує обидва інтерфейси
class ClinicReporter<T> where T : IExaminable, IPrintable
{
    public void GenerateReport(T worker)
    {
        Console.WriteLine($"--- Генерація звіту ---");
        worker.Examine();
        worker.Print();
        Console.WriteLine($"Зведення: {worker.Summary}");
    }
}

class Program
{
    static void Main()
    {
        var reporter = new ClinicReporter<Doctor>();
        reporter.GenerateReport(new Doctor("Ковальчук"));

        // ClinicReporter<Nurse> — помилка компіляції:
        // Nurse не реалізує IPrintable
    }
}
```

Зверніть увагу: ClinicReporter<Nurse> не скомпілюється взагалі — компілятор повідомить про порушення обмеження ще на етапі збірки. Це набагато безпечніше, ніж перевірка типу під час виконання.

Інтерфейс як параметр типу

Обмеження where T : IExaminable, IPrintable задовольняє не лише клас, а й **інтерфейс, що успадковує обидва**. Наприклад:

```C#
using System;

interface IExaminable
{
    string Summary { get; }
    void Examine();
}

interface IPrintable
{
    void Print();
}

// Інтерфейс, що об'єднує обидва — також задовольняє обмеження
interface IClinicWorker : IExaminable, IPrintable
{
    string Department { get; }
}

class Cardiologist : IClinicWorker
{
    public string Name { get; }
    public string Department => "Кардіологія";
    public string Summary => $"Кардіолог {Name} ({Department})";

    public Cardiologist(string name) { Name = name; }

    public void Examine() =>
        Console.WriteLine($"{Name}: ЕКГ + ехокардіографія.");

    public void Print() =>
        Console.WriteLine($"[Звіт] {Summary}");
}

class ClinicReporter<T> where T : IExaminable, IPrintable
{
    public void GenerateReport(T worker)
    {
        worker.Examine();
        worker.Print();
    }
}

class Program
{
    static void Main()
    {
        // Cardiologist реалізує IClinicWorker, який включає обидва інтерфейси
        var reporter = new ClinicReporter<Cardiologist>();
        reporter.GenerateReport(new Cardiologist("Іваненко"));

        // Можна також типізувати самим інтерфейсом IClinicWorker:
        // ClinicReporter<IClinicWorker> — теж допустимо
```

```
54      }
55  }
```

Узагальнені інтерфейси

Інтерфейси, як і класи, можуть бути **узагальненими** — тобто мати власний параметр типу T. Це дозволяє визначати гнучкі контракти, де конкретні типи підставляються під час реалізації.

Типовий приклад: інтерфейс сутності зі значенням ідентифікатора, тип якого може варіюватися:

```
C#
1  interface IEntity<TId>
2  {
3      TId Id { get; }
4  }
```

Клас Patient може використовувати int як тип ідентифікатора, а InsurancePolicy — рядок (GUID або номер поліса):

```
C#
 1  using System;
 2
 3  interface IEntity<TId>
 4  {
 5      TId Id { get; }
 6      string DisplayName { get; }
 7  }
 8
 9  class Patient : IEntity<int>
10  {
11      public int Id { get; }
12      public string Name { get; }
13      public string DisplayName => $"Пацієнт #{Id}: {Name}";
14
15      public Patient(int id, string name)
16      {
17          Id = id;
18          Name = name;
19      }
20  }
21
22  class InsurancePolicy : IEntity<string>
23  {
24      public string Id { get; }           // номер поліса — рядок
25      public string HolderName { get; }
26      public string DisplayName => $"Поліс {Id} — {HolderName}";
27
28      public InsurancePolicy(string policyNumber, string holder)
29      {
30          Id = policyNumber;
31          HolderName = holder;
32      }
33  }
34
```

```
class Program
{
    static void Main()
    {
        IEntity<int> patient = new Patient(1042, "Петренко Іван");
        Console.WriteLine(patient.DisplayName);
        Console.WriteLine($"ID (int): {patient.Id.ToString()}");

        Console.WriteLine("---");

        IEntity<string> policy = new InsurancePolicy("UA-2024-00789", "Петренко
Іван");
        Console.WriteLine(policy.DisplayName);
        Console.WriteLine($"ID (string): {policy.Id}");
    }
}
```

Фіксований параметр типу

Під час реалізації узагальненого інтерфейсу клас може відразу **зафіксувати** конкретний тип замість T, не стаючи сам узагальненим:

```
C#
using System;

interface IEntity<TId>
{
    TId Id { get; }
}

// Клас фіксує TId = int — сам вже не є узагальненим
class Doctor : IEntity<int>
{
    public int Id { get; }
    public string Name { get; }
    public string Specialty { get; }

    public Doctor(int id, string name, string specialty)
    {
        Id = id;
        Name = name;
        Specialty = specialty;
    }
}

// Клас фіксує TId = string — використовує ліцензійний номер
class MedicalLicense : IEntity<string>
{
    public string Id { get; }       // наприклад "UA-LIC-2024-5571"
    public string IssuedTo { get; }

    public MedicalLicense(string licenseNumber, string issuedTo)
    {
        Id = licenseNumber;
        IssuedTo = issuedTo;
    }
}

```

```
class Program
{
    static void Main()
    {
        Doctor doc = new Doctor(101, "Ковальчук О.П.", "Хірургія");
        IEntity<int> entity = doc;
        Console.WriteLine($"Лікар ID={entity.Id.ToString()}: {doc.Name},
{doc.Specialty}");

        MedicalLicense lic = new MedicalLicense("UA-LIC-2024-5571", "Ковальчук
О.П.");
        Console.WriteLine($"Ліцензія {lic.Id} видана: {lic.IssuedTo}");
    }
}
```

Фіксація типу корисна, коли клас точно знає, який тип ідентифікатора він використовує, і не потребує гнучкості: лікарі завжди мають числовий id, а ліцензії — рядковий номер.

Підсумок

Інтерфейси та узагальнення взаємодіють у двох напрямах:

- **Інтерфейс як обмеження** (where T : IExaminable, IPrintable) — гарантує, що тип T реалізує необхідний контракт. Допускається декілька інтерфейсів одночасно. Компілятор перевіряє обмеження під час збірки.
- **Узагальнений інтерфейс** (IEntity<TId>) — дозволяє параметризувати контракт типом. Клас реалізує інтерфейс або підставляючи конкретний тип, або залишаючись узагальненим сам.

7.6. Копіювання об'єктів. Інтерфейс ICloneable

Оскільки класи є **посилальними типами** (reference types), змінна класу зберігає не сам об'єкт, а адресу у пам'яті, де цей об'єкт знаходиться. Просте присвоєння b = a не створює нового об'єкта — воно лише копіює адресу: тепер обидві змінні вказують на **той самий** об'єкт у купі.

Проблема копіювання посилальних типів

```
C#
using System;

class Patient
{
    public string Name { get; set; }
    public int Age { get; set; }

```

```
    public Patient(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

class Program
{
    static void Main()
    {
        var patient1 = new Patient("Петренко", 45);
        var patient2 = patient1;  // копіюємо ПОСИЛАННЯ, не об'єкт

        patient2.Name = "Іваненко";  // змінюємо через patient2...

        // ...але змінилось в обох, бо це один об'єкт!
        Console.WriteLine(patient1.Name);  // Іваненко (а не Петренко!)
        Console.WriteLine(patient2.Name);  // Іваненко
    }
}
```

Щоб patient2 вказував на **новий**, незалежний об'єкт з такими самими даними, застосовують **клонування**.

Інтерфейс ICloneable

Стандартна бібліотека .NET пропонує інтерфейс ICloneable з єдиним методом:

```
C#
public interface ICloneable
{
    object Clone();
}
```

Метод Clone() повертає object — незалежну копію поточного об'єкта. Конкретний спосіб копіювання клас реалізує самостійно, обираючи між поверхневим і глибоким копіюванням.

Поверхневе копіювання

Поверхневе копіювання (shallow copy) створює новий об'єкт і копіює до нього всі поля поточного об'єкта «як є». Для полів-значень (int, bool, struct тощо) це повноцінна незалежна копія. Але для полів-посилань копіюється лише **адреса** — новий об'єкт і оригінал будуть вказувати на той самий вкладений об'єкт у пам'яті.

Метод MemberwiseClone(), успадкований від object, виконує саме поверхневе копіювання:

```
C#
```

```
using System;

class Patient : ICloneable
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Patient(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public object Clone() => MemberwiseClone();
}

class Program
{
    static void Main()
    {
        var patient1 = new Patient("Петренко", 45);
        var patient2 = (Patient)patient1.Clone();

        patient2.Name = "Іваненко";
        patient2.Age = 32;

        // Тепер це незалежні об'єкти
        Console.WriteLine($"patient1: {patient1.Name},
{patient1.Age.ToString()} р.");
        Console.WriteLine($"patient2: {patient2.Name},
{patient2.Age.ToString()} р.");
    }
}
```

Для полів-значень поверхневе копіювання достатнє. Але якщо клас містить поле-посилання на інший об'єкт, виникає проблема:

```
C#
using System;

class DiagnosisRecord
{
    public string Code { get; set; }
    public string Description { get; set; }

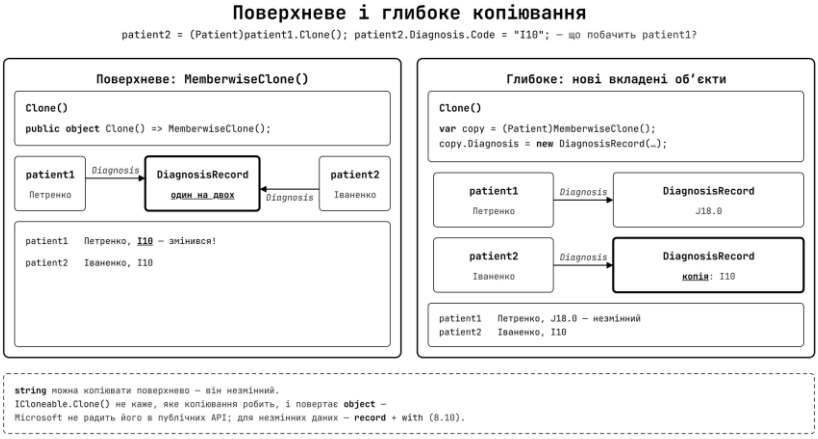
    public DiagnosisRecord(string code, string description)
    {
        Code = code;
        Description = description;
    }
}

class Patient : ICloneable
{
    public string Name { get; set; }
    public DiagnosisRecord Diagnosis { get; set; }

    public Patient(string name, DiagnosisRecord diagnosis)
```

```
21      {
22          Name = name;
23          Diagnosis = diagnosis;
24      }
25
26      // Поверхневе копіювання — копіює ПОСИЛАННЯ на Diagnosis
27      public object Clone() => MemberwiseClone();
28  }
29
30  class Program
31  {
32      static void Main()
33      {
34          var patient1 = new Patient("Петренко", new DiagnosisRecord("J18.0",
    "Пневмонія"));
35          var patient2 = (Patient)patient1.Clone();
36
37          patient2.Name = "Іваненко";
38          patient2.Diagnosis.Code = "I10";    // змінюємо діагноз у patient2...
39
40          // ...але patient1.Diagnosis також змінився!
41          Console.WriteLine($"patient1: {patient1.Name},
    {patient1.Diagnosis.Code}");
42          Console.WriteLine($"patient2: {patient2.Name},
    {patient2.Diagnosis.Code}");
43          // Обидва мають I10 — DiagnosisRecord спільний!
44      }
45  }
```

Глибоке копіювання



Поверхневе та глибоке копіювання

Глибоке копіювання (deep copy) вирішує проблему: для кожного поля-посилання вручну створюється новий об'єкт із тими самими даними. Clone() повністю будує незалежну копію всього графа об'єктів:

```C#
using System;

class DiagnosisRecord
{
    public string Code { get; set; }
    public string Description { get; set; }

    public DiagnosisRecord(string code, string description)
    {
        Code = code;
        Description = description;
    }
}

class Patient : ICloneable
{
    public string Name { get; set; }
    public DiagnosisRecord Diagnosis { get; set; }

    public Patient(string name, DiagnosisRecord diagnosis)
    {
        Name = name;
        Diagnosis = diagnosis;
    }

    // Глибоке копіювання — новий DiagnosisRecord для кожного клону
    public object Clone()
    {
        var diagnosisCopy = new DiagnosisRecord(Diagnosis.Code,
Diagnosis.Description);
        return new Patient(Name, diagnosisCopy);
    }
}

class Program
{
    static void Main()
    {
        var patient1 = new Patient("Петренко", new DiagnosisRecord("J18.0",
"Пневмонія"));
        var patient2 = (Patient)patient1.Clone();

        patient2.Name = "Іваненко";
        patient2.Diagnosis.Code = "I10";  // змінюємо лише в patient2

        Console.WriteLine($"patient1: {patient1.Name},
{patient1.Diagnosis.Code}");  // J18.0
        Console.WriteLine($"patient2: {patient2.Name},
{patient2.Diagnosis.Code}");  // I10
        // Тепер об'єкти повністю незалежні
    }
}
```

Правило вибору: якщо клас містить лише поля значимих типів або рядки (які незмінні в C#) — достатньо MemberwiseClone(). Якщо є вкладені об'єкти-посилання, що можуть змінюватися — потрібне глибоке копіювання.

Варто знати і позицію самих розробників .NET: настанови з проєктування бібліотек **не рекомендують** реалізовувати ICloneable у публічних API. Причина в тому, що контракт інтерфейсу не каже, яке саме копіювання виконує Clone() — поверхневе чи глибоке, — і код, що його викликає, не може на це покластися; до того ж метод повертає object, тож потрібне приведення типу. На практиці краще оголосити власний метод з очевидною назвою і типом результату (Patient DeepCopy()), а для незмінних даних — використати record з оператором with (розділ 8.10), де копіювання з частковою зміною вбудоване в мову.

Сортування об'єктів. Інтерфейс IComparable

Вбудовані типи C# — числа, рядки — вміють порівнювати себе між собою: Array.Sort на масиві int[] спрацює одразу. Але для власних класів компілятор не знає, який порядок вважати «правильним». Для цього існує інтерфейс IComparable<T>:

```
C#
1  public interface IComparable<T>
2  {
3      int CompareTo(T? other);
4  }
```

Метод CompareTo повертає:

— **від'ємне число** — поточний об'єкт стоїть **до** other у відсортованому порядку

— **нуль** — об'єкти рівні за критерієм порівняння

— **додатне число** — поточний об'єкт стоїть **після** other

Реалізуємо сортування пацієнтів за прізвищем:

```
C#
 1  using System;
 2
 3  class Patient : IComparable<Patient>
 4  {
 5      public string LastName { get; }
 6      public string FirstName { get; }
 7      public int Age { get; }
 8
 9      public Patient(string lastName, string firstName, int age)
10      {
11          LastName = lastName;
```

```
        FirstName = firstName;
        Age = age;
    }

    // Порівнюємо за прізвищем (алфавітний порядок)
    public int CompareTo(Patient? other)
    {
        if (other is null)
            throw new ArgumentException("Некоректне значення параметра");
        return LastName.CompareTo(other.LastName);
    }

    public override string ToString() =>
        $"{LastName} {FirstName}, {Age.ToString()} р.";
}

class Program
{
    static void Main()
    {
        Patient[] patients =
        {
            new Patient("Шевченко", "Оксана", 34),
            new Patient("Іваненко", "Петро",  52),
            new Patient("Бойко",    "Марія",  28),
            new Patient("Ковальчук","Андрій", 41),
        };

        Array.Sort(patients);  // використовує CompareTo

        Console.WriteLine("Пацієнти за алфавітом:");
        foreach (Patient p in patients)
            Console.WriteLine($"  {p.ToString()}");
    }
}
```

Застосування компаратора IComparer

Іноді потрібно сортувати за різними критеріями в різних ситуаціях — наприклад, за прізвищем в одному місці програми і за віком в іншому. Один клас не може мати два різні CompareTo. Для цього існує окремий інтерфейс IComparer<T>:

```
C#
public interface IComparer<in T>
{
    int Compare(T? x, T? y);
}
```

Компаратор — це окремий клас, що описує **один конкретний спосіб порівняння**. Він передається як другий аргумент у Array.Sort:

```
C#
using System;
using System.Collections.Generic;
```

```
 3
 4  Patient[] patients =
 5  {
 6      new Patient("Шевченко", 34),
 7      new Patient("Іваненко", 52),
 8      new Patient("Бойко",    28),
 9      new Patient("Ковальчук",41),
10  };
11
12  Array.Sort(patients, new PatientAgeComparer());
13  Console.WriteLine("За віком:");
14  foreach (Patient p in patients)
15      Console.WriteLine($"  {p.ToString()}");
16
17  Console.WriteLine();
18
19  Array.Sort(patients, new PatientNameComparer());
20  Console.WriteLine("За прізвищем:");
21  foreach (Patient p in patients)
22      Console.WriteLine($"  {p.ToString()}");
23
24  class Patient
25  {
26      public string LastName { get; }
27      public int Age { get; }
28
29      public Patient(string lastName, int age)
30      {
31          LastName = lastName;
32          Age = age;
33      }
34
35      public override string ToString() => $"{LastName}, {Age.ToString()} p.";
36  }
37
38  // Компаратор за віком (від молодшого до старшого)
39  class PatientAgeComparer : IComparer<Patient>
40  {
41      public int Compare(Patient? x, Patient? y)
42      {
43          if (x is null || y is null)
44              throw new ArgumentException("Некоректне значення параметра");
45          return x.Age.CompareTo(y.Age);
46      }
47  }
48
49  // Компаратор за прізвищем
50  class PatientNameComparer : IComparer<Patient>
51  {
52      public int Compare(Patient? x, Patient? y)
53      {
54          if (x is null || y is null)
55              throw new ArgumentException("Некоректне значення параметра");
56          return x.LastName.CompareTo(y.LastName);
57      }
58  }
```

Правила компаратора мають **вищий пріоритет** над CompareTo: якщо клас реалізує IComparable<T> і одночасно передається IComparer<T> — перемагає компаратор.

Підсумок

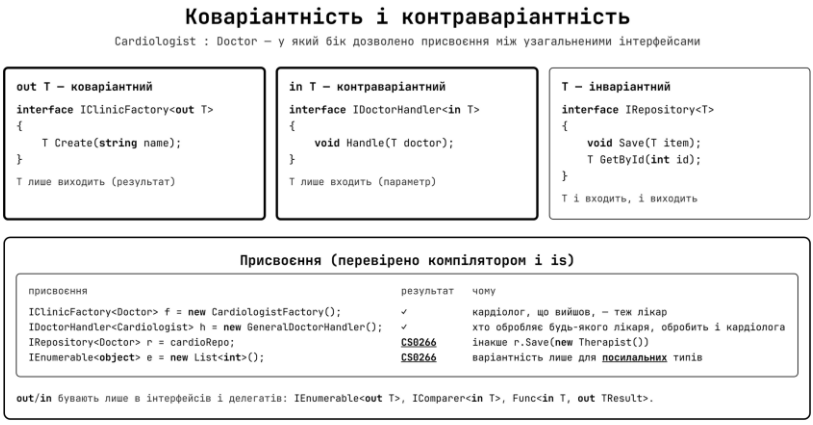
Інтерфейс	Призначення	Де реалізується
ICloneable	Копіювання об'єкта	У класі, що копіюється
IComparable<T>	Вбудований порядок сортування	У класі, що порівнюється
IComparer<T>	Зовнішній, змінний критерій сортування	В окремому класі-компараторі

Поверхневе копіювання (MemberwiseClone) підходить для об'єктів без вкладених посилальних типів. Для складних об'єктів з вкладеними посиланнями потрібне глибоке копіювання через ручне створення копій кожного вкладеного об'єкта.

7.7. Коваріантність та контраваріантність узагальнених інтерфейсів

Уявіть ієрархію: Cardiologist успадковує Doctor. Це означає, що Cardiologist **є** Doctor: де очікується Doctor, можна підставити Cardiologist. Але чи діє та ж логіка для узагальнених інтерфейсів? Чи можна підставити IClinicFactory<Cardiologist> туди, де очікується IClinicFactory<Doctor>?

Відповідь залежить від того, чи є інтерфейс **коваріантним**, **контраваріантним** або **інваріантним**. Ці три поняття описують, як напрям допустимого присвоєння між параметризованими типами співвідноситься з напрямом ієрархії між типами-аргументами.



Коваріантність та контраваріантність

Три форми варіантності:

— **Коваріантність** — присвоєння йде **в той самий бік**, що й ієрархія класів. Якщо Cardiologist : Doctor, то IFactory<Cardiologist> можна присвоїти IFactory<Doctor>.

— **Контраваріантність** — присвоєння йде **в зворотний бік**. IHandler<Doctor> можна присвоїти IHandler<Cardiologist>.

— **Інваріантність** — присвоєння неможливе в жодному напрямі (поведінка за замовчуванням).

Інваріантність — поведінка за замовчуванням

Без спеціальних ключових слів усі узагальнені інтерфейси є **інваріантними**: IRepository<Doctor> і IRepository<Cardiologist> — абсолютно несумісні типи, навіть якщо Cardiologist : Doctor. Жодне присвоєння в жодному напрямі не дозволяється.

```C#
interface IRepository<T>
{
    T GetById(int id);
    void Save(T item);
}

// IRepository<Cardiologist> НЕ можна присвоїти IRepository<Doctor> — помилка
// IRepository<Doctor> НЕ можна присвоїти IRepository<Cardiologist> — помилка
```

Це може здаватися надто суворим, але є підставовою безпекою: якби IRepository<Cardiologist> можна було підставити як IRepository<Doctor>, то через змінну типу IRepository<Doctor> хтось міг би викликати Save(new Therapist()) — а Therapist не є Cardiologist. Тому без явного дозволу компілятор забороняє такі присвоєння.

Коваріантні інтерфейси (out)

Коваріантний інтерфейс оголошується за допомогою ключового слова **out** перед параметром типу:

```
C#
1  interface IClinicFactory<out T>
2  {
3      T Create();
4  }
```

Ключове слово out означає: «параметр T використовується лише як тип **значення, що повертається**». Тобто об'єкт типу T **виходить** з інтерфейсу (звідси out). У такому разі компілятор може гарантувати безпеку: якщо Cardiologist : Doctor, то всі кардіологи, яких повертає IClinicFactory<Cardiologist>, є і лікарями, тому IClinicFactory<Cardiologist> безпечно замінює IClinicFactory<Doctor>.

```
C#
 1  using System;
 2
 3  class Doctor
 4  {
 5      public string Name { get; }
 6      public string Specialty { get; }
 7
 8      public Doctor(string name, string specialty)
 9      {
10          Name = name;
11          Specialty = specialty;
12      }
13
14      public virtual void Introduce() =>
15          Console.WriteLine($"Лікар {Name}, спеціальність: {Specialty}");
16  }
17
18  class Cardiologist : Doctor
19  {
20      public Cardiologist(string name) : base(name, "Кардіологія") { }
21
22      public override void Introduce() =>
23          Console.WriteLine($"Кардіолог {Name}");
24  }
25
26  interface IClinicFactory<out T>
27  {
```

```
    T Create(string name);
}

class CardiologistFactory : IClinicFactory<Cardiologist>
{
    public Cardiologist Create(string name) => new Cardiologist(name);
}

class Program
{
    static void Main()
    {
        // Пряме використання
        IClinicFactory<Cardiologist> cardFactory = new CardiologistFactory();
        Cardiologist card = cardFactory.Create("Іваненко");
        card.Introduce();

        // Коваріантність: IClinicFactory<Cardiologist> → IClinicFactory<Doctor>
        IClinicFactory<Doctor> docFactory = cardFactory;
        Doctor doc = docFactory.Create("Петренко");
        doc.Introduce();

        Console.WriteLine();

        // Через проміжну змінну
        IClinicFactory<Cardiologist> src = new CardiologistFactory();
        IClinicFactory<Doctor> dst = src;   // [OK] коваріантність дозволяє
        Doctor created = dst.Create("Ковальчук");
        created.Introduce();
    }
}
```

Обмеження коваріантного інтерфейсу: параметр T з out можна використовувати **лише як тип, що повертається** — як тип вхідного аргументу методу він заборонений. Це гарантує, що через коваріантний інтерфейс не можна «записати» об'єкт неправильного типу.

Контраваріантні інтерфейси (in)

Контраваріантний інтерфейс оголошується за допомогою ключового слова **in**:

```
C#
interface IDoctorHandler<in T>
{
    void Handle(T doctor);
}
```

Ключове слово in означає: «параметр T використовується лише як тип **вхідного аргументу**». Тобто об'єкт типу T **входить** у інтерфейс. Якщо IDoctorHandler<Doctor> вміє обробляти будь-якого лікаря, то він точно вміє обробляти і кардіолога (бо кардіолог є лікарем). Тому

IDoctorHandler<Doctor> можна присвоїти IDoctorHandler<Cardiologist> — **зворотний** напрям відносно ієрархії.

```C#
using System;

class Doctor
{
    public string Name { get; }
    public Doctor(string name) { Name = name; }
}

class Cardiologist : Doctor
{
    public Cardiologist(string name) : base(name) { }
}

interface IDoctorHandler<in T>
{
    void Handle(T doctor);
}

class GeneralDoctorHandler : IDoctorHandler<Doctor>
{
    public void Handle(Doctor doctor) =>
        Console.WriteLine($"Обробляємо лікаря: {doctor.Name}");
}

class Program
{
    static void Main()
    {
        // Пряме використання
        IDoctorHandler<Doctor> generalHandler = new GeneralDoctorHandler();
        generalHandler.Handle(new Doctor("Ковальчук"));

        // Контраваріантність: IDoctorHandler<Doctor> → IDoctorHandler<Cardiologist>
        IDoctorHandler<Cardiologist> cardHandler = generalHandler;
        cardHandler.Handle(new Cardiologist("Іваненко"));

        Console.WriteLine();

        // Через проміжну змінну — теж працює
        IDoctorHandler<Doctor> src = new GeneralDoctorHandler();
        IDoctorHandler<Cardiologist> dst = src;  // [OK] контраваріантність
        dst.Handle(new Cardiologist("Петренко"));
    }
}
```

Обмеження контраваріантного інтерфейсу: параметр T з in можна використовувати **лише як тип вхідного аргументу** — як тип значення, що повертається, він заборонений. Це гарантує, що через контраваріантний інтерфейс не можна «прочитати» об'єкт неправильного типу.

Поєднання коваріантності та контраваріантності

Один і той самий інтерфейс може мати кілька параметрів типу, одні з яких є out, а інші — in. Наприклад, інтерфейс клінічного месенджера, що **приймає** повідомлення одного типу і **повертає** відповідь іншого:

```C#
using System;

class Doctor
{
    public string Name { get; }
    public Doctor(string name) { Name = name; }
}

class Cardiologist : Doctor
{
    public Cardiologist(string name) : base(name) { }
}

class Report
{
    public string Text { get; }
    public Report(string text) { Text = text; }
}

class DetailedReport : Report
{
    public DetailedReport(string text) : base(text) { }
}

// in T — приймає повідомлення типу T
// out K — повертає відповідь типу K
interface IClinicMessenger<in T, out K>
{
    void Send(T request);
    K Receive();
}

class ClinicMessengerImpl : IClinicMessenger<Doctor, DetailedReport>
{
    private string _lastSender = "";

    public void Send(Doctor request)
    {
        _lastSender = request.Name;
        Console.WriteLine($"Повідомлення від {request.Name} отримано.");
    }

    public DetailedReport Receive() =>
        new DetailedReport($"Детальний звіт для {_lastSender}");
}

class Program
{
    static void Main()
    {
        ClinicMessengerImpl impl = new ClinicMessengerImpl();
```

```
52
53          // Завдяки in T: IClinicMessenger<Doctor,...> → IClinicMessenger<Cardiologist,...>
54          // Завдяки out K: IClinicMessenger<...,DetailedReport> → IClinicMessenger<...,Report>
55          IClinicMessenger<Cardiologist, Report> messenger = impl;
56
57          messenger.Send(new Cardiologist("Іваненко"));
58          Report report = messenger.Receive();
59          Console.WriteLine(report.Text);
60      }
61  }
```

Завдяки обом модифікаторам об'єкт ClinicMessengerImpl (тип IClinicMessenger<Doctor, DetailedReport>) можна безпечно використовувати як IClinicMessenger<Cardiologist, Report> — контраваріантність по T (Doctor → Cardiologist, зворотний напрям) і коваріантність по K (DetailedReport → Report, прямий напрям).

Правила і обмеження

Модифікатор	Назва	Дозволено для T	Напрям присвоєння
out T	Коваріантний	Лише тип результату методу	IFoo<Derived> → IFoo<Base>
in T	Контраваріантний	Лише тип аргументу методу	IFoo<Base> → IFoo<Derived>
(без модифікатора)	Інваріантний	Будь-яке використання	Присвоєння заборонено

Ці модифікатори доступні лише для **інтерфейсів** і **делегатів** — не для класів і структур. Друге обмеження стосується аргументів типу: варіантність працює лише для **посилальних типів**. IEnumerable<string> можна присвоїти IEnumerable<object>, а IEnumerable<int> — ні (CS0266), бо перетворення int → object вимагає упаковки, тобто зміни представлення значення в пам'яті, а варіантне присвоєння лише переінтерпретує те саме посилання (перевірено на .NET 10). Саме тому в стандартній бібліотеці .NET такі інтерфейси, як IEnumerable<out T> (коваріантний) і IComparer<in T> (контраваріантний), ведуть себе інтуїтивно: послідовність кардіологів є послідовністю лікарів, а компаратор лікарів може порівнювати кардіологів.

Питання для самоконтролю — Розділ 7. Інтерфейси

7.1. Визначення інтерфейсів

1. Що таке інтерфейс у C#? Чим він принципово відрізняється від класу і від абстрактного класу? Що дозволяє, а що забороняє мати інтерфейс серед своїх членів?
2. Чому за угодою C# назви інтерфейсів починаються з великої літери I? Чи є це вимогою компілятора? Наведіть 3–4 приклади стандартних інтерфейсів .NET, що ілюструють цю конвенцію.
3. Поясніть різницю у рівнях доступу: чому члени інтерфейсу за замовчуванням public, а не private, як у класах? Що відбудеться, якщо клас спробує реалізувати публічний метод інтерфейсу зі зниженим рівнем доступу (наприклад, protected)?
4. Що таке «реалізація за замовчуванням» (default interface implementation), доступна з C# 8.0? Яку практичну проблему вона вирішує? Чому без неї додавання нового методу до вже опублікованого інтерфейсу «ламало» б всіх реалізаторів?
5. Знайдіть помилку у цьому коді та поясніть її:

```C#
interface ILogger
{
    private int _count = 0; // поле екземпляра
    void Log(string message);
}
```

6. Що таке антипатерн «Бог-інтерфейс»? Наведіть приклад такого інтерфейсу в контексті медичної системи. Які конкретні проблеми він спричиняє при реалізації різними класами? Як правильно його розбити на вузькі інтерфейси?
7. Порівняйте інтерфейс і абстрактний клас за такими критеріями: стан (поля), конструктор, кількість реалізованих/успадкованих типів, default-реалізація, спадкування. В якому випадку краще обирати інтерфейс, а в якому — абстрактний клас?
8. Оголошено статичний метод безпосередньо в інтерфейсі. Як до нього звернутися? Чи успадковується він класами-реалізаторами? Поясніть різницю між статичним методом інтерфейсу і звичайним статичним методом класу-утиліти.

7.2. Застосування інтерфейсів

9. Чому не можна створити екземпляр інтерфейсу напряму через new? Що натомість зберігає змінна типу інтерфейсу? Покажіть, як змінна IDiagnosable d може «вказувати» на об'єкти різних класів.
10. Поясніть, що таке поліморфізм через інтерфейс. Напишіть метод RunAll(IDiagnosable[] items), який послідовно викликає RunDiagnostics() для кожного елемента. Покажіть, що цей метод однаково добре працює з об'єктами різних класів, що реалізують IDiagnosable.
11. Що виведе наступний код? Поясніть, чому виклик asConcrete.LogResult() може спричинити помилку компіляції:

```
C#
 1  interface ILogger
 2  {
 3      void LogResult() => Console.WriteLine("[LOG] Збережено");
 4  }
 5  class Patient : ILogger { }
 6
 7  ILogger asInterface = new Patient();
 8  asInterface.LogResult(); // рядок A
 9
10  Patient asConcrete = new Patient();
11  asConcrete.LogResult(); // рядок B
```

12. Опишіть різницю між розширювальним (widening) і звужувальним (narrowing) перетворенням типу через інтерфейс. Чому одне відбувається автоматично, а інше потребує явного приведення? Коли при явному приведенні виникне InvalidCastException?
13. Порівняйте три способи звужувального приведення: явний cast (T), оператор as, і оператор is з pattern matching. Коли кожен з них доречний? Який з них рекомендується як основний і чому?
14. Клас реалізує кілька інтерфейсів. Напишіть код, де один об'єкт класу Patient зберігається в трьох різних змінних типів інтерфейсів: IDiagnosable, IBillable, INotifiable. Покажіть, які методи доступні через кожну з них.
15. Структура також може реалізовувати інтерфейси. Чому це особливо важливо для структур? Що відбувається зі значенням структури при присвоєнні до змінної інтерфейсного типу (boxing)? Які наслідки це має для продуктивності?
16. Проаналізуйте код і поясніть його поведінку:

```C#
interface IRunnable { void Run(); }
class Base : IRunnable { public void Run() => Console.WriteLine("Base"); }
class Child : Base { public new void Run() => Console.WriteLine("Child"); }

IRunnable r = new Child();
r.Run(); // що виведе?
```

7.3. Явна реалізація інтерфейсів

17. Що таке явна реалізація інтерфейсу? Чим вона синтаксично відрізняється від звичайної (неявної) реалізації? Який рівень доступу має явно реалізований метод і чому?
18. Через яку змінну доступний явно реалізований метод — через змінну класу чи через змінну інтерфейсу? Покажіть три різних способи викликати явно реалізований метод RunDiagnostics() для об'єкта Patient.
19. Назвіть дві головні ситуації, коли явна реалізація є необхідною, а не лише зручною. Наведіть код, що ілюструє кожну із ситуацій.
20. Два інтерфейси IDiagnosable і ITreatment мають метод void RunProcedure(). Клас реалізує обидва. Що відбудеться, якщо метод реалізовано без явної реалізації (один спільний метод)? Як явна реалізація вирішує конфлікт? Напишіть обидва варіанти.
21. Проаналізуйте чотири варіанти зміни реалізації інтерфейсу у похідному класі: override, new, new + повторна реалізація, явна реалізація + new. Що виведе кожен варіант для змінних типів BaseClass, IDiagnosable і DerivedClass?
22. Є абстрактний клас MedicalRecord, що реалізує IDiagnosable, але оголошує RunDiagnostics() абстрактним. Поясніть механізм: чи виконує MedicalRecord контракт інтерфейсу? Що відбувається при спробі створити екземпляр MedicalRecord напряму?
23. Спроєктуйте клас SmartDevice, який реалізує два інтерфейси: IReadable з методом string Read() і IWriteable з методом string Write(). Реалізуйте Read() явно (доступно тільки через IReadable), а Write() — неявно (доступно також напряму). Покажіть різницю у доступі.
24. Чому не можна написати public void IDiagnosable.RunDiagnostics()? Що відбудеться, якщо поставити

public перед явною реалізацією? Поясніть логіку цього обмеження.

7.4. Успадкування інтерфейсів

25. Що таке успадкування інтерфейсів? Яку вимогу ставить компілятор до класу, що реалізує похідний інтерфейс, стосовно методів батьківського інтерфейсу?
26. Є ланцюжок: IExaminable → IDiagnosable → ITreatable. Клас Doctor реалізує ITreatable. Скільки методів зобов'язаний реалізувати Doctor? Які «зрізи» контракту доступні через змінні різних типів інтерфейсів?
27. Чому інтерфейс може успадковувати кілька базових інтерфейсів одночасно, а клас — лише один базовий клас? Яка концептуальна різниця між «успадкуванням реалізації» і «успадкуванням контракту»?
28. Що відбувається, коли в похідному інтерфейсі оголошується new void Examine(), а базовий вже має void Examine() з реалізацією за замовчуванням? Як поведінка залежить від типу змінної (IDerived чи IBase)? Чим це відрізняється від override?
29. Проаналізуйте наступне оголошення і поясніть, чи є воно коректним та чому:

```
C#
1  internal interface IBase { void Act(); }
2  public interface IDerived : IBase { void ExtendedAct(); }
```

30. Чому не можна застосувати sealed або abstract до інтерфейсу? Поясніть, чому ці модифікатори суперечать природі інтерфейсів.
31. Спроєктуйте систему інтерфейсів для медичної клініки: ISchedulable (запис пацієнта), IPayable (оплата), IClinicAppointment : ISchedulable, IPayable (прийом). Реалізуйте клас ConsultationAppointment і покажіть, як через різні змінні типу інтерфейсу отримати доступ до різних підмножин методів.
32. Два інтерфейси — ISchedulable і IPayable — обидва оголошують string GetSummary(). Клас реалізує обидва. Як правильно надати різну реалізацію для кожного інтерфейсу? Напишіть повний приклад.

7.5. Інтерфейси в узагальненнях

33. У чому полягають два різні способи взаємодії інтерфейсів та узагальнень у C#? Поясніть різницю між «інтерфейс як обмеження» і «узагальнений інтерфейс».
34. Оголошено class Repository<T> where T : IExaminable, IPrintable. Що саме гарантує компілятор всередині цього класу щодо об'єкта типу T? Чому це безпечніше, ніж перевірка через is у runtime?
35. Скільки інтерфейсних обмежень можна вказати у where? Чи можна поєднати обмеження-клас і обмеження-інтерфейси в одному where? Запишіть синтаксис для узагальненого класу, що вимагає базовий клас BaseMedical і одночасно реалізацію IExaminable, IPrintable.
36. Оголошено interface IEntity<TId> { TId Id { get; } }. Як клас Patient може реалізувати цей інтерфейс з int як типом ID, а клас InsurancePolicy — з string? Покажіть реалізацію обох класів.
37. Клас Cardiologist реалізує IClinicWorker, де IClinicWorker : IExaminable, IPrintable. Чи задовольняє Cardiologist обмеження where T : IExaminable, IPrintable? Поясніть логіку: чому «успадкована реалізація» інтерфейсу рахується.
38. Напишіть узагальнений метод PrintAll<T>(T[] items) where T : IPrintable, який викликає Print() для кожного елемента масиву. Покажіть, що цей метод прийме масив будь-якого класу, що реалізує IPrintable, але відмовить у компіляції для класу без цього інтерфейсу.
39. Порівняйте: interface IRepo<T> (без параметра типу) і interface IRepo<TId> (узагальнений). В яких сценаріях узагальнений інтерфейс є явно кращим рішенням? Назвіть принаймні два конкретних приклади.
40. Спроєктуйте узагальнений клас MedicalRegistry<T> where T : IEntity<int>, IExaminable, що зберігає колекцію медичних сутностей і надає метод T FindById(int id). Поясніть, чому подвійне обмеження тут доречне.

7.6. Копіювання об'єктів. Інтерфейс ICloneable

41. Що відбувається при простому присвоєнні b = a для об'єкта класу (reference type)? Чому зміна b.Name після такого

присвоєння також змінює a.Name? Чим ця поведінка відрізняється від присвоєння значимих типів (struct)?

42. Що таке інтерфейс ICloneable? Яку сигнатуру має його єдиний метод? Чому тип повернення object, а не конкретний тип? Як правильно викликати Clone() і отримати об'єкт конкретного типу?
43. Поясніть різницю між поверхневим (shallow) і глибоким (deep) копіюванням. Що робить MemberwiseClone()? Коли поверхневого копіювання достатньо, а коли потрібне глибоке?
44. Знайдіть помилку у наступному прикладі та поясніть, що відбуватиметься при виконанні коду:

```C#
class Patient : ICloneable
{
    public string Name { get; set; }
    public DiagnosisRecord Diagnosis { get; set; }
    public object Clone() => MemberwiseClone();
}
var p1 = new Patient { Name = "Іван", Diagnosis = new DiagnosisRecord("J18") };
var p2 = (Patient)p1.Clone();
p2.Diagnosis.Code = "I10"; // змінюємо через p2
Console.WriteLine(p1.Diagnosis.Code); // що виведе?
```

45. Реалізуйте глибоке копіювання для класу Patient, що містить поле DiagnosisRecord Diagnosis. Покажіть, що після клонування зміна Diagnosis.Code в копії не впливає на оригінал.
46. Що таке інтерфейс IComparable<T>? Який контракт він накладає на клас? Які значення повинен повертати метод CompareTo і що кожне з них означає?
47. Порівняйте IComparable<T> і IComparer<T>: у чому їх різниця? Коли кожен доречний? Чому один клас не може мати два різні CompareTo для різних критеріїв сортування — і як IComparer<T> вирішує цю проблему?
48. Напишіть клас Appointment (прийом у лікаря) з полями DateTime Date і string DoctorName. Реалізуйте IComparable<Appointment> для сортування за датою. Також напишіть окремий компаратор AppointmentByDoctorComparer : IComparer<Appointment> для сортування за ім'ям лікаря.

7.7. Коваріантність та контраваріантність узагальнених інтерфейсів

49. Що таке інваріантність узагальнених інтерфейсів? Чому за замовчуванням IRepository<Cardiologist> і IRepository<Doctor> несумісні навіть при Cardiologist : Doctor? Яку небезпеку усуває ця суворість?
50. Поясніть різницю між трьома формами варіантності: коваріантністю (out), контраваріантністю (in) та інваріантністю. У якому напрямі дозволяється присвоєння в кожному випадку?
51. Оголошено interface IFactory<out T> { T Create(); }. Що означає out? Чому IFactory<Cardiologist> можна безпечно присвоїти змінній типу IFactory<Doctor>? Яке обмеження накладає out на використання T у методах інтерфейсу?
52. Оголошено interface IHandler<in T> { void Handle(T item); }. Що означає in? Чому IHandler<Doctor> можна присвоїти змінній IHandler<Cardiologist>? Це інтуїтивно здається «зворотнім» — поясніть логіку цього явища.
53. Що виведе наступний код? Поясніть, яка форма варіантності тут задіяна і чому присвоєння у рядку 3 є коректним:

```
C#
1  interface IProducer<out T> { T Produce(); }
2  class AnimalProducer : IProducer<Animal> { public Animal Produce() => new
   Animal(); }
3  IProducer<object> p = new AnimalProducer(); // рядок 3 — Animal : object
4  object obj = p.Produce();
```

54. Поясніть, чому out не можна використовувати як тип параметра методу, а in — як тип результату методу. Яку безпеку гарантують ці обмеження?
55. Стандартний інтерфейс IEnumerable<out T> є коваріантним, а IComparer<in T> — контраваріантним. Поясніть, чому такі рішення прийняті для кожного з них: яким чином T використовується в кожному інтерфейсі?
56. Спроєктуйте інтерфейс IClinicMessenger<in TRequest, out TResponse> для системи медичних запитів. Покажіть, яке присвоєння стає можливим завдяки обом модифікаторам, і поясніть, чому таке присвоєння є безпечним.

Розділ 8. Додаткові можливості ООП у C#

8.1. Визначення операторів

Оператори +, -, >, <, == добре знайомі у контексті примітивних типів: int, double, bool. Але коли ми створюємо власний клас, компілятор не знає, що означає «скласти» або «порівняти» два об'єкти — адже у кожного класу своя семантика. **Перевантаження операторів** (operator overloading) — це механізм, який дозволяє визначити власну логіку для стандартних операторів стосовно об'єктів нашого типу.

Уявіть клінічну систему, де потрібно порівнювати температуру тіла двох пацієнтів або відстежувати її зміну після прийому жарознижуючого. Без перевантаження код виглядає громіздко і прив'язаний до деталей реалізації:

```
C#
1  if (patient1.Temperature.Celsius > patient2.Temperature.Celsius) { ... }
2  double change = afterMed.Temperature.Celsius - beforeMed.Temperature.Celsius;
```

З перевантаженням — читабельно і природно, як у математиці:

```
C#
1  if (tempAfter > tempBefore) { ... }
2  double change = tempAfter - tempBefore;
```

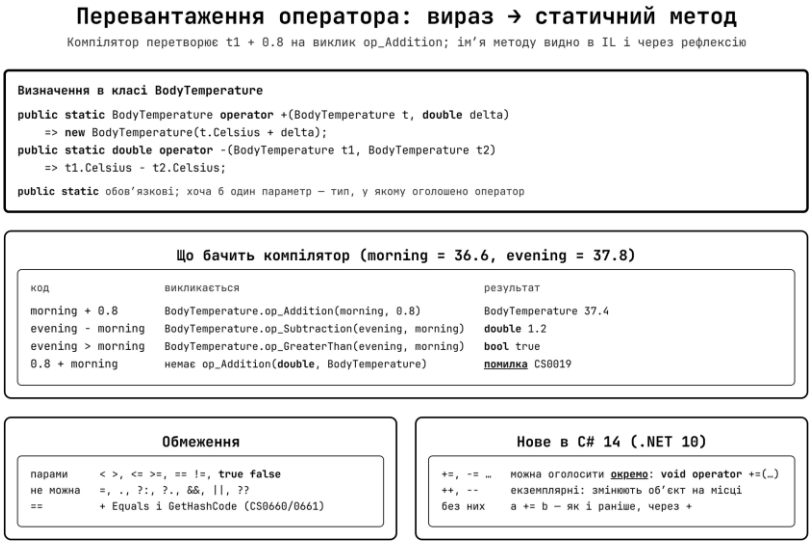
Перевантаження операторів не змінює їх пріоритет, не додає нових символів і не змінює правил мови — воно лише визначає, що відбувається, коли стандартний символ застосовується до об'єктів нашого типу.

Механізм: як компілятор знаходить оператор

Коли компілятор зустрічає вираз t1 + delta, де t1 має тип BodyTemperature, він шукає у цьому класі статичний метод із підходящою сигнатурою:

```
C#
1  public static BodyTemperature operator+(BodyTemperature t, double delta)
```

Якщо такий метод є — вираз t1 + delta автоматично перетворюється на виклик цього статичного методу. У скомпільованому коді він має спеціальне ім'я — op_Addition (для - — op_Subtraction, для > — op_GreaterThan, для == — op_Equality тощо), тож фактично виконується BodyTemperature.op_Addition(t1, delta). Ці імена видно через рефлексію і використовують мови .NET, що не мають синтаксису операторів; з C# так викликати метод не можна — лише через сам оператор. Якщо відповідного методу немає — помилка компіляції CS0019. Це і є «синтаксичний цукор»: зручний запис оператором, за яким приховується звичайний виклик статичного методу.



Перевантаження оператора: від виразу до виклику методу

Синтаксис визначення оператора

Оператор визначається як спеціальний метод у тілі класу або структури:

```
C#
1  public static ТипРезультату operator СИМВОЛ(параметр1[, параметр2])
2  {
3      // логіка
4  }
```

Обов'язкові вимоги:

— Модифікатори public static — оператор є методом типу, а не конкретного екземпляра.

— Ключове слово operator перед символом оператора.

— Принаймні один параметр повинен мати тип класу або структури, де визначається оператор.

— Бінарні оператори (два операнди) — приймають два параметри.

— Унарні оператори (один операнд) — приймають один параметр.

— Тип результату може бути будь-яким: тим самим класом, bool, double, будь-яким іншим.

Клінічний приклад: клас BodyTemperature

Розглянемо клас BodyTemperature, що представляє температуру тіла пацієнта. Цей клас — наочний приклад для перевантаження: температури порівнюють (чи є жар?), обчислюють різницю (наскільки знизилась?), застосовують зсув (зросла на 0.5°C).

Арифметичні оператори

Визначимо два оператори: + для зсуву температури на задану величину і - для обчислення різниці між двома вимірами. Зверніть увагу, що вони повертають різні типи — BodyTemperature і double відповідно, адже семантика операцій різна:

```C#
using System;

// Виконуваний код
BodyTemperature morning = new BodyTemperature(36.6);
BodyTemperature evening = new BodyTemperature(37.8);

// Зсув — наприклад, після фізичного навантаження
BodyTemperature afterExercise = morning + 0.8;
Console.WriteLine($"Після навантаження: {afterExercise}");

// Різниця між вечірнім і ранковим вимірами
double delta = evening - morning;
Console.WriteLine($"Приріст за день: {delta:+0.0;-0.0}°C");

// Клас
class BodyTemperature
{
    public double Celsius { get; }

    public BodyTemperature(double celsius) => Celsius = celsius;

    // Зсув температури: 36.6 + 0.5 → нова температура 37.1
```

```
    public static BodyTemperature operator +(BodyTemperature t, double delta)
    {
        return new BodyTemperature(t.Celsius + delta);
    }

    // Різниця між двома вимірами: повертає double, не BodyTemperature
    public static double operator -(BodyTemperature t1, BodyTemperature t2)
    {
        return t1.Celsius - t2.Celsius;
    }

    public override string ToString() => $"{Celsius:F1}°C";
}
```

Тип результату оператора — не обов'язково той самий клас. Оператор - між двома BodyTemperature повертає double, бо різниця температур — числова величина, а не нова температура. Тип результату визначається семантикою операції.

Оскільки в операторі + перший параметр — BodyTemperature, а другий — double, вираз morning + 0.8 коректний. Але 0.8 + morning вже не компілюється — параметри не збігаються в потрібному порядку. Якщо потрібно, можна визначити ще один перевантажений варіант із переставленими параметрами.

Оператори порівняння

Порівняння — найчастіший сценарій у клінічних системах: «чи є у пацієнта жар?», «яка температура вища?». Визначимо чотири оператори порівняння:

```
C#
using System;

// Виконуваний код
BodyTemperature t1 = new BodyTemperature(38.2);
BodyTemperature t2 = new BodyTemperature(36.9);
BodyTemperature t3 = new BodyTemperature(38.2);

Console.WriteLine($"{t1} > {t2}: {(t1 > t2 ? "так" : "ні")}");
Console.WriteLine($"{t1} < {t2}: {(t1 < t2 ? "так" : "ні")}");
Console.WriteLine($"{t1} == {t3}: {(t1 == t3 ? "так" : "ні")}");
Console.WriteLine($"Пацієнт має жар: {(t1.IsFever ? "так" : "ні")}");

// Клас
class BodyTemperature
{
    public double Celsius { get; }
    public bool IsFever => Celsius > 37.5;

    public BodyTemperature(double celsius) => Celsius = celsius;

    public static bool operator >(BodyTemperature t1, BodyTemperature t2)
        => t1.Celsius > t2.Celsius;
```

```
23
24      public static bool operator <(BodyTemperature t1, BodyTemperature t2)
25          => t1.Celsius < t2.Celsius;
26
27      public static bool operator ==(BodyTemperature t1, BodyTemperature t2)
28          => Math.Abs(t1.Celsius - t2.Celsius) < 0.05;
29
30      public static bool operator !=(BodyTemperature t1, BodyTemperature t2)
31          => !(t1 == t2);
32
33      public override string ToString() => $"{Celsius:F1}°C";
34      public override bool Equals(object obj) => obj is BodyTemperature t && this
    == t;
35      public override int GetHashCode() => Math.Round(Celsius, 1).GetHashCode();
36  }
```

Зверніть увагу на оператор ==: замість строгого t1.Celsius == t2.Celsius використовується допуск 0.05. Це важливо для чисел із плаваючою крапкою — через особливості їх бінарного представлення два «однакові» значення після арифметичних операцій можуть трохи відрізнятись.

Повний клас у клінічному сценарії

Об'єднаємо всі оператори в повному класі і побачимо, як природно вони виглядають у реальному коді:

```
C#
 1  using System;
 2
 3  // Клінічний сценарій: вимірювання до і після прийому жарознижуючого
 4  BodyTemperature before = new BodyTemperature(39.4);
 5  BodyTemperature after  = new BodyTemperature(37.1);
 6
 7  Console.WriteLine($"До прийому:    {before}");
 8  Console.WriteLine($"Після прийому: {after}");
 9
10  double drop = before - after;
11  Console.WriteLine($"Зниження: {drop:F1}°C");
12
13  Console.WriteLine($"Температура нормалізувалась: {(after.IsNormal ? "так" :
    "ні")}");
14
15  // Через 2 години температура трохи підвищилась
16  BodyTemperature twoHoursLater = after + 0.4;
17  Console.WriteLine($"Через 2 год: {twoHoursLater}");
18  Console.WriteLine($"Потрібен повторний прийом: {(twoHoursLater > after ? "так"
    : "ні")}");
19
20  // Клас
21  class BodyTemperature
22  {
23      public double Celsius { get; }
24
25      public bool IsNormal    => Celsius >= 36.0 && Celsius <= 37.0;
```

```
26      public bool IsFever     => Celsius > 37.5;
27      public bool IsHighFever => Celsius > 39.0;
28
29      public BodyTemperature(double celsius)
30      {
31          if (celsius < 30.0 || celsius > 45.0)
32              throw new ArgumentOutOfRangeException(nameof(celsius));
33          Celsius = celsius;
34      }
35
36      public static BodyTemperature operator +(BodyTemperature t, double delta)
37          => new BodyTemperature(t.Celsius + delta);
38
39      public static BodyTemperature operator -(BodyTemperature t, double delta)
40          => new BodyTemperature(t.Celsius - delta);
41
42      public static double operator -(BodyTemperature t1, BodyTemperature t2)
43          => t1.Celsius - t2.Celsius;
44
45      public static bool operator >(BodyTemperature t1, BodyTemperature t2)
46          => t1.Celsius > t2.Celsius;
47
48      public static bool operator <(BodyTemperature t1, BodyTemperature t2)
49          => t1.Celsius < t2.Celsius;
50
51      public static bool operator ==(BodyTemperature t1, BodyTemperature t2)
52          => Math.Abs(t1.Celsius - t2.Celsius) < 0.05;
53
54      public static bool operator !=(BodyTemperature t1, BodyTemperature t2)
55          => !(t1 == t2);
56
57      public override string ToString() =>
58          IsHighFever ? $"{Celsius:F1}°C (висока!)" :
59          IsFever     ? $"{Celsius:F1}°C (жар)"     :
60          IsNormal    ? $"{Celsius:F1}°C (норма)"   :
61                        $"{Celsius:F1}°C";
62
63      public override bool Equals(object obj) => obj is BodyTemperature t && this
    == t;
64      public override int GetHashCode() => Math.Round(Celsius, 1).GetHashCode();
65  }
```

Парні оператори

Деякі оператори компілятор вимагає визначати **парами**. Якщо ви визначаєте один — необхідно визначити і другий:

Пара	Причина
> і <	Логічна симетрія: якщо a > b, то b < a
>= і <=	Аналогічно
== і !=	Якщо об'єкти рівні, то вони не нерівні

Ця вимога вбудована в компілятор — він не дозволить визначити > без <. Мета обмеження — запобігти логічним суперечностям: ситуації, коли визначено лише a > b, а вираз b < a для того самого типу не компілюється, або коли a == b і a != b одночасно повертають true. Ту саму вимогу парності компілятор ставить і до операторів true/false (розділ 8.2). Зауважимо, що компілятор перевіряє лише наявність пари, але не узгодженість реалізацій — логіку обох операторів програміст пише сам.

При перевантаженні == компілятор також рекомендує перевизначити Equals(object) і GetHashCode(). Причина: за замовчуванням ці методи порівнюють посилання, а не значення. Якщо залишити їх без змін, поведінка стає непослідовною — == порівнює значення, а Equals() — посилання на об'єкт у пам'яті.

Таблиця операторів, що підтримують перевантаження

Категорія	**Оператори**	**Примітка**			
Унарні	+x, -x, !, ~, ++, --, true, false	true/false — для умовних виразів			
Арифметичні	+, -, *, /, %	—			
Порівняння	==, !=, <, >, <=, >=	Тільки парами			
Порозрядні	&, `\	, ^, <<, >>, >>>`	—		
Логічні	&&, `\	\	`	Визначаються через &, `\	, true, false`

Оператор присвоєння =, тернарний ?:, оператори доступу . та ?., ??, а також && і || напряму перевантажити **неможливо** (два останні працюють через &, |, true, false). Складені присвоєння +=, -= тощо за замовчуванням автоматично використовують перевантажений + і - — визначати їх окремо не потрібно: t += 0.5 компілюється у t = t + 0.5, тобто створює **новий** об'єкт.

Нове в C# 14 (.NET 10). Для типів, де створення нового об'єкта на кожну операцію дороге (великі буфери, вектори, матриці), C# 14 дозволяє оголосити складене присвоєння **окремим екземплярним оператором**, що змінює об'єкт на місці:

```C#
class DoseAccumulator
{
    public double TotalMg { get; private set; }

    // C# 14: екземплярний оператор, без static, повертає void
    public void operator +=(double mg) => TotalMg += mg;
}
```

Аналогічно можна оголосити екземплярні ++ і -- (public void operator ++()). Якщо такого оператора немає — працює старе правило через +. Перевірено на .NET 10: acc += 2 для такого класу змінює той самий об'єкт.

Коли варто, а коли не варто перевантажувати оператори

Варто, якщо:

— Тип моделює математичне або фізичне поняття: температура, відстань, маса, координата, інтервал часу.

— Оператор має очевидну і однозначну інтерпретацію для цього типу.

— Код з оператором читається так само природно, як математичний вираз.

Не варто, якщо:

— Семантика оператора неочевидна або може трактуватись по-різному.

— Ви перевантажуєте оператор виключно заради стислості — без виграшу в читабельності.

— Тип не є «значеннєвим» поняттям. Для сутностей-ідентичностей — пацієнт, лікар, прийом — оператор + між двома об'єктами не має природного змісту.

Перевантаження операторів — інструмент виразності, а не синтаксичний трюк. Добре спроектований оператор робить код читабельнішим; поганий вносить плутанину і ускладнює підтримку.

8.2. Визначення інкременту та декременту

Оператори ++ і -- — унарні, тобто вони приймають один операнд і повертають змінений стан. Для користувацьких типів їх можна перевантажити так само, як і будь-який інший оператор. Однак у них є важлива особливість, яку необхідно враховувати при визначенні.

Правило незмінності параметрів

Коли ми визначаємо оператор ++ або --, параметр методу представляє поточний об'єкт. Інтуїтивно здається, що потрібно просто змінити значення прямо у методі:

```
C#
1  // НЕПРАВИЛЬНО — мутуємо параметр
2  public static BodyTemperature operator ++(BodyTemperature t)
3  {
4      t.Celsius += 0.1; // не можна: Celsius — readonly
5      return t;
6  }
```

Це неправильний підхід з двох причин. По-перше, якщо властивість доступна лише для читання (що є стандартом для immutable-типів), такий код не скомпілюється. По-друге, і це головне: **оператор не повинен мутувати свої параметри**. Параметри — це входи, а не виходи. Правильне визначення — повернути **новий об'єкт** з оновленим значенням:

```
C#
1  // ПРАВИЛЬНО — повертаємо новий об'єкт
2  public static BodyTemperature operator ++(BodyTemperature t)
3  {
4      return new BodyTemperature(t.Celsius + 0.1);
5  }
```

Цей підхід відомий як «immutable update pattern»: замість зміни існуючого об'єкта ми створюємо новий із потрібним значенням. Він запобігає побічним ефектам і є стандартом при визначенні операторів у C#.

Синтаксис оператора інкременту та декременту

Оскільки ++ і -- — унарні оператори, вони приймають один параметр. Одне визначення охоплює і префіксну (++t), і постфіксну (t++) форму — компілятор сам реалізує різницю:

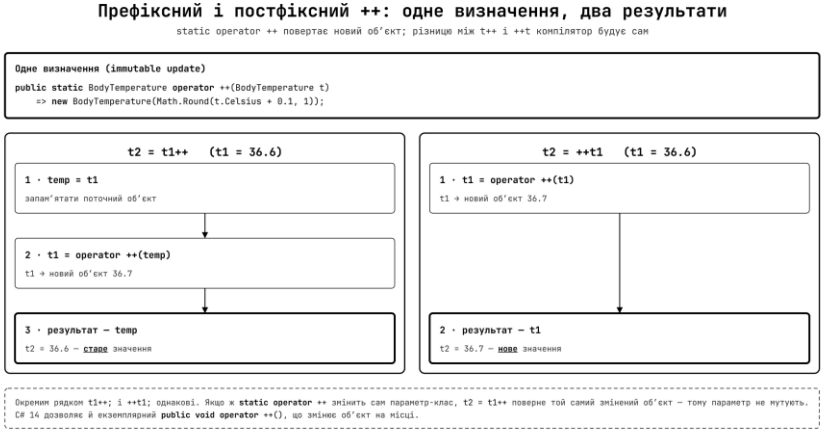
```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  BodyTemperature t = new BodyTemperature(36.6);
 5  Console.WriteLine($"Початкова: {t}");
 6
 7  t++;
 8  Console.WriteLine($"Після t++: {t}");
```

```
 9
10  ++t;
11  Console.WriteLine($"Після ++t: {t}");
12
13  t--;
14  Console.WriteLine($"Після t--: {t}");
15
16  // Клас
17  class BodyTemperature
18  {
19      public double Celsius { get; }
20
21      public BodyTemperature(double celsius) => Celsius = celsius;
22
23      // Один метод охоплює і ++t, і t++
24      public static BodyTemperature operator ++(BodyTemperature t)
25          => new BodyTemperature(Math.Round(t.Celsius + 0.1, 1));
26
27      public static BodyTemperature operator --(BodyTemperature t)
28          => new BodyTemperature(Math.Round(t.Celsius - 0.1, 1));
29
30      public override string ToString() => $"{Celsius:F1}°C";
31  }
```

Math.Round(..., 1) тут необхідний через особливості арифметики чисел із плаваючою крапкою: 36.6 + 0.1 без округлення може дати 36.699999... замість 36.7.

Префіксна та постфіксна форми: як компілятор розрізняє їх

Хоча ми визначаємо один метод, поведінка t++ і ++t — різна. Різниця не в тому, як змінюється об'єкт, а в тому, **яке значення повертається як результат виразу**. Компілятор реалізує це самостійно на основі одного нашого визначення.



Постфіксний і префіксний ++: одне визначення, різна поведінка

Побачимо різницю в дії:

```C#
using System;

// Виконуваний код
BodyTemperature t1 = new BodyTemperature(36.6);

// Постфіксний: t2 отримує СТАРУ копію, потім t1 інкрементується
BodyTemperature t2 = t1++;
Console.WriteLine($"Постфіксний: t1={t1},  t2={t2}");
// t1 = 36.7, t2 = 36.6 — t2 зафіксувала стан ДО зміни

BodyTemperature t3 = new BodyTemperature(36.6);

// Префіксний: t3 спочатку інкрементується, потім t4 отримує НОВУ копію
BodyTemperature t4 = ++t3;
Console.WriteLine($"Префіксний:  t3={t3},  t4={t4}");
// t3 = 36.7, t4 = 36.7 — t4 отримала стан ПІСЛЯ зміни

// Клас
class BodyTemperature
{
    public double Celsius { get; }

    public BodyTemperature(double celsius) => Celsius = celsius;

    public static BodyTemperature operator ++(BodyTemperature t)
        => new BodyTemperature(Math.Round(t.Celsius + 0.1, 1));

    public override string ToString() => $"{Celsius:F1}°C";
}
```

При постфіксному t1++ компілятор виконує три дії: зберігає поточний об'єкт у тимчасову змінну, замінює t1 результатом operator++(t1), а як результат виразу повертає тимчасову (стару) копію. При префіксному ++t3 — викликає operator++(t3), присвоює результат t3 і повертає його ж.

Ця різниця важлива лише тоді, коли результат виразу використовується: t2 = t1++ та t2 = ++t1 поводяться по-різному. Якщо ж ми пишемо просто t++; або ++t; в окремому рядку — різниці немає.

Оператори true та false

Окрема пара унарних операторів — true і false. Вони визначаються, коли ми хочемо використовувати об'єкт нашого класу **безпосередньо як умову** в if, while або тернарному операторі — без явного виклику методу чи порівняння з bool.

У клінічній системі природним кандидатом є клас MedicalDevice. Пристрій може бути активним або офлайн, і зручно писати if (device) замість if (device.IsOnline):

```C#
using System;

// Виконуваний код
MedicalDevice monitor = new MedicalDevice("Кардіомонітор", true);
MedicalDevice scanner = new MedicalDevice("МРТ-сканер", false);

// Використання в умові if — без явного .IsOnline
if (monitor)
    Console.WriteLine($"{monitor.Name}: готовий до роботи");

if (!scanner)
    Console.WriteLine($"{scanner.Name}: потрібне технічне обслуговування");

// Тернарний оператор
string status = monitor ? "підключений" : "відключений";
Console.WriteLine($"Статус монітора: {status}");

// Клас
class MedicalDevice
{
    public string Name { get; }
    public bool IsOnline { get; }

    public MedicalDevice(string name, bool isOnline)
    {
        Name = name;
        IsOnline = isOnline;
    }

    // Пристрій «true» — якщо активний
    public static bool operator true(MedicalDevice d)  => d.IsOnline;
```

```
32
33      // Пристрій «false» — якщо офлайн
34      public static bool operator false(MedicalDevice d) => !d.IsOnline;
35
36      // Логічне заперечення: !device
37      public static bool operator !(MedicalDevice d)     => !d.IsOnline;
38
39      public override string ToString()
40          => IsOnline ? $"{Name} [активний]" : $"{Name} [офлайн]";
41  }
```

Оператори true і false завжди визначаються **парою** — компілятор вимагає наявності обох. Оператор ! не є обов'язковим, але без нього конструкція if (!device) не компілюється. Семантично !device збігається з оператором false — обидва перевіряють, що пристрій «не true».

Повний приклад: температурний моніторинг

Об'єднаємо ++, -- і true/false у реалістичному клінічному сценарії — автоматичному відстеженні температури пацієнта після лікування:

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  BodyTemperature temp = new BodyTemperature(37.8);
 5  Console.WriteLine($"Початкова температура: {temp}");
 6
 7  if (!temp)
 8      Console.WriteLine("Стан відхиляється від норми — призначено лікування");
 9
10  Console.WriteLine("Динаміка після лікування:");
11  for (int i = 0; i < 10; i++)
12  {
13      temp--;
14      Console.WriteLine($"  вимірювання {i + 1}: {temp}");
15      if (temp)
16      {
17          Console.WriteLine("Температура нормалізувалась. Лікування завершено.");
18          break;
19      }
20  }
21
22  // Клас
23  class BodyTemperature
24  {
25      public double Celsius { get; }
26
27      public bool IsNormal => Celsius >= 36.0 && Celsius <= 37.0;
28      public bool IsFever  => Celsius > 37.5;
29
30      public BodyTemperature(double celsius) => Celsius = celsius;
31
32      public static BodyTemperature operator ++(BodyTemperature t)
33          => new BodyTemperature(Math.Round(t.Celsius + 0.1, 1));
```

```
34
35      public static BodyTemperature operator --(BodyTemperature t)
36          => new BodyTemperature(Math.Round(t.Celsius - 0.1, 1));
37
38      // true: температура в нормі
39      public static bool operator true(BodyTemperature t)  => t.IsNormal;
40      public static bool operator false(BodyTemperature t) => !t.IsNormal;
41      public static bool operator !(BodyTemperature t)     => !t.IsNormal;
42
43      public override string ToString() =>
44          IsFever  ? $"{Celsius:F1}°C (жар)"   :
45          IsNormal ? $"{Celsius:F1}°C (норма)" :
46                     $"{Celsius:F1}°C";
47  }
```

У цьому прикладі if (!temp) перевіряє аномальний стан, а if (temp) у циклі фіксує повернення до норми. Код читається природно і не вимагає явного звертання до IsNormal або IsFever у кожній умові.

8.3. Перевантаження операцій перетворення типів

У розділі 2 ми розглядали неявні та явні перетворення між примітивними типами: int → long відбувається автоматично (implicit), а double → int потребує явного cast-у (explicit). Той самий механізм можна визначити і для власних класів — це **перевантаження операторів перетворення типів**.

Завдяки цьому ми можемо навчити компілятор, як перетворювати об'єкт нашого типу на інший тип (і навпаки), і контролювати — чи це відбуватиметься автоматично, чи лише при явному приведенні. Наприклад, у клінічній системі зручно присвоювати числове значення температури безпосередньо об'єкту BodyTemperature:

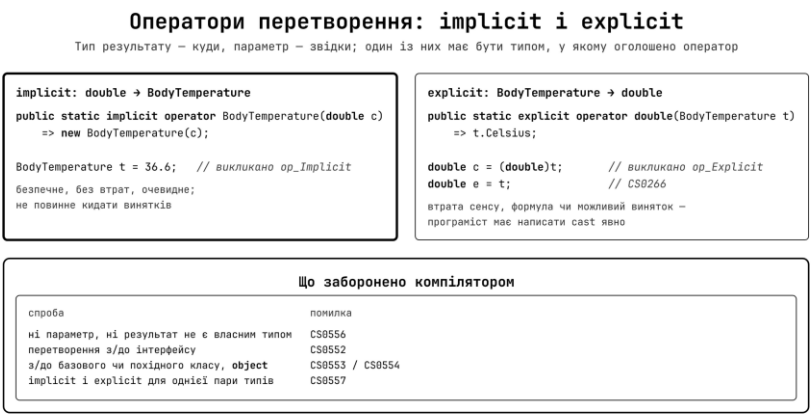
```
C#
1  BodyTemperature t = 36.6;  // implicit: double → BodyTemperature
2  double c = (double)t;      // explicit: BodyTemperature → double
```

Синтаксис оператора перетворення

Оператор перетворення визначається як статичний метод у класі або структурі:

```
C#
1  public static implicit|explicit operator ТипРезультату(ВхіднийТип param)
2  {
3      // логіка перетворення
4  }
```

— public static — обов'язкові модифікатори.
— implicit або explicit — визначає, чи потрібен явний cast.
— operator — ключове слово.
— ТипРезультату — тип, **до** якого перетворюємо.
— ВхіднийТип param — тип і параметр, **з** якого перетворюємо.



Оператор перетворення типів: implicit vs explicit

Обмеження: оператор перетворення повинен або приймати параметром об'єкт свого типу, або повертати об'єкт свого типу. Тобто оператор визначений у класі BodyTemperature може перетворювати double → BodyTemperature (повертає свій тип) або BodyTemperature → double (приймає свій тип). Визначити оператор між двома чужими типами в середині третього класу — неможливо.

implicit: неявне перетворення

Неявне перетворення відбувається автоматично — без жодного синтаксису з боку програміста. Використовується тоді, коли перетворення безпечне, без втрати даних і семантично очевидне.

Визначимо implicit-перетворення з double у BodyTemperature — створення температури з числового значення:

```
C#
1  using System;
2
3  // Виконуваний код
```

```
// Неявне перетворення: double → BodyTemperature (без cast)
BodyTemperature morning = 36.6;
BodyTemperature fever   = 38.5;

Console.WriteLine($"Ранкова температура: {morning}");
Console.WriteLine($"Температура при жарі: {fever}");
Console.WriteLine($"Є жар: {(fever.IsFever ? "так" : "ні")}");

// Клас
class BodyTemperature
{
    public double Celsius { get; }
    public bool IsFever => Celsius > 37.5;

    public BodyTemperature(double celsius) => Celsius = celsius;

    // implicit: double → BodyTemperature (безпечно, без втрат)
    public static implicit operator BodyTemperature(double celsius)
        => new BodyTemperature(celsius);

    public override string ToString() => $"{Celsius:F1}°C";
}
```

Завдяки implicit ми можемо писати BodyTemperature t = 36.6 — компілятор сам викличе оператор перетворення. Це зручно і читабельно, адже семантика очевидна: число 36.6 — це температура в градусах Цельсія.

explicit: явне перетворення

Явне перетворення вимагає від програміста написати cast явно: (ТипРезультату). Це сигнал: «я свідомо роблю це перетворення і розумію наслідки».

Визначимо explicit-перетворення з BodyTemperature у double — вилучення числового значення температури:

C#
```
using System;

// Виконуваний код
BodyTemperature t = new BodyTemperature(38.2);

// Явне перетворення: BodyTemperature → double (потрібен cast)
double celsius = (double)t;
Console.WriteLine($"Температура як число: {celsius}");

// Якщо написати просто: double c = t; — буде помилка компіляції
// (немає implicit оператора в цьому напрямку)

// Клас
class BodyTemperature
{
    public double Celsius { get; }

```

```
    public BodyTemperature(double celsius) => Celsius = celsius;

    // explicit: BodyTemperature → double (явно — бо "просто число" без контексту)
    public static explicit operator double(BodyTemperature t)
        => t.Celsius;

    public override string ToString() => $"{Celsius:F1}°C";
}
```

Чому тут explicit, а не implicit? Коли ми витягуємо double з BodyTemperature, результат втрачає семантичний контекст — це вже «просто число», а не температура. Програміст має явно підтвердити, що він хоче саме це. Якщо б перетворення було неявним, можна було б випадково отримати число там, де очікувався об'єкт BodyTemperature, і не помітити помилки.

Перетворення між двома власними типами

Оператори перетворення можна визначити і між двома складовими типами. Розглянемо конвертацію між BodyTemperature (градуси Цельсія) і FahrenheitTemperature (градуси Фаренгейта) — типова задача в міжнародних клінічних системах:

C#
```
using System;

// Виконуваний код
BodyTemperature celsius = new BodyTemperature(38.5);

// explicit: BodyTemperature → FahrenheitTemperature (формульне перетворення)
FahrenheitTemperature fahrenheit = (FahrenheitTemperature)celsius;
Console.WriteLine($"Цельсій:    {celsius}");
Console.WriteLine($"Фаренгейт:  {fahrenheit}");

// explicit: FahrenheitTemperature → BodyTemperature (назад)
BodyTemperature backToCelsius = (BodyTemperature)fahrenheit;
Console.WriteLine($"Назад:      {backToCelsius}");

// Клас температури в Цельсіях
class BodyTemperature
{
    public double Celsius { get; }

    public BodyTemperature(double celsius) => Celsius = celsius;

    // explicit → FahrenheitTemperature (формула: F = C * 9/5 + 32)
    public static explicit operator FahrenheitTemperature(BodyTemperature t)
        => new FahrenheitTemperature(t.Celsius * 9.0 / 5.0 + 32.0);

    public override string ToString() => $"{Celsius:F1}°C";
}

```

```
// Клас температури у Фаренгейтах
class FahrenheitTemperature
{
    public double Fahrenheit { get; }

    public FahrenheitTemperature(double fahrenheit) => Fahrenheit = fahrenheit;

    // explicit → BodyTemperature (формула: C = (F - 32) * 5/9)
    public static explicit operator BodyTemperature(FahrenheitTemperature f)
        => new BodyTemperature((f.Fahrenheit - 32.0) * 5.0 / 9.0);

    public override string ToString() => $"{Fahrenheit:F1}°F";
}
```

Обидва перетворення тут explicit, бо вони нетривіальні: задіяна формула, є заокруглення, і читач коду повинен бачити, що відбувається перетворення між системами вимірювання. Неявне перетворення тут приховало б важливий факт.

Коли обирати implicit, а коли explicit

Критерій	**implicit**	**explicit**
Можлива втрата даних?	Ні	Так (або невідомо)
Семантика очевидна?	Так	Неочевидна або потребує уваги
Задіяна нетривіальна формула?	Ні	Так
Перетворення між різними одиницями вимірювання?	Ні	Так
Приклад у клінічному домені	double → BodyTemperature	BodyTemperature → FahrenheitTemperature

Загальне правило: **якщо є хоч найменший сумнів — робіть explicit**. Явний cast у коді — це документація: він повідомляє читачеві, що тут відбувається свідоме перетворення типів.

8.4. Індексатори

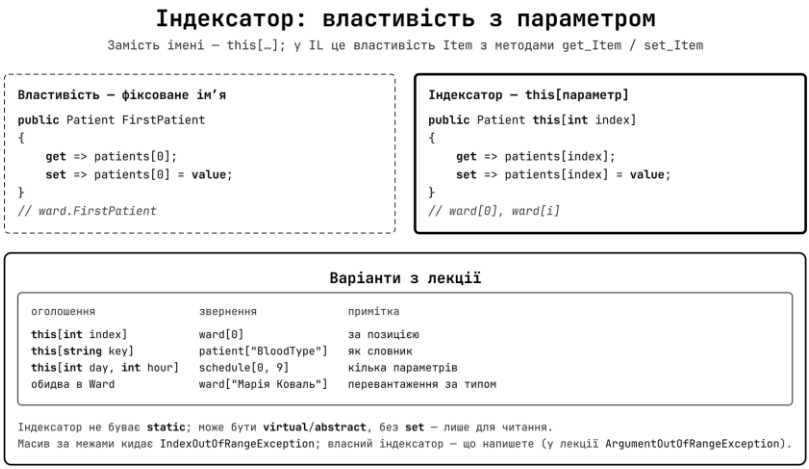
Масиви зручні тим, що до елементів можна звертатися за індексом: arr[0], arr[i]. Але іноді ми хочемо, щоб і власний клас підтримував таку синтаксичну форму — наприклад, звертатися до пацієнтів відділення як ward[0], ward[i], або до атрибутів пацієнта як patient["diagnosis"]. Саме для цього у C# існують **індексатори** (indexers).

Індексатор — це спеціальний член класу, який дозволяє звертатися до об'єкта за допомогою квадратних дужок, як до масиву або словника. За своєю формою він нагадує властивість: також має блоки get і set, також має тип. Ключова відмінність — замість фіксованої назви використовується ключове слово this і параметр у квадратних дужках.

Синтаксис індексатора

```
C#
тип_результату this[Тип параметр, ...]
{
    get { ... }
    set { ... }
}
```

- this — ключове слово замість назви. Означає «доступ до цього об'єкта за індексом».
- Тип у квадратних дужках — тип індексу (може бути int, string або будь-яким іншим).
- get — повертає значення за вказаним індексом.
- set — записує значення; переданий об'єкт доступний через неявний параметр value.
- Індексатор не може бути статичним — він завжди прив'язаний до екземпляра.



Індексатор: синтаксис та порівняння з властивістю

Числовий індексатор

Найпоширеніший варіант — індексатор з параметром int, який дозволяє звертатися до елементів внутрішньої колекції об'єкта. Визначимо клас Ward (відділення лікарні), що зберігає масив пацієнтів і надає доступ до них за позицією:

```C#
using System;

// Виконуваний код
Ward cardiology = new Ward("Кардіологія", new Patient[]
{
    new Patient("Іван Петренко"),
    new Patient("Марія Коваль"),
    new Patient("Олег Сидоренко"),
});

// Доступ через індексатор — як у масиві
Patient first = cardiology[0];
Console.WriteLine($"Перший пацієнт: {first.Name}");

// Заміна елемента через set
cardiology[0] = new Patient("Андрій Мельник");
Console.WriteLine($"Після виписки: {cardiology[0].Name}");

// Ітерація
for (int i = 0; i < 3; i++)
    Console.WriteLine($"  [{i}] {cardiology[i].Name}");

```

```
// Класи
class Patient
{
    public string Name { get; }
    public Patient(string name) => Name = name;
}

class Ward
{
    public string Name { get; }
    private Patient[] _patients;

    public Ward(string name, Patient[] patients)
    {
        Name = name;
        _patients = patients;
    }

    public Patient this[int index]
    {
        get
        {
            if (index < 0 || index >= _patients.Length)
                throw new ArgumentOutOfRangeException(nameof(index));
            return _patients[index];
        }
        set
        {
            if (index < 0 || index >= _patients.Length)
                throw new ArgumentOutOfRangeException(nameof(index));
            _patients[index] = value;
        }
    }
}
```

Блок get перевіряє межі масиву і генерує ArgumentOutOfRangeException, якщо індекс некоректний — Зауважимо, що сам масив у такій ситуації кидає інший виняток — IndexOutOfRangeException (перевірено на .NET 10); індексатор власного класу може обрати виняток сам, і ArgumentOutOfRangeException тут доречніший, бо описує неправильний аргумент методу get/set. За замовчуванням компілятор перетворює індексатор на властивість з ім'ям Item і методами get_Item/set_Item — саме так його бачать рефлексія та інші мови .NET. Блок set виконує ту саму перевірку перед записом.

Рядковий індексатор

Індекс не обов'язково має бути цілим числом. Рядковий індекс зручний, коли об'єкт зберігає набір іменованих атрибутів. Визначимо Patient із доступом до клінічних даних за ключовим рядком:

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  Patient patient = new Patient("Іван Петренко");
 5  patient["diagnosis"] = "Гіпертонія II ст.";
 6  patient["allergies"] = "Пеніцилін";
 7  patient["notes"]     = "Контроль АТ двічі на день";
 8
 9  Console.WriteLine($"Пацієнт: {patient.Name}");
10  Console.WriteLine($"Діагноз:  {patient["diagnosis"]}");
11  Console.WriteLine($"Алергії:  {patient["allergies"]}");
12  Console.WriteLine($"Примітки: {patient["notes"]}");
13
14  // Клас
15  class Patient
16  {
17      public string Name { get; }
18      private string _diagnosis = "";
19      private string _allergies = "";
20      private string _notes = "";
21
22      public Patient(string name) => Name = name;
23
24      public string this[string attribute]
25      {
26          get
27          {
28              switch (attribute)
29              {
30                  case "diagnosis": return _diagnosis;
31                  case "allergies": return _allergies;
32                  case "notes":     return _notes;
33                  default: throw new ArgumentException($"Невідомий атрибут:
    {attribute}");
34              }
35          }
36          set
37          {
38              switch (attribute)
39              {
40                  case "diagnosis": _diagnosis = value; break;
41                  case "allergies": _allergies = value; break;
42                  case "notes":     _notes = value;     break;
43                  default: throw new ArgumentException($"Невідомий атрибут:
    {attribute}");
44              }
45          }
46      }
47  }
```

Рядковий індексатор дає інтерфейс, схожий на словник, але з чіткими дозволеними ключами і можливістю додати валідацію або логіку перетворення прямо в get/set.

Індексатор із кількома параметрами

Індексатор може приймати більше одного параметра. Це корисно для двовимірних структур, наприклад, для розкладу прийомів лікаря, де день і час — це два незалежні виміри:

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  AppointmentGrid schedule = new AppointmentGrid(5, 16);
 5
 6  schedule[0, 9]  = "Кардіологія — Петренко І.";
 7  schedule[0, 11] = "Неврологія  — Коваль М.";
 8  schedule[1, 10] = "Ортопедія   — Сидоренко О.";
 9
10  Console.WriteLine($"Пн 09:00 — {schedule[0, 9]}");
11  Console.WriteLine($"Пн 11:00 — {schedule[0, 11]}");
12  Console.WriteLine($"Вт 10:00 — {schedule[1, 10]}");
13
14  string slot = schedule[0, 14];
15  Console.WriteLine($"Пн 14:00 — {(slot != null ? slot : "(вільно)")}");
16
17  // Клас
18  class AppointmentGrid
19  {
20      private string[,] _slots;
21
22      public AppointmentGrid(int days, int hours)
23      {
24          _slots = new string[days, hours];
25      }
26
27      // Два параметри: день (0=Пн..4=Пт) і година (8..17)
28      public string this[int day, int hour]
29      {
30          get => _slots[day, hour];
31          set => _slots[day, hour] = value;
32      }
33  }
```

Синтаксис звернення schedule[0, 9] виглядає так само природно, як доступ до двовимірного масиву. Компілятор транслює це в виклик індексатора з двома аргументами.

Модифікатори доступу та readonly-індексатор

Як і у властивостей, в індексаторах можна опустити блок set (тоді індексатор лише для читання) або обмежити його доступ модифікатором:

```
C#
 1  // Тільки для читання — set відсутній
```

```
 2  public Patient this[int index]
 3  {
 4      get => _patients[index];
 5  }
 6
 7  // set приватний — зовні не можна записати
 8  public Patient this[int index]
 9  {
10      get => _patients[index];
11      private set => _patients[index] = value;
12  }
```

Readonly-індексатор корисний для незмінних колекцій (наприклад, архівних записів), де зовнішній код має тільки читати дані, але не змінювати їх.

Перевантаження індексаторів

Так само як і методи, індексатори можна **перевантажувати** — визначити кілька версій з різними типами або кількістю параметрів. Визначимо Ward одночасно з числовим доступом (за позицією) і рядковим (за ім'ям пацієнта):

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  Ward ward = new Ward(new Patient[]
 5  {
 6      new Patient("Іван Петренко"),
 7      new Patient("Марія Коваль"),
 8      new Patient("Олег Сидоренко"),
 9  });
10
11  // Доступ за числовим індексом
12  Console.WriteLine($"Пацієнт [0]: {ward[0].Name}");
13  Console.WriteLine($"Пацієнт [2]: {ward[2].Name}");
14
15  // Доступ за ім'ям
16  Patient found = ward["Марія Коваль"];
17  Console.WriteLine($"Знайдено за ім'ям: {found.Name}");
18
19  // Класи
20  class Patient
21  {
22      public string Name { get; }
23      public Patient(string name) => Name = name;
24  }
25
26  class Ward
27  {
28      private Patient[] _patients;
29
30      public Ward(Patient[] patients) => _patients = patients;
31
```

```
32      // Перший індексатор — доступ за позицією
33      public Patient this[int index] => _patients[index];
34
35      // Другий індексатор — пошук за ім'ям
36      public Patient this[string name]
37      {
38          get
39          {
40              foreach (var p in _patients)
41                  if (p.Name == name) return p;
42              throw new ArgumentException($"Пацієнт '{name}' не знайдений");
43          }
44      }
45  }
```

Компілятор розрізняє перевантажені варіанти за типом аргументу у дужках: ward[0] — це int, тому викликається перший індексатор; ward["Марія Коваль"] — це string, тому другий. Перевантаження за кількістю параметрів також дозволене.

Індексатори можуть бути **віртуальними** і **абстрактними** — їх можна перевизначати у похідних класах так само, як і методи, що дозволяє будувати гнучкі ієрархії класів із уніфікованим синтаксисом доступу.

8.5. Змінні-посилання та повернення посилання

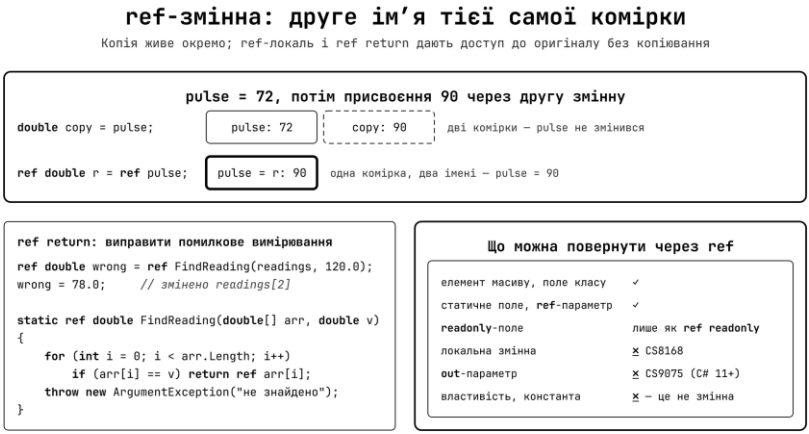
Ми вже знаємо, що параметр із модифікатором ref дозволяє методу змінювати значення змінної, що знаходиться у викликаючому коді — бо передається не копія значення, а посилання на ділянку пам'яті. У C# ці можливості йдуть далі: можна визначити **локальну змінну-посилання** (ref local) і повертати **посилання з методу** (ref return). Це дозволяє маніпулювати значеннями в масивах і структурах безпосередньо в пам'яті, без копіювання.

Змінна-посилання (ref local)

Звичайне присвоєння double copy = x створює **копію** — нову ділянку пам'яті з тим самим значенням. Зміна copy не впливає на x. Якщо ж визначити змінну з ключовим словом ref, вона стане **псевдонімом** (alias) для вже існуючої ділянки пам'яті:

```
C#
1  double pulse = 72.0;
2  ref double pulseRef = ref pulse; // pulseRef — псевдонім для pulse
```

Тепер pulseRef і pulse — це два імені для одного й того самого місця в пам'яті. Зміна одного одразу відображається в іншому.



ref-змінна: псевдонім vs копія

Ref-змінну обов'язково потрібно ініціалізувати при оголошенні — визначити, на що вона вказує. Просте оголошення без ініціалізації не скомпілюється:

```
C#
1  ref double r; // помилка — ref local без ініціалізації
```

Побачимо поведінку на практиці:

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  double pulse = 72.0;
 5  ref double pulseRef = ref pulse;
 6
 7  Console.WriteLine($"pulse:    {pulse}");
 8  Console.WriteLine($"pulseRef: {pulseRef}");
 9
10  // Змінюємо через pulseRef — змінюється pulse
11  pulseRef = 90.0;
12  Console.WriteLine($"Після pulseRef = 90.0:");
13  Console.WriteLine($"  pulse:    {pulse}");
14  Console.WriteLine($"  pulseRef: {pulseRef}");
15
16  // Змінюємо pulse — pulseRef теж змінюється
17  pulse = 65.0;
```

```
Console.WriteLine($"Після pulse = 65.0:");
Console.WriteLine($"  pulse:    {pulse}");
Console.WriteLine($"  pulseRef: {pulseRef}");
```

Повернення посилання з методу (ref return)

Окрім локальних ref-змінних, C# дозволяє **повернути посилання з методу**. Це корисно, коли потрібно дати зовнішньому коду прямий доступ до елемента масиву або поля структури — без копіювання значення туди й назад.

Для цього в сигнатурі методу перед типом результату вказується ref, і після return також ставиться ref:

```
C#
ref double Find(double target, double[] array)
{
    for (int i = 0; i < array.Length; i++)
        if (array[i] == target)
            return ref array[i]; // повертаємо посилання на елемент
    throw new IndexOutOfRangeException("значення не знайдено");
}
```

Та при виклику змінна, що приймає результат, теж визначається з ref:

```
C#
ref double element = ref Find(target, array);
element = newValue; // змінює безпосередньо елемент масиву
```

Застосуємо це в клінічному сценарії — знайдемо конкретний показник пульсу в журналі вимірювань і замінимо його виправленим значенням:

```
C#
using System;

// Виконуваний код
double[] readings = { 68.0, 75.0, 120.0, 72.0, 80.0 }; // 120 — помилкове
вимірювання

Console.WriteLine("До виправлення:");
for (int i = 0; i < readings.Length; i++)
    Console.WriteLine($"  [{i}] = {readings[i]}");

// Отримуємо ref на конкретний елемент і виправляємо його напряму
ref double wrong = ref FindReading(120.0, readings);
wrong = 78.0;

Console.WriteLine("Після виправлення:");
for (int i = 0; i < readings.Length; i++)
    Console.WriteLine($"  [{i}] = {readings[i]}");
```

```
17
18  // Метод — повертає ref на елемент масиву
19  ref double FindReading(double target, double[] arr)
20  {
21      for (int i = 0; i < arr.Length; i++)
22          if (arr[i] == target)
23              return ref arr[i];
24      throw new ArgumentException($"Значення {target} не знайдено");
25  }
```

Зверніть увагу: метод FindReading повертає не значення 120.0, а **посилання** на комірку пам'яті arr[2]. Тому присвоєння wrong = 78.0 змінює саме цю комірку — елемент оригінального масиву readings.

ref return для двох значень

Ще один практичний приклад — метод, що повертає посилання на **більше** з двох значень. Це дозволяє безпосередньо змінити максимальний показник:

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  double systolic  = 130.0;
 5  double diastolic = 85.0;
 6
 7  Console.WriteLine($"Систолічний:  {systolic}");
 8  Console.WriteLine($"Діастолічний: {diastolic}");
 9
10  // Отримуємо ref на більший показник і коригуємо його
11  ref double higher = ref MaxRef(ref systolic, ref diastolic);
12  higher -= 10.0;
13
14  Console.WriteLine($"Після зниження більшого на 10:");
15  Console.WriteLine($"  Систолічний:  {systolic}");
16  Console.WriteLine($"  Діастолічний: {diastolic}");
17
18  // Метод — повертає ref на більший з двох параметрів
19  ref double MaxRef(ref double a, ref double b)
20  {
21      if (a >= b) return ref a;
22      else        return ref b;
23  }
```

Параметри методу MaxRef також визначені з ref — бо метод повертає посилання на один із них, а посилання повинно вказувати на змінну, що існує поза межами методу.

ref readonly: посилання тільки для читання

Якщо потрібно дати доступ до елемента за посиланням (без копіювання), але захистити його від зміни, використовується ref readonly:

```
C#
1  ref readonly double element = ref array[i]; // не можна змінити через element
2  // element = 99.0; // помилка компіляції
```

Це зручно для read-only доступу до великих структур — ми уникаємо копіювання, але гарантуємо, що значення не зміниться через цю змінну.

Обмеження методів із ref return

Метод, що повертає посилання, підпорядковується певним правилам. Він **не може** повертати посилання на:

Що заборонено	Причина
null	Посилання повинно вказувати на реальну ділянку пам'яті
Константу або літерал	Константи не мають адреси у купі
Значення enum	Аналогічно
Властивість класу або структури	Властивість не є змінною — це метод
Поле з модифікатором readonly (звичайним ref)	Захищено від змін; дозволено лише ref readonly return
Локальну змінну всередині методу	Вона зникає після виходу з методу

Повертати можна посилання на елементи масивів, поля екземпляра класу (без readonly), статичні поля та параметри ref — тобто на те, що існує поза межами поточного стекового кадру. out-параметри з C# 11 вважаються обмеженими поточним методом (scoped), і return ref на них дає помилку CS9075; локальна змінна — CS8168 (обидва випадки перевірено на .NET 10).

8.6. Методи розширення

Уявіть ситуацію: вам потрібно додати корисний метод до класу string або структури DateTime — але ви не маєте доступу до їх вихідного коду і не можете успадкуватися від них (string — sealed-клас, а від структур успадковуватися неможливо взагалі). Або ви хочете додати утиліти до типів зі сторонньої бібліотеки, не чіпаючи її код. Саме для цього існують **методи розширення** (extension methods) — механізм, який дозволяє «додавати» нові методи до будь-якого типу ззовні, без зміни його вихідного коду.

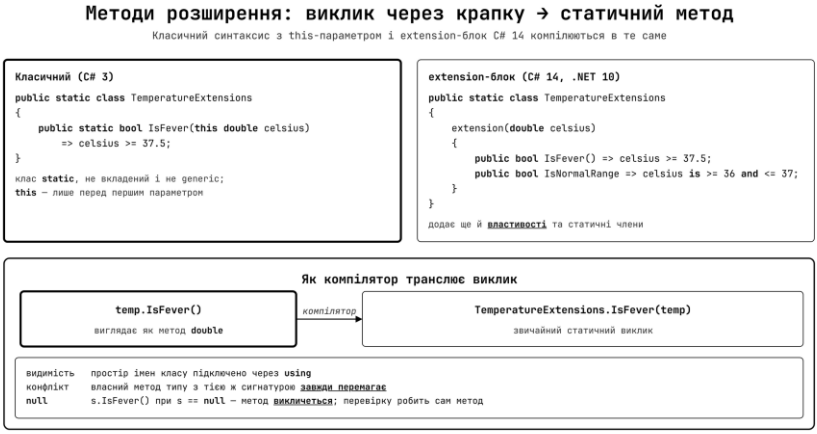
Методи розширення — це не магія. За лаштунками вони залишаються звичайними статичними методами. Але компілятор дозволяє викликати їх через крапку, як ніби вони є рідними методами типу:

```
C#
1  double temp = 38.5;
2  bool hasFever = temp.IsFever(); // виглядає як метод double
3  // компілятор транслює це в:
4  bool hasFever = TemperatureExtensions.IsFever(temp);
```

Це **синтаксичний цукор** — зручний запис, що приховує звичайний статичний виклик.

Синтаксис методу розширення

Метод розширення визначається у public static класі як public static метод. Перший параметр — особливий: він позначається ключовим словом this і вказує, який тип цей метод розширює. При виклику цей параметр не передається явно — компілятор підставляє значення зліва від крапки.



Методи розширення: синтаксис і механізм компіляції

```
C#
1  public static class TemperatureExtensions
2  {
3      public static bool IsFever(this double celsius) => celsius > 37.5;
4      //                          ^^^^ — розширюємо тип double
5  }
```

Розширення вбудованих типів: double

Додамо кілька корисних методів до double для роботи з температурними показниками — типовий сценарій у клінічних системах:

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  double morning  = 36.6;
 5  double evening  = 38.2;
 6  double critical = 40.1;
 7
 8  Console.WriteLine($"Ранкова {morning}°C — жар: {(morning.IsFever() ? "так" :
    "ні")}");
 9  Console.WriteLine($"Вечірня {evening}°C — жар: {(evening.IsFever() ? "так" :
    "ні")}");
10  Console.WriteLine($"Критична {critical}°C — небезпечна:
    {(critical.IsDangerous() ? "так" : "ні")}");
11  Console.WriteLine($"37.8°C у Фаренгейтах: {37.8.ToFahrenheit():F1}°F");
12
13  // Статичний клас з методами розширення для double
14  public static class TemperatureExtensions
15  {
```

```
    public static bool IsFever(this double celsius)
        => celsius > 37.5;

    public static bool IsDangerous(this double celsius)
        => celsius > 39.5;

    public static double ToFahrenheit(this double celsius)
        => celsius * 9.0 / 5.0 + 32.0;
}
```

Розширення вбудованих типів: string

Методи розширення особливо корисні для string — дозволяють додавати специфічну бізнес-логіку без успадкування. Додамо методи для роботи з медичними рядками:

```
C#
using System;

// Виконуваний код
string diagnosis = "   гіпертонія іі ст.   ";
string code = "ICD-10";
string emptyCode = "";

Console.WriteLine($"Нормалізований діагноз:
'{diagnosis.NormalizeDiagnosis()}'");
Console.WriteLine($"Код '{code}' валідний: {(code.IsValidMedicalCode() ? "так"
: "ні")}");
Console.WriteLine($"Код '{emptyCode}' валідний:
{(emptyCode.IsValidMedicalCode() ? "так" : "ні")}");
Console.WriteLine($"Скорочено: '{("Гострий інфаркт міокарда, ускладнений
серцевою недостатністю").Truncate(30)}'");

// Статичний клас з методами розширення для string
public static class StringMedicalExtensions
{
    // Trim + нормалізація регістру
    public static string NormalizeDiagnosis(this string s)
        => string.IsNullOrWhiteSpace(s) ? "" : s.Trim().ToLower();

    // Простий код: не порожній і довший за 3 символи
    public static bool IsValidMedicalCode(this string s)
        => !string.IsNullOrWhiteSpace(s) && s.Trim().Length >= 3;

    // Обрізати до maxLen символів, додати "..."
    public static string Truncate(this string s, int maxLen)
    {
        if (s == null || s.Length <= maxLen) return s;
        return s.Substring(0, maxLen) + "...";
    }
}
```

Розширення DateTime

Методи розширення для DateTime зручні для бізнес-логіки, яка не належить до самого класу DateTime, але часто потрібна в конкретному домені:

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  DateTime appointment1 = new DateTime(2027, 6, 8, 10, 30, 0);  // вівторок,
    10:30
 5  DateTime appointment2 = new DateTime(2027, 6, 12, 19, 0, 0);  // субота, 19:00
 6  DateTime appointment3 = new DateTime(2027, 6, 13, 8, 0, 0);   // неділя, 8:00
 7
 8  Console.WriteLine($"{appointment1:ddd HH:mm} — робочий час:
    {(appointment1.IsWorkingHours() ? "так" : "ні")}");
 9  Console.WriteLine($"{appointment2:ddd HH:mm} — робочий час:
    {(appointment2.IsWorkingHours() ? "так" : "ні")}");
10  Console.WriteLine($"{appointment3:ddd HH:mm} — робочий час:
    {(appointment3.IsWorkingHours() ? "так" : "ні")}");
11
12  Console.WriteLine($"Днів до прийому: {appointment1.DaysUntil()}");
13
14  // Статичний клас з методами розширення для DateTime
15  public static class DateTimeClinicExtensions
16  {
17      // Пн-Пт, 8:00–18:00
18      public static bool IsWorkingHours(this DateTime dt)
19          => dt.DayOfWeek >= DayOfWeek.Monday
20          && dt.DayOfWeek <= DayOfWeek.Friday
21          && dt.Hour >= 8 && dt.Hour < 18;
22
23      // Кількість днів від сьогодні
24      public static int DaysUntil(this DateTime dt)
25          => (int)(dt.Date - DateTime.Today).TotalDays;
26  }
```

Правила та обмеження

Простір імен. Методи розширення діють у межах простору імен (namespace). Якщо статичний клас із методами розширення знаходиться в іншому namespace, його потрібно підключити через using:

```
C#
1  using ClinicApp.Extensions; // підключаємо простір імен зі статик-класом
2
3  string code = "ICD-10";
4  bool valid = code.IsValidMedicalCode(); // тепер доступний
```

Пріоритет instance-методу. Якщо тип вже має метод із тією самою сигнатурою, що й метод розширення, буде викликаний **рідний метод**

— метод розширення ігнорується. Ця поведінка захищає від випадкового перевизначення поведінки існуючого типу.

Де доцільно використовувати extension methods:

— Додавання утилітарних методів до типів, яких ми не контролюємо (string, DateTime, типи з бібліотек).

— Розширення sealed-класів, від яких не можна успадкуватися.

— Організація допоміжної логіки окремо від основного класу (принцип єдиної відповідальності).

— Побудова fluent API: patient.GetAge().IsAdult().HasInsurance().

Варто знати: весь механізм **LINQ** (Where, Select, OrderBy, GroupBy тощо) реалізований саме через методи розширення для IEnumerable<T>. Коли ви пишете list.Where(p => p.Age > 18), ви викликаєте статичний метод Enumerable.Where(list, p => p.Age > 18) — і це саме метод розширення.

Члени розширення в C# 14

Класичний синтаксис дозволяє додати до типу лише **методи**. C# 14 (.NET 10) вводить **блоки розширення** (extension), у яких можна оголошувати також **властивості** і **статичні члени** розширеного типу:

```
C#
 1  public static class TemperatureExtensions
 2  {
 3      extension(double celsius)            // «приймач» — як this-параметр
 4      {
 5          public bool IsFever() => celsius >= 37.5;            // метод
 6          public bool IsNormalRange => celsius is >= 36.0 and <= 37.0; //
    властивість!
 7      }
 8  }
 9
10  // Використання:
11  double t = 36.6;
12  bool fever  = t.IsFever();   // false
13  bool normal = t.IsNormalRange;   // true — властивість без дужок
```

Для коду, що викликає, нічого не змінюється: це ті самі статичні члени статичного класу, а класичні методи з this і нові блоки сумісні між собою. Перевірено на .NET 10.

Одна пастка, знайдена під час перевірки: властивість спершу хотілося назвати IsNormal, але в double вже є **статичний** метод double.IsNormal(double). Правило пріоритету діє й тут — член самого типу перемагає розширення, тому t.IsNormal компілятор прив'язує до

групи методів double.IsNormal, і рядок не компілюється (CS1503). Звідси ім'я IsNormalRange.

8.7. Часткові класи та методи

Зазвичай клас визначається в одному файлі. Але бувають ситуації, коли один і той самий клас зручно або необхідно розподілити між кількома файлами. Саме для цього у C# існує механізм **часткових класів** (partial classes) — можливість визначати один тип у декількох місцях, а компілятор під час збірки об'єднає всі частини в єдине ціле.

Навіщо потрібні часткові класи

Найважливіший практичний сценарій — **генератори коду**. Коли середовище розробки або фреймворк автоматично генерує частину класу (наприклад, WinForms-дизайнер генерує ініціалізацію UI-компонентів, а Entity Framework — опис таблиць бази даних), зручно тримати цей згенерований код окремо від коду, написаного вручну. При кожному оновленні інструмент перегенеровує «свій» файл, не торкаючись «вашого». Без partial це було б неможливо без постійних конфліктів злиття.

Другий сценарій — **великі класи**, в яких логічно виділити окремі «зони відповідальності»: базові властивості, медична інформація, допоміжні методи. Замість одного файлу на 2000 рядків ми розбиваємо клас на кілька файлів — кожен відповідає за свою ділянку. При цьому з точки зору мови це все ще один тип.

Третій сценарій — **розподіл роботи в команді**: різні розробники можуть паралельно додавати функціональність до одного класу у різних файлах, не отримуючи конфліктів.

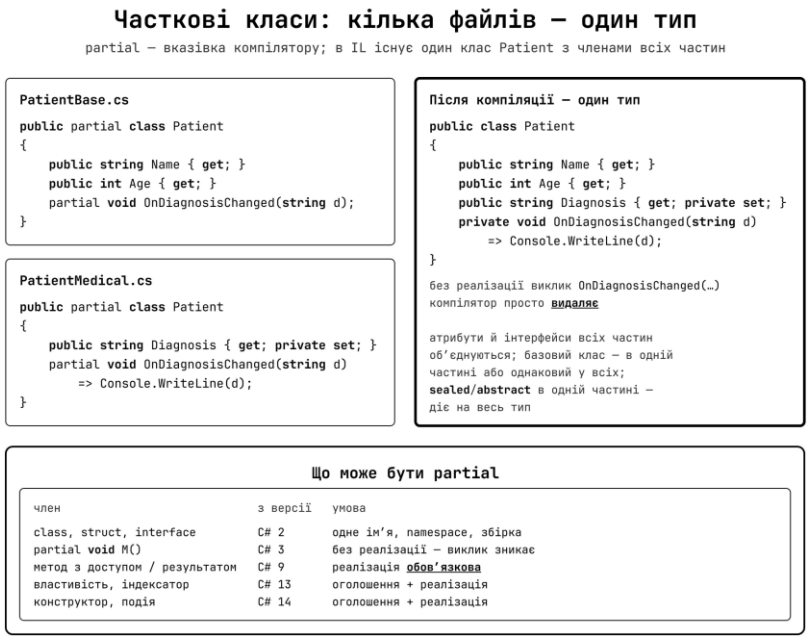
Синтаксис і механізм компіляції

Для визначення часткового класу перед ключовим словом class (або struct, interface, record) додається модифікатор partial:

```
C#
1  public partial class Patient { ... } // у першому файлі
2  public partial class Patient { ... } // у другому файлі
```

Слово partial — це виключно директива для компілятора, а не частина метаданих типу. У скомпільованому IL-коді існує один-єдиний

тип Patient з усіма членами з усіх частин. Жодного накладу у виконанні немає.



Часткові класи: два визначення — один тип після компіляції

Правила сумісності часткових визначень. Всі частини одного partial типу повинні:

— Перебувати в **одному просторі імен** (namespace).

— Перебувати в **одному assembly** (проєкті).

— Мати **однакове ім'я**; модифікатор доступу (public, internal тощо) можна вказати не в усіх частинах, але там, де він указаний, він має збігатися.

— Якщо тип успадковує інший — базовий клас можна вказати лише в одному partial-визначенні, або в усіх, але однаково.

Якщо одна частина оголошує partial class Patient : Person, а інша просто partial class Patient — це допустимо. Але якщо дві частини вказують різні базові класи — помилка компіляції.

Що може бути partial

Модифікатор partial можна застосувати до:

Тип	Синтаксис
Клас	public partial class Patient { }
Структура	public partial struct BloodPressure { }
Інтерфейс	public partial interface IPatientRecord { }
Record (C# 9+)	public partial record Patient(string Name, int Age);

Вкладені типи також можуть бути partial. Серед членів класу partial можуть бути **методи** (C# 3, розширені — C# 9, про них далі), **властивості та індексатори** (C# 13), **конструктори та події** (C# 14). У всіх цих випадках одна частина містить оголошення, а інша — реалізацію. Поля partial бути не можуть. Перевірено на .NET 10: public partial string Name { get; } в одній частині і public partial string Name => _n; в іншій компілюються в одну властивість.

Клінічний приклад: розподіл класу Patient

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  Patient patient = new Patient("Іван Петренко", 45);
 5  patient.Diagnosis = "Гіпертонія II ст.";
 6  patient.PrintBasicInfo();
 7  patient.PrintMedicalInfo();
 8  Console.WriteLine($"Потребує госпіталізації:
    {(patient.RequiresHospitalization() ? "так" : "ні")}");
 9
10  // ── Частина 1: базова інформація (PatientBase.cs) ──
11  public partial class Patient
12  {
13      public string Name { get; }
14      public int Age { get; }
15
16      public Patient(string name, int age)
17      {
18          Name = name;
19          Age = age;
20      }
21
22      public void PrintBasicInfo()
```

```
        => Console.WriteLine($"Пацієнт: {Name}, вік: {Age} р.");
}

// ── Частина 2: медична інформація (PatientMedical.cs) ──
public partial class Patient
{
    public string Diagnosis { get; set; } = "не встановлено";
    public bool IsHighRisk { get; set; }

    public void PrintMedicalInfo()
        => Console.WriteLine($"Діагноз: {Diagnosis} | Група ризику:
{(IsHighRisk ? "висока" : "стандартна")}");

    public bool RequiresHospitalization()
        => IsHighRisk || Diagnosis.Contains("інфаркт") ||
Diagnosis.Contains("інсульт");
}
```

У реальному проєкті ці два partial блоки знаходились би у різних файлах. У runnable-прикладі вони в одному файлі — C# це допускає, оскільки partial є лише підказкою компілятору про дозвіл на розподіл, а не вимогою.

Часткові методи

Часткові класи можуть містити **часткові методи** (partial methods). Ідея полягає в тому, що одна частина класу оголошує **сигнатуру** методу (без тіла), а інша — надає **реалізацію**. Це корисно саме в контексті code generation: генератор може оголосити «точки розширення», а розробник заповнює їх реалізацією.

```
C#
// Частина 1 (згенерована автоматично):
public partial class Patient
{
    partial void OnDiagnosisChanged(string newDiagnosis); // тільки оголошення

    public void SetDiagnosis(string diagnosis)
    {
        Diagnosis = diagnosis;
        OnDiagnosisChanged(diagnosis); // викликаємо — можливо порожній виклик
    }
}

// Частина 2 (написана вручну розробником):
public partial class Patient
{
    partial void OnDiagnosisChanged(string newDiagnosis)
    {
        Console.WriteLine($"Діагноз змінено: {newDiagnosis}");
        // тут можна надіслати сповіщення, зробити логування тощо
    }
}
```

Ключова особливість: якщо реалізація часткового методу **відсутня**, компілятор просто **видаляє виклик** OnDiagnosisChanged(diagnosis) з коду. Жодної помилки компіляції, жодних витрат на виконання — виклик зникає повністю. Це фундаментально відрізняє часткові методи від абстрактних: абстрактний метод без реалізації — помилка компіляції; частковий — тиха відмова.

```C#
using System;

// Виконуваний код
Patient p = new Patient("Марія Коваль");
Console.WriteLine("--- Зміна діагнозу ---");
p.SetDiagnosis("Цукровий діабет II тип");
p.SetDiagnosis("Гіпертонія I ст.");

// ── Частина 1: каркас з оголошенням ──
public partial class Patient
{
    public string Name { get; }
    public string Diagnosis { get; private set; } = "не встановлено";

    public Patient(string name) => Name = name;

    partial void OnDiagnosisChanged(string newDiagnosis);

    public void SetDiagnosis(string diagnosis)
    {
        string previous = Diagnosis;
        Diagnosis = diagnosis;
        OnDiagnosisChanged(diagnosis); // якщо реалізації немає — виклик видаляється
        Console.WriteLine($"[{Name}] Діагноз: {previous} → {Diagnosis}");
    }
}

// ── Частина 2: реалізація ──
public partial class Patient
{
    partial void OnDiagnosisChanged(string newDiagnosis)
    {
        // Логіка, яку розробник додає вручну
        if (newDiagnosis.Contains("діабет") || newDiagnosis.Contains("інфаркт"))
            Console.WriteLine($"  ⚑ Увага: встановлено хронічне захворювання — потрібен план лікування");
    }
}
```

Обмеження базових часткових методів

У базовій формі часткові методи мають низку обмежень. Всі вони пов'язані з однією причиною: компілятор повинен мати можливість **безслідно видалити виклик**, якщо реалізація відсутня.

Обмеження	Причина
Немає модифікатора доступу (за замовчуванням private)	public метод без реалізації — порушення контракту типу
Тип результату — лише void	Не можна видалити виклик, якщо код очікує повернуте значення
Немає out-параметрів	Невизначені out після видаленого виклику — помилка
Немає virtual, override, sealed, new, extern	Ці модифікатори вимагають реальної реалізації у типі

Розширені часткові методи (C# 9+)

Починаючи з C# 9, часткові методи можуть мати **будь-які** модифікатори доступу та будь-який тип результату — за однієї умови: реалізація **обов'язкова**. Такий варіант не може бути «тихо видалений», тому компілятор вимагає наявності тіла.

```
C#
 1  // Оголошення з public і non-void:
 2  public partial class PatientRecord
 3  {
 4      public partial string GenerateSummary(); // C# 9 — реалізація обов'язкова
 5  }
 6
 7  // Реалізація — ОБОВ'ЯЗКОВО в іншій частині:
 8  public partial class PatientRecord
 9  {
10      public partial string GenerateSummary()
11          => $"Пацієнт: {Name}, вік: {Age}, діагноз: {Diagnosis}";
12  }
```

Розширені часткові методи корисні у Source Generators (з'явились разом із C# 9 у .NET 5), де генератор оголошує метод-«заглушку», а розробник надає конкретну реалізацію, яка може повертати значення і бути публічною.

Підсумок: базові vs розширені часткові методи

Характеристика	Базовий partial (C# 3)	Розширений partial (C# 9)
Модифікатор доступу	Тільки private (implicit)	Будь-який
Тип результату	Тільки void	Будь-який
out-параметри	Заборонені	Дозволені
Реалізація	Необов'язкова	**Обов'язкова**
Без реалізації	Виклик видаляється	Помилка компіляції

8.8. Анонімні типи

Іноді нам потрібен об'єкт лише для одного конкретного місця в коді: щоб тимчасово зберегти кілька пов'язаних значень, передати у функцію виведення або зібрати результат вибірки. Створювати повноцінний клас заради одноразового використання — надлишок. Для таких ситуацій у C# існують **анонімні типи** (anonymous types): об'єкти без явно оголошеного класу, де компілятор сам генерує внутрішній тип на льоту.

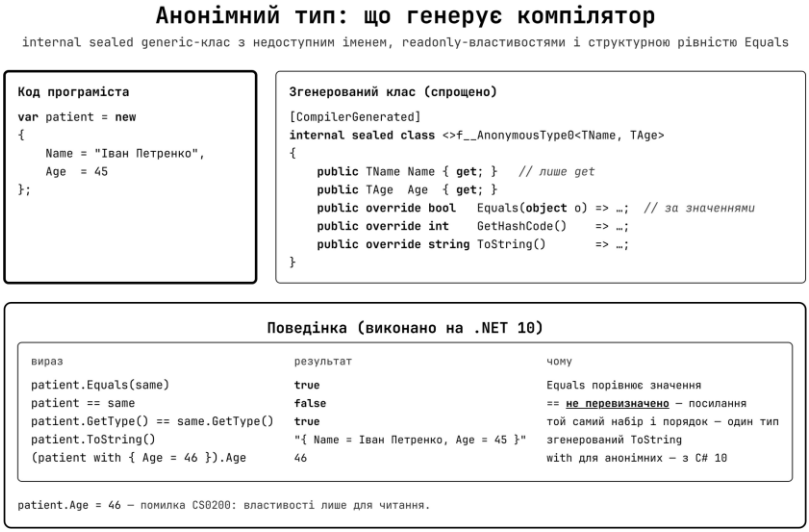
Анонімний тип визначається через var і ініціалізатор об'єкта без вказівки класу:

```
C#
1  var patient = new { Name = "Іван Петренко", Age = 45 };
2  Console.WriteLine(patient.Name); // Іван Петренко
3  Console.WriteLine(patient.Age);  // 45
```

Тут немає жодного класу Patient — компілятор сам порозбирається з типом.

Що насправді генерує компілятор

Анонімний тип — це не магія. Компілятор генерує внутрішній клас з автоматичним іменем і підставляє його скрізь, де використовується var. Цей клас має конкретну структуру:



Анонімний тип: що генерує компілятор

sealed — анонімний тип не можна розширити спадкуванням.

readonly властивості — усі властивості анонімного типу доступні тільки для читання. Після ініціалізації змінити значення неможливо. Це свідомий дизайн: анонімні типи — immutable value containers. Отримати змінену копію можна виразом with (C# 10): patient with { Age = 46 } створює новий анонімний об'єкт.

Автоматичне ім'я — компілятор дає класу ім'я виду <>f__AnonymousType0. Кутові дужки <> — заборонений символ в ідентифікаторах C#. Це навмисно: програміст не може написати це ім'я у коді, що гарантує — ніхто не зможе зберегти посилання на тип у явно типізованій змінній.

Equals і GetHashCode — компілятор перевизначає обидва методи на основі структурного порівняння. Два анонімних об'єкти з однаковими іменами властивостей, типами та значеннями — рівні за Equals. Але оператор == для анонімних типів **не перевизначено**: a == b порівнює посилання і для двох різних об'єктів з однаковими даними повертає false (перевірено на .NET 10). Порівнюйте анонімні об'єкти через Equals.

Узагальнений клас — насправді компілятор генерує generic-клас <>f__AnonymousType0<TName, TAge>, а для кожного набору типів

властивостей використовує його закрите варіювання. Тому два анонімні типи з однаковими іменами властивостей, але різними типами значень, — це різні закриті типи одного generic-шаблону.

ToString — повертає рядок виду { Name = Іван Петренко, Age = 45 } — зручно для діагностики та виведення.

Основний синтаксис та ініціалізатори з проекцією

Крім явного присвоєння значень, анонімні типи підтримують **ініціалізатори з проекцією** (projection initializers): можна передати ідентифікатор або вираз, і ім'я змінної стане ім'ям властивості:

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  // Явний ініціалізатор
 5  var appointment = new
 6  {
 7      PatientName = "Марія Коваль",
 8      DoctorName  = "Олег Петренко",
 9      Date        = "2026-06-10",
10      Room        = 12
11  };
12  Console.WriteLine($"Прийом: {appointment.PatientName} до
    {appointment.DoctorName}");
13  Console.WriteLine($"Дата: {appointment.Date}, кабінет {appointment.Room}");
14
15  // Проекція: ім'я поля = ім'я властивості
16  string diagnosis = "Гіпертонія II ст.";
17  int roomNumber   = 7;
18  var record = new { diagnosis, roomNumber }; // → record.diagnosis,
    record.roomNumber
19  Console.WriteLine($"Діагноз: {record.diagnosis}, палата: {record.roomNumber}");
20
21  // Вкладені анонімні типи
22  var summary = new
23  {
24      Patient = new { Name = "Іван Петренко", Age = 45 },
25      Visit   = new { Date = "2026-06-10", Reason = "Плановий огляд" }
26  };
27  Console.WriteLine($"{summary.Patient.Name} ({summary.Patient.Age} р.) —
    {summary.Visit.Reason}");
```

Рівність анонімних типів

Як вже зазначалось, компілятор генерує структурну рівність. Два об'єкти анонімного типу **рівні**, якщо мають однакові назви властивостей, однакові типи — і однакові значення. Більше того: якщо у програмі визначено кілька анонімних об'єктів з однаковим **набором,**

порядком та типами властивостей, компілятор створить для них **одне** визначення класу:

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  var patient1 = new { Name = "Іван Петренко", Age = 45 };
 5  var patient2 = new { Name = "Марія Коваль",  Age = 32 };
 6  var patient3 = new { Name = "Іван Петренко", Age = 45 }; // те саме що patient1
 7
 8  // Той самий тип? — перевіряємо через GetType
 9  Console.WriteLine($"patient1 тип: {patient1.GetType().Name}");
10  Console.WriteLine($"patient2 тип: {patient2.GetType().Name}");
11  Console.WriteLine($"patient3 тип: {patient3.GetType().Name}");
12  // patient1 і patient2 і patient3 мають ОДНАКОВИЙ тип — бо однакові поля
13
14  // Рівність через Equals (структурна):
15  Console.WriteLine($"patient1 == patient3: {patient1.Equals(patient3)}"); //
    true
16  Console.WriteLine($"patient1 == patient2: {patient1.Equals(patient2)}"); //
    false
17
18  // Об'єкт із ІНШИМ набором полів — ІНШИЙ тип:
19  var extended = new { Name = "Тест", Age = 20, Room = 5 };
20  Console.WriteLine($"extended тип: {extended.GetType().Name}"); // відмінний від
    попередніх
21
22  // ToString — вбудований діагностичний формат:
23  Console.WriteLine(patient1.ToString());
```

Умова спільного визначення: однаковий **набір** властивостей, однаковий **порядок**, однакові **типи**. Якщо хоч щось відрізняється — компілятор створить окремий тип.

Масиви анонімних типів

Масив анонімних типів визначається через new[] — компілятор виведе тип елементів із першого ініціалізатора:

```
C#
1  var patients = new[]
2  {
3      new { Name = "Іван Петренко", Age = 45, Diagnosis = "Гіпертонія" },
4      new { Name = "Марія Коваль",  Age = 32, Diagnosis = "Діабет"     },
5      new { Name = "Олег Сидоренко",Age = 58, Diagnosis = "Аритмія"    },
6  };
7
8  foreach (var p in patients)
9      Console.WriteLine($"{p.Name} ({p.Age} р.) — {p.Diagnosis}");
```

Усі елементи масиву повинні мати однаковий набір властивостей. Якщо хоч один відрізняється — помилка компіляції.

Область видимості та обмеження

Анонімний тип обмежений своєю областю видимості. Через var ми можемо зберегти об'єкт у локальній змінній, але **не можна** явно вказати тип анонімного об'єкта в сигнатурі методу, у полі класу чи в параметрі. Якщо потрібно передати анонімний об'єкт в інший метод, є лише два способи: через object (з втратою типізації) або через dynamic (з відкладеною перевіркою типів).

```
C#
// НЕ можна:
// Patient GetAnonymous() => new { Name = "Іван", Age = 45 }; // помилка

// Можна через object (але доведеться кастити або використовувати рефлексію):
object obj = new { Name = "Іван", Age = 45 };

// Або через dynamic (повільно, без підтримки IDE):
dynamic dyn = new { Name = "Іван", Age = 45 };
Console.WriteLine(dyn.Name);
```

Саме ця обмеженість і визначає, де анонімні типи доречні: **у межах одного методу**, для тимчасового зберігання або перетворення даних.

Readonly. Після створення об'єкта змінити його властивості не можна:

```
C#
var patient = new { Name = "Іван", Age = 45 };
patient.Age = 46; // помилка компіляції — властивість readonly
```

Якщо потрібна змінювана структура — краще використовувати кортеж (8.9) або звичайний клас.

Зв'язок з LINQ

Анонімні типи є основним механізмом для **проекцій у LINQ**. Оператор select часто повертає не вихідний об'єкт, а лише потрібний набір полів — і саме тут анонімні типи незамінні:

```
C#
// LINQ-вибірка лише потрібних полів (буде детально в розділі 13):
var summaries = patients.Select(p => new
{
    p.Name,
    p.Diagnosis,
    IsRisk = p.Age > 50
});
```

Кожен елемент summaries — це анонімний об'єкт з трьома полями. Клас для цього не потрібен — і не потрібен буде ніколи, бо ці об'єкти живуть лише в межах одного запиту.

Коли анонімний тип, а коли щось інше

Потреба	Рекомендація
Тимчасова проекція в межах методу / LINQ-запиту	Анонімний тип
Передача між методами з типізацією	Кортеж (string Name, int Age) (розділ 8.9)
Багаторазово використовуваний іменований тип	record (розділ 8.10) або клас
Потрібна мутабельність (зміна після створення)	Клас зі звичайними властивостями
Потрібне успадкування	Клас

Анонімні типи добре підходять для «одноразових» проекцій, де ім'я типу взагалі не має значення. Якщо ж структура даних потрібна більше ніж в одному місці коду — варто оголосити явний тип.

8.9. Кортежі

Класичне обмеження методу в C# — він може повертати лише одне значення. Щоб повернути два пов'язаних результати (наприклад, мінімум і максимум з масиву вимірювань), раніше доводилось або створювати спеціальний клас-обгортку, або використовувати out-параметри, або повертати масив. Усі ці підходи або зайво громіздкі, або не виразні. **Кортежі** (tuples), додані у C# 7.0, вирішують цю проблему елегантно: вони дозволяють групувати кілька значень в одну структуру без оголошення окремого класу.

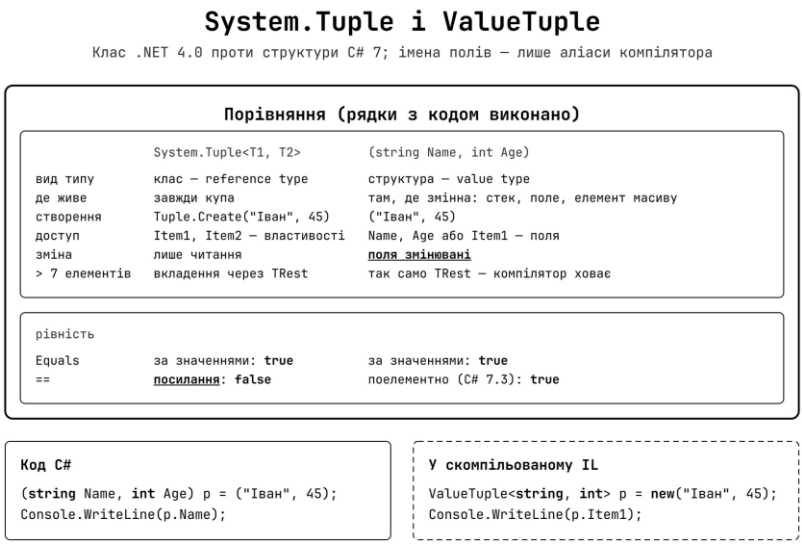
System.Tuple vs ValueTuple: важлива різниця

У .NET існують два різних механізми кортежів, які легко переплутати.

System.Tuple<T1,T2,...> з'явився ще у .NET 4.0. Це звичайний **клас** — тобто reference type. Об'єкт розміщується у heap, а доступ до

елементів — лише через Item1, Item2... Властивості readonly, синтаксис громіздкий: Tuple.Create("Іван", 45). Метод Equals у Tuple перевизначено і порівнює елементи за значеннями, але оператор == — ні, тому Tuple.Create("Іван", 45) == Tuple.Create("Іван", 45) дає false (порівняння посилань), а .Equals(...) — true (перевірено на .NET 10).

System.ValueTuple<T1,T2,...> з'явився у C# 7.0. Це **структура** — value type. Як і будь-яка структура, він зберігається там, де оголошена змінна: локальна змінна — у стеку, поле класу чи елемент масиву — разом з об'єктом у купі. Головне — окремого об'єкта в купі не створюється, тому немає виділень пам'яті і навантаження на збирач сміття. Синтаксис короткий: ("Іван", 45). Поля мутабельні. Рівність — структурна (== порівнює значення). Саме цей варіант і має на увазі слово «кортеж» у сучасному C#.



Кортежі в C#: System.Tuple (старий) vs ValueTuple (новий)

Надалі, говорячи про кортежі, ми маємо на увазі виключно ValueTuple — сучасний та рекомендований варіант.

Синтаксис: оголошення та доступ

Кортеж визначається переліком значень у круглих дужках. Якщо не вказувати назви полів, доступ відбувається через Item1, Item2 і так далі:

```
C#
1  var measurement = (36.6, 72);  // (double, int)
2  Console.WriteLine(measurement.Item1); // 36.6
3  Console.WriteLine(measurement.Item2); // 72
```

Рекомендований підхід — **іменовані поля**: вони роблять код значно зрозумілішим:

```
C#
1  var measurement = (Temperature: 36.6, Pulse: 72);
2  Console.WriteLine(measurement.Temperature); // 36.6
3  Console.WriteLine(measurement.Pulse);       // 72
```

Іменовані поля — це виключно **compile-time аліаси**. Компілятор перетворює measurement.Temperature у measurement.Item1 під час компіляції. У скомпільованому IL-коді Temperature не існує — лише Item1. Це означає, що якщо ви передаєте кортеж у бібліотеку або зберігаєте у object, назви полів там будуть недоступні.

Тип кортежу можна вказати явно:

```
C#
1  (double Temperature, int Pulse) measurement = (36.6, 72);
```

Або визначити псевдонім типу через using (C# 12+):

```
C#
1  using Measurement = (double Temperature, int Pulse);
2  Measurement m = (37.2, 88);
```

Декомпозиція кортежу

Кортеж можна **декомпозувати** — розкласти на окремі змінні. Це особливо зручно при отриманні результату методу:

```
C#
1  var (temperature, pulse) = GetVitals(patientId);
2  Console.WriteLine($"Температура: {temperature}, пульс: {pulse}");
```

Якщо деякі елементи кортежу непотрібні, їх можна пропустити за допомогою **discard** (_):

```
C#
1  var (temperature, _) = GetVitals(patientId); // пульс не потрібен
```

Discard — це не змінна, це явна вказівка компілятору, що значення нас не цікавить. Він не займає пам'яті і не створює змінної.

Основний синтаксис у клінічному контексті

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  // Явний кортеж
 5  (string Name, int Age, double Temp) patient = ("Іван Петренко", 45, 38.2);
 6  Console.WriteLine($"Пацієнт: {patient.Name}, вік: {patient.Age}, t°:
    {patient.Temp}");
 7
 8  // Декомпозиція
 9  var (name, age, temp) = patient;
10  Console.WriteLine($"Після декомпозиції: {name}, {age} р., {temp}°C");
11
12  // Discard: нас цікавить лише ім'я
13  var (patientName, _, _) = patient;
14  Console.WriteLine($"Ім'я: {patientName}");
15
16  // Мутабельність — на відміну від анонімних типів
17  var visit = (Diagnosis: "Гіпертонія", Room: 7);
18  visit.Room = 12; // можна змінити
19  Console.WriteLine($"Діагноз: {visit.Diagnosis}, палата: {visit.Room}");
20
21  // Обмін значеннями через кортеж
22  string doctorA = "Олег Петренко";
23  string doctorB = "Марія Іванова";
24  (doctorA, doctorB) = (doctorB, doctorA);
25  Console.WriteLine($"Після ротації: {doctorA}, {doctorB}");
```

Кортеж як результат методу

Найцінніше застосування кортежів — **повернення кількох значень з методу**. До C# 7 для цього використовували out-параметри або окремі класи. Тепер можна описати результат прямо в сигнатурі:

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  double[] readings = { 36.6, 37.1, 38.5, 37.8, 36.9, 39.2, 37.4 };
 5
 6  var stats = AnalyzeReadings(readings);
 7  Console.WriteLine($"Мін: {stats.Min:F1}°C");
 8  Console.WriteLine($"Макс: {stats.Max:F1}°C");
 9  Console.WriteLine($"Середнє: {stats.Average:F1}°C");
10  Console.WriteLine($"Кількість вимірювань з жаром: {stats.FeverCount}");
11
```

```
// Декомпозиція без збереження всього кортежу:
var (min, max, avg, _) = AnalyzeReadings(readings);
Console.WriteLine($"Діапазон: {min:F1}–{max:F1}°C");

// Метод повертає іменований кортеж
(double Min, double Max, double Average, int FeverCount)
    AnalyzeReadings(double[] values)
{
    double min = values[0], max = values[0], sum = 0;
    int feverCount = 0;
    foreach (var v in values)
    {
        if (v < min) min = v;
        if (v > max) max = v;
        sum += v;
        if (v > 37.5) feverCount++;
    }
    return (min, max, sum / values.Length, feverCount);
}
```

Іменований кортеж у сигнатурі методу виконує роль «легкого DTO»: він описує структуру результату прямо там, де метод визначений, без необхідності оголошувати окремий клас. Але на відміну від анонімного типу — може бути явно типізований і переданий між методами.

Кортеж як параметр методу

Кортеж можна передати і як параметр:

C#
```
using System;

// Виконуваний код
PrintVitals(("Іван Петренко", 38.2, 90));
PrintVitals(("Марія Коваль",  36.7, 72));

// Сортування масиву через tuple-swap
int[] pulseReadings = { 88, 62, 95, 71, 83, 55, 79 };
for (int i = 0; i < pulseReadings.Length - 1; i++)
    for (int j = i + 1; j < pulseReadings.Length; j++)
        if (pulseReadings[i] > pulseReadings[j])
            (pulseReadings[i], pulseReadings[j]) = (pulseReadings[j],
pulseReadings[i]);

Console.WriteLine("Відсортований пульс:");
foreach (var p in pulseReadings)
    Console.WriteLine($"  {p} уд/хв");

// Метод із кортежем як параметром
void PrintVitals((string Name, double Temp, int Pulse) vitals)
{
    string status = vitals.Temp > 37.5 ? " [жар]" : "";
    Console.WriteLine($"{vitals.Name}: {vitals.Temp:F1}°C, {vitals.Pulse}
уд/хв{status}");
```

```
23  }
```

Рівність кортежів

ValueTuple підтримує структурну рівність через ==: два кортежі рівні, якщо рівні всі їхні елементи в тому самому порядку. Назви полів при порівнянні не враховуються.

```
C#
1  var a = (Name: "Іван", Age: 45);
2  var b = (PatientName: "Іван", Years: 45); // інші назви — але однакові типи і
   значення
3  Console.WriteLine(a == b); // true — назви полів в IL відсутні
```

Коли кортеж, а коли щось інше

Потреба	**Інструмент**
Повернути 2–4 пов'язаних значення з методу	**Кортеж** (T1, T2, ...)
Тимчасова проекція тільки в межах методу	Анонімний тип (8.8)
Тип потрібний в кількох місцях, має поведінку	record (8.10) або клас
Потрібне успадкування або складний конструктор	Клас
Просто обміняти два значення	Кортеж (a, b) = (b, a)

Головна перевага кортежу перед анонімним типом — **можна вказати тип явно** і передати між методами. Головна перевага перед класом — не треба оголошувати тип, якщо структура використовується лише локально.

8.10. Records

У більшості програм існують об'єкти двох принципово різних призначень. Одні — **сутності з ідентичністю**: конкретний пацієнт, конкретний лікар. Їх ідентифікують не за значеннями полів, а за посиланням на конкретний об'єкт у пам'яті. Два різні пацієнти з

однаковим іменем — це різні люди. Для таких об'єктів добре підходить звичайний клас з посилальною рівністю.

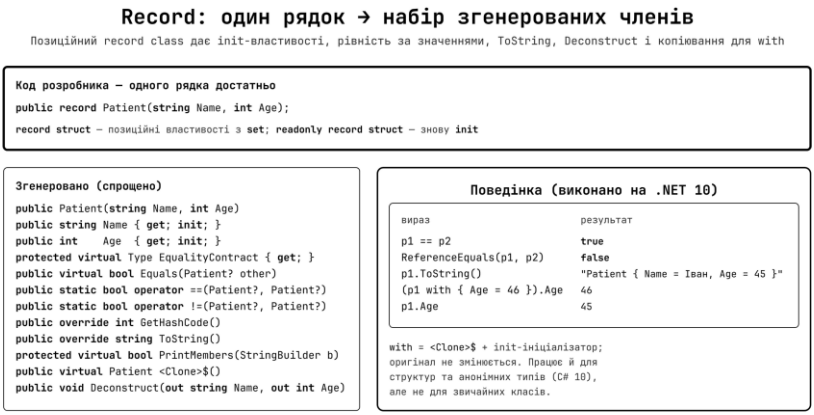
Інші — **контейнери значень** (value objects): діагноз, дата прийому, результат аналізу, адреса, медичний запис. Тут два об'єкти з однаковим вмістом семантично рівні. «Діагноз "гіпертонія"» і ще один «діагноз "гіпертонія"» — це одне й те саме. Для таких об'єктів ми хочемо **структурну рівність** (як у string) і незмінність після створення. Раніше для цього доводилось вручну перевизначати Equals, GetHashCode, ToString та реалізовувати operator ==. **Records** (C# 9) вирішують цю задачу: компілятор генерує весь цей код автоматично.

Що компілятор генерує автоматично

Один рядок позиційного record:

```
C#
1  public record Patient(string Name, int Age);
```

замінює близько 50 рядків класу з усіма необхідними членами.



Record: один рядок коду ⟶ що генерує компілятор

Конструктор — приймає всі параметри позиційного запису.

init-властивості — автоматично генеруються з readonly-семантикою: public string Name { get; init; }. Встановити можна тільки при конструюванні або в ініціалізаторі об'єкта — після цього значення зафіксоване.

Структурна рівність — Equals, ==, != порівнюють за значеннями всіх властивостей, а не за посиланням.

GetHashCode — генерується на основі значень властивостей, узгоджений з Equals.

ToString — виводить Patient { Name = Іван, Age = 45 } — зручно для логування та діагностики.

Deconstruct — дозволяє декомпозицію var (name, age) = patient;.

<Clone>$() — метод копіювання для підтримки оператора with. Ім'я з символами <> і $ недоступне з C#, тому викликати його напряму не можна. Крім того, компілятор генерує EqualityContract і PrintMembers (про них нижче).

Модифікатор init

init — це особливий тип сеттера, введений у C# 9. Властивість з init можна встановити **тільки** в двох місцях:

— у конструкторі;

— в ініціалізаторі об'єкта (new Patient { Name = "Іван" }).

Після виходу з конструктора або ініціалізатора значення «замерзає» — будь-яка подальша спроба присвоєння спричиняє помилку компіляції. Це принципово відрізняється від set (можна змінити будь-коли) і від readonly-поля (можна встановити тільки в конструкторі, але не в ініціалізаторі).

```C#
public record Patient
{
    public string Name { get; init; } = "";
    public int Age    { get; init; }
}

var p = new Patient { Name = "Іван", Age = 45 }; // OK — ініціалізатор
p.Name = "Марія"; // помилка компіляції — init вже завершено
```

Синтаксис: повний та позиційний

Повний синтаксис — схожий на клас, але з ключовим словом record:

```C#
public record Patient
{
    public string Name { get; init; }
    public int Age     { get; init; }

    public Patient(string name, int age)
    {
```

```
 8          Name = name;
 9          Age = age;
10      }
11  }
```

Позиційний синтаксис — вся інформація в одному рядку:

```
C#
1  public record Patient(string Name, int Age);
```

Компілятор сам створює конструктор, init-властивості та Deconstruct. Позиційний синтаксис — це не скорочення повного, це повноцінна альтернатива. Якщо потрібні додаткові члени — їх можна дописати у тілі:

```
C#
1  public record Patient(string Name, int Age)
2  {
3      public bool IsAdult => Age >= 18;
4      public string Diagnosis { get; set; } = ""; // звичайна мутабельна
   властивість
5  }
```

Оператор with

with — оператор немутабельного оновлення. Він створює **копію** record зі зміненими вказаними властивостями:

```
C#
1  var original = new Patient("Іван Петренко", 45);
2  var updated  = original with { Age = 46 }; // новий об'єкт; Name = "Іван
   Петренко"
```

original при цьому не змінюється. Це ключова особливість: замість того, щоб мутувати об'єкт, ми отримуємо новий з оновленими даними. Такий підхід — «незмінні дані + нові версії» — є основою функціонального програмування і безпечний при роботі з кількома потоками.

Внутрішньо with для record class викликає <Clone>$() і потім встановлює init-властивості через ініціалізатор. З C# 10 with працює також з будь-якими **структурами** (у тому числі record struct і кортежами ValueTuple) та **анонімними типами** — для них копія створюється простим копіюванням значення. Для звичайних класів with недоступний.

Структурна рівність

Для record компілятор генерує рівність за значеннями, а не за посиланням. Два різні об'єкти з однаковими даними рівні:

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  var p1 = new Patient("Іван Петренко", 45);
 5  var p2 = new Patient("Іван Петренко", 45); // різні об'єкти, однакові дані
 6  var p3 = new Patient("Марія Коваль",  32);
 7
 8  Console.WriteLine($"p1 == p2: {p1 == p2}");      // true
 9  Console.WriteLine($"p1 == p3: {p1 == p3}");      // false
10  Console.WriteLine($"Equals:   {p1.Equals(p2)}"); // true
11
12  // with — незмінне оновлення
13  var updated = p1 with { Age = 46 };
14  Console.WriteLine($"p1:      {p1}");     // вихідний не змінився
15  Console.WriteLine($"updated: {updated}"); // новий з Age=46
16
17  // Deconstruct
18  var (name, age) = p1;
19  Console.WriteLine($"Декомпозиція: {name}, {age}");
20
21  // record
22  public record Patient(string Name, int Age);
```

Порівняємо з класом:

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  // record — структурна рівність
 5  var r1 = new MedicalRecord("Гіпертонія II ст.", "2026-06-10");
 6  var r2 = new MedicalRecord("Гіпертонія II ст.", "2026-06-10");
 7  Console.WriteLine($"record r1 == r2: {r1 == r2}");  // true
 8
 9  // class — посилальна рівність (різні об'єкти — не рівні)
10  var c1 = new AppointmentClass("Гіпертонія II ст.", "2026-06-10");
11  var c2 = new AppointmentClass("Гіпертонія II ст.", "2026-06-10");
12  Console.WriteLine($"class  c1 == c2: {c1 == c2}");  // false
13
14  // ToString: record виводить вміст автоматично
15  Console.WriteLine(r1.ToString()); // MedicalRecord { Diagnosis = ..., Date =
    ... }
16
17  // record
18  public record MedicalRecord(string Diagnosis, string Date);
19
20  // class — без перевизначення Equals
21  public class AppointmentClass
22  {
23      public string Diagnosis { get; }
24      public string Date { get; }
```

```
25      public AppointmentClass(string d, string date) { Diagnosis = d; Date =
    date; }
26  }
```

Успадкування records

Record-класи можна успадковувати так само, як звичайні класи. Але є важлива деталь: при рівності враховується **фактичний тип** об'єкта. Компілятор генерує захищену властивість EqualityContract, яка повертає typeof(Patient), typeof(Doctor) тощо. При порівнянні типи повинні збігатися.

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  var person    = new MedicalPerson("Іван Петренко", 45);
 5  var patient   = new ClinicPatient("Іван Петренко", 45, "Гіпертонія");
 6  var patient2  = new ClinicPatient("Іван Петренко", 45, "Гіпертонія");
 7
 8  // Рівність record враховує тип!
 9  Console.WriteLine($"person == patient:  {person.Equals(patient)}");   // false
    – різні типи
10  Console.WriteLine($"patient == patient2: {patient.Equals(patient2)}"); // true
    – однаковий тип і дані
11
12  Console.WriteLine(person);  // MedicalPerson { Name = Іван Петренко, Age = 45 }
13  Console.WriteLine(patient); // ClinicPatient { Name = ..., Age = ..., Diagnosis
    = ... }
14
15  // with працює і для похідних
16  var updated = patient with { Diagnosis = "Гіпертонія I ст." };
17  Console.WriteLine(updated);
18
19  // Records
20  public record MedicalPerson(string Name, int Age);
21  public record ClinicPatient(string Name, int Age, string Diagnosis) :
    MedicalPerson(Name, Age);
```

EqualityContract — це зв'язуюча ланка, яка гарантує: Person { Name="Tom", Age=37 } та Employee { Name="Tom", Age=37 } не рівні навіть при однакових значеннях полів. Це правильна поведінка: різні типи — різна ідентичність.

record struct vs record class

Починаючи з C# 10, тип record можна визначити і як структуру:

```
C#
1  public record struct BloodPressure(int Systolic, int Diastolic);
```

Ключова відмінність від record class:

Характеристика	record class	record struct	readonly record struct
Тип даних	reference (купа)	value (там, де змінна)	value (там, де змінна)
Null	може бути null	ні	ні
Мутабельність (позиційні)	init — immutable	set — **mutable**	init — immutable
Успадкування	так	ні	ні
with	так	так	так
Структурна рівність	так	так	так

Важлива різниця: для **позиційного** record struct компілятор генерує звичайні set-сеттери (мутабельні), а для record class — init-сеттери (immutable). Щоб зробити record struct незмінним, потрібне readonly:

```
C#
1  public readonly record struct BloodPressure(int Systolic, int Diastolic);
```

record struct доречний для невеликих value objects, які часто копіюються: точки, координати, вимірювання — там де value type дає виграш у продуктивності.

Підсумкова таблиця: record vs інші підходи

	record	Анонімний тип	Кортеж	Клас
Іменований тип	✓	✗	✓ (частково)	✓
Передача між методами	✓	✗ (тільки object)	✓	✓
Структурна рівність	✓ (авто)	✓ (авто)	✓ (авто)	✗ (потрібно визначити)

	record	Анонімний тип	Кортеж	Клас
Immutable	✓ (init)	✓	✗ (мутабельні)	✓ (потрібно визначити)
with	✓	✓ (C# 10)	✓ (C# 10, структура)	✗
Успадкування	✓ (class)	✗	✗	✓
ToString (стан)	✓ (авто)	✓ (авто)	✓ (авто)	✗ (потрібно визначити)
Boilerplate	мінімальний	немає	немає	максимальний

Records — ідеальний інструмент для **DTO** (Data Transfer Objects), **value objects** у доменній моделі, **результатів запитів**, **конфігураційних об'єктів**: скрізь, де об'єкт несе дані і ідентифікується за своїм вмістом, а не за посиланням на конкретний екземпляр.

Питання для самоконтролю — Розділ 8. Додаткові можливості ООП у C#

8.1. Визначення операторів

1. Що таке перевантаження операторів у C#? Яку проблему воно вирішує? Поясніть на прикладі, чому код if (tempAfter > tempBefore) читабельніший за if (tempAfter.Celsius > tempBefore.Celsius).
2. Опишіть механізм роботи перевантаженого оператора: що робить компілятор, коли зустрічає вираз t1 + delta, де t1 — об'єкт класу BodyTemperature? Яку сигнатуру шукає компілятор?
3. Які обов'язкові вимоги до методу-оператора? Назвіть усі: модифікатори, ключові слова, вимоги до параметрів, різниця між унарними і бінарними операторами.
4. Знайдіть та поясніть помилку у наступному коді:

```
C#
 1  class Temperature
 2  {
```

```
    public double Value { get; }
    public Temperature(double v) => Value = v;

    public static Temperature operator +(Temperature t1, Temperature t2)
        => new Temperature(t1.Value + t2.Value);

    public static bool operator >(Temperature t1, Temperature t2)
        => t1.Value > t2.Value;
}
```

5. Що означає «парні оператори» у C#? Які пари компілятор вимагає визначати разом? Що трапиться, якщо визначити > без <? Чому це обмеження введено?
6. При перевантаженні == компілятор рекомендує також перевизначити Equals(object) і GetHashCode(). Чому? Яка непослідовна поведінка виникає, якщо == порівнює значення, а Equals() — посилання?
7. Оператор t1 + 0.5 компілюється успішно, а 0.5 + t1 — ні. Чому? Що потрібно зробити, щоб обидві форми працювали?
8. Коли варто перевантажувати оператори, а коли це є антипатерном? Наведіть по два приклади доречного і недоречного перевантаження у контексті реальної предметної галузі.

8.2. Визначення інкременту та декременту

9. Чим визначення операторів ++ і -- для класу відрізняється від визначення арифметичних операторів? Скільки параметрів приймає оператор ++? Яку кількість перевантажених варіантів потрібно написати для підтримки і t++, і ++t?
10. Чому наступний підхід є неправильним? Поясніть обидві причини:

```
C#
public static BodyTemperature operator ++(BodyTemperature t)
{
    t.Celsius += 0.1; // безпосередня мутація параметра
    return t;
}
```

11. Що таке «immutable update pattern» при визначенні операторів? Чому він є стандартом у C#? Напишіть правильну реалізацію ++ для класу BodyTemperature, що зберігає температуру в градусах Цельсія.

12. Поясніть різницю між постфіксним t++ і префіксним ++t. Хоча ми визначаємо один метод — компілятор розрізняє їх поведінку. Що саме відрізняється? Що виведе наступний код:

```
C#
1  BodyTemperature t1 = new BodyTemperature(36.6);
2  BodyTemperature t2 = t1++;
3  Console.WriteLine($"t1={t1.Celsius:F1}, t2={t2.Celsius:F1}");
```

13. Коли різниця між t++ і ++t має практичне значення, а коли — ні? Наведіть конкретний приклад, де вибір між ними впливає на результат.
14. Що таке оператори true і false у C#? Навіщо вони потрібні? Що відбудеться після їх визначення, коли ви пишете if (device) або if (!device)?
15. Оператори true і false завжди визначаються парою. Чому? Що відбувається при спробі визначити тільки один із них?
16. Спроєктуйте клас BloodPressure з полями Systolic і Diastolic. Реалізуйте оператори ++ і -- (збільшення/зменшення систолічного на 1), а також true/false (норма — якщо систолічний до 120 і діастолічний до 80). Покажіть використання у if-умові.

8.3. Перевантаження операцій перетворення типів

17. Що таке перевантаження операторів перетворення типів у C#? Яку проблему воно вирішує? Наведіть приклад, коли без нього код стає громіздким.
18. Поясніть різницю між implicit і explicit оператором перетворення. Коли кожен з них доречний? Яку відповідальність несе програміст, коли пише явний cast?
19. Які обов'язкові вимоги до оператора перетворення: модифікатори, ключові слова, обмеження на типи параметра і результату? Чому не можна визначити перетворення між двома чужими типами в третьому класі?
20. Знайдіть помилку та поясніть:

```
C#
1  class Celsius { public double Value { get; } public Celsius(double v) => Value =
   v; }
2  class Fahrenheit { public double Value { get; } public Fahrenheit(double v) =>
   Value = v; }
3  class Converter
```

```
4  {
5      public static implicit operator Celsius(Fahrenheit f)
6          => new Celsius((f.Value - 32) * 5.0 / 9.0);
7  }
```

21. Напишіть клас Pulse (пульс у ударах на хвилину). Визначте implicit-перетворення з int у Pulse і explicit-перетворення з Pulse у int. Поясніть свій вибір напрямів.
22. За яким критерієм обирати implicit або explicit? Заповніть таблицю: «Можлива втрата даних?», «Задіяна нетривіальна формула?», «Перетворення між різними одиницями?» — і вкажіть для кожного рядка, який оператор доречніший.
23. Що виведе наступний код? Поясніть, яке перетворення відбувається і коли:

```
C#
1  class Kg { public double Value { get; }
2             public Kg(double v) => Value = v;
3             public static implicit operator Kg(double v) => new Kg(v); }
4  Kg weight = 72.5;
5  Console.WriteLine(weight.Value);
```

24. Чи можна визначити два оператори перетворення у протилежних напрямах між одними й тими самими типами (наприклад, BodyTemperature ↔ double)? Чи є якісь обмеження на комбінацію implicit у одному напрямі і explicit в іншому?

8.4. Індексатори

25. Що таке індексатор у C#? Чим він схожий на властивість і чим від неї відрізняється? Що використовується замість імені властивості в синтаксисі індексатора?
26. Запишіть синтаксис числового індексатора з блоками get і set для класу Ward, що дає доступ до пацієнтів за позицією. Яку перевірку доречно додати в get і set?
27. Чим рядковий індексатор відрізняється від числового? Наведіть практичний приклад, де рядковий індексатор є більш виразним рішенням, ніж числовий або набір окремих властивостей.
28. Що виведе наступний код? Чи є помилка у використанні індексатора?

```
C#
 1  class Config
 2  {
```

```
    private string[] _items = { "alpha", "beta", "gamma" };
    public string this[int i]
    {
        get => _items[i];
        set => _items[i] = value;
    }
}
var cfg = new Config();
cfg[1] = "delta";
Console.WriteLine(cfg[0] + " " + cfg[1]);
```

29. Як визначити індексатор лише для читання (readonly)? Коли це доречно? Напишіть клас ImmutableRecord, де індексатор надає доступ до даних за числовим індексом, але не дозволяє їх змінювати.
30. Чи можна перевантажити індексатор — визначити кілька індексаторів з різними типами або кількістю параметрів? Що визначає, який саме індексатор викликається? Наведіть приклад.
31. Що таке двопараметровий індексатор? Де і коли його доречно застосовувати? Запишіть синтаксис і покажіть приклад із розкладом прийомів schedule[day, hour].
32. Порівняйте індексатор і словник Dictionary<TKey, TValue>: обидва надають доступ за ключем. Коли краще використовувати власний клас з індексатором, а коли достатньо словника?

8.5. Змінні-посилання та повернення посилання

33. Що таке ref-змінна (ref local)? Чим вона відрізняється від звичайного присвоєння? Що означає термін «псевдонім» (alias) у цьому контексті?
34. Що виведе наступний код? Поясніть крок за кроком:

```
C#
double x = 10.0;
ref double r = ref x;
r = 20.0;
Console.WriteLine(x);
x = 30.0;
Console.WriteLine(r);
```

35. Чи можна оголосити ref-змінну без ініціалізації (ref double r;)? Що відбудеться? Яке значення обов'язково надається при оголошенні ref-змінної?

36. Що таке ref return? Яку практичну перевагу дає повернення посилання замість значення? Яка синтаксична відмінність у сигнатурі методу і у виклику?
37. Напишіть метод ref double FindFirst(double[] arr, double value), що повертає посилання на перший елемент масиву, рівний value. Покажіть, як через повернене посилання змінити цей елемент безпосередньо в масиві.
38. Що таке ref readonly? Яку мету воно переслідує? Чим відрізняється від звичайного ref?
39. Перелічіть обмеження методів із ref return: що не можна повертати як посилання і чому? Чому не можна повертати посилання на локальну змінну методу?
40. Коли ref local і ref return є обґрунтованими інструментами, а коли вони лише ускладнюють код? Назвіть конкретний сценарій, де ref return дає реальний виграш у порівнянні зі звичайним поверненням значення.

8.6. Методи розширення

41. Що таке метод розширення у C#? Яку практичну проблему він вирішує? Наведіть приклад: коли метод розширення є єдиним варіантом, а успадкування — неможливим.
42. Опишіть синтаксис методу розширення: де він визначається, які модифікатори потрібні, що означає this перед першим параметром?
43. Що насправді відбувається при виклику 38.5.IsFever()? Як компілятор транслює цей виклик? Поясніть, що мають на увазі терміном «синтаксичний цукор».
44. Що виведе наступний код? Поясніть, що відбувається при конфлікті між методом розширення і власним методом типу:

```
C#
1  public static class Ext { public static string Info(this string s) =>
   "extension"; }
2  // string вже має метод, але не 'Info' — в реальному коді, але уявімо:
3  string s = "hello";
4  Console.WriteLine(s.Info());
```

45. Напишіть статичний клас DateTimeExtensions з методом розширення для DateTime, що визначає, чи є дата вихідним днем (IsWeekend()). Покажіть виклик.

46. Поясніть: якщо методи розширення для типу визначені в просторі імен MyApp.Extensions, а у файлі використовується using MyApp, чи будуть вони доступні? Що потрібно вказати?
47. LINQ-методи (Where, Select, OrderBy) реалізовані як методи розширення. Поясніть: для якого типу вони є розширеннями? Що це означає для будь-якого класу, що реалізує IEnumerable<T>?
48. Порівняйте методи розширення і статичні методи-утиліти (helper methods): обидва досягають схожого результату. Яка головна перевага методів розширення з точки зору читабельності та стилю fluent API?

8.7. Часткові класи та методи

49. Що таке частковий клас (partial class)? Яке ключове слово використовується? Чи існують два «об'єкти» або один? Що відбувається з усіма частинами під час компіляції?
50. Назвіть три практичних сценарії використання часткових класів. Який із них є найважливішим у промисловій розробці (підказка: генератори коду)?
51. Які правила сумісності повинні виконуватися для всіх частин одного partial типу? Що станеться, якщо одна частина оголошує public partial class Patient, а інша — internal partial class Patient?
52. Що таке частковий метод (partial method)? Яку роль відіграє частина з оголошенням і яку — частина з реалізацією? Що відбувається з викликом часткового методу, якщо реалізація відсутня?
53. Визначте обмеження базових часткових методів (C# 3). Чому кожне з обмежень пов'язане з можливістю компілятора «видалити виклик»: відсутність public, тільки void, заборона out?
54. Знайдіть помилку у наступному коді:

```C#
public partial class Report
{
    partial string GenerateSummary(); // C# 3 стиль
}
public partial class Report
{
    partial string GenerateSummary() => "Summary text";
}
```

55. Чим розширені часткові методи (C# 9) відрізняються від базових? Які нові можливості вони надають і яку обов'язкову умову вводять натомість?
56. Спроєктуйте частковий клас PatientRecord у двох частинах: перша зберігає базові дані (ім'я, вік, id) і оголошує partial void OnDataUpdated(string field), друга — медичні дані (діагноз, алергії) і надає реалізацію OnDataUpdated, що логує зміну в консоль.

8.8. Анонімні типи

57. Що таке анонімний тип у C#? Де і навіщо він використовується? Який синтаксис для його визначення? Чому обов'язково потрібне var?
58. Що компілятор генерує для анонімного типу «під капотом»: клас, модифікатор, властивості, методи? Чому ім'я генерованого класу містить символи <>, і що це практично означає?
59. Чому властивості анонімних типів є readonly (тільки для читання)? Що відбудеться при спробі змінити значення після ініціалізації? Яку альтернативу слід використовувати, якщо потрібна мутабельність?
60. Що виведе наступний код і чому? Поясніть механізм структурної рівності анонімних типів:

```
C#
1  var a = new { Name = "Іван", Age = 45 };
2  var b = new { Name = "Іван", Age = 45 };
3  var c = new { Age = 45, Name = "Іван" }; // інший порядок!
4  Console.WriteLine(a.Equals(b));
5  Console.WriteLine(a.Equals(c));
6  Console.WriteLine(a.GetType() == b.GetType());
7  Console.WriteLine(a.GetType() == c.GetType());
```

61. Поясніть «ініціалізатор з проекцією» (projection initializer): що означає new { diagnosis, roomNumber } замість new { Diagnosis = diagnosis, RoomNumber = roomNumber }? Звідки беруться імена властивостей?
62. Чому анонімний тип не можна передати як параметр методу або повернути як результат методу зі збереженням типізації? Які два «обходи» існують і які їхні недоліки?

63. Де анонімні типи є незамінними? Поясніть їх роль у LINQ-запитах, зокрема в операторі select. Чому саме тут анонімний тип краще, ніж повноцінний клас?
64. Порівняйте анонімний тип, кортеж і record: коли кожен з них є правильним вибором? Складіть таблицю або опишіть по 1–2 конкретних сценарії для кожного.

8.9. Кортежі

65. Яку проблему вирішують кортежі у C#? Які альтернативи існували до C# 7.0 і чим вони незручні? Наведіть типовий сценарій, де кортеж є найлаконічнішим рішенням.
66. Поясніть принципову різницю між System.Tuple<T1, T2> і System.ValueTuple<T1, T2>: де розміщується в пам'яті кожен, яка рівність, чи є мутабельність, як виглядає синтаксис?
67. Що таке «іменовані поля» кортежу? Чи існують вони у скомпільованому IL-коді? Що це означає при передачі кортежу в зовнішню бібліотеку або збереженні у object?
68. Що виведе наступний код? Поясніть, як відбувається структурна рівність і чому назви полів не враховуються:

```
C#
1  var a = (Name: "Іван", Age: 45);
2  var b = (PatientName: "Іван", Years: 45);
3  Console.WriteLine(a == b);
```

69. Що таке декомпозиція кортежу? Що означає discard (_)? Напишіть метод, що повертає (double Min, double Max, double Average), і покажіть два варіанти його виклику: зі збереженням усіх значень і з відкиданням непотрібного.
70. Порівняйте out-параметри і кортежі для повернення кількох значень. Який підхід є кращим у сучасному C# і чому?
71. Кортежі є мутабельними (на відміну від анонімних типів). Покажіть трюк обміну двох змінних через кортеж (a, b) = (b, a). Чому цей запис є лаконічнішим за традиційний обмін через тимчасову змінну?
72. Спроєктуйте метод AnalyzeBloodPressure(int[] systolicReadings), що повертає іменований кортеж (int Min, int Max, double Average, bool HasHypertension). Покажіть виклик методу з декомпозицією та відкиданням непотрібного поля.

8.10. Records

73. Що таке record у C# 9? Яку проблему він вирішує порівняно зі звичайним класом? Назвіть три–чотири члени, які компілятор генерує автоматично для record.
74. Порівняйте рівність для record і для звичайного класу. Що виведе наступний код і чому?

```
C#
1  public record Patient(string Name, int Age);
2  public class Doctor { public string Name { get; } public int Age { get; }
3                        public Doctor(string n, int a) { Name = n; Age = a; } }
4  var p1 = new Patient("Іван", 45); var p2 = new Patient("Іван", 45);
5  var d1 = new Doctor("Іван", 45);  var d2 = new Doctor("Іван", 45);
6  Console.WriteLine(p1 == p2);
7  Console.WriteLine(d1 == d2);
```

75. Що таке модифікатор init? Чим він відрізняється від set і від readonly-поля? Які два місця у коді дозволяють встановити init-властивість?
76. Поясніть оператор with. Що він робить? Чому він важливий для незмінних об'єктів? Що відбувається з оригінальним об'єктом після var updated = original with { Age = 46 }?
77. Є ієрархія records: MedicalPerson(Name, Age) і ClinicPatient(Name, Age, Diagnosis) : MedicalPerson. Чому new MedicalPerson("Іван", 45).Equals(new ClinicPatient("Іван", 45, "Норма")) поверне false, хоча поля Name і Age однакові? Що таке EqualityContract?
78. Порівняйте позиційний і повний синтаксис record. Яка різниця між ними? Коли потрібно обирати повний синтаксис? Покажіть, як до позиційного record додати обчислювану властивість і звичайну мутабельну властивість.
79. Поясніть різницю між record class, record struct і readonly record struct. Де розміщується кожен у пам'яті? Яка різниця у мутабельності позиційних властивостей між record struct і record class?
80. Спроєктуйте record DiagnosisRecord(string Code, string Description, DateTime Date) для медичної системи. Покажіть: структурну рівність двох однакових записів, операцію with для оновлення дати, декомпозицію через var (code, desc, date) = record, та автоматичний ToString.

Розділ 9. Pattern matching

9.1. Патерн типів

Що таке pattern matching

Pattern matching (зіставлення із зразком) — це механізм, який дозволяє перевірити значення на відповідність певному **зразку** (pattern) і, якщо перевірка успішна, виконати відповідну дію або отримати результат. Зразок може описувати тип, конкретне значення, структуру об'єкта, діапазон, логічну комбінацію умов тощо.

До появи pattern matching у C# 7 (і його розширень у 8, 9, 10, 11) програмісти виконували перевірки вручну через ланцюги if-else, оператори as і явні приведення типів. Ця система працювала, але була багатослівною, схильною до помилок і погано читалась. Pattern matching замінює її компактним, виразним і типобезпечним синтаксисом.

У розділі 9 ми розглянемо всі ключові типи зразків: типів, властивостей, кортежів, позиційних, реляційних, логічних і списків.

Патерн типів (type pattern)

Найпростіший і найбільш використовуваний різновид — **патерн типів**. Він перевіряє, чи є значення екземпляром певного типу, і якщо так — одразу прив'язує його до нової змінної потрібного типу:

```
1  значення  is  Тип  змінна
```

Три дії в одному виразі: перевірка, приведення і прив'язка. Якщо перевірка не проходить — змінна **не отримує значення** (вона оголошена, але не присвоєна), і виконання йде в гілку else. Формально запис staff is Doctor doc у специфікації називається **патерном оголошення** (declaration pattern, C# 7); сам «патерн типу» без змінної — Doctor => … у switch expression — з'явився в C# 9. Нижче обидва називаємо патерном типів, як це прийнято на практиці.

Стара форма vs pattern matching

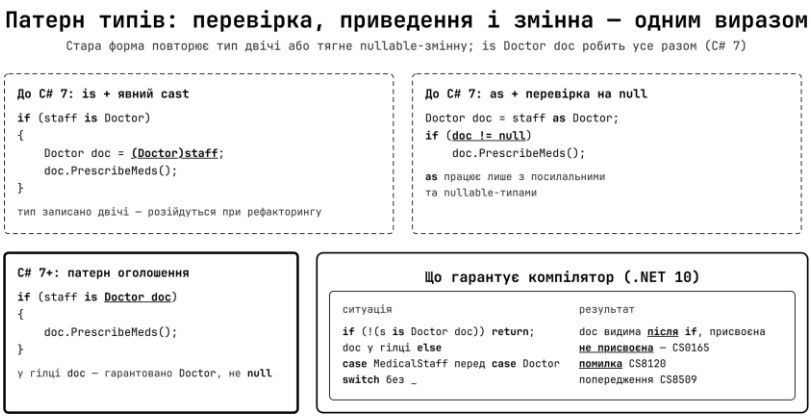
До C# 7 перевірка типу з подальшим cast виглядала так:

```
C#
 1  // Стара форма — два кроки, дублювання
```

```
 2  if (staff is Doctor)
 3  {
 4      Doctor doc = (Doctor)staff; // cast треба повторити явно
 5      doc.PrescribeMeds();
 6  }
 7
 8  // Або через as + null-check
 9  Doctor? doc = staff as Doctor;
10  if (doc != null)
11      doc.PrescribeMeds();
```

З патерном типів:

```
C#
1  // Нова форма — одна операція
2  if (staff is Doctor doc)
3  {
4      doc.PrescribeMeds(); // doc вже типу Doctor, не потрібен додатковий cast
5  }
```



Патерн типів: стара форма vs pattern matching

Компілятор гарантує: якщо умова is Doctor doc виконалась, то doc не є null і гарантовано має тип Doctor. Ні зайвого cast-у, ні зайвої null-перевірки.

Клінічний приклад: ієрархія медичного персоналу

Визначимо клінічну ієрархію і застосуємо патерн типів:

```
C#
 1  using System;
```

```
 2
 3  // Виконуваний код
 4  MedicalStaff[] staff =
 5  {
 6      new Doctor("Олег Петренко", "Кардіологія"),
 7      new Nurse("Марія Іванова", "Кардіологія"),
 8      new Doctor("Тетяна Мельник", "Неврологія"),
 9      new MedicalStaff("Адмін Сидоренко"),
10  };
11
12  foreach (var person in staff)
13      ProcessStaff(person);
14
15  void ProcessStaff(MedicalStaff s)
16  {
17      if (s is Doctor doc)
18      {
19          Console.WriteLine($"Лікар {doc.Name} ({doc.Specialty}) — може
    виписувати рецепти");
20      }
21      else if (s is Nurse nurse)
22      {
23          Console.WriteLine($"Медсестра {nurse.Name} — виконує призначення
    лікаря");
24      }
25      else
26      {
27          Console.WriteLine($"Персонал {s.Name} — адміністративна роль");
28      }
29  }
30
31  // Ієрархія
32  class MedicalStaff
33  {
34      public string Name { get; }
35      public MedicalStaff(string name) => Name = name;
36  }
37  class Doctor : MedicalStaff
38  {
39      public string Specialty { get; }
40      public Doctor(string name, string specialty) : base(name) => Specialty =
    specialty;
41  }
42  class Nurse : MedicalStaff
43  {
44      public string Ward { get; }
45      public Nurse(string name, string ward) : base(name) => Ward = ward;
46  }
```

Тут варто бути точним щодо області видимості. Змінна патерну, оголошена в умові if, належить не блоку if, а **блоку, що його охоплює** (правило «ширшої області видимості», C# 7). Тому doc і nurse видимі в усьому тілі ProcessStaff, і оголосити там іншу змінну doc не вдасться (CS0128). Захищає нас інше правило — **певне присвоєння** (definite assignment): компілятор дозволяє читати doc лише там, де патерн

гарантовано спрацював. У гілці else чи після if без виходу звернення до doc дає помилку CS0165 «використання неприсвоєної змінної».

Саме ця комбінація робить можливим популярний прийом «ранній вихід»:

```
C#
1  if (s is not Doctor doc)
2      return;                 // не лікар — виходимо
3
4  doc.PrescribeMeds();         // тут doc видима і гарантовано присвоєна
```

Перевірено на .NET 10: doc після такого if доступна, а звернення до неї в else звичайного if (s is Doctor doc) не компілюється (CS0165).

Патерн типів у switch

Патерни зручно застосовувати в конструкції switch. Тут є дві форми: switch statement приймає патерни в case з C# 7, а switch expression з'явився в C# 8:

switch statement (традиційний) — більш детальний, дозволяє break/return в кожному case:

```
C#
 1  switch (staff)
 2  {
 3      case Doctor doc when !doc.IsOnLeave:
 4          doc.PrescribeMeds();
 5          break;
 6      case Nurse nurse:
 7          nurse.ExecuteOrders();
 8          break;
 9      case null:
10          Console.WriteLine("Об'єкт null");
11          break;
12      default:
13          Console.WriteLine("Невідома роль");
14          break;
15  }
```

switch expression (C# 8+) — компактний, повертає значення:

```
C#
1  string role = staff switch
2  {
3      Doctor doc   => $"Лікар: {doc.Specialty}",
4      Nurse  nurse => $"Медсестра: {nurse.Ward}",
5      null         => "null",
6      _            => "Інший персонал"
7  };
```

Switch expression — це **вираз**, він повертає значення і може стояти в правій частині присвоєння. Switch statement — це **оператор**, він виконує дії. При pattern matching switch expression зазвичай компактніший і виразніший.

Додаткові умови: when

У switch statement до кожного case можна додати умову через when:

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  MedicalStaff[] staff =
 5  {
 6      new Doctor("Олег Петренко",  isOnLeave: false),
 7      new Doctor("Тетяна Мельник", isOnLeave: true),
 8      new Nurse("Марія Іванова"),
 9  };
10
11  foreach (var person in staff)
12  {
13      string status = GetStatus(person);
14      Console.WriteLine($"{person.Name}: {status}");
15  }
16
17  string GetStatus(MedicalStaff s) => s switch
18  {
19      Doctor { IsOnLeave: false } doc => $"Лікар на зміні ({doc.Name})",
20      Doctor { IsOnLeave: true }      => "Лікар у відпустці",
21      Nurse nurse                     => $"Медсестра: {nurse.Name}",
22      null                            => "порожньо",
23      _                               => "інший персонал"
24  };
25
26  // Ієрархія
27  class MedicalStaff
28  {
29      public string Name { get; }
30      public MedicalStaff(string name) => Name = name;
31  }
32  class Doctor : MedicalStaff
33  {
34      public bool IsOnLeave { get; }
35      public Doctor(string name, bool isOnLeave) : base(name) => IsOnLeave =
    isOnLeave;
36  }
37  class Nurse : MedicalStaff
38  {
39      public Nurse(string name) : base(name) { }
40  }
```

Constant pattern і null-перевірка

Патерн може порівнювати значення з **константою**, включаючи null:

```C#
// Перевірка конкретного рядка
if (diagnosis is "Гіпертонія II ст.")
    Console.WriteLine("Стандартний протокол лікування");

// null-перевірка через патерн
if (patient is not null)
    patient.PrintInfo();

// Або
if (patient is null)
    throw new ArgumentNullException(nameof(patient));
```

is null / is not null (друга форма — з C# 9) — це не просто синтаксичний цукор. Головна відмінність від == null / != null: патерн **ніколи не викликає перевантажений оператор ==**, а перевіряє саме посилання. Якщо в класі == перевантажено (навіть некоректно), x == null може повернути що завгодно, а x is null — завжди правду. Перевірено на .NET 10: для класу, чий == завжди повертає true, obj == null дає True, а obj is null — False. Крім того, такий запис виразно показує намір: «переконатись, що об'єкт існує».

Порядок case у switch: важливо

Switch обробляє варіанти **зверху вниз** і зупиняється на першому збігу. Якщо базовий тип іде першим — до похідних ніколи не дійде:

```C#
// НЕПРАВИЛЬНО — Doctor ніколи не буде спіймано:
switch (staff)
{
    case MedicalStaff s:  // спрацює першим для будь-якого типу!
        // ...
    case Doctor doc:      // недосяжний код
        // ...
}

// ПРАВИЛЬНО — конкретніші типи першими:
switch (staff)
{
    case Doctor doc:      // спочатку — конкретний
    case Nurse nurse:
    case MedicalStaff s:  // потім — загальний
}
```

Компілятор тут не просто попереджає — недосяжний case є **помилкою компіляції**: CS8120 у switch statement і CS8510 у switch expression («патерн уже оброблено попереднім arm-ом»). Тож неправильний порядок для типів з однієї ієрархії не пройде непоміченим. Але компілятор ловить лише очевидну недосяжність;

якщо загальний case має умову when, він уже не вважається таким, що «з'їдає» все, і відповідальність за порядок повертається до програміста.

Антипатерн: ланцюг if-else з is та явним cast

До появи pattern matching стандартним способом перевірки типу була комбінація is для перевірки і явного cast (Тип)значення для доступу до членів. Цей підхід досі зустрічається у старому коді та є антипатерном у сучасному C#:

```
C#
 1  // АНТИПАТЕРН — стара форма: тип перевіряється двічі, потім cast вручну
 2  void ProcessStaff(MedicalStaff s)
 3  {
 4      if (s is Doctor)
 5      {
 6          Doctor doc = (Doctor)s;              // тип дублюється: is Doctor і
    (Doctor)
 7          Console.WriteLine(doc.Specialty);
 8      }
 9      else if (s is Nurse)
10      {
11          Nurse nurse = (Nurse)s;
12          Console.WriteLine(nurse.Ward);
13      }
14      else if (s is Administrator)
15      {
16          Administrator admin = (Administrator)s;
17          Console.WriteLine(admin.Department);
18      }
19  }
```

Цей стиль має три проблеми. По-перше, **дублювання типу**: перевірка is Doctor і наступний (Doctor)s — це дві операції над одним і тим самим фактом. По-друге, **крихкість при рефакторингу**: якщо хтось змінить перевірку is Doctor на is Surgeon, але забуде оновити cast — отримає InvalidCastException у runtime. По-третє, **відсутність вичерпності**: якщо до ієрархії додати новий тип Paramedic, компілятор не попередить, що він не оброблений у жодній гілці — виконання мовчки пройде мимо.

Стара форма через as + null-check трохи краща (немає подвійного cast), але все одно потребує двох кроків, а змінна doc має тип Doctor? і лишається доступною після if — компілятор дозволить звернутися до неї там, де вона може бути null (буде лише попередження nullable-аналізу). Змінна патерну теж видима після if, але правило певного присвоєння не дасть прочитати її там, де збігу не було:

```
C#
```

```
// as + null-check: один cast, але binding поза блоком if
Doctor? doc = s as Doctor;
if (doc != null)
    Console.WriteLine(doc.Specialty);

// doc досі видимий тут — хоч може бути null
```

Правильний підхід — switch expression із type pattern. Одна операція перевіряє тип і прив'язує змінну; вираз повертає значення; exhaustiveness перевіряється компілятором через _:

```
C#
using System;

MedicalStaff[] staff =
{
    new Doctor("Петренко О.", "Кардіологія"),
    new Doctor("Мельник Т.", "Неврологія") { IsOnLeave = true },
    new Nurse("Іванова М.", "Кардіологія"),
    new Administrator("Бойко Р.", "Реєстратура"),
};

foreach (var person in staff)
    Console.WriteLine(Describe(person));

// switch expression: тип + властивість в одному місці, повертає рядок
string Describe(MedicalStaff s) => s switch
{
    Doctor { IsOnLeave: true } doc
        => $"Лікар {doc.Name} ({doc.Specialty}) — у відпустці",
    Doctor doc
        => $"Лікар {doc.Name} ({doc.Specialty}) — на зміні",
    Nurse nurse
        => $"Медсестра {nurse.Name}, відділення {nurse.Ward}",
    Administrator admin
        => $"Адміністратор {admin.Name}, {admin.Department}",
    null
        => "(порожнє посилання)",
    _
        => $"Невідома роль: {s.GetType().Name}"
};

class MedicalStaff
{
    public string Name { get; }
    public MedicalStaff(string name) => Name = name;
}
class Doctor : MedicalStaff
{
    public string Specialty { get; }
    public bool IsOnLeave { get; init; }
    public Doctor(string name, string specialty) : base(name) => Specialty =
specialty;
}
class Nurse : MedicalStaff
{
    public string Ward { get; }
    public Nurse(string name, string ward) : base(name) => Ward = ward;
```

```
46  }
47  class Administrator : MedicalStaff
48  {
49      public string Department { get; }
50      public Administrator(string name, string dept) : base(name) => Department =
    dept;
51  }
```

Switch expression перевіряє тип і прив'язує змінну **в одній операції** (Doctor doc) — жодного зайвого cast, жодного дублювання. Вбудований property pattern ({ IsOnLeave: true }) дозволяє уточнити умову без окремого if. Гілка _ — запасний варіант для нових типів: IDE одразу підказує додати case Paramedic, щойно такий клас з'явиться у ієрархії.

	Старий if-else + cast	**Switch expression**
Кількість операцій для перевірки типу	2 (is + явний cast)	1 (type pattern)
Ризик InvalidCastException	Є при неузгодженому рефакторингу	Відсутній
Прив'язка змінної	Окрема оголошення після блоку if	Вбудована (Doctor doc)
Перевірка властивостей	Окремий вкладений if	Вбудований property pattern
Повертає значення	Ні — лише виконує дії	Так — вираз, можна присвоїти
Контроль нових типів	Мовчазний пропуск	IDE пропонує додати case

9.2. Патерн властивостей

Патерн типів (9.1) перевіряє, **чим є** об'єкт. Але часто нас цікавить не тільки тип, а й **конкретні значення** його властивостей: «чи є це пацієнт з активним статусом і пріоритетом вище 2?». Для цього існує **патерн властивостей** (property pattern) — він дозволяє перевіряти відповідність значень полів об'єкта прямо у виразі is або switch.

Синтаксис і анатомія

Патерн властивостей записується у фігурних дужках: { Властивість: значення }. Двокрапка тут не присвоєння, а **порівняння**: «властивість має це значення». Значенням може бути константа, null, інший патерн (у тому числі реляційний > 2) або var для захоплення.

Патерн властивостей: анатомія і варіанти

Двокрапка — не присвоєння, а «властивість відповідає вкладеному патерну»; кожен рядок таблиці виконано

Анатомія

```
p is Patient { Status: "Active", Priority: > 2 }
//   ↑ тип      ↑ властивість ↑ константа  ↑ реляційний патерн
```

тип — необов'язковий; будь-який { … } спершу перевіряє, що об'єкт **не null**

Варіанти: p — активний пацієнт кардіології, Priority = 3

вираз	результат	що перевіряє
`p is Patient { Status: "Active", Priority: > 2 }`	**true**	тип + дві властивості
`p is { Priority: > 3 }`	**false**	лише властивість
`p is { Department.Name: "Кардіологія" }`	**true**	вкладена властивість (C# 10)
`p is { Department: { Name: "Кардіологія" } }`	**true**	те саме до C# 10
`p is { }`	**true**	будь-який не-null
`nobody is { }`	**false**	**null** не збігається

захоплення значення

`p is { Department.Name: var dept } ? dept : ""`	"Кардіологія"	**var** збігається з будь-чим

Патерн властивостей: анатомія та варіанти

Патерн властивостей може стояти окремо або бути частиною складнішого патерну, де зліва вказано тип:

```
C#
1  // Тільки властивості (без перевірки типу)
2  if (patient is { Status: "Active" })
3
4  // Тип + властивості разом
5  if (patient is Patient { Status: "Active", Priority: > 2 })
```

У другій формі спочатку перевіряється тип (Patient), і якщо він збігається — перевіряються властивості.

Базовий приклад у клінічному контексті

```
C#
 1  using System;
```

```
 2
 3  // Виконуваний код
 4  var patients = new[]
 5  {
 6      new Patient("Іван Петренко",  "Active",   priority: 3, "Кардіологія"),
 7      new Patient("Марія Коваль",   "Active",   priority: 1, "Неврологія"),
 8      new Patient("Олег Сидоренко", "Discharge",priority: 2, "Хірургія"),
 9      new Patient("Ганна Мельник",  "Active",   priority: 5, "Реанімація"),
10  };
11
12  foreach (var p in patients)
13      Console.WriteLine($"{p.Name}: {GetPriority(p)}");
14
15  string GetPriority(Patient p) => p switch
16  {
17      { Status: "Active", Priority: > 3 }  => "ТЕРМІНОВО — негайна увага",
18      { Status: "Active", Priority: > 1 }  => "Активний — плановий огляд",
19      { Status: "Active" }                 => "Активний — рутинний",
20      { Status: "Discharge" }              => "Виписка оформляється",
21      _                                    => "Невизначено",
22  };
23
24  // Клас
25  class Patient
26  {
27      public string Name       { get; }
28      public string Status     { get; }
29      public int    Priority   { get; }
30      public string Department { get; }
31
32      public Patient(string name, string status, int priority, string dept)
33      {
34          Name = name; Status = status; Priority = priority; Department = dept;
35      }
36  }
```

Зверніть увагу: Priority: > 3 — це поєднання патерну властивостей з реляційним патерном. Значення Priority перевіряється не на рівність, а на виконання умови > 3. Патерни в C# **вкладаються** один в одного.

Захоплення значення через var

Іноді нам потрібно не лише перевірити значення властивості, а й **зберегти** його у змінну для подальшого використання у виразі результату. Для цього замість конкретного значення пишуть var ім'я:

```
C#
1  string GetWelcome(Patient? p) => p switch
2  {
3      { Status: "Active", Department: var dept } => $"Пацієнт активний,
   відділення: {dept}",
4      { Status: var s }                          => $"Статус: {s}",
5      null                                       => "Пацієнт не знайдений"
6  };
```

Department: var dept означає: «захопити значення властивості Department у змінну dept» — і тоді використати dept у рядку результату. Перевірка типу при цьому не виконується: var завжди успішно збігається з будь-яким значенням (у тому числі null).

Вкладені об'єкти (C# 10+)

Якщо об'єкт містить вкладені об'єкти, патерн властивостей може заглиблюватись у них. До C# 10 для цього потрібен був вкладений патерн:

```
C#
1  // До C# 10 — явне вкладення
2  if (patient is { Department: { Name: "Кардіологія" } })
3
4  // C# 10+ — скорочений точковий запис
5  if (patient is { Department.Name: "Кардіологія" })
```

Обидва варіанти еквівалентні, але точковий синтаксис C# 10 значно компактніший.

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  var patients = new[]
 5  {
 6      new Patient("Іван Петренко",  new Department("Кардіологія", isIntensive:
    false)),
 7      new Patient("Марія Коваль",   new Department("Реанімація",  isIntensive:
    true)),
 8      new Patient("Олег Сидоренко", new Department("Хірургія",    isIntensive:
    false)),
 9  };
10
11  foreach (var p in patients)
12      Console.WriteLine($"{p.Name}: {GetAlert(p)}");
13
14  string GetAlert(Patient p) => p switch
15  {
16      // Вкладений патерн — перевірка властивості вкладеного об'єкта
17      { Department.IsIntensive: true }             => "РЕАНІМАЦІЯ — постійний
    моніторинг",
18      { Department.Name: "Кардіологія" }           => "Кардіо-відділення — ЕКГ
    щодня",
19      { Department: { Name: var name } }           => $"Відділення {name} —
    стандартний режим",
20      _                                            => "Невідомо",
21  };
22
23  // Класи
24  class Department
25  {
```

```
26      public string Name        { get; }
27      public bool   IsIntensive { get; }
28      public Department(string name, bool isIntensive)
29      { Name = name; IsIntensive = isIntensive; }
30  }
31  class Patient
32  {
33      public string     Name       { get; }
34      public Department Department { get; }
35      public Patient(string name, Department dept) { Name = name; Department =
    dept; }
36  }
```

Порожній патерн властивостей {}

Фігурні дужки без вмісту {} — це особливий випадок: він **збігається з будь-яким non-null об'єктом**:

```
C#
1  string Describe(Patient? p) => p switch
2  {
3      { Status: "Active" } => "Активний пацієнт",
4      { }                  => "Пацієнт (не активний)", // будь-який не-null
5      null                 => "Null"
6  };
```

{} часто стоїть передостанньою гілкою в switch як «запасний варіант для не-null» перед явною перевіркою на null.

Поєднання типового та властивісного патернів

Обидва патерни можна поєднувати: спочатку перевірка типу, потім — властивостей. Це особливо зручно в ієрархіях:

```
C#
1  string GetInfo(MedicalStaff s) => s switch
2  {
3      Doctor { IsOnLeave: false, Specialty: var spec } => $"Лікар на зміні:
   {spec}",
4      Doctor { IsOnLeave: true }                       => "Лікар у відпустці",
5      Nurse  { Ward: var ward }                        => $"Медсестра: палата
   {ward}",
6      _                                                => "Персонал"
7  };
```

Тут Doctor { IsOnLeave: false, Specialty: var spec } — патерн типу Doctor (type pattern), що одночасно перевіряє властивість IsOnLeave (property pattern) і захоплює Specialty у змінну spec (var capture).

Такі поєднання роблять switch expression потужним інструментом для опису складної бізнес-логіки у компактній, декларативній формі — без глибоких вкладених if-else.

9.3. Патерни кортежів

У попередніх секціях ми перевіряли один об'єкт або одне значення. Але багато практичних рішень залежать від **кількох незалежних вимірів** одночасно: «який протокол лікування призначити пацієнту, якщо його тиск категорії X і вік у діапазоні Y?». Перевіряти кожну комбінацію через вкладені if-else — громіздко і важко читати. **Патерн кортежів** вирішує це елегантно: передаємо кілька значень як кортеж у switch і описуємо кожну значущу комбінацію окремим arm.

Механізм: як це компілюється

Коли ви пишете:

```
C#
1  (specialty, shift) switch
2  {
3      ("Кардіологія", "День") => ...,
4      ...
5  }
```

Логічно це switch над кортежем — тим самим ValueTuple з розділу 8.9, лише використаним не для зберігання і передачі, а як ключ для перемикання. Але фізично компілятор **не створює** кортеж: коли в switch стоїть кортежний літерал (a, b), він обчислює елементи в тимчасові змінні й порівнює кожну позицію напряму. Це перевірено на .NET 10 аналізом IL методу зі (specialty, shift) switch — інструкції newobj ValueTuple там немає. Тож кортежний патерн нічого не коштує в пам'яті й працює так само швидко, як вкладені if.

Вкладені патерни в позиціях кортежу

Кожна позиція кортежу може містити **будь-який патерн** — не лише рядок або число. В позицію можна вставити:

- константу ("Кардіологія", 42);
- discard _ — «будь-яке значення»;
- реляційний патерн (> 140, < 18);
- var ім'я — захопити значення у змінну;

— навіть вкладений property pattern.

Це робить кортежний патерн потужним: один switch замінює складну ієрархію перевірок.

Патерн кортежів: рішення за двома вимірами

(specialty, shift) switch — arm-и перевіряються згори вниз, перший збіг виграє

Arm-и з лекції

```
(specialty, shift) switch
{
    ("Кардіологія", "День") => "Плановий прийом + ЕКГ",
    ("Кардіологія", "Ніч")  => "Термінова допомога — чергово",
    ("Хірургія",    "День") => "Операційний зал — за розкладом",
    ("Хірургія",    "Ніч")  => "Ургентна хірургія",
    ("Неврологія",  _)      => "Неврологічний огляд",
    (_,             "День") => "Денний прийом (стандарт)",
    _                       => "Позаплановий прийом"
}
```

_ у позиції — будь-яке значення; _ замість кортежу — будь-яка пара

Що повертає

(specialty, shift)	arm
("Кардіологія", "Ніч")	"Термінова допомога — чергово"
("Неврологія", "Позапланова")	"Неврологічний огляд"
("Педіатрія", "День")	"Денний прийом (стандарт)"
("Педіатрія", "Ніч")	"Позаплановий прийом"

Компілятор **не створює** ValueTuple: елементи порівнюються напряму (перевірено в IL).
Без останнього _ — попередження CS8509, а в runtime — SwitchExpressionException.

Патерн кортежів: двовимірне зіставлення

Клінічний приклад: розклад та правила призначень

Визначимо логіку призначення лікаря на прийом залежно від спеціалізації і зміни:

```C#
using System;

// Виконуваний код
// Комбінації: (спеціалізація, зміна)
PrintProtocol("Кардіологія", "День");
PrintProtocol("Кардіологія", "Ніч");
```

```
 7  PrintProtocol("Хірургія",    "День");
 8  PrintProtocol("Хірургія",    "Ніч");
 9  PrintProtocol("Неврологія",  "День");
10  PrintProtocol("Неврологія",  "Позапланова");
11
12  void PrintProtocol(string specialty, string shift)
13  {
14      string protocol = (specialty, shift) switch
15      {
16          // Точні комбінації — найвищий пріоритет
17          ("Кардіологія", "День")  => "Плановий прийом + ЕКГ",
18          ("Кардіологія", "Ніч")   => "Термінова допомога — чергово",
19          ("Хірургія",    "День")  => "Операційний зал — за розкладом",
20          ("Хірургія",    "Ніч")   => "Ургентна хірургія",
21
22          // Discard: будь-яка зміна для відділення
23          ("Неврологія",  _)       => "Неврологічний огляд",
24
25          // Discard: відома зміна для будь-якого відділення
26          (_,             "День")  => "Денний прийом (стандарт)",
27
28          // Загальний випадок
29          _                        => "Позаплановий прийом"
30      };
31
32      Console.WriteLine($"[{specialty} / {shift}] → {protocol}");
33  }
```

Arm-и обробляються **зверху вниз**: перший збіг виграє. Тому порядок важливий: конкретні комбінації мають іти першими, загальні (з _) — в кінці. Якщо поставити _ на початок — він поглине всі наступні arm-и.

Вкладені реляційні патерни: медичне рішення

Особливо потужна можливість — реляційні патерни в позиціях кортежу. Можна перевіряти не рівність значення, а діапазон:

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  // Протокол лікування за (систолічний тиск, вік)
 5  Console.WriteLine(GetTreatment(120, 35));  // норма, молодий
 6  Console.WriteLine(GetTreatment(145, 60));  // підвищений, похилий
 7  Console.WriteLine(GetTreatment(180, 45));  // гіпертонічний криз
 8  Console.WriteLine(GetTreatment(85,  25));  // гіпотонія, молодий
 9  Console.WriteLine(GetTreatment(85,  75));  // гіпотонія, похилий
10
11  string GetTreatment(int systolic, int age) => (systolic, age) switch
12  {
13      // Гіпертонічний криз — незалежно від віку
14      (>= 180, _)              => "КРИЗ — негайна госпіталізація",
15
16      // Підвищений + похилий вік — підвищений ризик
```

```
    (>= 140, >= 60)          => "Гіпертонія + похилий вік — інтенсивна
терапія",

    // Підвищений, молодший
    (>= 140, _)              => "Гіпертонія — медикаментозне лікування",

    // Норма
    (>= 90, >= 60 and < 120) => "Норма для похилого — спостереження",
    (>= 90, _)               => "Нормальний тиск",

    // Гіпотонія
    (_, >= 60)               => "Гіпотонія + похилий вік — УВАГА",
    _                        => "Гіпотонія — консультація"
};
```

Зверніть: у другій позиції кортежу стоїть >= 60 and < 120 — це логічний патерн всередині arm-а кортежу. Позиції кортежу дозволяють будь-який патерн, включаючи складені логічні вирази.

Вичерпність (exhaustiveness)

Компілятор аналізує, чи всі можливі комбінації значень покриті. Якщо ні і відсутній _ — попередження (warning). У switch expression це важливо: якщо жоден arm не збігається — кидається System.Runtime.CompilerServices.SwitchExpressionException у runtime (перевірено на .NET 10; винятку MatchFailedException у .NET немає). Для double компілятор вимагає покрити навіть double.NaN — без _ буде попередження CS8509.

Тому для кортежних патернів **завжди** завершуйте switch arm-ом _ або логічно повним набором умов, що покривають всі випадки.

Коли обирати кортежний патерн

Кортежний патерн найкраще підходить, коли:

- рішення залежить від **двох або більше незалежних вимірів** одночасно;
- кожна комбінація вимірів дає унікальний результат;
- альтернатива — вкладені if-else або таблиця умов.

Якщо рішення залежить від **властивостей одного об'єкта** — краще використовувати патерн властивостей (9.2). Якщо рішення залежить від **типу** об'єкта в ієрархії — патерн типів (9.1). Кортежний патерн — для **кількох незалежних вхідних змінних**.

9.4. Позиційний патерн

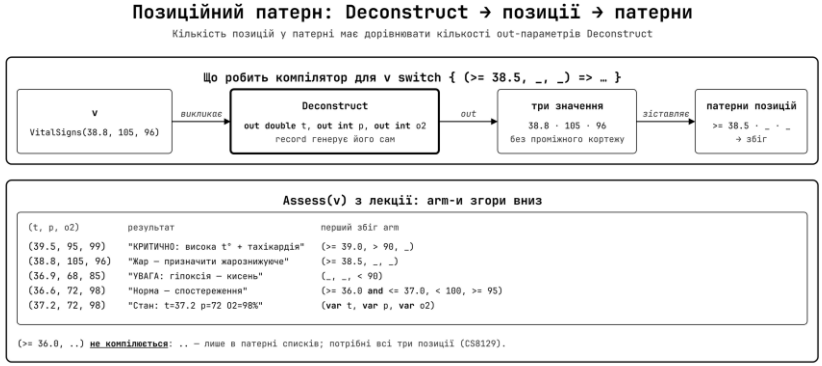
У секції 9.3 ми передавали у switch явно створений кортеж: (specialty, shift) switch. Але що якщо дані вже зібрані в об'єкт? Розбирати об'єкт на окремі змінні перед switch — зайва церемонія. **Позиційний патерн** вирішує це: він автоматично «розкладає» об'єкт через метод Deconstruct і потім зіставляє позиції — так само, як кортежний патерн.

Механізм: Deconstruct як міст

Щоб об'єкт підтримував позиційний патерн, він повинен мати метод Deconstruct з out-параметрами. Компілятор виконує три кроки:

1. Викликає obj.Deconstruct(out var p1, out var p2, ...) — отримує набір out-значень.
2. Нумерує отримані значення за позиціями [0], [1], [2]... — проміжний кортеж при цьому не створюється, значення лежать у тимчасових змінних.
3. Зіставляє кожну позицію з відповідним патерном у дужках (pat1, pat2, ...).

Таким чином, позиційний патерн — це **кортежний патерн над об'єктом, що вміє себе деконструювати**. Зв'язок між 9.3 (кортежний) і 9.4 (позиційний) прямий: вони використовують один і той самий механізм зіставлення по позиції.



Позиційний патерн: Deconstruct → кортеж → зіставлення

Зв'язок з record (8.10)

У розділі 8.10 ми дізнались, що позиційний record автоматично генерує Deconstruct. Це означає: **будь-який позиційний record вже підтримує позиційний патерн** без жодного додаткового коду:

```
C#
 1  public record VitalSigns(double Temperature, int Pulse, int OxygenSat);
 2
 3  // Позиційний патерн одразу доступний:
 4  vitals switch
 5  {
 6      (> 38.5, > 100, _)        => "Жар і тахікардія",
 7      (_, _, < 90)              => "Гіпоксія",
 8      (>= 36.0 and <= 37.0, _, _) => "Норма",
 9      _                         => "Потрібен огляд"
10  }
```

Зверніть увагу на третій arm: кількість позицій у патерні має **точно** дорівнювати кількості out-параметрів Deconstruct. Запис (>= 36.0 and <= 37.0, ..) не компілюється: slice .. дозволений лише в патернах списків (9.6), а компілятор шукає Deconstruct з двома параметрами й не знаходить (CS8129). Пропущені позиції заповнюються _.

record і позиційний патерн — природна пара: record компактно описує структуру, позиційний патерн компактно описує умови на цю структуру.

Приклад: record з позиційним патерном

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  var measurements = new VitalSigns[]
 5  {
 6      new(37.2, 72,  98),
 7      new(38.8, 105, 96),
 8      new(36.9, 68,  85),
 9      new(39.5, 95,  99),
10  };
11
12  foreach (var v in measurements)
13      Console.WriteLine($"t={v.Temperature} p={v.Pulse} O2={v.OxygenSat}% →
    {Assess(v)}");
14
15  string Assess(VitalSigns v) => v switch
16  {
17      // Позиційний патерн: [Temperature, Pulse, OxygenSat]
18      (>= 39.0, > 90, _)             => "КРИТИЧНО: висока t° + тахікардія",
19      (>= 38.5, _, _)                => "Жар — призначити жарознижуюче",
20      (_, _, < 90)                   => "УВАГА: гіпоксія — кисень",
21      (>= 36.0 and <= 37.0, < 100, >= 95) => "Норма — спостереження",
```

```
    (var t, var p, var o2)         => $"Стан: t={t:F1} p={p} O2={o2}%"
};

// record з автоматичним Deconstruct
public record VitalSigns(double Temperature, int Pulse, int OxygenSat);
```

У передостанньому arm (>= 36.0 and <= 37.0, < 100, >= 95) кожна позиція містить реляційний або логічний патерн. В останньому arm (var t, var p, var o2) — захоплення всіх позицій у змінні.

Кастомний Deconstruct

Якщо клас не є record, Deconstruct визначається вручну — метод без типу результату (void) з параметрами out:

C#
```
using System;

// Виконуваний код
var appointments = new Appointment[]
{
    new("Олег Петренко",   "Плановий",    "Кардіологія"),
    new("Марія Коваль",    "Ургентний",   "Хірургія"),
    new("Іван Сидоренко",  "Плановий",    "Неврологія"),
    new("Ганна Мельник",   "Ургентний",   "Реанімація"),
};

foreach (var a in appointments)
    Console.WriteLine($"{a.PatientName}: {GetPriority(a)}");

string GetPriority(Appointment a) => a switch
{
    // Позиційний патерн: [type, department]
    ("Ургентний", "Реанімація") => "НЕГАЙНО — реанімація",
    ("Ургентний", var dept)     => $"Терміново — {dept}",
    ("Плановий",  "Кардіологія")=> "Плановий + ЕКГ",
    ("Плановий",  var dept)     => $"Плановий — {dept}",
    _                           => "Невизначено"
};

// Клас з кастомним Deconstruct
class Appointment
{
    public string PatientName  { get; }
    public string Type         { get; }
    public string Department   { get; }

    public Appointment(string name, string type, string dept)
    { PatientName = name; Type = type; Department = dept; }

    // Deconstruct — розкладає на Type та Department
    public void Deconstruct(out string type, out string department)
    { type = Type; department = Department; }
}
```

Зверніть: Deconstruct розкриває лише Type і Department — не PatientName. Ми самі вирішуємо, які поля «виставити» позиційному патерну. Решту полів можна перевіряти через property pattern окремо або через var з подальшою умовою.

Порівняння: кортежний vs позиційний патерн

Аспект	Кортежний (9.3)	Позиційний (9.4)
Вхід	явний кортеж (a, b)	об'єкт з Deconstruct
Вимога	будь-які два значення	Deconstruct в класі/record
Зв'язок	кортеж з 8.9	record з 8.10
Коли краще	незалежні зовнішні значення	вже є об'єкт з даними

Якщо дані вже «живуть» в об'єкті — позиційний патерн. Якщо дані приходять як окремі змінні — кортежний.

9.5. Реляційний та логічний патерни

У попередніх секціях патерни порівнювали значення **на рівність**: "Active", true, null. Але в медицині (та загалом у числовій логіці) рішення ухвалюються на основі **діапазонів**: температура вище 38.5°C — жар; тиск від 140 — гіпертонія. Для цього у C# 9 введено **реляційний** та **логічний** патерни.

Реляційний патерн

Реляційний патерн (relational pattern) перевіряє значення за допомогою операторів <, >, <=, >=:

```
C#
1  temperature switch
2  {
3      < 36.0  => "Гіпотермія",
4      >= 38.5 => "Жар",
5      _       => "Норма"
6  }
```

Важливо: реляційний патерн **не є оператором порівняння**. Це **патерн** — тобто він може стояти не лише у switch, а й після is, у позиції кортежного патерну, всередині патерну властивостей тощо:

```
C#
1  if (temperature is > 38.5) { ... }          // після is
2  (temp, pulse) switch { (> 38.5, > 100) => } // в позиції кортежу
3  { Priority: >= 3 }                           // в патерні властивостей
```



Реляційний патерн: температура тіла — діапазони та рішення

Логічний патерн: and, or, not

Логічний патерн дозволяє комбінувати кілька патернів через and, or, not. Ключова особливість: and, or, not тут — це **патерні оператори**, а не логічні оператори &&, ||, !.

Різниця принципова:

— && і || — оператори над булевими **значеннями**: x > 5 && x < 10

— and і or у патернах — оператори над **патернами**: > 5 and < 10

```
C#
1  // Звичайна умова (bool-вирази):
2  if (temp > 37.5 && temp < 39.0) { ... }
3
4  // Логічний патерн (комбінація патернів):
5  if (temp is > 37.5 and < 39.0) { ... }
```

Результат той самий, але форма запису в switch є значно виразнішою.

Пріоритет: and вищий за or, як множення вище за додавання. Для явного групування використовуйте дужки:

```
C#
1  // (< 35 or > 39) — критичні значення
2  temperature is < 35 or > 39
3
4  // not (>= 36 and <= 37) — не нормальна
5  temperature is not (>= 36.0 and <= 37.0)
```

Реляційний патерн у клінічному сценарії

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  double[] temperatures = { 34.8, 36.5, 37.2, 38.1, 39.0, 40.5 };
 5
 6  foreach (var t in temperatures)
 7      Console.WriteLine($"{t:F1}°C → {AssessTemperature(t)}");
 8
 9  string AssessTemperature(double t) => t switch
10  {
11      < 35.0                    => "Гіпотермія — термінова допомога",
12      >= 35.0 and < 36.0        => "Нижче норми — спостереження",
13      >= 36.0 and <= 37.0       => "Нормальна температура",
14      > 37.0  and < 37.5        => "Субфебрильна — спостереження",
15      >= 37.5 and < 38.5        => "Жар — призначити жарознижуюче",
16      >= 38.5 and < 39.5        => "Висока — термінова оцінка",
17      >= 39.5                   => "Критична — госпіталізація"
18  };
```

Кожен arm покриває рівно один інтервал, і разом вони покривають усю числову вісь. Проте компілятор все одно видає попередження CS8509: для double існує ще значення double.NaN, яке не збігається з жодним реляційним патерном (будь-яке порівняння з NaN — хибне). Якщо в такий switch потрапить NaN, буде SwitchExpressionException. Щоб switch став вичерпним, додайте arm double.NaN => … або завершальний _.

Ще одне обмеження: праворуч від <, >, <=, >= у патерні може стояти лише **константа**. Запис t is > limit, де limit — змінна, не компілюється (CS9135); для порівняння зі змінною використовуйте звичайний вираз t > limit або when.

Логічний патерн: and, or, not у дії

```
C#
 1  using System;
 2
```

```
// Виконуваний код
// Комплексна оцінка стану пацієнта
int[] pulseReadings = { 45, 60, 82, 105, 130 };
int[] systolicValues = { 85, 110, 135, 160, 195 };

Console.WriteLine("--- Оцінка пульсу ---");
foreach (var p in pulseReadings)
    Console.WriteLine($"  {p} уд/хв → {AssessPulse(p)}");

Console.WriteLine("--- Оцінка тиску ---");
foreach (var s in systolicValues)
    Console.WriteLine($"  {s} мм рт.ст. → {AssessPressure(s)}");

string AssessPulse(int pulse) => pulse switch
{
    // not + or: «не (нижче 60 або вище 100)» — тобто 60…100
    not (< 60 or > 100) => "Норма",
    < 60                => "Брадикардія",
    _                   => "Тахікардія",
};

string AssessPressure(int systolic) => systolic switch
{
    // критичні значення з двох боків — окремі arm-и, бо повідомлення різні
    // (з одним повідомленням їх можна об'єднати: > 180 or < 90)
    > 180           => "КРИТИЧНО (Гіпертонічний криз)",
    < 90            => "КРИТИЧНО (Гіпотонія)",
    >= 90 and < 120 => "Норма",
    >= 120 and < 140=> "Підвищений",
    _               => "Гіпертонія",  // >= 140 and <= 180
};
```

not з типовим патерном

not може заперечувати не лише реляційні вирази, а й **типові патерни**:

```
C#
if (staff is not Doctor)
    Console.WriteLine("Не лікар — не може виписувати рецепти");

// В switch:
string role = staff switch
{
    not null and not Doctor => "Медперсонал (не лікар)",
    Doctor doc              => $"Лікар: {doc.Specialty}",
    null                    => "Відсутній"
};
```

not null — це патернний еквівалент перевірки != null. Компілятор трактує його як null-check pattern і може застосовувати null-state analysis.

Де можна використовувати реляційні та логічні патерни

Реляційні та логічні патерни **не прив'язані до switch** — вони можуть стояти у будь-якому місці, де стоїть патерн:

```
C#
 1  // Після is:
 2  if (temperature is >= 37.5 and < 39.5) { ... }
 3
 4  // В позиції кортежного патерну:
 5  (temp, pulse) switch { (>= 37.5, > 100) => "..." }
 6
 7  // В патерні властивостей:
 8  patient is { Age: > 60 and < 90, Priority: >= 3 }
 9
10  // Навіть у декларативному присвоєнні:
11  bool isFever = temperature is >= 37.5;
```

Саме ця **композиційність** — можливість вкладати патерни один в одний — є головною силою pattern matching у C#.

9.6. Патерни списків

Патерни списків (list patterns) — найновіший різновид патернів у C#, доданий у версії **C# 11** (.NET 7, 2022). Вони дозволяють зіставляти масиви та колекції зі структурними зразками: перевіряти кількість елементів, значення на конкретних позиціях, першого і останнього елементів, а також довільний «зріз» посередині.

Які типи підтримуються

Патерн списків працює з будь-яким типом, що є **лічильним** (має властивість Length або Count) і **індексованим** (має індексатор з int або Index). Інтерфейс тут не потрібен — компілятор шукає члени за іменами:

— T[] — масиви будь-якого типу;

— List<T>, IList<T> — списки;

— Span<T>, ReadOnlySpan<T> — span-и.

Для slice з захопленням (.. var x) потрібно більше: індексатор з Range або метод Slice(int, int). Він є в масивів, string, Span<T> і List<T> (метод Slice додано в .NET 8), але **немає** в IList<T>: il is [var a, .. var rest] дає помилку CS1503, хоча il is [1, ..] (slice без захоплення) компілюється. Захоплений slice масиву — це новий масив-копія.

IEnumerable<T> без Count/Length **не підтримується** — компілятор не може ефективно перевіряти довжину ліниво-обчислюваних послідовностей.

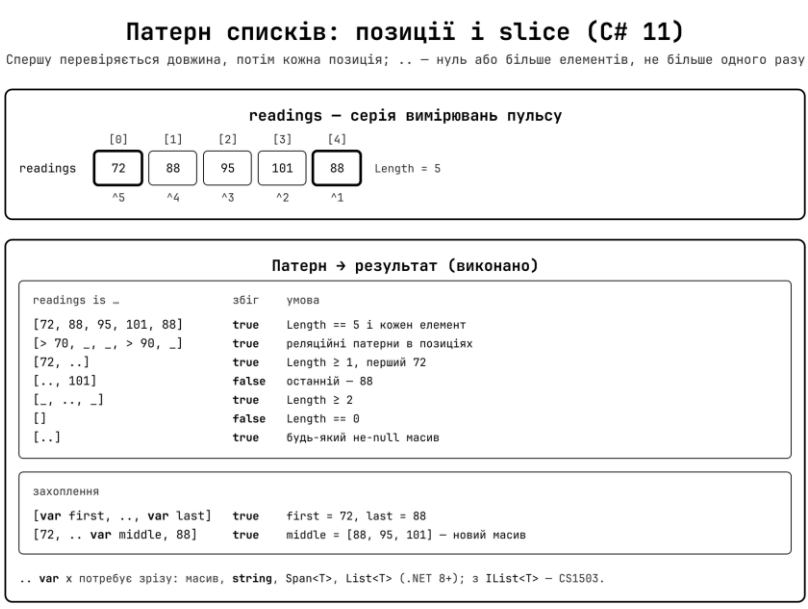
Як компілятор перевіряє патерн

Для патерну [a, b, c] компілятор генерує приблизно:

```
C#
1  collection.Length == 3 &&
2  collection[0] matches a &&
3  collection[1] matches b &&
4  collection[2] matches c
```

Тобто спочатку перевіряється довжина, потім — кожна позиція через відповідний патерн. Це важливо для продуктивності: якщо довжина не збігається — позиції не перевіряються.

Синтаксис: елементи та slice



Патерн списків: синтаксис елементів та slice

Елементи у дужках перевіряють позиції. В кожній позиції може стояти будь-який патерн — константа, _, реляційний, var:

```
C#
1  readings is [72, 88, 95, 101, 88]  // точний збіг усіх 5 елементів
2  readings is [> 70, _, _, > 90, _]  // реляційні в позиціях
3  readings is [var first, .., var last] // захопити крайні
```

Slice .. — це «довільна кількість елементів» (включно з нулем). Він може стояти лише один раз у патерні:

```
C#
1  [72, ..]       // починається з 72
2  [.., 88]       // закінчується на 88
3  [72, .., 88]   // починається 72, закінчується 88, між ними будь-що
4  [_, .., _]     // ≥2 елементи
5  [..]           // будь-яка довжина ≥0
```

Сам slice теж можна захопити у змінну: .. var назва (див. приклад нижче).

Клінічний приклад: аналіз серії вимірювань

```
C#
 1  using System;
 2
 3  // Виконуваний код
 4  int[][] pulseSeries = {
 5      new[] { 72, 75, 78 },              // стабільний діапазон
 6      new[] { 95, 102, 115, 122 },       // зростаючий
 7      new[] { 130, 120, 95, 72 },        // спадний (лікування)
 8      new[] { 68 },                      // одне вимірювання
 9      new int[] { },                     // порожньо
10      new[] { 72, 85, 91, 88, 72 },      // перший = останній
11  };
12
13  foreach (var series in pulseSeries)
14      Console.WriteLine($"[{string.Join(",",series)}] →
    {AnalyzeSeries(series)}");
15
16  string AnalyzeSeries(int[] readings) => readings switch
17  {
18      // Порожній або одне значення
19      []                              => "Немає даних",
20      [var single]                    => $"Одне вимірювання: {single}",
21
22      // Перший і останній — тахікардія
23      [> 100, .., > 100]              => "Стійка тахікардія",
24
25      // Зростаючий тренд — перший менший від останнього
26      [var first, .., var last]
27          when last > first + 20      => $"Тренд зростання: {first}→{last}",
28
```

```
    // Спадний тренд
    [var first, .., var last]
        when first > last + 20      => $"Тренд спадання: {first}→{last} (ефект
лікування?)",

    // Рівний перший і останній
    [var first, .., var last]
        when first == last          => $"Стабільний: {first}",

    // Загальний випадок — показати перший, останній
    [var f, .., var l]              => $"Стан: {f}..{l}"
};
```

Захоплення slice у змінну

Частину між крайніми елементами можна захопити у змінну через .. var назва. Це дозволяє далі обробляти «середину» масиву:

```
C#
using System;

// Виконуваний код
// Формат медичного запису: [ідентифікатор, прізвище, ім'я, ...дані...]
string[][] records = {
    new[] { "P001", "Петренко", "Іван", "36.6", "72" },
    new[] { "P002", "Коваль",   "Марія" },           // без показань
    new[] { "P003" },                                  // тільки ID
    new string[] { },                                  // порожній
};

foreach (var rec in records)
    Console.WriteLine(ParseRecord(rec));

string ParseRecord(string[] record) => record switch
{
    // Мінімальний запис: ID + прізвище + ім'я + дані
    [var id, var last, var first, .. var data]
        when data.Length > 0
        => $"{id}: {last} {first} — показання: {string.Join(", ", data)}",

    // Тільки ПІБ без показань
    [var id, var last, var first]
        => $"{id}: {last} {first} — показань немає",

    // Тільки ID
    [var id]
        => $"{id}: неповний запис",

    // Порожній
    []  => "Порожній рядок"
};
```

Поєднання патерну списків з патерном властивостей

Якщо потрібно перевірити і структуру списку, і властивості колекції, обидва патерни можна поєднати через and:

```
C#
1  if (readings is { Length: >= 3 } and [var first, _, _, .. var rest])
2  {
3      Console.WriteLine($"Перший: {first}, решта: {string.Join(",", rest)}");
4  }
```

Тут { Length: >= 3 } — патерн властивостей (перевіряє довжину), and [...] — патерн списків (зіставляє структуру). Разом — єдина умова. Щоправда, в цьому конкретному прикладі перевірка довжини надлишкова: [var first, _, _, .. var rest] сам по собі вимагає щонайменше трьох елементів. Комбінація має сенс, коли властивість обмежує те, чого патерн списку не виражає, — наприклад, верхню межу: { Length: <= 10 } and [> 100, ..] («коротка серія, що почалась із тахікардії»).

Де патерни списків практично корисні

Патерни списків особливо виразні в задачах, де структура масиву сама по собі несе інформацію:

- **Парсинг рядків команд**: ["add", var name, var value], ["remove", var id], []
- **Валідація форматів**: масив токенів повинен починатись і закінчуватись певними маркерами
- **Обробка кадрів протоколу**: перший байт — тип, останній — контрольна сума
- **Аналіз трендів**: [> threshold, .., < threshold] — значення перетнуло поріг

Для великих або нескінченних послідовностей (IEnumerable<T>) патерни списків не підходять — там слід використовувати LINQ або явні цикли.

Питання для самоконтролю — Розділ 9. Pattern matching

9.1. Патерн типів

1. Що таке **pattern matching** у C# і яку проблему він вирішує порівняно зі старим підходом через is + явний cast?
2. Що виведе наступний код і чому?

```
C#
object value = "Гіпертонія";
if (value is string s)
    Console.WriteLine(s.ToUpper());
else
    Console.WriteLine("не рядок");
Console.WriteLine(s); // чи компілюється цей рядок?
```

3. Поясніть, у чому різниця між трьома формами:

```
C#
// Форма A
if (staff is Doctor) { Doctor doc = (Doctor)staff; doc.PrescribeMeds(); }
// Форма B
Doctor? doc = staff as Doctor;
if (doc != null) doc.PrescribeMeds();
// Форма C
if (staff is Doctor doc) doc.PrescribeMeds();
```

Яка форма є найкращою і чому?

4. Поясніть різницю між **switch statement** і **switch expression**. Коли варто обирати кожен варіант? Який з них повертає значення?
5. Чому в switch конкретніші типи мають іти **перед** базовими? Що станеться, якщо розмістити MedicalStaff перед Doctor? Чи попередить компілятор про помилку?
6. Напишіть метод Describe(object obj), який через **switch expression** повертає рядок опису для: int (значення числа), string (значення рядка), null (рядок "null"), будь-якого іншого типу (назва типу через GetType().Name).
7. Що таке **constant pattern** і де він використовується? Поясніть, чим is not null відрізняється від != null з точки зору компілятора.
8. Порівняйте switch expression з type pattern і класичний ланцюг if-else + cast за такими критеріями: кількість операцій для перевірки типу, ризик InvalidCastException, можливість повертати значення, підтримка нових типів у ієрархії.

9.2. Патерн властивостей

9. Що таке **патерн властивостей** і чим він доповнює патерн типів? Запишіть синтаксис у двох формах: без перевірки типу і з перевіркою типу одночасно.
10. Що виведе наступний код?

```
C#
1  var p = new Patient("Коваль", "Active", priority: 2);
2  string result = p switch
3  {
4      { Status: "Active", Priority: > 3 } => "Терміново",
5      { Status: "Active", Priority: > 1 } => "Плановий",
6      { Status: "Active" }                => "Рутинний",
7      _                                   => "Інше",
8  };
9  Console.WriteLine(result);
```

11. Навіщо в патерні властивостей використовується var ім'я замість конкретного значення? Наведіть приклад, коли це зручно.
12. Поясніть різницю між синтаксисом вкладених об'єктів до C# 10 і після C# 10:

```
C#
1  // До C# 10
2  if (patient is { Department: { Name: "Кардіологія" } })
3  // C# 10+
4  if (patient is { Department.Name: "Кардіологія" })
```

Чи є між ними функціональна різниця?

13. Що означає патерн {} (порожні фігурні дужки) у switch expression? Чим він відрізняється від _?
14. Спроєктуйте switch expression для класу MedicalStaff з підкласами Doctor і Nurse, де для лікаря потрібно перевіряти IsOnLeave і Specialty, а для медсестри — Ward. Поєднайте type pattern та property pattern.
15. Чому вираз Priority: > 3 у патерні властивостей є поєднанням **двох** різних видів патернів? Поясніть, як вони вкладаються.
16. Знайдіть помилку у наступному коді і поясніть, як її виправити:

```
C#
1  string GetStatus(Patient? p) => p switch
2  {
3      { Status: "Active" } => "Активний",
4      {}                   => "Неактивний",
5      null                 => "Null"
6  };
```

9.3. Патерни кортежів

17. Що таке **кортежний патерн** і коли його слід обирати замість патерну властивостей або патерну типів? Наведіть відмінну рису сценарію застосування.
18. Що виведе наступний код?

```
C#
1  string result = ("Хірургія", "Ніч") switch
2  {
3      ("Кардіологія", "День")  => "Плановий",
4      ("Хірургія",    "День")  => "Операційний зал",
5      ("Хірургія",    _)       => "Ургентна хірургія",
6      (_,             "День")  => "Денний прийом",
7      _                        => "Позаплановий",
8  };
9  Console.WriteLine(result);
```

19. Поясніть роль discard (_) в кортежному патерні. Чим він відрізняється від var?
20. Чому **порядок arm-ів** у кортежному switch має принципове значення? Що станеться, якщо arm з _ розмістити першим?
21. Напишіть функцію string GetRiskLevel(int systolic, int age), яка за допомогою кортежного патерну визначає рівень ризику: критичний (тиск >= 180), підвищений + похилий вік (>= 140, >= 60), підвищений молодший, норма.
22. Що означає поява >= 60 and < 120 в позиції кортежного патерну? Які ще патерни можна вставляти в позиції кортежу?
23. Що відбувається у runtime, якщо жоден arm у switch expression не збігається і відсутній _? Як цього уникнути?
24. Порівняйте кортежний патерн і вкладені if-else для логіки, яка залежить від двох незалежних вхідних значень. Які переваги і недоліки кожного підходу?

9.4. Позиційний патерн

25. Що таке **позиційний патерн** і в чому його відмінність від кортежного патерну? Яка технічна вимога до класу, щоб він підтримував позиційний патерн?
26. Поясніть три кроки, які виконує компілятор при зіставленні об'єкта з позиційним патерном (pat1, pat2).

27. Чому позиційний record **автоматично** підтримує позиційний патерн без додаткового коду? Що генерує компілятор для record VitalSigns(double Temperature, int Pulse)?
28. Що виведе наступний код?

```C#
public record VitalSigns(double Temperature, int Pulse, int OxygenSat);
var v = new VitalSigns(38.8, 105, 96);
string result = v switch
{
    (>= 39.0, > 90, _)          => "КРИТИЧНО",
    (>= 38.5, _, _)             => "Жар",
    (_, _, < 90)                => "Гіпоксія",
    (var t, var p, var o2)      => $"Стан: {t} {p} {o2}",
};
Console.WriteLine(result);
```

29. Клас Appointment містить три поля: PatientName, Type, Department. У Deconstruct розкрито лише Type і Department. Чому не PatientName? Як тоді отримати PatientName при зіставленні?
30. Напишіть клас BloodPressure з полями Systolic і Diastolic, додайте Deconstruct, і реалізуйте метод string Assess(BloodPressure bp) через позиційний патерн.
31. Порівняйте кортежний та позиційний патерни за такими критеріями: тип вхідних даних, вимоги до класу, типовий сценарій використання.
32. Чи можна поєднати позиційний патерн з патерном властивостей в одному arm? Якщо так — наведіть приклад; якщо ні — поясніть чому.

9.5. Реляційний та логічний патерни

33. Що таке **реляційний патерн** і чим він відрізняється від звичайного оператора порівняння >? У яких місцях коду може стояти реляційний патерн?
34. Поясніть принципову різницю між && та and у наступних фрагментах:

```C#
if (temp > 37.5 && temp < 39.0) { ... }   // A
if (temp is > 37.5 and < 39.0) { ... }    // B
```

Чи однаковий результат? Чи однакова природа операторів?

35. Що виведе наступний код?

```
C#
 1  double[] temps = { 34.5, 36.6, 37.8, 39.2 };
 2  foreach (var t in temps)
 3  {
 4      string r = t switch
 5      {
 6          < 35.0              => "Гіпотермія",
 7          >= 35.0 and < 36.0  => "Нижче норми",
 8          >= 36.0 and <= 37.0 => "Норма",
 9          > 37.0 and < 38.5   => "Субфебрильна",
10          _                   => "Жар або вище",
11      };
12      Console.WriteLine($"{t}: {r}");
13  }
```

36. Яка різниця між not null як патерном і != null як умовою? Як компілятор трактує is not null?
37. Поясніть пріоритет операторів and та or у патернах. Що означає вираз temperature is < 35 or > 39? Чи потрібні дужки для явного групування?
38. Напишіть метод string AssessPulse(int pulse), який через логічний патерн визначає: брадикардія (< 60), тахікардія (> 100), норма (60–100), критична тахікардія (> 150).
39. Покажіть три місця, де можна використовувати реляційний патерн: після is, в позиції кортежного патерну, в патерні властивостей. Наведіть конкретні приклади коду.
40. Чому not Doctor у контексті staff is not Doctor є **патерним оператором**, а не логічним запереченням? Що саме заперечує not у цьому виразі?

9.6. Патерни списків

41. Що таке **патерн списків** (list patterns)? В якій версії C# вони з'явились і які типи підтримують цей патерн?
42. Чому IEnumerable<T> без Count/Length **не підтримується** патерном списків? Що генерує компілятор для перевірки [a, b, c]?
43. Що виведе наступний код?

```
C#
 1  int[] data = { 130, 120, 95, 72 };
 2  string result = data switch
 3  {
```

```
    []              => "Порожній",
    [var single]    => $"Одне: {single}",
    [> 100, .., > 100] => "Стійка тахікардія",
    [var f, .., var l] when f > l + 20 => $"Спадання: {f}→{l}",
    [var f, .., var l] => $"Стан: {f}..{l}",
};
Console.WriteLine(result);
```

44. Поясніть синтаксис .. (slice) у патерні списків. Скільки разів він може з'являтися в одному патерні і чому?
45. Що означає патерн [var first, .., var last]? Що буде, якщо масив містить рівно 2 елементи? Що якщо 1 елемент?
46. Напишіть метод string ParseCommand(string[] args), який через патерн списків обробляє команди: ["add", var name] → додати, ["remove", var id] → видалити, ["list"] → список, [] → порожня команда, _ → невідома команда.
47. Як поєднати патерн списків з патерном властивостей в одній умові? Наведіть приклад з and.
48. Для яких практичних задач патерн списків є особливо виразним інструментом? Назвіть принаймні три сценарії. Чому він непридатний для IEnumerable<T> з великою кількістю елементів?

Розділ 10. Колекції

10.1. Список List<T>

Масив — найпростіша структура для зберігання однотипних елементів, але він має фундаментальне обмеження: **фіксований розмір**. Після створення масиву до нього не можна додати новий елемент без виділення нового масиву більшого розміру і копіювання всіх даних. Якщо ми наперед не знаємо скільки пацієнтів буде в листі очікування, або скільки записів потрапить у добовий журнал — масив незручний.

Клас List<T> із простору імен System.Collections.Generic вирішує цю проблему: він надає **динамічний масив** — структуру, що автоматично розширюється при потребі, зберігаючи при цьому зручний індексований доступ. List<T> є найбільш уживаною колекцією в C# і в більшості задач є правильним вибором за замовчуванням.

Внутрішня структура: динамічний масив

List<T> — це не магія. Всередині він зберігає звичайний масив T[]. Але коли цей масив заповнюється, List<T> автоматично виконує три кроки:

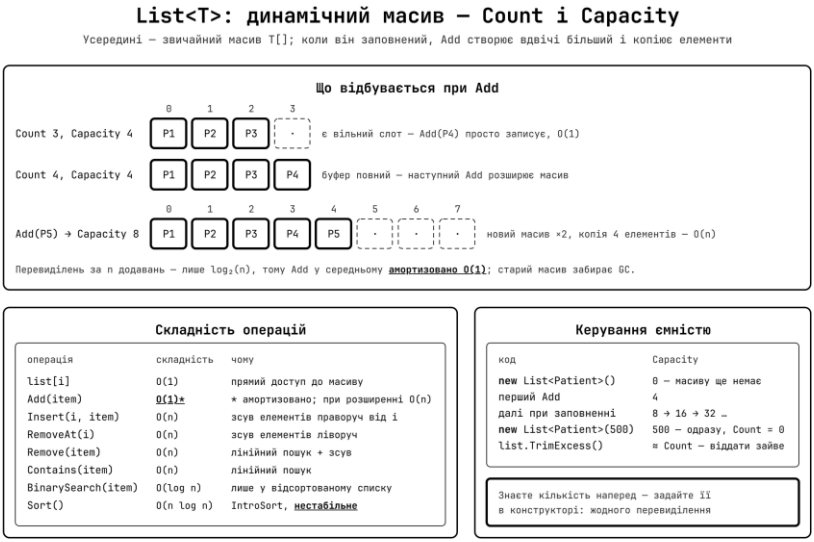
1. Виділяє новий масив **вдвічі більшого розміру**.
2. Копіює всі існуючі елементи в новий масив.
3. Відпускає старий масив — він стає кандидатом для збирача сміття.

Уточнимо, звідки починається це подвоєння. Список, створений конструктором без параметрів, **взагалі не виділяє масиву**: його Capacity дорівнює 0 (усередині — спільний порожній масив). Перший Add виділяє буфер на 4 елементи, а далі ємність при кожному заповненні подвоюється: 4 → 8 → 16 → 32 і так далі. Саме цю послідовність показує схема нижче: при Count = Capacity = 4 наступний Add(P5) створює масив на 8 комірок і копіює туди чотири наявні елементи.

Тому List<T> має дві характеристики розміру:

— **Count** — кількість реальних елементів у списку. Саме це значення повертається при list.Count.

— **Capacity** — розмір внутрішнього масиву-буфера (скільки елементів можна вмістити до наступного розширення).



List<T>: динамічний масив — Capacity vs Count

Стратегія подвоєння забезпечує **амортизовану O(1)** складність для Add: більшість операцій записують елемент у вільний слот (O(1)), а перевиділення пам'яті відбувається $\log_2(n)$ разів за n додавань. Якщо ви знаєте заздалегідь скільки елементів очікується — вкажіть Capacity у конструкторі, щоб уникнути зайвих перевиділень:

```
C#
1  List<Patient> patients = new List<Patient>(500); // буфер одразу на 500
```

Зворотна операція — TrimExcess(): якщо список уже наповнено і більше не зростатиме, цей метод зменшує Capacity до Count (точніше — робить це, коли зайвого місця понад 10 %), щоб не тримати в пам'яті порожні комірки.

Складність ключових операцій

Операція	Складність	Причина
Add(item)	O(1)*	Запис у кінець; * амортизована

Операція	Складність	Причина
list[i]	O(1)	Прямий доступ за індексом
Insert(i, item)	O(n)	Зсув елементів праворуч від i
Remove(item)	O(n)	Лінійний пошук + зсув
RemoveAt(i)	O(n)	Зсув елементів ліворуч від i
Contains(item)	O(n)	Лінійний пошук
Sort()	O(n log n)	IntroSort, нестабільне

Варто знати, як саме сортує List<T>.Sort(). Це **інтроспективне сортування** (IntroSort): воно починає як швидке сортування (QuickSort), на дуже глибокій рекурсії перемикається на пірамідальне (HeapSort), а малі відрізки дочищає сортуванням вставками. Так гарантується O(n log n) навіть у найгіршому випадку, на якому «чистий» QuickSort деградує до $O(n^2)$. Важливий наслідок: сортування **нестабільне** — елементи з однаковим ключем можуть помінятися місцями відносно початкового порядку. Якщо порядок рівних важливий (наприклад, пацієнти з однаковим пріоритетом мають лишитися в порядку надходження), використовуйте стабільне сортування OrderBy з LINQ або додавайте до порівняння другий ключ.

Вставка і видалення в середині — O(n). Це ключовий момент: якщо операції вставки/видалення в середині часті — розгляньте LinkedList<T>. Якщо потрібен тільки швидкий пошук — Dictionary<K,V>.

Синтаксис: створення та ініціалізація

List<T> типізується елементом, що зберігається:

```
C#
1  // Порожній список
2  List<string> diagnoses = new List<string>();
3
4  // Ініціалізація з елементами одразу
5  var diagnoses = new List<string> { "Гіпертонія", "Діабет", "Аритмія" };
6
7  // Заданою початковою ємністю (без елементів!)
8  var patients = new List<Patient>(100);
```

```
 9
10  // З іншої колекції або масиву
11  string[] arr = { "P001", "P002" };
12  var ids = new List<string>(arr);
```

У C# 9+ допускається new() без повторення типу, якщо він виводиться з контексту:

```
C#
1  List<string> list = new() { "A", "B", "C" };
```

Основні операції: CRUD пацієнтів у відділенні

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3
 4  // Виконуваний код
 5  var ward = new List<string> { "Петренко І.", "Коваль М.", "Сидоренко О." };
 6
 7  Console.WriteLine($"Пацієнтів у відділенні: {ward.Count}");
 8
 9  // Додавання одного і кількох
10  ward.Add("Мельник Г.");
11  ward.AddRange(new[] { "Бойко Т.", "Шевченко В." });
12  Console.WriteLine($"Після надходжень: {ward.Count}");
13
14  // Перебір — foreach через IEnumerable<T>
15  Console.WriteLine("Поточний список:");
16  foreach (var name in ward)
17      Console.WriteLine($"  {name}");
18
19  // Доступ за індексом — O(1)
20  Console.WriteLine($"\nПерший: {ward[0]}");
21  Console.WriteLine($"Останній: {ward[ward.Count - 1]}");
22
23  // Вставка ПЕРЕД конкретним індексом — O(n)
24  ward.Insert(0, "ТЕРМІНОВО: Іванченко Д.");
25  Console.WriteLine($"\nПісля термінової вставки: {ward[0]}");
26
27  // Видалення за значенням і за індексом
28  ward.Remove("Коваль М.");     // O(n) — знайти і видалити
29  ward.RemoveAt(0);              // O(n) — зсув після видалення
30  Console.WriteLine($"Після виписки: {ward.Count} пацієнтів");
```

Пошук, фільтрація, сортування

List<T> містить методи вищого порядку, що приймають предикати — делегати або лямбда-вирази:

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3
```

```
// Виконуваний код
var patients = new List<string>
{
    "Петренко І. (Кардіо)",
    "Коваль М. (Невро)",
    "Сидоренко О. (Кардіо)",
    "Мельник Г. (Хірург)",
    "Бойко Т. (Кардіо)",
};

// Contains — точний збіг
bool hasMelnyk = patients.Contains("Мельник Г. (Хірург)");
Console.WriteLine($"Мельник є в списку: {(hasMelnyk ? "так" : "ні")}");

// Find — перший що відповідає умові
string first = patients.Find(p => p.Contains("Кардіо"));
Console.WriteLine($"Перший кардіо: {first}");

// FindAll — всі що відповідають
var cardio = patients.FindAll(p => p.Contains("Кардіо"));
Console.WriteLine($"Кардіологія ({cardio.Count}):");
foreach (var p in cardio) Console.WriteLine($"  {p}");

// RemoveAll — видалити всіх виписаних (умова)
int removed = patients.RemoveAll(p => p.Contains("Хірург"));
Console.WriteLine($"Виписано з хірургії: {removed}");

// Sort — алфавітне сортування
patients.Sort();
Console.WriteLine("Відсортований список:");
foreach (var p in patients) Console.WriteLine($"  {p}");
```

Методи Find, FindAll, Exists, RemoveAll приймають Predicate<T> — делегат виду bool(T item). Зазвичай передається лямбда. Це значно виразніше, ніж ручний цикл з перевіркою умови.

Довідник методів

Метод	Що робить	Складність
Add(item)	Додати в кінець	O(1)*
AddRange(coll)	Додати колекцію в кінець	O(k)
Insert(i, item)	Вставити за індексом	O(n)
InsertRange(i, coll)	Вставити колекцію за індексом	O(n+k)
Remove(item)	Видалити перше входження	O(n)

Метод	Що робить	Складність
RemoveAt(i)	Видалити за індексом	O(n)
RemoveRange(i, count)	Видалити діапазон	O(n)
RemoveAll(pred)	Видалити всі що відповідають умові	O(n)
Clear()	Очистити список	O(n)
Contains(item)	Чи є елемент	O(n)
Exists(pred)	Чи є елемент що відповідає умові	O(n)
Find(pred)	Перший що відповідає умові	O(n)
FindLast(pred)	Останній що відповідає умові	O(n)
FindAll(pred)	Всі що відповідають умові	O(n)
IndexOf(item)	Індекс першого входження	O(n)
GetRange(i, count)	Підсписок	O(k)
Sort()	Сортування	O(n log n)
Reverse()	Розвернути порядок	O(n)
CopyTo(arr)	Скопіювати в масив	O(n)
BinarySearch(item)	Бінарний пошук (список має бути відсортованим)	O(log n)

Коли List<T>, а коли масив?

Критерій	List<T>	T[] (масив)
Розмір відомий наперед	Можна обидва	Масив дещо ефективніший

Критерій	List<T>	T[] (масив)
Розмір змінюється	**List**	Незручний — треба перевиділяти
Часте додавання в кінець	**List**	Масив не підходить
Мінімальний overhead пам'яті	Масив	—
Потрібен foreach + LINQ	Обидва (обидва IEnumerable)	—
Повернення з методу	Обидва	—

Загальна рекомендація: **за замовчуванням використовуйте List<T>**. Переходьте на масив лише коли є конкретна причина: фіксований розмір відомий наперед і продуктивність критична.

10.2. Двозв'язаний список LinkedList<T>

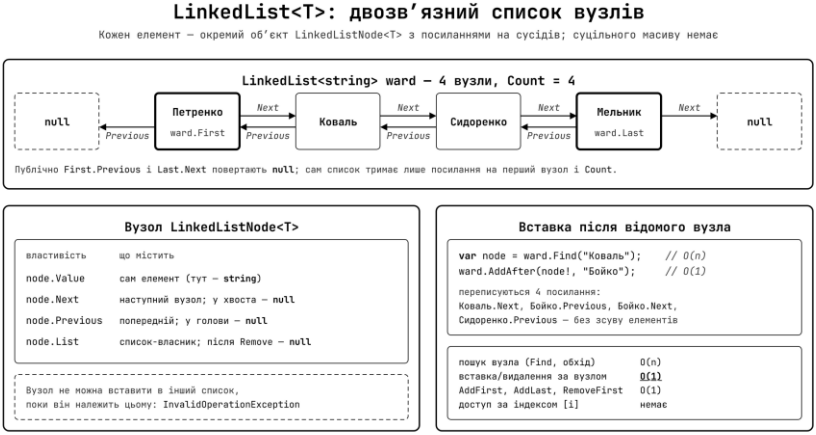
List<T> — потужна колекція загального призначення, але вона побудована на масиві. Це означає, що вставка або видалення елемента **не в кінці** списку вимагає зсуву всіх сусідніх елементів: O(n) за часом. Якщо ваш алгоритм часто додає або видаляє елементи на початку, у кінці або після конкретного відомого елемента — LinkedList<T> може бути значно ефективнішим вибором.

Клас LinkedList<T> з простору імен System.Collections.Generic представляє **двозв'язний список** — лінійну структуру даних, в якій кожен елемент зберігає посилання одночасно на **наступний** і **попередній** елемент. Завдяки цьому вставка та видалення на відомій позиції виконуються за O(1) — без жодних зсувів.

Внутрішня структура

На відміну від List<T>, де елементи зберігаються в суцільному масиві пам'яті, LinkedList<T> зберігає кожен елемент окремо у вигляді **вузла** — об'єкта класу LinkedListNode<T>. Вузли з'єднані між собою посиланнями: кожен знає свого сусіда ліворуч і праворуч. Логічно список має **голову** (перший вузол, First) і **хвіст** (останній вузол, Last); обидва доступні за O(1).

Цікава деталь реалізації .NET: усередині LinkedList<T> зберігає **лише посилання на голову** (поле head) і лічильник. Внутрішні посилання вузлів замкнені в кільце — «попередній» для голови насправді є хвостом, тому Last обчислюється як head.prev без жодного перебору. Назовні це кільце не видно: публічні властивості First.Previous і Last.Next повертають null, і саме так список і варто уявляти, як на схемі.



LinkedList<T> — структура двозв'язного списку

Коли потрібно вставити вузол між двома існуючими, достатньо переписати чотири посилання — без копіювання даних. Саме тому вставка і видалення на відомій позиції — O(1). Але є і зворотний бік: **індексований доступ** (list[i]) відсутній — щоб дістатись до і-го елемента, треба пройти від голови до нього крок за кроком, що дає O(n).

LinkedListNode<T>: вузол списку

Якщо у List<T> кожен елемент — це просто значення типу T, то в LinkedList<T> кожен елемент обгортається у вузол LinkedListNode<T>. Саме через вузли виконуються всі операції вставки після/перед конкретним елементом.

Клас LinkedListNode<T> має чотири основні властивості:

— Value — значення вузла типу T. Єдине місце, де зберігається сам елемент.

— Next — посилання на наступний вузол типу LinkedListNode<T>. У хвостового вузла — null.

— Previous — посилання на попередній вузол типу LinkedListNode<T>. У головного вузла — null.

— List — посилання на список-власник LinkedList<T>. Після видалення вузла зі списку стає null.

Властивість List пояснює важливе обмеження: вузол завжди належить щонайбільше одному списку. Спроба передати в AddLast, AddAfter чи інший метод вузол, що вже входить до якогось списку, кидає InvalidOperationException; так само AddAfter(node, …) перевіряє, що node належить саме цьому списку. (З .NET 7 є ще ValueRef — посилання ref на значення, щоб змінювати великі структури без копіювання.)

Зберігши посилання на конкретний вузол у змінній, ви отримуєте точку вставки: можна додати нового сусіда за O(1), не шукаючи позицію повторно.

Властивості LinkedList<T>

Сам клас LinkedList<T> визначає три властивості:

— Count — кількість елементів у списку.

— First — перший вузол списку як об'єкт LinkedListNode<T>. Для порожнього списку — null.

— Last — останній вузол списку. Для порожнього списку — null.

Створення та ініціалізація

LinkedList<T> можна створити порожнім або ініціалізувати з будь-якої колекції, що реалізує IEnumerable<T>:

```
C#
1  // Порожній список
2  LinkedList<string> queue = new LinkedList<string>();
3
4  // З іншої колекції
5  var names = new List<string> { "Петренко", "Коваль", "Сидоренко" };
6  LinkedList<string> fromList = new LinkedList<string>(names);
7
8  // З масиву
9  LinkedList<string> fromArray = new LinkedList<string>(new[] { "A", "B", "C" });
```

Обхід вперед і назад — runnable приклад

Одна з практичних переваг LinkedList<T> — можливість рухатись у **будь-якому напрямку** через властивості Next і Previous. Це корисно, наприклад, при роботі з хронологічними записами, де потрібен перегляд як від найстаршого до найновішого, так і у зворотному порядку:

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3
 4  // Денний журнал прийомів у відділенні
 5  var journal = new LinkedList<string>(new[]
 6  {
 7      "08:00 — Петренко І. (кардіо)",
 8      "09:30 — Коваль М. (невро)",
 9      "11:00 — Сидоренко О. (кардіо)",
10      "13:15 — Мельник Г. (хірург)"
11  });
12
13  Console.WriteLine($"Прийомів за день: {journal.Count}");
14  Console.WriteLine($"Перший:  {journal.First?.Value}");
15  Console.WriteLine($"Останній: {journal.Last?.Value}");
16
17  // Прямий обхід — від ранку до вечора
18  Console.WriteLine("\n--- Журнал (хронологічно) ---");
19  var node = journal.First;
20  while (node != null)
21  {
22      Console.WriteLine($"  {node.Value}");
23      node = node.Next;
24  }
25
26  // Зворотній обхід — від останнього прийому
27  Console.WriteLine("\n--- Журнал (від останнього) ---");
28  node = journal.Last;
29  while (node != null)
30  {
31      Console.WriteLine($"  {node.Value}");
32      node = node.Previous;
33  }
```

Методи LinkedList<T>

LinkedList<T> надає повний набір методів для роботи з вузлами. Ключова особливість: методи AddAfter, AddBefore, Remove(node) приймають **вузол** (LinkedListNode<T>), а не значення — тому для цільової вставки потрібно спочатку отримати або зберегти посилання на вузол.

Додавання:

— AddFirst(T value) / AddFirst(LinkedListNode<T> node) — вставляє на початок списку.

— AddLast(T value) / AddLast(LinkedListNode<T> node) — вставляє в кінець списку.

— AddAfter(LinkedListNode<T> node, T value) / AddAfter(node, LinkedListNode<T> newNode) — вставляє після вказаного вузла.

— AddBefore(LinkedListNode<T> node, T value) / AddBefore(node, LinkedListNode<T> newNode) — вставляє перед вказаним вузлом.

Всі методи Add* повертають LinkedListNode<T> — посилання на щойно створений вузол. Збережіть його, якщо плануєте звертатись до цього вузла пізніше.

Видалення:

— RemoveFirst() — видаляє перший вузол. Новим першим стає наступний за ним.

— RemoveLast() — видаляє останній вузол.

— Remove(LinkedListNode<T> node) — видаляє конкретний вузол за посиланням: O(1).

— Remove(T value) — знаходить перший вузол із вказаним значенням і видаляє його: O(n).

Пошук:

— Find(T value) — повертає перший LinkedListNode<T> із заданим значенням або null.

— FindLast(T value) — повертає останній відповідний вузол.

— Contains(T value) — перевіряє, чи є значення у списку.

Інше:

— Clear() — очищає весь список.

— CopyTo(T[] array, int index) — копіює елементи в масив.

Маніпуляції з хірургічною чергою — runnable приклад

Розглянемо реалістичний сценарій: черга на хірургічну операцію, де нові пацієнти можуть додаватись як планово (в кінець), так і позачергово (на початок або після конкретного пацієнта):

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3
 4  void PrintQueue(LinkedList<string> q)
```

```
{
    int i = 1;
    var n = q.First;
    while (n != null)
    {
        Console.WriteLine($"  {i++}. {n.Value}");
        n = n.Next;
    }
}

var surgeryQueue = new LinkedList<string>();

// Планові записи — в кінець
surgeryQueue.AddLast("Петренко І. — апендектомія");
surgeryQueue.AddLast("Коваль М.  — холецистектомія");
surgeryQueue.AddLast("Мельник Г. — герніопластика");

Console.WriteLine("=== Первинна черга ===");
PrintQueue(surgeryQueue);

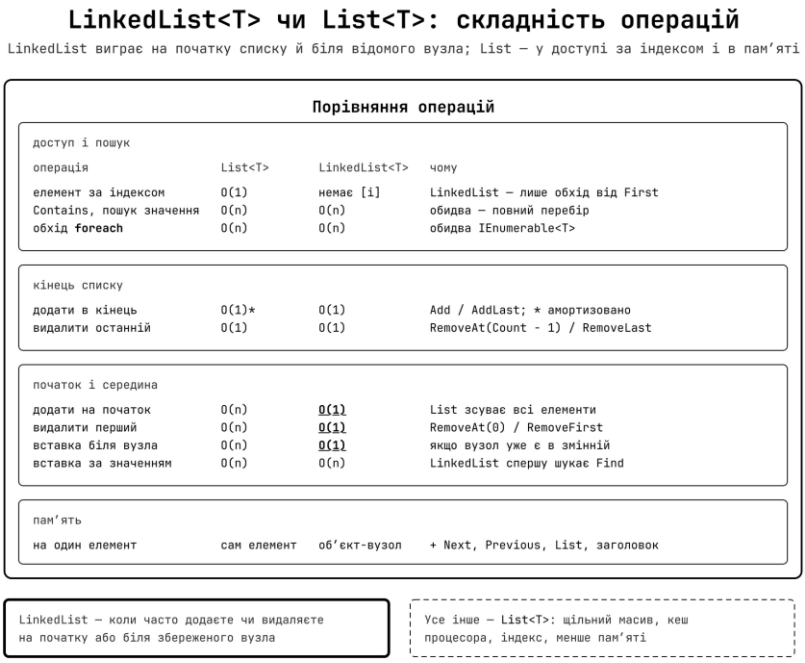
// Термінова операція — першочергово
surgeryQueue.AddFirst("ТЕРМІНОВО: Бойко Т. — перитоніт");
Console.WriteLine($"\nДодано термінового. Перший:
{surgeryQueue.First?.Value}");

// Зберігаємо посилання на вузол для подальшої вставки після нього
var kovalNode = surgeryQueue.Find("Коваль М.  — холецистектомія");
if (kovalNode != null)
{
    surgeryQueue.AddAfter(kovalNode, "Сидоренко О. — лапароскопія");
    Console.WriteLine("Вставлено Сидоренка після Коваля.");
}

Console.WriteLine($"\nВсього в черзі: {surgeryQueue.Count}");
Console.WriteLine("\n=== Фінальна черга ===");
PrintQueue(surgeryQueue);

// Перша операція завершена — видаляємо
surgeryQueue.RemoveFirst();
Console.WriteLine($"\nПісля завершення першої операції: {surgeryQueue.Count} в
черзі.");
Console.WriteLine($"Наступний: {surgeryQueue.First?.Value}");
```

Складність операцій



LinkedList<T> vs List<T> — складність ключових операцій

Ключовий висновок: LinkedList<T> виграє там, де List<T> витрачає O(n) на зсув елементів — а саме при роботі з **початком і серединою** колекції. Але LinkedList<T> програє, де потрібен **прямий доступ за індексом**: його просто немає.

Зверніть увагу й на кінець списку: видалення останнього елемента в List<T> — це RemoveAt(Count - 1), яке нічого не зсуває і теж виконується за O(1). Тож перевага LinkedList<T> — саме початок списку і позиції біля вже відомого вузла, а не кінець.

Також слід враховувати **витрати пам'яті**: кожен елемент LinkedList<T> — це окремий об'єкт LinkedListNode<T> у купі. Окрім значення він зберігає три посилання (Next, Previous і List) і службовий заголовок об'єкта, тоді як List<T> тримає елементи щільно в одному масиві. До того ж вузли розкидані по пам'яті, тож обхід LinkedList<T>

гірше використовує кеш процесора — на практиці навіть O(n)-перебір List<T> часто швидший.

Коли LinkedList<T>, а коли List<T>?

Сценарій	Рекомендація
Часте додавання/видалення на початку або в середині	**LinkedList** — O(1) після знаходження вузла
Реалізація черги з пріоритетами (вставка між елементами)	**LinkedList** — природна модель
Потрібен доступ за індексом (list[i])	**List** — O(1) vs відсутність у LinkedList
Більшість операцій — додавання в кінець і читання	**List** — менший overhead пам'яті
Колекція невелика (< 100 елементів)	**List** — різниця несуттєва, код простіший
LINQ-запити, сортування, пошук за умовою	**List** — краща підтримка в стандартній бібліотеці

Загальна рекомендація: **за замовчуванням використовуйте List<T>**. Обирайте LinkedList<T> лише тоді, коли профіль операцій вашого алгоритму явно вигравав би від O(1) вставки/видалення на краях або після відомого вузла — і ви готові пожертвувати індексованим доступом та додатковою пам'яттю.

10.3. Черга Queue<T>

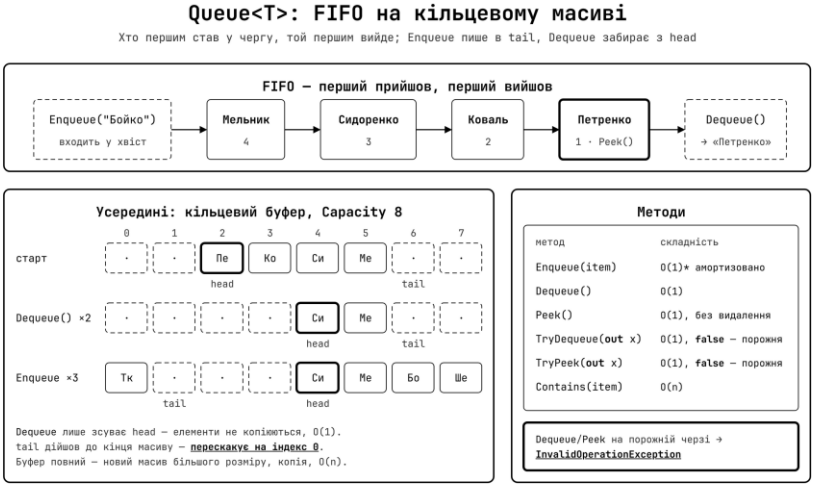
У реальному житті черга — інтуїтивна структура: хто прийшов першим, той і обслуговується першим. У програмуванні цей принцип називається **FIFO** (First In — First Out): елемент, доданий до колекції раніше, вилучається з неї раніше.

Клас Queue<T> з простору імен System.Collections.Generic реалізує саме цей принцип. На відміну від List<T>, де є індексований доступ і вставка в будь-яку позицію, Queue<T> надає лише дві основні операції: **Enqueue** (додати в кінець) і **Dequeue** (забрати з початку). Такий обмежений, але чіткий контракт робить черги ідеальними там, де

важливий порядок обробки: черга пацієнтів на прийом, задачі у виконавця, повідомлення в черзі обробки.

Внутрішня структура

Всередині Queue<T> зберігає елементи у **кільцевому масиві** (circular buffer). На відміну від List<T>, де при видаленні з початку всі елементи зсуваються, черга використовує два індекси — **head** (вказівник на перший елемент) і **tail** (вказівник на наступну вільну позицію). При Dequeue лише head зсувається вперед — жодного копіювання. Тому обидві основні операції виконуються за **O(1)**.



Queue<T> — принцип FIFO

Якщо кільцевий масив заповнюється, Queue<T> виділяє новий масив більшого розміру і копіює дані — аналогічно до List<T>. Тому Enqueue — O(1) **амортизована**: більшість операцій O(1), але іноді виникає O(n) перевиділення.

Створення черги

Queue<T> типізується елементом, що зберігається, і підтримує кілька способів ініціалізації:

```
C#
```

```
// Порожня черга
Queue<string> waitingRoom = new Queue<string>();

// З початковою ємністю (без елементів — лише буфер)
Queue<string> withCapacity = new Queue<string>(32);

// З іншої колекції або масиву
var names = new List<string> { "Петренко", "Коваль", "Мельник" };
Queue<string> fromList = new Queue<string>(names);
```

Властивість Count повертає поточну кількість елементів у черзі.

Методи Queue<T>

Enqueue та Dequeue

Enqueue(T item) додає елемент до хвоста черги, Dequeue() вилучає і повертає елемент з голови:

```
C#
var queue = new Queue<string>();
queue.Enqueue("А");  // { А }
queue.Enqueue("Б");  // { А, Б }
queue.Enqueue("В");  // { А, Б, В }

var first = queue.Dequeue();  // повертає "А", черга стає { Б, В }
```

Важливо: якщо викликати Dequeue() або Peek() на **порожній** черзі — програма кине InvalidOperationException. Завжди перевіряйте Count > 0 або використовуйте Try-варіанти методів.

Peek — підглянути без видалення

Peek() повертає перший елемент черги, але не видаляє його. Корисно, коли потрібно прийняти рішення на основі поточного елемента, не обробляючи його одразу:

```
C#
if (queue.Count > 0)
{
    var next = queue.Peek();   // прочитали, не видалили
    // ... перевірка ...
    var processed = queue.Dequeue();  // тепер забираємо
}
```

TryDequeue і TryPeek

Безпечні варіанти, що не кидають виняток на порожній черзі. Повертають true, якщо операція успішна, і передають результат через out-параметр:

— bool TryDequeue(out T result) — вилучає перший елемент, якщо черга не порожня.
— bool TryPeek(out T result) — читає перший елемент без видалення, якщо черга не порожня.

Повний перелік методів

Метод	Що робить	Складність
Enqueue(item)	Додає елемент до хвоста	O(1)*
Dequeue()	Вилучає і повертає елемент з голови	O(1)
Peek()	Повертає перший елемент без видалення	O(1)
TryDequeue(out T)	Безпечний Dequeue; повертає false якщо порожня	O(1)
TryPeek(out T)	Безпечний Peek; повертає false якщо порожня	O(1)
Contains(item)	Чи є елемент у черзі	O(n)
Clear()	Очищає чергу	O(n)
CopyTo(arr, i)	Копіює в масив починаючи з індексу i	O(n)
ToArray()	Повертає новий масив зі збереженням порядку	O(n)

Черга реєстратури — runnable приклад

Змоделюємо ранкову чергу пацієнтів у реєстратурі. Нові пацієнти займають чергу, реєстратор обслуговує по одному в порядку надходження:

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3
 4  var registrationDesk = new Queue<string>();
```

```
 5
 6  // Пацієнти записуються в чергу
 7  registrationDesk.Enqueue("Петренко І. — прийом 08:00");
 8  registrationDesk.Enqueue("Коваль М.   — аналізи");
 9  registrationDesk.Enqueue("Сидоренко О. — кардіолог");
10  registrationDesk.Enqueue("Мельник Г.  — терапевт");
11
12  Console.WriteLine($"Черга на реєстрацію: {registrationDesk.Count} осіб");
13  Console.WriteLine($"Перший у черзі: {registrationDesk.Peek()}");
14  Console.WriteLine("");
15
16  // Реєстратор обслуговує по одному
17  Console.WriteLine("=== Процес реєстрації ===");
18  while (registrationDesk.Count > 0)
19  {
20      var patient = registrationDesk.Dequeue();
21      Console.WriteLine($"✓ Зареєстровано: {patient}");
22      Console.WriteLine($"  Залишилось у черзі: {registrationDesk.Count}");
23  }
24
25  Console.WriteLine("\nРеєстрацію завершено.");
26
27  // TryDequeue на порожній черзі — безпечний варіант
28  var success = registrationDesk.TryDequeue(out var next);
29  Console.WriteLine($"\nСпроба Dequeue з порожньої черги: {(success ? "успішно" :
    "черга порожня")}");
```

Черга пацієнтів у лікаря — runnable приклад

Розширений приклад із класами: черга пацієнтів передається лікарю, який послідовно їх приймає:

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3
 4  // Виконуваний код
 5  var patients = new Queue<Patient>();
 6  patients.Enqueue(new Patient("Петренко Іван",    "гіпертонія"));
 7  patients.Enqueue(new Patient("Коваль Марія",     "діабет"));
 8  patients.Enqueue(new Patient("Сидоренко Олег",   "аритмія"));
 9  patients.Enqueue(new Patient("Мельник Ганна",    "мігрень"));
10
11  Console.WriteLine($"Очікують прийому: {patients.Count} пацієнтів\n");
12
13  var doctor = new Doctor("Іванченко В.О.", "кардіолог");
14  doctor.ConductReception(patients);
15
16  Console.WriteLine($"\nЧерга після прийому: {patients.Count} осіб");
17
18  // Класи — після виконуваного коду
19  class Patient
20  {
21      public string Name { get; }
22      public string Diagnosis { get; }
23      public Patient(string name, string diagnosis)
24      {
```

```
25          Name = name;
26          Diagnosis = diagnosis;
27      }
28  }
29
30  class Doctor
31  {
32      private string _name;
33      private string _speciality;
34
35      public Doctor(string name, string speciality)
36      {
37          _name = name;
38          _speciality = speciality;
39      }
40
41      public void ConductReception(Queue<Patient> queue)
42      {
43          Console.WriteLine($"Лікар {_name} ({_speciality}) розпочинає прийом:");
44          int num = 1;
45          while (queue.TryDequeue(out var patient))
46          {
47              Console.WriteLine($"  {num++}. {patient.Name} — скарга:
    {patient.Diagnosis}");
48          }
49          Console.WriteLine("Прийом завершено.");
50      }
51  }
```

Складність операцій

Queue<T>, Stack<T> і List<T>: складність операцій

Черга й стек працюють лише з кінцями — зате обидва кінці коштують O(1)

Порівняння

операція	Queue<T>	Stack<T>	List<T>	примітка
додати				
в кінець / на вершину	Enqueue O(1)*	Push O(1)*	Add O(1)*	* амортизовано: буфер ×2
забрати				
перший (FIFO)	Dequeue **O(1)**	—	RemoveAt(0) O(n)	List зсуває всі елементи
останній (LIFO)	—	Pop **O(1)**	RemoveAt(Count-1) O(1)	кінець масиву — без зсуву
підглянути	Peek O(1)	Peek O(1)	list[0] O(1)	без видалення
решта				
Contains	O(n)	O(n)	O(n)	лінійний пошук
доступ за індексом	немає	немає	O(1)	черга й стек не індексовані
foreach	від head	від вершини	від 0	порядок обходу = порядок вилучення

Черга на List<T> із RemoveAt(0) — O(n) на кожне вилучення; Queue<T> робить те саме за O(1).

Queue<T> — складність операцій та порівняння з List<T>

Ключова перевага Queue<T> над List<T> — **O(1) для Dequeue**. Якщо ви реалізуєте чергу через List<T> і видаляєте елемент з початку через RemoveAt(0) — кожна така операція зсуває весь список: O(n). При великих обсягах даних це критично.

Коли Queue<T>?

Queue<T> — правильний вибір, коли:

- **Порядок обробки важливий**: потрібно зберегти FIFO-порядок (черга задач, повідомлень, запитів).
- **Операції тільки з кінцями**: ніколи не потрібен доступ до середини колекції.
- **Видалення з початку часте**: List<T> для цього невигідний — O(n) на кожне RemoveAt(0).

Якщо ж потрібен доступ за індексом, пошук по середині, або порядок **LIFO** (останній прийшов — перший вийшов) — розгляньте відповідно List<T> або Stack<T>.

10.4. Stack<T>

Стек — одна з фундаментальних структур даних у програмуванні. Він реалізує принцип **LIFO** (Last In — First Out): останній доданий елемент вилучається першим. Уявіть стопку тарілок: нову тарілку кладуть зверху, і знімають теж зверху. Або одяг у зимовий день — одягаєш шарами (майка → сорочка → светр → куртка), а знімаєш у зворотному порядку.

У медичному контексті стек природно описує **журнал процедур у зворотному порядку**: скасувати потрібно останню введену процедуру, а не першу. Або стек навігації в медичній системі — кнопка «назад» завжди повертає на попередній екран, тобто на вершину стека переходів.

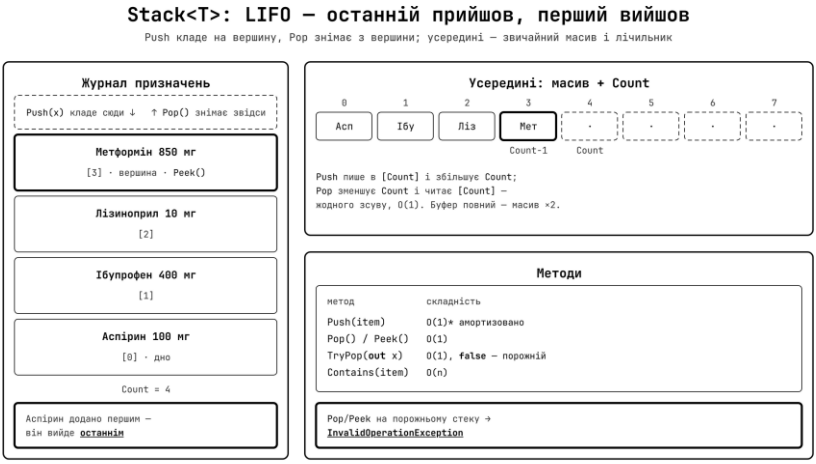
Клас Stack<T> з простору імен System.Collections.Generic реалізує цей принцип ефективно: і Push, і Pop виконуються за **O(1)**.

Внутрішня структура

Всередині Stack<T> зберігає елементи у звичайному масиві та лічильник Count (у коді .NET — поле _size). Вершина стека — це комірка з індексом Count - 1, тобто останній доданий елемент. При Push елемент записується в комірку [Count], після чого Count збільшується;

при Pop спершу зменшується Count, а потім зчитується значення з комірки [Count] (для посилальних типів комірка ще й очищається, щоб не тримати об'єкт від збирача сміття). Жодного зсуву даних — лише зміна одного числа. Звідси O(1) для обох операцій.

При заповненні буфера відбувається стандартне подвоєння — аналогічно List<T>.



Stack<T> — принцип LIFO

Створення стека

Stack<T> підтримує ті ж способи ініціалізації, що й Queue<T>:

```
C#
// Порожній стек
Stack<string> procedureLog = new Stack<string>();

// З початковою ємністю (без елементів)
Stack<string> withCapacity = new Stack<string>(32);

// З іншої колекції — елементи потрапляють у стек у порядку ітерації,
// тобто останній елемент колекції стане вершиною стека
var meds = new List<string> { "Аспірин", "Ібупрофен", "Лізиноприл" };
Stack<string> fromList = new Stack<string>(meds);
// Вершина: Лізиноприл (останній у List — перший у Pop)
```

Властивість Count повертає кількість елементів у стеку.

Важливо: foreach по Stack<T> повертає елементи у порядку **LIFO** — спочатку вершина, потім вниз. Це відрізняється від Queue<T>, де foreach іде від голови до хвоста.

Методи Stack<T>

Push та Pop

Push(T item) кладе елемент на вершину стека, Pop() знімає і повертає вершину:

```
C#
1  var stack = new Stack<string>();
2  stack.Push("А");   // стек: { А }
3  stack.Push("Б");   // стек: { Б, А }   ← Б на вершині
4  stack.Push("В");   // стек: { В, Б, А } ← В на вершині
5
6  var top = stack.Pop();   // повертає "В", стек стає { Б, А }
```

Виклик Pop() або Peek() на **порожньому** стеку кидає InvalidOperationException. Перевіряйте Count > 0 або використовуйте Try-варіанти.

Peek — підглянути без видалення

Peek() повертає вершину стека, але не видаляє її. Корисно, наприклад, щоб перевірити що саме буде знято, перш ніж приймати рішення про Pop:

```
C#
1  if (stack.Count > 0)
2  {
3      var top = stack.Peek();    // прочитали вершину
4      // ... перевірка ...
5      stack.Pop();               // тепер знімаємо
6  }
```

TryPop і TryPeek

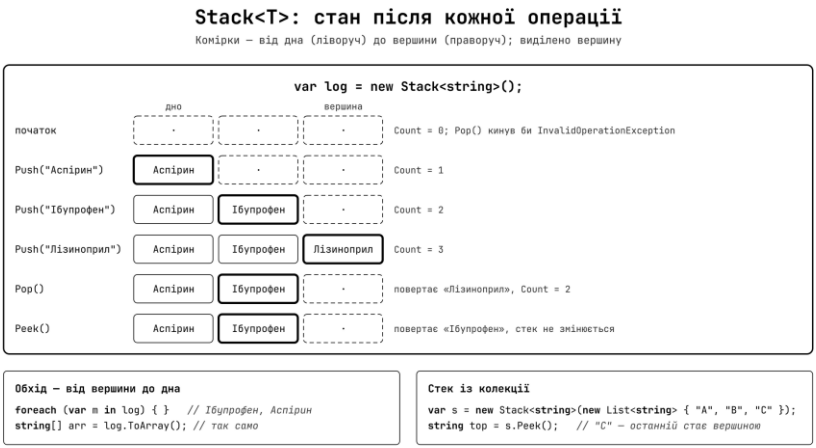
Безпечні варіанти без ризику виключення:

- bool TryPop(out T result) — знімає вершину, якщо стек не порожній; false якщо порожній.
- bool TryPeek(out T result) — читає вершину без видалення; false якщо порожній.

Повний перелік методів

Метод	Що робить	Складність
Push(item)	Кладе елемент на вершину	O(1)*
Pop()	Знімає і повертає вершину	O(1)
Peek()	Повертає вершину без видалення	O(1)
TryPop(out T)	Безпечний Pop; false якщо порожній	O(1)
TryPeek(out T)	Безпечний Peek; false якщо порожній	O(1)
Contains(item)	Чи є елемент у стеку	O(n)
Clear()	Очищає стек	O(n)
ToArray()	Масив у порядку LIFO (вершина першою)	O(n)

Покрокова зміна стану



Stack<T> — покрокова зміна стану при Push та Pop

Журнал призначень — runnable приклад

Стек ідеально підходить для журналу, де важливо обробляти записи у зворотному порядку — наприклад, переглянути або скасувати останнє призначення:

```
C#
using System;
using System.Collections.Generic;

var prescriptionLog = new Stack<string>();

// Лікар вносить призначення протягом дня
prescriptionLog.Push("08:15 — Аспірин 100мг (Петренко І.)");
prescriptionLog.Push("09:40 — Ібупрофен 400мг (Коваль М.)");
prescriptionLog.Push("11:00 — Лізиноприл 10мг (Сидоренко О.)");
prescriptionLog.Push("13:30 — Метформін 850мг (Мельник Г.)");

Console.WriteLine($"Призначень у журналі: {prescriptionLog.Count}");
Console.WriteLine($"Останнє призначення: {prescriptionLog.Peek()}");
Console.WriteLine("");

// Перегляд усіх призначень у зворотному порядку (LIFO)
Console.WriteLine("=== Журнал призначень (від останнього) ===");
foreach (var entry in prescriptionLog)
    Console.WriteLine($"  {entry}");

Console.WriteLine("");

// Скасування останнього призначення — помилка лікаря
var cancelled = prescriptionLog.Pop();
Console.WriteLine($"Скасовано: {cancelled}");
Console.WriteLine($"Поточне останнє: {prescriptionLog.Peek()}");
Console.WriteLine($"Залишилось у журналі: {prescriptionLog.Count}");

// Безпечне зняття всіх записів
Console.WriteLine("\n=== Обробка всіх призначень ===");
while (prescriptionLog.TryPop(out var record))
    Console.WriteLine($"  Оброблено: {record}");

Console.WriteLine($"Стек порожній: {prescriptionLog.Count == 0}");
```

Навігація в медичній карті — runnable приклад

Класичний патерн «undo / назад» — стек переходів між розділами. Кожне відкриття нового розділу кладе поточний на стек; кнопка «назад» знімає зі стеку і повертає до попереднього:

```
C#
using System;
using System.Collections.Generic;

// Виконуваний код
var navigator = new CardNavigator();
navigator.Open("Головна сторінка");
```

```
navigator.Open("Пацієнт: Петренко І.");
navigator.Open("Вкладка: Аналізи");
navigator.Open("Аналіз крові від 2026-01-15");

Console.WriteLine($"Поточний розділ: {navigator.Current}");
Console.WriteLine($"Глибина навігації: {navigator.Depth}");
Console.WriteLine("");

navigator.GoBack();
Console.WriteLine($"Після 'назад': {navigator.Current}");

navigator.GoBack();
Console.WriteLine($"Після 'назад': {navigator.Current}");

navigator.GoBack();
Console.WriteLine($"Після 'назад': {navigator.Current}");

navigator.GoBack(); // спроба з першої сторінки
Console.WriteLine($"Спроба 'назад' з початку: {navigator.Current}");

// Клас — після виконуваного коду
class CardNavigator
{
    private Stack<string> _history = new Stack<string>();
    private string _current = "—";

    public string Current => _current;
    public int Depth => _history.Count;

    public void Open(string section)
    {
        if (_current != "—")
            _history.Push(_current);
        _current = section;
        Console.WriteLine($"  Відкрито: {section}  (history depth: {_history.Count})");
    }

    public void GoBack()
    {
        if (_history.TryPop(out var previous))
            _current = previous;
        else
            Console.WriteLine("  Вже на початковій сторінці — нікуди повертатись.");
    }
}
```

Коли Stack<T>?

Stack<T> — правильний вибір коли:

— Потрібен **зворотний порядок обробки**: останній доданий — перший оброблений (undo, backtrack, відкат).

— Реалізується **рекурсія ітеративно**: стек замінює системний стек викликів.

— Потрібно **відстежувати вкладені стани**: навігація, відкриті теги, баланс дужок.

Питання	Відповідь
Порядок LIFO (скасування, «назад»)?	**Stack**
Порядок FIFO (черга обробки)?	**Queue**
Довільний доступ за індексом?	**List**
Вставка/видалення в середині?	**LinkedList**

10.5. Словник Dictionary<K, V>

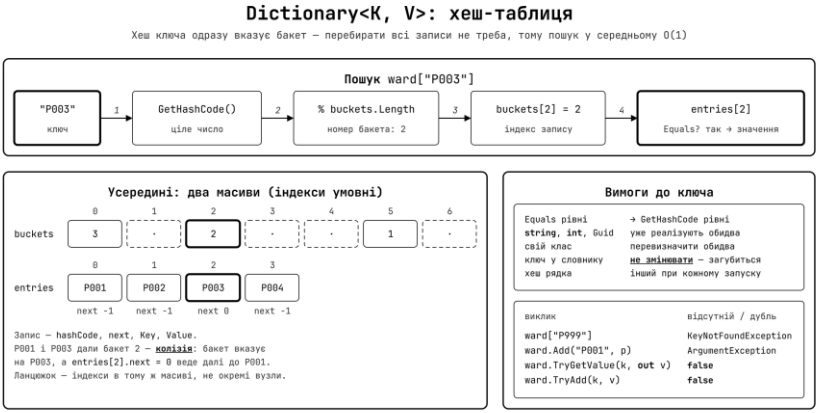
Уявіть задачу: є список пацієнтів, і потрібно швидко знайти пацієнта за його ідентифікатором "P001". При використанні List<T> доведеться перебирати весь список поелементно — O(n). При 10 000 пацієнтів це 10 000 порівнянь у гіршому випадку.

Dictionary<K, V> вирішує цю задачу за **O(1)** в середньому. Це колекція пар **ключ → значення**, де ключ унікальний і забезпечує миттєвий доступ до відповідного значення. Dictionary<K, V> — правильний вибір щоразу, коли потрібен швидкий пошук за ідентифікатором, кодом або будь-яким унікальним атрибутом.

Внутрішня структура: хеш-таблиця

Всередині Dictionary<K, V> зберігає дані у **хеш-таблиці**. Принцип роботи:

4. При додаванні пари key → value обчислюється хеш ключа: key.GetHashCode().
5. За хешем визначається **індекс бакету** (bucket): index = hash % bucketCount.
6. Пара зберігається у відповідному бакеті.
7. При пошуку за ключем ті самі два кроки дають індекс — і значення знаходиться без перебору.



Dictionary<K,V> — внутрішня структура (хеш-таблиця)

Якщо два різних ключі дають однаковий індекс бакету (**колізія**), елементи утворюють у цьому бакеті ланцюжок. У .NET цей ланцюжок влаштовано економно: словник тримає два масиви — buckets (номер бакета → індекс першого запису) і entries (самі записи: збережений хеш, ключ, значення і поле next — індекс наступного запису в ланцюжку). Тобто ланцюжок — це індекси всередині того самого масиву, а не окремі об'єкти-вузли; при пошуку словник проходить ним і порівнює ключі через Equals. Розмір таблиці — просте число (3, 7, 17, …), що краще розподіляє залишки від ділення. Добре реалізований GetHashCode() мінімізує колізії, тому на практиці пошук майже завжди O(1).

Ще одна деталь, яку легко пропустити: хеш рядка в .NET **рандомізований** — "P001".GetHashCode() дає різні числа при кожному запуску програми (захист від атак, що підбирають ключі з однаковим хешем). Тому хеш ніколи не можна зберігати у файл чи базу як «ідентифікатор» рядка.

Тип ключа K повинен коректно реалізувати GetHashCode() і Equals(). Для вбудованих типів (string, int, Guid тощо) це вже зроблено. Для власних класів — потрібно явно перевизначити ці методи або використовувати незмінні типи-значення. Правило узгодженості: якщо a.Equals(b), то a.GetHashCode() == b.GetHashCode(). І ще одне: ключ, що вже лежить у словнику, **не можна змінювати** так, щоб змінився його хеш, — запис залишиться в «старому» бакеті, і словник його більше не знайде.

Створення та ініціалізація

Dictionary<K, V> типізується двома параметрами: тип ключа K і тип значення V:

```
C#
// Порожній словник: int-ключ → рядкове значення
Dictionary<int, string> rooms = new Dictionary<int, string>();

// Ключ — рядок (код пацієнта), значення — рядок (ПІБ)
var patientIndex = new Dictionary<string, string>();

// Ініціалізатор з фігурними дужками — синтаксис { ключ, значення }
var diagnoses = new Dictionary<string, string>()
{
    { "P001", "Гіпертонія" },
    { "P002", "Діабет" },
    { "P003", "Аритмія" }
};

// Альтернативний синтаксис з індексатором [ключ] = значення
var medications = new Dictionary<string, string>()
{
    ["ASP100"] = "Аспірин 100мг",
    ["IBU400"] = "Ібупрофен 400мг",
    ["MET850"] = "Метформін 850мг"
};
```

Обидва синтаксиси рівноцінні — використовуйте той, що читається краще для конкретної задачі.

KeyValuePair<TKey, TValue>

Кожен елемент словника внутрішньо представляється структурою KeyValuePair<TKey, TValue>. Ця структура має дві властивості: Key — ключ елемента і Value — його значення. Вона з'являється при ітерації словника через foreach, при ініціалізації через конструктор, і при роботі з колекцією Keys/Values.

```
C#
// Створення через KeyValuePair (рідкісний сценарій — для ініціалізації з
   колекції)
var pair = new KeyValuePair<string, string>("P001", "Петренко Іван");
var pairs = new List<KeyValuePair<string, string>> { pair };
var dict = new Dictionary<string, string>(pairs);
```

Отримання та зміна елементів

Доступ до елемента відбувається через індексатор dict[key]:

```
C#
```

```
var patientNames = new Dictionary<string, string>()
{
    ["P001"] = "Петренко Іван",
    ["P002"] = "Коваль Марія",
};

// Читання
string name = patientNames["P001"];   // "Петренко Іван"

// Оновлення існуючого ключа
patientNames["P002"] = "Коваль Марія Степанівна";

// Додавання нового ключа — якщо ключ відсутній
patientNames["P003"] = "Сидоренко Олег";
```

Якщо звернутися за **неіснуючим ключем** через dict[key] — буде кинуто KeyNotFoundException. Для безпечного доступу використовуйте TryGetValue.

Властивості Dictionary<K, V>

— Count — кількість пар у словнику.

— Keys — колекція всіх ключів типу Dictionary<K, V>.KeyCollection (реалізує ICollection<K>). Це не копія, а «вікно» на ключі словника; ітерується через foreach.

— Values — колекція всіх значень типу Dictionary<K, V>.ValueCollection (реалізує ICollection<V>).

Порядок перебору словника (і його Keys/Values) **не гарантований** документацією. Поки з словника нічого не видаляли, він на практиці збігається з порядком додавання, але після Remove нові записи займають звільнені місця — покладатися на порядок не можна. Якщо він потрібен, сортуйте ключі явно або використовуйте SortedDictionary<K, V>.

```C#
var rooms = new Dictionary<int, string>()
{
    [101] = "Кардіологія",
    [102] = "Неврологія",
    [201] = "Хірургія"
};

Console.WriteLine($"Палат: {rooms.Count}");

foreach (var roomNum in rooms.Keys)
    Console.WriteLine($"Палата {roomNum}");

foreach (var dept in rooms.Values)
    Console.WriteLine($"Відділення: {dept}");
```

Перебір словника

При foreach кожен елемент словника — це KeyValuePair<K, V>:

```
C#
 1  var patientNames = new Dictionary<string, string>()
 2  {
 3      ["P001"] = "Петренко Іван",
 4      ["P002"] = "Коваль Марія",
 5      ["P003"] = "Сидоренко Олег"
 6  };
 7
 8  foreach (var entry in patientNames)
 9  {
10      Console.WriteLine($"ID: {entry.Key}  Пацієнт: {entry.Value}");
11  }
```

Можна одразу деструктурувати пару. Синтаксис деконструкції з'явився в C# 7, але для KeyValuePair він працює завдяки методу KeyValuePair<K, V>.Deconstruct, доданому в .NET Core 2.0 (у старому .NET Framework його немає):

```
C#
1  foreach (var (id, name) in patientNames)
2  {
3      Console.WriteLine($"{id} → {name}");
4  }
```

Методи Dictionary<K, V>

Метод	Що робить	Складність
Add(key, value)	Додає пару; виняток якщо ключ вже є	O(1)*
TryAdd(key, value)	Додає якщо ключ відсутній; повертає bool	O(1)*
Remove(key)	Видаляє пару за ключем; повертає bool	O(1)*
Remove(key, out V)	Видаляє і повертає значення через out	O(1)*
ContainsKey(key)	Чи є ключ у словнику	O(1)*
ContainsValue(value)	Чи є значення у словнику	O(n)

Метод	Що робить	Складність
TryGetValue(key, out V)	Безпечне читання; false якщо ключ відсутній	O(1)*
Clear()	Очищає словник	O(n)

TryGetValue — найважливіший метод безпечного доступу

TryGetValue — рекомендований спосіб читання зі словника в production-коді. Він не кидає виняток на відсутньому ключі і повертає false:

```
C#
1  if (patientNames.TryGetValue("P999", out var name))
2      Console.WriteLine($"Знайдено: {name}");
3  else
4      Console.WriteLine("Пацієнт P999 не знайдений.");
```

Порівняйте з небезпечним варіантом:

```
C#
1  // НЕБЕЗПЕЧНО — кидає KeyNotFoundException якщо ключ відсутній:
2  var name = patientNames["P999"];
```

Каталог медикаментів — runnable приклад

Медичний каталог: пошук препарату за кодом, оновлення, видалення:

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3
 4  // Каталог: код препарату → назва
 5  var catalog = new Dictionary<string, string>()
 6  {
 7      ["ASP100"] = "Аспірин 100мг",
 8      ["IBU400"] = "Ібупрофен 400мг",
 9      ["MET850"] = "Метформін 850мг",
10      ["LIS10"]  = "Лізиноприл 10мг"
11  };
12
13  Console.WriteLine($"Препаратів у каталозі: {catalog.Count}");
14
15  // Перебір усіх записів
16  Console.WriteLine("\n=== Каталог медикаментів ===");
17  foreach (var (code, name) in catalog)
18      Console.WriteLine($"  [{code}]  {name}");
19
20  // Пошук за кодом — O(1)
```

```
Console.WriteLine("");
string[] lookupCodes = { "IBU400", "PARA500", "LIS10" };
foreach (var code in lookupCodes)
{
    if (catalog.TryGetValue(code, out var med))
        Console.WriteLine($"Знайдено [{code}]: {med}");
    else
        Console.WriteLine($"Не знайдено: [{code}]");
}

// Оновлення назви та додавання нового
catalog["MET850"] = "Метформін 850мг (пролонгований)";
catalog.TryAdd("PARA500", "Парацетамол 500мг");
Console.WriteLine($"\nПісля оновлення: {catalog["MET850"]}");
Console.WriteLine($"Новий препарат: {catalog["PARA500"]}");

// Видалення
bool removed = catalog.Remove("ASP100", out var removedName);
Console.WriteLine($"\nВидалено ASP100 ({removedName}): {removed}");
Console.WriteLine($"Препаратів у каталозі: {catalog.Count}");

// Ключі відділень
Console.WriteLine("\nДоступні коди:");
foreach (var key in catalog.Keys)
    Console.Write($"{key}  ");
Console.WriteLine("");
```

Реєстратура пацієнтів — runnable приклад

Реалістичний приклад з класом: реєстр пацієнтів за ідентифікатором:

```
C#
using System;
using System.Collections.Generic;

// Виконуваний код
var registry = new PatientRegistry();

registry.Register("P001", new Patient("Петренко Іван",  42, "Гіпертонія"));
registry.Register("P002", new Patient("Коваль Марія",   35, "Діабет 2 типу"));
registry.Register("P003", new Patient("Сидоренко Олег", 58, "Аритмія"));
registry.Register("P004", new Patient("Мельник Ганна",  29, "Мігрень"));

Console.WriteLine($"Пацієнтів у реєстрі: {registry.Count}\n");

registry.PrintAll();

Console.WriteLine("");
registry.Lookup("P002");
registry.Lookup("P999");

registry.Discharge("P003");
Console.WriteLine($"\nПісля виписки: {registry.Count} пацієнтів");

// Класи — після виконуваного коду
class Patient
{
    public string Name { get; }
```

```
    public int Age { get; }
    public string Diagnosis { get; }

    public Patient(string name, int age, string diagnosis)
    {
        Name = name;
        Age = age;
        Diagnosis = diagnosis;
    }

    public override string ToString() =>
        $"{Name}, {Age} p. — {Diagnosis}";
}

class PatientRegistry
{
    private Dictionary<string, Patient> _data = new Dictionary<string,
Patient>();

    public int Count => _data.Count;

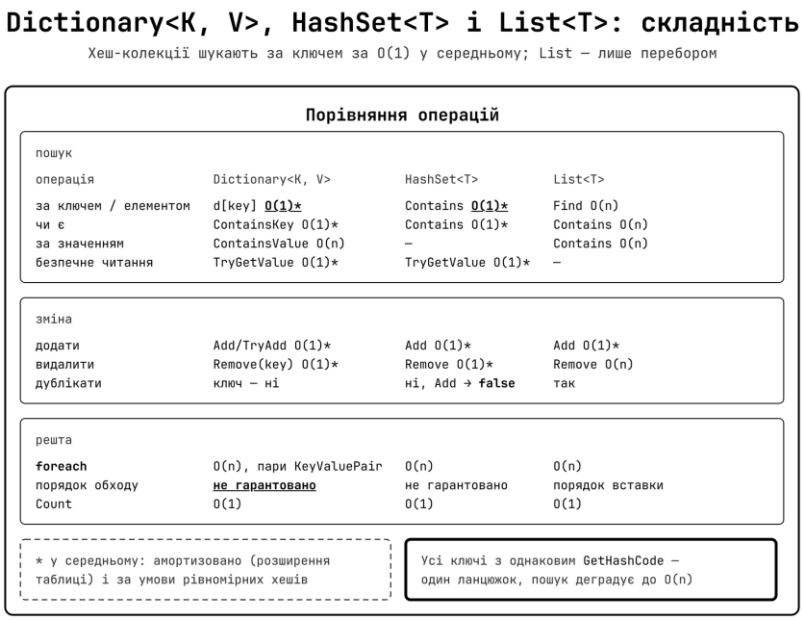
    public void Register(string id, Patient patient)
    {
        if (_data.TryAdd(id, patient))
            Console.WriteLine($"  Зареєстровано [{id}]: {patient.Name}");
        else
            Console.WriteLine($"  Помилка: ID {id} вже існує.");
    }

    public void Lookup(string id)
    {
        if (_data.TryGetValue(id, out var p))
            Console.WriteLine($"Пацієнт [{id}]: {p}");
        else
            Console.WriteLine($"Пацієнт [{id}] не знайдений.");
    }

    public void Discharge(string id)
    {
        if (_data.Remove(id, out var p))
            Console.WriteLine($"Виписано [{id}]: {p.Name}");
        else
            Console.WriteLine($"Пацієнт [{id}] не знайдений.");
    }

    public void PrintAll()
    {
        Console.WriteLine("=== Поточний реєстр ===");
        foreach (var (id, p) in _data)
            Console.WriteLine($"  [{id}]  {p}");
    }
}
```

Складність операцій



Dictionary<K,V> — складність операцій vs List<T>

Коли Dictionary<K, V>?

Dictionary<K, V> — правильний вибір коли:

— Потрібен **швидкий пошук за ключем**: O(1) vs O(n) у List<T>.

— Дані мають природну структуру **ключ → значення**: ID пацієнта → пацієнт, код препарату → препарат.

— Потрібно перевіряти **наявність** елемента за ідентифікатором.

— Потрібен **підрахунок** (ключ → кількість) або **групування** (ключ → список).

List<T> залишається кращим вибором, коли:

— Порядок елементів важливий і ключова операція — перебір.

— Немає унікального ідентифікатора для елементів.

— Колекція мала (< ~20 елементів) — різниця O(1) vs O(n) несуттєва.

HashSet<T> — множина унікальних елементів

HashSet<T> — «словник без значень». Він зберігає **лише унікальні елементи**, дає O(1) у середньому для додавання, видалення і пошуку (так само, як Dictionary, — за умови рівномірних хешів), але на відміну від Dictionary не зберігає пар ключ-значення — тільки самі елементи.

Ключові властивості HashSet<T>:

— **Не допускає дублікатів**: повторне Add(element) ігнорується (повертає false)

— **Немає гарантій порядку**: елементи перебираються у довільному порядку

— **O(1) для Contains**: значно швидший за List<T>.Contains для великих колекцій

— **Операції теорії множин**: UnionWith, IntersectWith, ExceptWith

```C#
using System;
using System.Collections.Generic;

// Реєстр унікальних діагностичних кодів (ICD-10) у відділенні
var diagnosisCodes = new HashSet<string>();

// Додавання — дублікати ігноруються
bool added1 = diagnosisCodes.Add("I10.9");   // Гіпертонія — додано
bool added2 = diagnosisCodes.Add("J45.0");   // Астма — додано
bool added3 = diagnosisCodes.Add("E11.9");   // Діабет — додано
bool added4 = diagnosisCodes.Add("I10.9");   // Гіпертонія — ВЖЕ Є, повертає false

Console.WriteLine($"Унікальних кодів: {diagnosisCodes.Count}"); // 3, не 4
Console.WriteLine($"Додано 4-й (дублікат): {added4}"); // False

// O(1) пошук
bool hasHypertension = diagnosisCodes.Contains("I10.9"); // true
bool hasCancer       = diagnosisCodes.Contains("C34.1"); // false
Console.WriteLine($"Є гіпертонія (I10.9): {hasHypertension}");
Console.WriteLine($"Є рак легень (C34.1): {hasCancer}");

// Видалення
diagnosisCodes.Remove("J45.0");
Console.WriteLine($"Після видалення астми: {diagnosisCodes.Count} кодів");
```

```C#
using System;
using System.Collections.Generic;

// Операції теорії множин — корисні для порівняння груп пацієнтів
var cardioPatients   = new HashSet<string> { "P001", "P002", "P003", "P004" };
var diabeticPatients = new HashSet<string> { "P003", "P004", "P005", "P006" };

// Перетин: пацієнти з обома захворюваннями
var both = new HashSet<string>(cardioPatients);
```

```
both.IntersectWith(diabeticPatients);
Console.WriteLine("Кардіо + Діабет: " + string.Join(", ", both)); // P003, P004

// Об'єднання: всі пацієнти хоча б з одним захворюванням
var either = new HashSet<string>(cardioPatients);
either.UnionWith(diabeticPatients);
Console.WriteLine("Кардіо або Діабет: " + string.Join(", ", either)); // P001-P006

// Різниця: тільки кардіо пацієнти (без діабетиків)
var onlyCardio = new HashSet<string>(cardioPatients);
onlyCardio.ExceptWith(diabeticPatients);
Console.WriteLine("Тільки кардіо: " + string.Join(", ", onlyCardio)); // P001, P002

// IsSubsetOf: чи всі кардіо-пацієнти входять до загальної групи?
bool isSubset = cardioPatients.IsSubsetOf(either);
Console.WriteLine($"Кардіо ⊆ Об'єднання: {isSubset}"); // True

// Видалення дублікатів зі списку
var rawList = new List<string> { "I10.9", "J45.0", "I10.9", "E11.9", "J45.0" };
var uniqueCodes = new HashSet<string>(rawList); // автоматично лише унікальні
Console.WriteLine($"\nRaw: {rawList.Count} кодів → Unique: {uniqueCodes.Count} кодів");
Console.WriteLine(string.Join(", ", uniqueCodes));
```

Операція	**List<T>**	**HashSet<T>**
Add	O(1) (amortized)	O(1) — дублікати ігноруються
Contains	O(n)	**O(1)**
Remove	O(n)	O(1)
Порядок	Зберігається	Не гарантується
Дублікати	Дозволені	Заборонені
Теорія множин	Ні	UnionWith, IntersectWith, ExceptWith

Використовуйте HashSet<T> коли: потрібна колекція без дублікатів, або потрібна швидка перевірка Contains для великих наборів даних, або потрібні операції над множинами. **Використовуйте List<T> коли:** важливий порядок або допустимі дублікати.

10.6. Клас ObservableCollection<T>

List<T>, Queue<T>, Dictionary<K,V> — всі ці колекції ефективно зберігають і організовують дані. Але жодна з них **не повідомляє** зовнішній код про те, що щось змінилось. Якщо хтось додав пацієнта до списку — UI-компонент, що відображає цей список, нічого про це не знатиме і не перемалює себе.

Клас ObservableCollection<T> з простору імен System.Collections.ObjectModel вирішує саме цю задачу. За базовими операціями він схожий на List<T> (індексований доступ, Add, Remove, Insert, а також власний Move), але кожна зміна колекції генерує **подію CollectionChanged**. Ідентичним List<T> він не є: ObservableCollection<T> успадковується від Collection<T> і не має AddRange, Sort, Find, FindAll, RemoveAll, BinarySearch — їх доводиться реалізовувати циклом (кожен виклик тоді дає окрему подію) або через LINQ. Будь-який підписник може відреагувати: перемалювати UI, зберегти лог, синхронізувати стан.

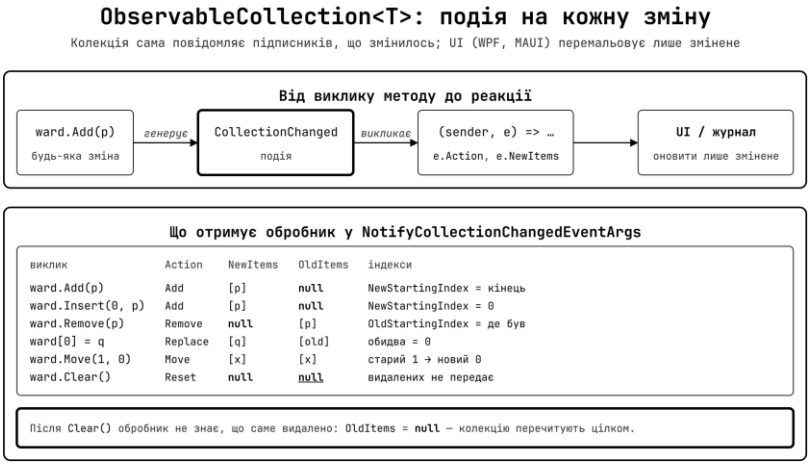
ObservableCollection<T> є базою для прив'язки даних у WPF, MAUI, Xamarin. Якщо ви прив'язуєте колекцію до ListView чи DataGrid — майже завжди потрібна саме ObservableCollection<T>.

Внутрішня структура: подія CollectionChanged

При кожній зміні ObservableCollection<T> генерує подію CollectionChanged і передає підписникам об'єкт NotifyCollectionChangedEventArgs, де зберігається:

- Action — тип зміни: Add, Remove, Replace, Move, Reset.
- NewItems — список доданих або нових елементів (для Add, Replace).
- OldItems — список видалених або замінених елементів (для Remove, Replace).
- NewStartingIndex / OldStartingIndex — позиції, куди додано і звідки видалено (для Move — обидві).

Для Move обидва списки містять той самий переміщений елемент. Для Reset, який генерує Clear(), і NewItems, і OldItems дорівнюють null: обробник не дізнається, що саме видалено, і має перечитати колекцію цілком. Окрім CollectionChanged, колекція реалізує INotifyPropertyChanged і повідомляє про зміну властивостей Count та індексатора ("Item[]") — саме на це реагує прив'язка даних у WPF/MAUI.



ObservableCollection<T> — сповіщення про зміни

Створення та ініціалізація

```C#
using System.Collections.ObjectModel;

// Порожня колекція
ObservableCollection<string> ward = new ObservableCollection<string>();

// З масиву
var fromArray = new ObservableCollection<string>(new[] { "Петренко", "Коваль" });

// Через ініціалізатор
var fromInit = new ObservableCollection<string>
{
    "Петренко І.",
    "Коваль М.",
    "Сидоренко О."
};
```

Доступ до елементів та методи

ObservableCollection<T> надає основні операції List<T> — індексатор, перебір, додавання й видалення:

```C#
var patients = new ObservableCollection<string> { "Петренко І.", "Коваль М." };

// Доступ за індексом
Console.WriteLine($"Перший: {patients[0]}");
```

```
 5
 6  // Зміна елемента за індексом (генерує Replace)
 7  patients[0] = "Петренко Іван Степанович";
 8
 9  // Перебір
10  foreach (var p in patients)
11      Console.WriteLine(p);
```

Повний перелік методів:

Метод	Що робить
Add(item)	Додати в кінець
Insert(index, item)	Вставити за індексом
Remove(item)	Видалити перше входження
RemoveAt(index)	Видалити за індексом
Move(oldIndex, newIndex)	Перемістити елемент на нову позицію
Clear()	Очистити (генерує Reset)
Contains(item)	Перевірити наявність
IndexOf(item)	Індекс першого входження
CopyTo(array, index)	Скопіювати в масив

Метод Move — унікальна особливість ObservableCollection<T>, якої немає в List<T>. Він переміщує елемент і генерує подію Move, що дозволяє UI-компонентам правильно анімувати переупорядкування.

Підписка на CollectionChanged — runnable приклад

Базовий сценарій: відстежуємо всі зміни списку пацієнтів у відділенні:

```
C#
 1  using System;
 2  using System.Collections.ObjectModel;
 3  using System.Collections.Specialized;
 4
 5  var ward = new ObservableCollection<string>
 6  {
 7      "Петренко І.",
 8      "Коваль М."
 9  };
10
11  // Підписка на подію
```

```
ward.CollectionChanged += (sender, e) =>
{
    switch (e.Action)
    {
        case NotifyCollectionChangedAction.Add:
            Console.WriteLine($"  [+] Надійшов: {e.NewItems?[0]}");
            break;
        case NotifyCollectionChangedAction.Remove:
            Console.WriteLine($"  [-] Виписано: {e.OldItems?[0]}");
            break;
        case NotifyCollectionChangedAction.Replace:
            Console.WriteLine($"  [~] Замінено: {e.OldItems?[0]} ->
{e.NewItems?[0]}");
            break;
        case NotifyCollectionChangedAction.Move:
            Console.WriteLine($"  [m] Переміщено: {e.OldItems?[0]}");
            break;
        case NotifyCollectionChangedAction.Reset:
            Console.WriteLine($"  [!] Список очищено");
            break;
    }
};

Console.WriteLine("=== Зміни у відділенні ===");
ward.Add("Сидоренко О.");
ward.Add("Мельник Г.");
ward.Remove("Коваль М.");
ward[0] = "Петренко Іван Степанович";
ward.Move(1, 0);
ward.Clear();

Console.WriteLine($"\nПалата: {ward.Count} пацієнтів");
```

Відстеження з об'єктами Patient — runnable приклад

Розширений приклад з класом: журналюємо кожну зміну складу відділення:

```
C#
using System;
using System.Collections.ObjectModel;
using System.Collections.Specialized;

// Виконуваний код
var patients = new ObservableCollection<Patient>();

patients.CollectionChanged += WardChanged;

patients.Add(new Patient("Петренко Іван",  "кардіологія"));
patients.Add(new Patient("Коваль Марія",   "неврологія"));
patients.Add(new Patient("Сидоренко Олег", "хірургія"));

patients.RemoveAt(1);

patients[0] = new Patient("Петренко Іван Степанович", "кардіологія");

Console.WriteLine("\n=== Поточний склад відділення ===");
```

```
for (int i = 0; i < patients.Count; i++)
    Console.WriteLine($"  {i+1}. {patients[i]}");

void WardChanged(object? sender, NotifyCollectionChangedEventArgs e)
{
    switch (e.Action)
    {
        case NotifyCollectionChangedAction.Add:
        {
            if (e.NewItems?[0] is Patient p)
                Console.WriteLine($"[Прийом]  {p.Name} ({p.Department})");
            break;
        }
        case NotifyCollectionChangedAction.Remove:
        {
            if (e.OldItems?[0] is Patient p)
                Console.WriteLine($"[Виписка] {p.Name}");
            break;
        }
        case NotifyCollectionChangedAction.Replace:
        {
            if (e.OldItems?[0] is Patient old && e.NewItems?[0] is Patient
newPat)
                Console.WriteLine($"[Оновлення] {old.Name} -> {newPat.Name}");
            break;
        }
    }
}

class Patient
{
    public string Name { get; }
    public string Department { get; }
    public Patient(string name, string dept) { Name = name; Department = dept;
}
    public override string ToString() => $"{Name} ({Department})";
}
```

Коли ObservableCollection<T>?

Ситуація	Рекомендація
WPF / MAUI / Xamarin прив'язка до UI	**ObservableCollection** — обов'язково
Потрібно реагувати на зміни ззовні	**ObservableCollection** + CollectionChanged
Звичайне зберігання і перебір даних	**List<T>** — легший overhead
Великі батч-операції без UI	**List<T>** — немає зайвих подій

ObservableCollection<T> генерує подію на **кожну** окрему зміну. Якщо потрібно додати 10 000 елементів — краще створити List<T>, наповнити його, а потім передати в конструктор ObservableCollection<T>(list), щоб уникнути 10 000 подій.

10.7. Інтерфейси IEnumerable<T> та IEnumerator<T>

Щоразу, коли ви пишете foreach (var p in ward), C# виконує певний контракт — звертається до колекції через стандартизовані інтерфейси. Цей контракт описується двома інтерфейсами: IEnumerable<T> і IEnumerator<T>.

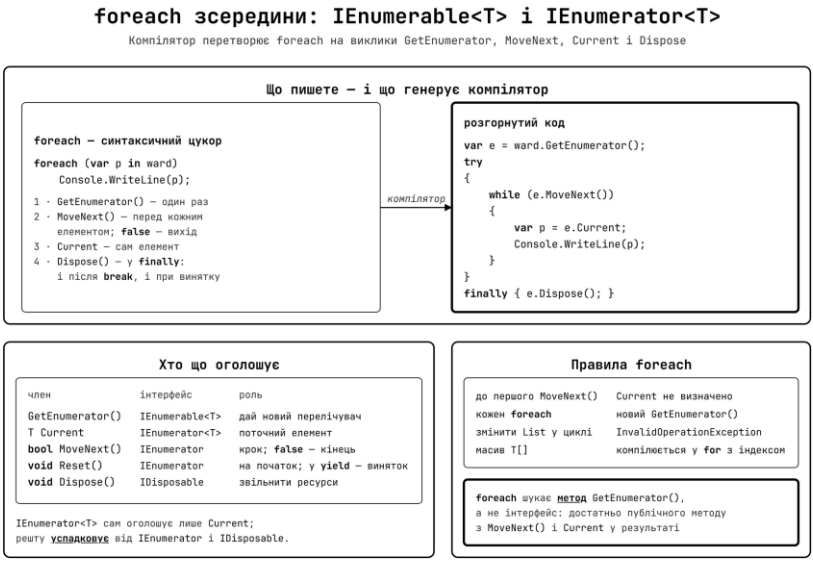
Розуміння цих інтерфейсів пояснює, чому foreach однаково працює з масивом, List<T>, Queue<T>, Dictionary<K,V>, власним класом пацієнтів або лінивим генератором. Усі вони реалізують один і той самий контракт.

Що компілятор робить з foreach

Конструкція foreach — синтаксичний цукор. Компілятор розгортає її у явний виклик методів перелічувача:

```
C#
 1  // Що ви пишете:
 2  foreach (var p in ward)
 3      Console.WriteLine(p);
 4
 5  // Що компілятор генерує:
 6  var e = ward.GetEnumerator();
 7  try
 8  {
 9      while (e.MoveNext())
10      {
11          var p = e.Current;
12          Console.WriteLine(p);
13      }
14  }
15  finally
16  {
17      e.Dispose();
18  }
```

Ці методи GetEnumerator(), MoveNext(), Current, Dispose() — і є контрактом двох інтерфейсів.



IEnumerable<T> та IEnumerator<T> — як працює foreach

Інтерфейс IEnumerable<T>

```
C#
1  public interface IEnumerable<out T> : IEnumerable
2  {
3      IEnumerator<T> GetEnumerator();
4  }
```

IEnumerable<T> відповідає на одне питання: «дай мені перелічувач». Реалізація цього інтерфейсу — стандартний і рекомендований спосіб зробити об'єкт придатним для foreach, а також для LINQ і всіх методів, що приймають IEnumerable<T>.

Строго кажучи, foreach працює **за патерном**, а не за інтерфейсом: компілятор шукає публічний метод GetEnumerator(), результат якого має метод bool MoveNext() і властивість Current. Тому foreach працює і з класом, який нічого не реалізує, але має такий метод (з C# 9 підходить навіть метод-розширення GetEnumerator). Цим користується сам .NET: List<T>.GetEnumerator() повертає структуру List<T>.Enumerator, а не інтерфейс, — і foreach по List<T> не виділяє пам'яті в купі. Для масивів компілятор узагалі генерує звичайний цикл for з індексом.

Реалізують: List<T>, T[], Queue<T>, Stack<T>, Dictionary<K,V>, ObservableCollection<T>, LinkedList<T> та будь-який власний клас, якому ви додасте цей інтерфейс.

Інтерфейс IEnumerator<T>

```
C#
 1  public interface IEnumerator<out T> : IDisposable, IEnumerator
 2  {
 3      T Current { get; }  // повернути поточний елемент (типізовано)
 4  }
 5
 6  // успадковані члени:
 7  public interface IEnumerator
 8  {
 9      bool MoveNext();          // перейти до наступного елемента
10      object Current { get; }   // поточний елемент як object
11      void Reset();             // повернутись на початок (рідко
    використовується)
12  }
13
14  public interface IDisposable
15  {
16      void Dispose();           // звільнити ресурси
17  }
```

Зверніть увагу: сам IEnumerator<T> оголошує **лише** типізовану властивість Current. Методи MoveNext() і Reset() він успадковує від старого негенеричного IEnumerator (часів .NET 1.0), а Dispose() — від IDisposable. Тому клас-перелічувач мусить реалізувати всі чотири члени (плюс негенеричний object Current, як у прикладі RangeEnumerator нижче). Reset() на практиці майже не використовують: перелічувачі, згенеровані з yield return, на виклик Reset() кидають NotSupportedException; для повторного обходу просто беруть новий перелічувач через GetEnumerator().

IEnumerator<T> — сам перелічувач. Він зберігає **поточну позицію** всередині колекції. Кожен виклик MoveNext() зсуває позицію на один крок і повертає true, поки є елементи; коли елементи вичерпані — повертає false і цикл while завершується.

Власний IEnumerable<T> — runnable приклад

Щоб власний клас підтримував foreach, достатньо реалізувати IEnumerable<T>. Нижче — клас Ward (відділення), який перебирає тільки критичних пацієнтів:

```
C#
```

```
using System;
using System.Collections;
using System.Collections.Generic;

var ward = new Ward("Кардіологія");
ward.AddPatient(new Patient("Петренко І.", severity: 1));
ward.AddPatient(new Patient("Коваль М.",  severity: 3));
ward.AddPatient(new Patient("Сидоренко О.", severity: 2));
ward.AddPatient(new Patient("Мельник Г.", severity: 3));

Console.WriteLine("=== Критичні пацієнти відділення ===");
foreach (var p in ward)
    Console.WriteLine($"  {p.Name} (рівень {p.Severity})");

class Patient
{
    public string Name { get; }
    public int Severity { get; }
    public Patient(string name, int severity) { Name = name; Severity =
severity; }
}

class Ward : IEnumerable<Patient>
{
    private readonly List<Patient> _patients = new();
    public string Name { get; }

    public Ward(string name) { Name = name; }

    public void AddPatient(Patient p) => _patients.Add(p);

    // IEnumerable<Patient>: повертаємо тільки критичних (severity >= 2)
    public IEnumerator<Patient> GetEnumerator()
    {
        foreach (var p in _patients)
            if (p.Severity >= 2)
                yield return p;
    }

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

Зверніть увагу: всередині GetEnumerator() використовується yield return (детально — розділ 10.8). Це найпростіший спосіб реалізувати власний перелічувач без написання окремого класу.

Явна реалізація IEnumerator<T> — runnable приклад

Для повного розуміння механізму — клас RangeEnumerator, який перебирає числа від from до to без зберігання їх у пам'яті:

```
C#
using System;
using System.Collections;
using System.Collections.Generic;

```

```
// Перебираємо ліжко-місця відділення: 101..105
var beds = new BedRange(101, 105);
Console.WriteLine("Ліжко-місця кардіологічного відділення:");
foreach (var num in beds)
    Console.WriteLine($"  Палата {num}");

// Клас діапазону
class BedRange : IEnumerable<int>
{
    private readonly int _from;
    private readonly int _to;
    public BedRange(int from, int to) { _from = from; _to = to; }

    public IEnumerator<int> GetEnumerator() => new RangeEnumerator(_from, _to);
    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}

// Перелічувач: стан зберігається у полі _current
class RangeEnumerator : IEnumerator<int>
{
    private readonly int _from;
    private readonly int _to;
    private int _current;

    public RangeEnumerator(int from, int to)
    {
        _from = from;
        _to = to;
        _current = from - 1; // до першого MoveNext()
    }

    public int  Current    => _current;
    object? IEnumerator.Current => _current;

    public bool MoveNext()
    {
        _current++;
        return _current <= _to;
    }

    public void Reset()   => _current = _from - 1;
    public void Dispose() { }
}
```

Цей приклад показує, що IEnumerator<T> — звичайний клас зі станом (_current). MoveNext() зсуває стан; Current повертає поточне значення. foreach просто викликає їх у потрібному порядку.

Коли реалізувати IEnumerable<T>?

Сценарій	Рішення
Власний клас-колекція	Реалізуйте IEnumerable<T>

Сценарій	Рішення
Метод повертає послідовність	Використовуйте yield return (10.8)
Тільки читати чужу колекцію	Оголошуйте параметр як IEnumerable<T>
Потрібні індексація та Count	Використовуйте IList<T> або List<T>

Параметр методу типу IEnumerable<T> замість List<T> — хороша практика: метод прийматиме масив, List<T>, Queue<T> та будь-яку іншу послідовність без змін.

Відкладене виконання (deferred execution)

Одна з найважливіших і найбільш неочікуваних особливостей IEnumerable<T> — **відкладене виконання**. Метод із yield return (або LINQ-запит, що повертає IEnumerable<T>) **не виконується** в момент виклику — він виконується лише тоді, коли результат фактично перебирається (у foreach, при виклику .ToList(), .First() тощо).

```C#
using System;
using System.Collections.Generic;

IEnumerable<string> GetCriticalPatients()
{
    Console.WriteLine("[Generator] Початок обходу пацієнтів...");

    string[] patients = { "Петренко І.", "Коваль М.", "Сидоренко О.", "Мельник Г." };
    bool[]   critical = { false, true, false, true };

    for (int i = 0; i < patients.Length; i++)
    {
        Console.WriteLine($"[Generator] Перевіряю: {patients[i]}");
        if (critical[i])
            yield return patients[i]; // виконання «призупиняється» тут
    }

    Console.WriteLine("[Generator] Кінець обходу");
}

// Виклик методу — код ще НЕ виконується!
Console.WriteLine("Викликаю GetCriticalPatients()...");
IEnumerable<string> sequence = GetCriticalPatients();
Console.WriteLine("GetCriticalPatients() повернув IEnumerable. [Generator] ЩЕ НЕ ЗАПУЩЕНИЙ!");

Console.WriteLine("\nПочинаємо foreach:");
foreach (string patient in sequence) // ось тепер генератор запускається
```

```
28      Console.WriteLine($"  Критичний: {patient}");
```

Генератор запускається тільки при першому MoveNext(). Після кожного yield return виконання **призупиняється** і повернеться до генератора лише при наступному MoveNext().

Багаторазове виконання — «подвійна пастка»

Оскільки кожен foreach викликає GetEnumerator() знову, генератор виконується **з нуля** при кожному переборі:

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4
 5  int callCount = 0;
 6
 7  IEnumerable<int> ExpensiveQuery()
 8  {
 9      callCount++;
10      Console.WriteLine($"[Запит #{callCount}] Відправляю запит до БД...");
11      yield return 101; // Петренко
12      yield return 205; // Коваль
13      Console.WriteLine($"[Запит #{callCount}] Результати отримано");
14  }
15
16  var results = ExpensiveQuery(); // нічого ще не відбулось
17
18  Console.WriteLine("=== Перший прохід (foreach) ===");
19  foreach (var id in results)
20      Console.WriteLine($"  Пацієнт ID: {id}");
21
22  Console.WriteLine("\n=== Другий прохід (foreach знову) ===");
23  foreach (var id in results) // ПОВТОРНИЙ ЗАПИТ ДО БД!
24      Console.WriteLine($"  Пацієнт ID: {id}");
25
26  Console.WriteLine($"\nЗапитів виконано: {callCount} (замість 1!)");
27  Console.WriteLine("\n=== Правильно: матеріалізуємо один раз через ToList()
    ===");
28  callCount = 0;
29  var cached = ExpensiveQuery().ToList(); // один запит, всі дані в пам'яті
30  foreach (var id in cached) Console.WriteLine($"  {id}");
31  foreach (var id in cached) Console.WriteLine($"  {id}"); // перебір List — нові
    запити не йдуть
32  Console.WriteLine($"Запитів: {callCount}"); // 1
```

Правило: якщо IEnumerable<T> потребує дорогої операції (запит до БД, HTTP-запит, читання файлу) — матеріалізуйте результат через **.ToList() або .ToArray()** перед повторним використанням. Якщо послідовність велика і потрібна лише для одного проходу — залиште IEnumerable<T> для економії пам'яті.

10.8. Ітератори та yield return

У розділі 10.7 ми побачили, що для реалізації IEnumerable<T> необхідно або написати окремий клас-перелічувач (IEnumerator<T>), або скористатися yield return у тілі GetEnumerator(). Другий варіант — набагато простіший і значно поширеніший на практиці.

yield return — це ключове слово, яке перетворює звичайний метод на **ітераторний метод**. Компілятор сам генерує клас-перелічувач з усіма необхідними полями та методами. Ви описуєте логіку обходу; C# будує машину стану.

Синтаксис і базовий принцип

Метод вважається ітераторним, якщо:

8. Тип повернення — IEnumerable<T>, IEnumerator<T> або їхні негенерик-варіанти (а для асинхронних ітераторів з C# 8 — IAsyncEnumerable<T> чи IAsyncEnumerator<T> разом з модифікатором async, розділ 17.6).
9. Тіло містить хоча б один yield return або yield break.

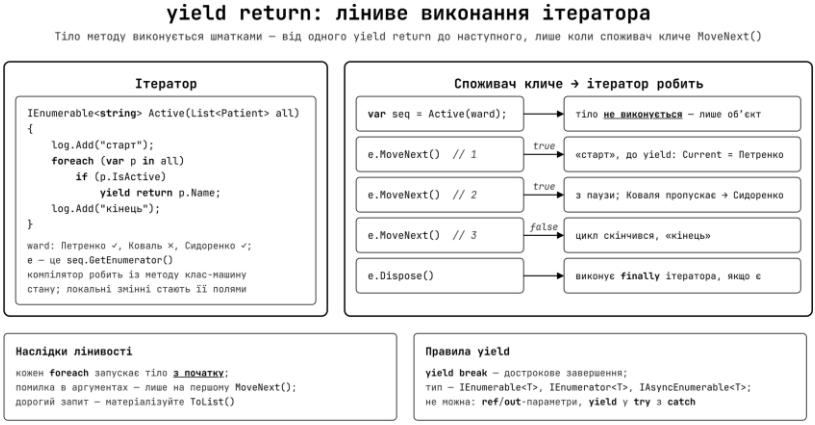
```
C#
1  IEnumerable<int> Sequence()
2  {
3      yield return 1;
4      yield return 2;
5      yield return 3;
6  }
```

Кожен виклик yield return повертає один елемент і **призупиняє виконання**. При наступному виклику MoveNext() виконання відновлюється з тієї ж точки. yield break завершує ітерацію достроково. Якщо ж виконання просто дійшло до кінця тіла методу, ітерація теж завершується (MoveNext() повертає false) — писати yield break наприкінці не потрібно.

На ітераторні методи компілятор накладає кілька обмежень, що випливають з перетворення на машину стану: параметри не можуть бути ref, out чи in; yield return не можна писати в блоці try, який має catch (у try з finally — можна), і в самому catch/finally; у тілі не можна використовувати unsafe-код. Блоки finally в ітераторі виконуються, коли перебір завершився або коли споживач викликав Dispose() перелічувача (наприклад, після break у foreach).

Ліниве виконання

Ключова властивість ітераторних методів: тіло методу **не виконується в момент виклику**. Воно запускається лише тоді, коли foreach або інший споживач запросить перший елемент.



yield return — ліниве виконання ітератора

Схема показує цей процес покроково: виклик методу лише створює об'єкт; перший MoveNext() виконує тіло до першого yield return; кожен наступний — продовжує з місця паузи (і пропускає неактивних пацієнтів), а коли метод доходить до кінця, MoveNext() повертає false.

Із лінивості є неочевидний наслідок: перевірки аргументів усередині ітератора теж відкладаються. Якщо метод на початку робить if (all == null) throw new ArgumentNullException(...), виняток вилетить не в рядку виклику, а лише на першому MoveNext() — можливо, далеко від місця помилки. Тому в бібліотечному коді перевірки виносять у звичайний (не ітераторний) метод-обгортку, який перевіряє аргументи й одразу повертає результат внутрішньої локальної функції-ітератора.

```C#
IEnumerable<string> GetActivePatients(List<Patient> all)
{
    Console.WriteLine("Початок перебору");  // виконається при першому MoveNext()
    foreach (var p in all)
        if (p.IsActive)
            yield return p.Name;            // пауза до наступного MoveNext()
```

```
    Console.WriteLine("Перебір завершено");
}

// Тут метод ще НЕ виконується:
var seq = GetActivePatients(patients);

// Тут починається фактичне виконання:
foreach (var name in seq)
    Console.WriteLine(name);
```

yield return у GetEnumerator — runnable приклад

Найпоширеніший патерн: реалізація IEnumerable<T> без окремого класу-перелічувача. Клас Department фільтрує пацієнтів за статусом прямо в GetEnumerator():

```
C#
using System;
using System.Collections;
using System.Collections.Generic;

var dept = new Department("Хірургія");
dept.Add(new Patient("Петренко І.", active: true));
dept.Add(new Patient("Коваль М.",  active: false));
dept.Add(new Patient("Сидоренко О.", active: true));
dept.Add(new Patient("Мельник Г.", active: false));
dept.Add(new Patient("Гонта В.",   active: true));

Console.WriteLine("=== Активні пацієнти ===");
int count = 0;
foreach (var p in dept)
{
    count++;
    Console.WriteLine($"  {count}. {p.Name}");
}
Console.WriteLine($"Всього активних: {count}");

class Patient
{
    public string Name   { get; }
    public bool   Active { get; }
    public Patient(string name, bool active) { Name = name; Active = active; }
}

class Department : IEnumerable<Patient>
{
    private readonly List<Patient> _all = new();
    public string Name { get; }

    public Department(string name) { Name = name; }

    public void Add(Patient p) => _all.Add(p);

    public IEnumerator<Patient> GetEnumerator()
    {
        foreach (var p in _all)
            if (p.Active)
```

```
                yield return p;
    }

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

yield return як самостійний метод — runnable приклад

yield return може бути у будь-якому методі з типом повернення IEnumerable<T>, не лише в GetEnumerator(). Це дозволяє будувати ланцюжки фільтрів і генераторів:

```
C#
using System;
using System.Collections.Generic;

var patients = new List<Patient>
{
    new Patient("Петренко І.", dept: "Кардіологія", age: 67),
    new Patient("Коваль М.",   dept: "Неврологія",  age: 45),
    new Patient("Сидоренко О.",dept: "Кардіологія", age: 71),
    new Patient("Мельник Г.", dept: "Хірургія",    age: 38),
    new Patient("Гонта В.",   dept: "Кардіологія", age: 58),
};

Console.WriteLine("=== Кардіологія, вік 60+ ===");
foreach (var p in FilterByDeptAndAge(patients, "Кардіологія", 60))
    Console.WriteLine($"  {p.Name}, {p.Age} р.");

Console.WriteLine("\n=== Перші 3 пацієнти (yield break) ===");
foreach (var p in TakeFirst(patients, 3))
    Console.WriteLine($"  {p.Name}");

IEnumerable<Patient> FilterByDeptAndAge(
    IEnumerable<Patient> source, string dept, int minAge)
{
    foreach (var p in source)
        if (p.Department == dept && p.Age >= minAge)
            yield return p;
}

IEnumerable<Patient> TakeFirst(IEnumerable<Patient> source, int count)
{
    int i = 0;
    foreach (var p in source)
    {
        if (i >= count) yield break;
        yield return p;
        i++;
    }
}

class Patient
{
    public string Name       { get; }
    public string Department { get; }
    public int    Age        { get; }
```

```
45      public Patient(string name, string dept, int age)
46      { Name = name; Department = dept; Age = age; }
47  }
```

Що компілятор генерує з yield return

yield return — це синтаксичний цукор. Компілятор перетворює ітераторний метод на клас-машину стану з полями для всіх локальних змінних та покажчиком поточного кроку:

```
1  Ваш код:
2    IEnumerable<T> MyMethod() { yield return a; yield return b; }
3
4  Генерований код (спрощено):
5    class MyMethod_StateMachine : IEnumerator<T>
6    {
7        int _state = 0;
8        T   _current;
9
10       bool MoveNext() {
11           switch (_state) {
12               case 0: _current = a; _state = 1; return true;
13               case 1: _current = b; _state = 2; return true;
14               default: return false;
15           }
16       }
17       T Current => _current;
18   }
```

Саме тому між двома yield return зберігається стан усіх локальних змінних: вони стають полями класу.

yield return vs явний IEnumerator<T>

Критерій	**yield return**	**Явний IEnumerator<T>**
Обсяг коду	Мінімальний	Значний (клас зі станом)
Читабельність	Висока	Середня
Повний контроль	Частковий (Reset() кидає NotSupportedException)	Повний (Reset, Dispose)
Ліниве виконання	Автоматично	Вручну
Коли використовувати	95% випадків	Складні нестандартні сценарії

yield return та IEnumerable<T> — підсумок розділу 10

Розділ 10 охоплює повний ланцюжок від конкретних колекцій до абстракцій:

```
List<T>               — індексована, динамічна
LinkedList<T>         — двозв'язний список, O(1) вставка
Queue<T>              — FIFO-черга
Stack<T>              — LIFO-стек
Dictionary<K,V>       — хеш-таблиця
ObservableCollection  — з подією CollectionChanged
IEnumerable<T>        — контракт «можна перебрати foreach»
yield return          — генерує IEnumerator<T> без зайвого коду
```

Кожен рівень дає додатковий рівень абстракції: foreach не знає, що за нею стоїть — список у пам'яті, черга у БД чи генератор з yield return. Вона просто викликає MoveNext().

Питання для самоконтролю — Розділ 10. Колекції

10.1. Список List<T>

1. Що таке List<T> з точки зору внутрішньої реалізації? Поясніть різницю між Count і Capacity. Що відбувається, коли Count досягає Capacity?
2. Що виведе наступний код? Поясніть складність кожної операції.

```
C#
var list = new List<string> { "А", "Б", "В" };
list.Insert(0, "Початок");
list.Remove("Б");
list.Add("Г");
Console.WriteLine(list.Count);
Console.WriteLine(list[0]);
```

3. Поясніть стратегію **подвоєння ємності** в List<T>. Чому складність Add є амортизованою O(1), а не завжди O(1)?
4. Порівняйте List<T> і T[] (масив) за такими критеріями: зміна розміру, індексований доступ, частота вставки в середину. Коли варто обирати масив?
5. Чим відрізняються методи Remove(item) і RemoveAt(i)? Яка складність кожного і чому?
6. Напишіть код, який з колекції List<Patient> (де Patient має поля Name та IsActive) видаляє всіх неактивних пацієнтів одним викликом методу. Який метод для цього потрібен?

7. Поясніть, що таке Predicate<T> і як він використовується в методах Find, FindAll, Exists, RemoveAll. Наведіть приклад лямбда-вираза для кожного.
8. Якщо відомо наперед, що список міститиме близько 10 000 елементів, як ефективно ініціалізувати List<T>, щоб уникнути зайвих перевиділень пам'яті? Запишіть відповідний код.

10.2. Двозв'язаний список LinkedList<T>

9. Що таке **двозв'язаний список** і як він відрізняється від List<T> на рівні внутрішньої структури? Що зберігає LinkedListNode<T> крім самого значення?
10. Поясніть, чому вставка та видалення у LinkedList<T> на відомій позиції є O(1), тоді як пошук конкретного елемента — O(n).
11. Що виведе наступний код?

```C#
var list = new LinkedList<int>(new[] { 1, 2, 3, 4 });
var node = list.Find(2);
list.AddAfter(node, 99);
list.RemoveFirst();
Console.WriteLine(list.First.Value);
Console.WriteLine(list.Count);
```

12. Поясніть різницю між Remove(T value) і Remove(LinkedListNode<T> node) у LinkedList<T>. Яка складність кожного варіанту і чому?
13. Для якого сценарію LinkedList<T> є кращим вибором ніж List<T>? Наведіть конкретний приклад задачі (не з лекції), де це дає реальну перевагу у складності.
14. Реалізуйте функцію обходу LinkedList<string> у **зворотному порядку** (від Last до First) через цикл while. Яка властивість вузла для цього потрібна?
15. Порівняйте LinkedList<T> і List<T> за витратами пам'яті. Чому LinkedList<T> використовує більше пам'яті на один елемент?
16. Ситуація: вам потрібна черга з пріоритетами, де ургентні пацієнти вставляються на початок, а планові — в кінець. Яку колекцію обрати і чому?

10.3. Черга Queue<T>

17. Що означає принцип **FIFO**? Поясніть його на прикладі черги пацієнтів у реєстратурі. Які два методи є ключовими для Queue<T>?
18. Яка внутрішня структура Queue<T> і чому видалення з початку (Dequeue) є O(1), на відміну від List<T>.RemoveAt(0) — O(n)?
19. Що виведе наступний код? Що станеться при останньому Dequeue?

```
C#
 1  var q = new Queue<string>();
 2  q.Enqueue("Перший");
 3  q.Enqueue("Другий");
 4  q.Enqueue("Третій");
 5  Console.WriteLine(q.Dequeue());
 6  Console.WriteLine(q.Peek());
 7  Console.WriteLine(q.Count);
 8  q.Dequeue();
 9  q.Dequeue();
10  q.Dequeue(); // черга порожня!
```

20. Чим TryDequeue відрізняється від Dequeue? Коли слід використовувати Try-варіант?
21. Поясніть різницю між Peek() і Dequeue(). Для чого корисний Peek() у реальному коді?
22. Напишіть клас AppointmentQueue, який обгортає Queue<Patient> і має методи: Enqueue(Patient), ServeNext() (обслуговує і повертає наступного пацієнта або null, якщо черга порожня), Count.
23. Реалізація черги через List<T> з видаленням через RemoveAt(0) — чому це антипатерн при великій кількості елементів? Оцініть складність для N операцій.
24. У яких ситуаціях Queue<T> є **неправильним** вибором? Наведіть конкретні сценарії, де краще підійдуть Stack<T> або List<T>.

10.4. Stack<T>

25. Що означає принцип **LIFO**? Наведіть два практичних приклади з реального програмування (не з лекції), де LIFO є природньою моделлю.
26. Що виведе наступний код? Поясніть порядок виведення.

```
C#
```

```
1  var stack = new Stack<string>();
2  stack.Push("A");
3  stack.Push("Б");
4  stack.Push("В");
5  foreach (var item in stack)
6      Console.Write(item + " ");
7  Console.WriteLine();
8  Console.WriteLine(stack.Pop());
9  Console.WriteLine(stack.Peek());
```

27. Stack<T> ініціалізується з List<string> { "А", "Б", "В" }. Який елемент стане вершиною і чому? Перевірте свою відповідь логікою.
28. Поясніть внутрішню структуру Stack<T>: чому Push і Pop є O(1), і в якому випадку виникає O(n)?
29. Реалізуйте функцію bool IsBalanced(string expression), яка перевіряє правильність розстановки дужок ()[]{} через Stack<char>.
30. Порівняйте Stack<T> і Queue<T> за такими критеріями: принцип обробки, типовий сценарій, методи додавання/вилучення, поведінка foreach.
31. Чому ToArray() у Stack<T> повертає елементи у порядку **LIFO** (вершина першою)? Як це може ввести в оману програміста?
32. Спроєктуйте клас HistoryNavigator<T>, який через Stack<T> реалізує функціональність "назад/вперед": методи Navigate(page), GoBack(), GoForward(). Які стеки потрібні?

10.5. Словник Dictionary<K, V>

33. Що таке **хеш-таблиця** і як вона забезпечує O(1) пошук за ключем? Поясніть поняття **бакету** і **колізії**.
34. Що виведе наступний код? Знайдіть потенційну проблему.

```
C#
1  var dict = new Dictionary<string, int> { ["A"] = 1, ["B"] = 2 };
2  dict["C"] = 3;
3  dict["A"] = 10;
4  Console.WriteLine(dict["A"]);
5  Console.WriteLine(dict["D"]); // ?
```

35. Поясніть різницю між dict["key"] і dict.TryGetValue("key", out var value). Коли виникне виняток при першому варіанті?
36. Чим Add(key, value) відрізняється від dict[key] = value при вже існуючому ключі? Що відбудеться в кожному випадку?

37. Напишіть метод Dictionary<string, int> CountDiagnoses(List<Patient> patients), який підраховує, скільки разів зустрічається кожен діагноз. Використайте TryGetValue або ContainsKey.
38. Поясніть, чому метод ContainsValue є O(n), тоді як ContainsKey — O(1). Як це пов'язано з внутрішньою структурою словника?
39. Що таке HashSet<T> і чим він відрізняється від Dictionary<K, V>? Наведіть три сценарії, де HashSet<T> є більш підходящим вибором, ніж List<T>.
40. Для правильної роботи Dictionary<K, V> тип ключа K повинен коректно реалізовувати два методи. Які саме і чому? Що відбудеться, якщо власний клас використовується як ключ без перевизначення цих методів?

10.6. Клас ObservableCollection<T>

41. Яку ключову проблему вирішує ObservableCollection<T>, яку не вирішує List<T>? В яких фреймворках і технологіях вона є обов'язковою?
42. Що таке подія CollectionChanged і який об'єкт вона передає підписникам? Перелічіть типи дій (Action), про які сигналізує ця подія.
43. Що виведе наступний код? У якому порядку з'являться повідомлення?

```C#
var col = new ObservableCollection<string>();
col.CollectionChanged += (s, e) =>
    Console.WriteLine($"{e.Action}: {e.NewItems?[0] ?? e.OldItems?[0]}");
col.Add("Петренко");
col.Add("Коваль");
col.Remove("Петренко");
col[0] = "Мельник";
```

44. Чим метод Move(oldIndex, newIndex) відрізняється від RemoveAt + Insert? Чому він є унікальним для ObservableCollection<T> і чому він важливий для UI?
45. Порівняйте ObservableCollection<T> і List<T> за такими критеріями: сповіщення про зміни, методи, застосування, продуктивність при пакетних операціях.

46. Поясніть антипатерн: чому не слід додавати 10 000 елементів до ObservableCollection<T> по одному в циклі? Як правильно вирішити цю задачу?
47. Напишіть клас WardMonitor, який підписується на CollectionChanged і веде рахунок поточної кількості пацієнтів (враховуючи Add, Remove, Reset).
48. В якому просторі імен знаходиться ObservableCollection<T>? Чим він відрізняється від System.Collections.Generic? Що потрібно додати у using?

10.7. Інтерфейси IEnumerable<T> та IEnumerator<T>

49. Що таке IEnumerable<T> і IEnumerator<T>? Яку роль кожен з них грає в роботі foreach?
50. Запишіть, у що компілятор розгортає наступний foreach:

```
C#
1  foreach (var p in ward)
2      Console.WriteLine(p);
```

Перелічіть методи та властивості, що викликаються.

51. Що таке **відкладене виконання** (deferred execution) в контексті IEnumerable<T>? Чому наступний код виводить [Generator] **після** рядка "Отримали IEnumerable"?

```
C#
1  IEnumerable<int> Gen() { Console.WriteLine("[Generator]"); yield return 1; }
2  var seq = Gen();
3  Console.WriteLine("Отримали IEnumerable");
4  foreach (var x in seq) Console.WriteLine(x);
```

52. Поясніть, що таке **«подвійна пастка»** відкладеного виконання. Коли виклик foreach двічі по одному IEnumerable<T> може призвести до подвійного запиту до бази даних?
53. Реалізуйте клас EvenNumbers : IEnumerable<int>, який перебирає парні числа від 0 до заданого max. Використайте yield return в GetEnumerator.
54. Що містить інтерфейс IEnumerator<T>: методи і властивості? Чому foreach загортає виклики в try/finally з Dispose()?
55. Порівняйте оголошення параметра методу як IEnumerable<T> та List<T>. Який варіант є більш гнучким і чому? Коли все ж потрібен конкретний тип?

56. Знайдіть і поясніть помилку у наступному коді:

```
C#
1  IEnumerable<string> GetFromDb() { /* дорогий запит */ yield return "A"; yield
   return "B"; }
2  var results = GetFromDb();
3  int count = results.Count(); // System.Linq
4  foreach (var r in results) Console.WriteLine(r);
```

10.8. Ітератори та yield return

57. Що таке **ітераторний метод** і які умови робить звичайний метод ітераторним? Що компілятор генерує замість методу з yield return?
58. Поясніть різницю між yield return і yield break. Коли потрібен yield break?
59. Що виведе наступний код? Поясніть порядок виведення крок за кроком.

```
C#
 1  IEnumerable<int> Numbers()
 2  {
 3      Console.WriteLine("Старт");
 4      yield return 1;
 5      Console.WriteLine("Після 1");
 6      yield return 2;
 7      Console.WriteLine("Після 2");
 8  }
 9  foreach (var n in Numbers())
10      Console.WriteLine($"Отримано: {n}");
```

60. Поясніть, як компілятор перетворює ітераторний метод на **клас-машину стану**. Де зберігаються локальні змінні між двома викликами MoveNext()?
61. Напишіть ітераторний метод IEnumerable<T> TakeWhile<T>(IEnumerable<T> source, Func<T, bool> predicate), який повертає елементи доки виконується умова, а потім зупиняється через yield break.
62. Порівняйте yield return з явним написанням класу IEnumerator<T> за обсягом коду, читабельністю та можливостями. В яких 5% випадків явний IEnumerator<T> все ж кращий?
63. Спроєктуйте ітераторний метод IEnumerable<Patient> GetCritical(IEnumerable<Patient> all, int minSeverity), який

лінивим чином фільтрує пацієнтів. Чому це краще за FindAll у контексті великих послідовностей?

64. Що відбудеться, якщо викликати foreach **двічі** по одному і тому ж ітераторному методу? Чи виконається тіло методу двічі? Як це пов'язано з GetEnumerator()?

Розділ 11. Робота з рядками

11.1. Рядки та клас String

Значна частина завдань у розробці медичних застосунків пов'язана з обробкою текстових даних: парсинг діагнозів за МКХ-10, перевірка формату номера телефону пацієнта, побудова звітів, пошук у клінічних нотатках. У мові C# рядкові значення представляє тип string, а вся функціональність роботи з ним зосереджена в класі System.String. Власне string — це псевдонім (alias) класу String; обидва записи еквівалентні і компілятор не робить між ними різниці. Прийнято використовувати string (з малої літери) в оголошеннях змінних, а String (з великої) — при зверненні до статичних членів: String.IsNullOrEmpty, String.Format.

Об'єкти String зберігають текст як послідовність символів Unicode у кодуванні UTF-16. Це означає, що кожен символ займає 2 або 4 байти — залежно від того, чи потрапляє він у базову багатомовну площину Unicode (BMP) чи виходить за її межі. Більшість символів кирилиці, латиниці, цифр і знаків пунктуації кодуються двома байтами. Максимальний розмір об'єкта String у пам'яті — близько 2 ГБ або приблизно 1 мільярд символів — цей ліміт рідко досягається на практиці, однак він важливий для розуміння масштабованості при обробці великих текстових файлів.

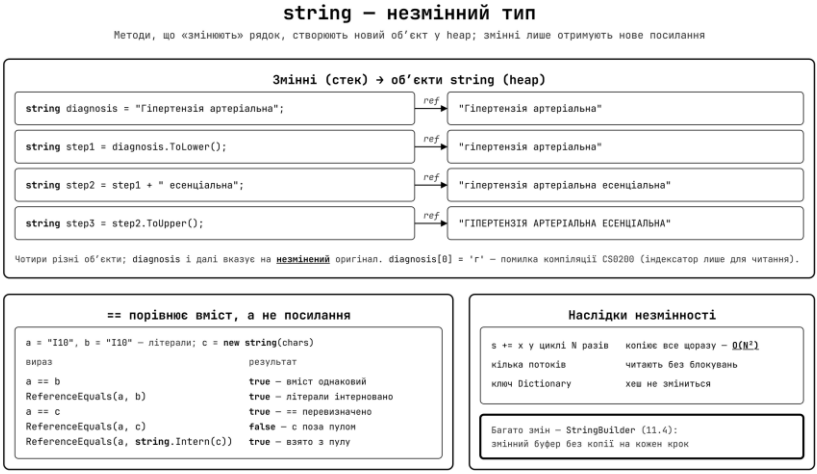
Незмінність рядків (immutability)

Найважливіша властивість string у C# — **незмінність** (immutability). Після того як об'єкт рядка створено і поміщено у heap, його вміст **не може бути змінений жодним засобом**. Це не обмеження платформи — це свідоме архітектурне рішення: незмінність дозволяє безпечно передавати рядки між потоками без синхронізації, ділити один об'єкт між багатьма змінними (і навіть інтернувати однакові літерали в один об'єкт) та використовувати рядки як ключі словників: хеш-код рядка залежить лише від його вмісту, а вміст не може змінитися після того, як ключ потрапив у хеш-таблицю. (Сам хеш-код .NET не кешує в об'єкті рядка — він обчислюється при кожному виклику GetHashCode(), і, як ми бачили в розділі 10.5, ще й рандомізований між запусками програми.)

Наслідок цього рішення: кожен метод класу String, який «змінює» рядок — ToUpper, Replace, Trim, конкатенація через + — насправді **не змінює оригінал**, а **створює новий об'єкт у heap** і повертає посилання на нього. Оригінальна змінна продовжує вказувати на незмінений об'єкт:

```
C#
1  string diagnosis = "Гіпертензія артеріальна";
2  string step1 = diagnosis.ToLower();      // новий об'єкт у heap
3  string step2 = step1 + " есенціальна";  // ще один новий об'єкт
4  string step3 = step2.ToUpper();          // ще один
5
6  Console.WriteLine(diagnosis); // "Гіпертензія артеріальна" — не змінився!
```

На діаграмі нижче видно: змінні на стеку зберігають **посилання** на різні об'єкти в heap. Оригінал залишається незачепленим — і це є фундаментальною особливістю, яку необхідно розуміти перед тим як працювати з будь-якими рядковими операціями.



string — незмінний тип: кожна операція створює новий об'єкт у heap

Практичний наслідок: якщо у циклі виконується багато операцій конкатенації, кожна ітерація виділяє нову пам'ять і копіює весь попередній вміст. При N ітераціях загальна кількість скопійованих символів складає 1+2+3+...+N = N(N+1)/2, тобто складність — O(N²). Для таких сценаріїв існує StringBuilder (розд. 11.4), який працює з мутабельним буфером і не копіює вміст при кожному додаванні.

String interning

Компілятор C# застосовує оптимізацію, яку називають **string interning** (інтернування рядків): однакові рядкові літерали, які зустрічаються в коді, компілюються в **один спільний об'єкт** у спеціальному пулі рядків (string intern pool). Цей пул зберігається у керованій купі (managed heap) і не підлягає збиранню сміттям протягом усього часу роботи програми.

Важливо не плутати інтернування з причиною того, що == «правильно» порівнює рядки. Оператор == для рядків порівнює **значення** (вміст) тому, що він **перевизначений** у класі String, — а зовсім не завдяки інтернуванню. Інтернування лише економить пам'ять і робить однакові літерали одним об'єктом; == дає очікуваний результат і тоді, коли обидва рядки є різними об'єктами:

```
C#
1  string a = "I10";
2  string b = "I10";
3
4  Console.WriteLine(a == b);                // true  — порівняння значень
5  Console.WriteLine(ReferenceEquals(a, b)); // true  — один об'єкт у пулі (literal
   interning)
6
7  string c = new string(new char[] { 'I', '1', '0' });
8  Console.WriteLine(a == c);                // true  — значення однакові
9  Console.WriteLine(ReferenceEquals(a, c)); // false — с створено через new, поза
   пулом
```

Рядки, створені динамічно (через конкатенацію, new string(...), читання з файлу тощо), за замовчуванням **не потрапляють у пул**. Метод string.Intern(s) дозволяє вручну помістити рядок у пул, щоб наступні порівняння через ReferenceEquals давали true. Але на практиці для перевірки рівності рядків **завжди достатньо ==** — він коректно порівнює значення незалежно від того, де зберігається об'єкт.

Важливо також розуміти, що перевантажений оператор == для рядків — це не той самий оператор, що для об'єктів. Для більшості класів == перевіряє посилання; для string він перевизначений і порівнює саме вміст. Тому "hello" == new string(new char[]{'h','e','l','l','o'}) дає true, хоча технічно це два різних об'єкти.

Тип char

Кожен елемент рядка — це значення типу char. Тип char — це **значущий тип** (struct) розміром 2 байти, що представляє один Unicode-символ у кодуванні UTF-16. На відміну від string, char не є посилальним

типом і не розміщується в heap окремим об'єктом — він зберігається безпосередньо у полі або на стеку.

```
C#
1  char letter = 'A';           // одинарні лапки — обов'язково
2  string word  = "Артеріальна"; // подвійні лапки
```

Відмінність між char і string принципова: 'A' — це значення з кодом 65 (його можна неявно перетворити на int), а "A" — це об'єкт у heap з одним символом. Конвертування між ними виконується явно: char c = "A"[0] або string s = c.ToString().

Структура char (System.Char) містить статичні методи для класифікації символів — вони широко використовуються при покрокому аналізі рядків, наприклад, для перевірки форматів ідентифікаторів:

Метод	Що перевіряє
char.IsLetter(c)	Чи є символ літерою (будь-якого алфавіту)
char.IsDigit(c)	Чи є символ десятковою цифрою
char.IsLetterOrDigit(c)	Літера або цифра
char.IsWhiteSpace(c)	Пробіл, табуляція, перенос рядка
char.IsUpper(c)	Верхній регістр
char.IsLower(c)	Нижній регістр
char.IsPunctuation(c)	Знак пунктуації
char.ToUpper(c)	Перетворити на верхній регістр
char.ToLower(c)	Перетворити на нижній регістр

Створення рядків

Рядок можна створити кількома способами. Найчастіше використовується рядковий літерал, проте конструктори класу String відкривають додаткові можливості:

```
C#
 1  // Рядковий літерал — найпоширеніший спосіб
```

```
 2  string diagnosis = "Гіпертензія артеріальна";
 3
 4  // Конструктор: повторити символ N разів — корисно для роздільників
 5  string separator = new string('-', 40);  // "----------------------------------
    ------"
 6
 7  // Конструктор: із масиву символів — корисно при побудові рядка посимвольно
 8  string code = new string(new char[] { 'I', '1', '0' });  // "I10"
 9
10  // Конструктор: частина масиву (startIndex, count) — витягує підпослідовність
11  string sub = new string(new char[] { 'I', '1', '0', '.', '9' }, 0, 3);  //
    "I10"
12
13  // Порожній рядок — два еквівалентні способи
14  string empty1 = "";
15  string empty2 = string.Empty; // той самий об'єкт, що й "" — нових алокацій
    немає в обох випадках
```

Варто окремо зупинитися на string.Empty. Це статичне поле, що завжди посилається на єдиний порожній рядок. Поширена думка, ніби літерал "" щоразу створює новий об'єкт, а string.Empty — ні, хибна: літерал "" інтернується, і в сучасному .NET ReferenceEquals("", string.Empty) дає true — це буквально той самий об'єкт. Тож різниця між ними лише стильова: явне string.Empty підкреслює намір («тут свідомо порожній рядок») і не плутається з пропущеним пробілом " ". Відмінність є лише в контекстах, де потрібна **константа часу компіляції** (значення параметра за замовчуванням, case у switch, атрибут): там допустимий тільки "", бо string.Empty — поле static readonly, а не const.

Рядок як масив символів

Клас String реалізує інтерфейс IEnumerable<char>, тому рядок можна перебирати як послідовність символів у циклі foreach. Крім того, визначено **індексатор** для доступу до окремих символів за позицією:

```
C#
 1  string icd = "I10.9";
 2
 3  char first = icd[0];            // 'I'
 4  int length  = icd.Length;       // 5
 5
 6  // Перебір через for з індексом
 7  for (int i = 0; i < icd.Length; i++)
 8      Console.Write(icd[i] + " "); // I 1 0 . 9
 9
10  // Перебір через foreach — без доступу до індексу
11  foreach (char c in icd)
12      Console.Write(c + " ");      // I 1 0 . 9
```

Ключова відмінність від масиву: індексатор рядка доступний **тільки для читання** — спроба присвоїти icd[0] = 'X' призведе до помилки компіляції. Це ще раз підкреслює незмінність рядка на рівні мови. Якщо потрібно отримати масив символів для редагування — використовується метод ToCharArray(), який повертає **копію** символів у вигляді звичайного масиву char[], після роботи з яким можна знову створити рядок через конструктор new string(chars).

Властивість Length повертає кількість об'єктів char у рядку. Для більшості символів це еквівалентно кількості видимих символів, але для деяких рідкісних Unicode-символів (так звані сурогатні пари), які кодуються двома char, значення Length може бути більшим, ніж кількість гліфів.

Перевірка на порожній рядок

Ситуація, коли рядок не містить корисного значення, зустрічається постійно: незаповнені поля форми, відсутні дані з бази, порожні рядки при парсингу. Для таких перевірок клас String надає два статичні методи:

```
C#
1  string.IsNullOrEmpty(s)      // true якщо s == null або s == ""
2  string.IsNullOrWhiteSpace(s) // true якщо s == null, "" або "   " (тільки
   пробіли)
```

IsNullOrWhiteSpace є **кращим вибором у більшості випадків**: він захищає не лише від null і порожнього рядка, а й від рядків, що складаються виключно з пробільних символів (' ', '\t', '\n'). Такі рядки технічно не є порожніми з точки зору IsNullOrEmpty, але з ділової логіки — це відсутність значення. Наприклад, якщо лікар ввів у поле діагнозу кілька пробілів, це та сама ситуація, що й порожнє поле:

```
C#
1  void PrintDiagnosis(string? text)
2  {
3      if (string.IsNullOrWhiteSpace(text))
4      {
5          Console.WriteLine("Діагноз не вказано");
6          return;
7      }
8      Console.WriteLine($"Діагноз: {text.Trim()}");
9  }
```

Зверніть увагу: параметр позначено як string? (nullable string) — це C# 8+ функціональність, яка на рівні компілятора попереджає, що

значення може бути null. Методи IsNullOrEmpty і IsNullOrWhiteSpace коректно обробляють null без виключень.

Порівняння рядків

На відміну від більшості класів, оператор == для рядків перевизначений і порівнює **значення** (вміст), а не посилання. Це дає природну семантику: два рядки з однаковим текстом вважаються рівними.

Однак оператор == за замовчуванням є **чутливим до регістру**. Для порівняння без урахування регістру використовується метод string.Equals з параметром StringComparison:

```
C#
1  string d1 = "Гіпертензія";
2  string d2 = "гіпертензія";
3
4  Console.WriteLine(d1 == d2);  // false — різний регістр
5  Console.WriteLine(string.Equals(d1, d2, StringComparison.OrdinalIgnoreCase)); //
   true
```

Перерахування StringComparison надає кілька режимів порівняння. Для медичних ідентифікаторів та кодів найбільш підходить OrdinalIgnoreCase — він порівнює символи за їхніми байтовими (Unicode code point) значеннями без урахування регістру і **без прив'язки до локалі** системи. Це важливо: режими CurrentCulture і InvariantCulture можуть давати різні результати залежно від регіональних налаштувань ОС, що небажано для алгоритмів обробки стандартизованих кодів.

Перелік основних методів

Клас String містить десятки методів. Нижче — найбільш уживані у повсякденній розробці:

Метод	Що робить
Contains(s)	Чи містить рядок підрядок
StartsWith(s) / EndsWith(s)	Чи починається / закінчується підрядком
IndexOf(s) / LastIndexOf(s)	Індекс першого / останнього входження
Replace(old, new)	Замінити всі входження

Метод	Що робить
Split(separator)	Розбити на масив підрядків
Substring(start, length)	Витягти підрядок
Trim() / TrimStart() / TrimEnd()	Видалити пробіли
ToUpper() / ToLower()	Зміна регістру
Insert(index, value)	Вставити підрядок
Remove(start, count)	Видалити символи
PadLeft(n) / PadRight(n)	Доповнити пробілами до ширини n
ToCharArray()	Отримати копію символів як char[]
Equals(s, comparison)	Порівняти з урахуванням режиму
Compare(s1, s2)	Лексикографічне порівняння

Кожен з цих методів розглядається детально в розділі 11.2.

Рядки у клінічному контексті — runnable приклад

Базові операції зі рядками на прикладі даних медичної картки:

```
C#
 1  using System;
 2
 3  string rawName   = "  Петренко Іван Степанович  ";
 4  string diagnosis = "Гіпертензія артеріальна";
 5  string icdCode   = "I10";
 6
 7  Console.WriteLine("=== Дані пацієнта ===");
 8  Console.WriteLine($"Ім'я (raw):  '{rawName}'");
 9  Console.WriteLine($"Ім'я (trim): '{rawName.Trim()}'");
10  Console.WriteLine($"Довжина (trim): {rawName.Trim().Length}");
11
12  Console.WriteLine("\n=== Аналіз діагнозу ===");
13  Console.WriteLine($"Містить 'артеріальна':
    {diagnosis.Contains("артеріальна")}");
14  Console.WriteLine($"Код МКХ починається з 'I': {icdCode.StartsWith("I")}");
15  Console.WriteLine($"Позиція 'артеріальна':
    {diagnosis.IndexOf("артеріальна")}");
16
17  Console.WriteLine("\n=== Immutability: кожна операція — новий об'єкт ===");
18  string d1 = diagnosis.ToUpper();
19  string d2 = diagnosis.Replace("артеріальна", "есенціальна");
20  Console.WriteLine($"Original : {diagnosis}");
21  Console.WriteLine($"ToUpper(): {d1}");
```

```
Console.WriteLine($"Replace(): {d2}");
Console.WriteLine($"Original не змінився: {diagnosis == "Гіпертензія артеріальна"}");
```

Символьний аналіз рядка — runnable приклад

Перевірка формату коду МКХ-10 через char-методи та IsNullOrWhiteSpace:

```
C#
using System;

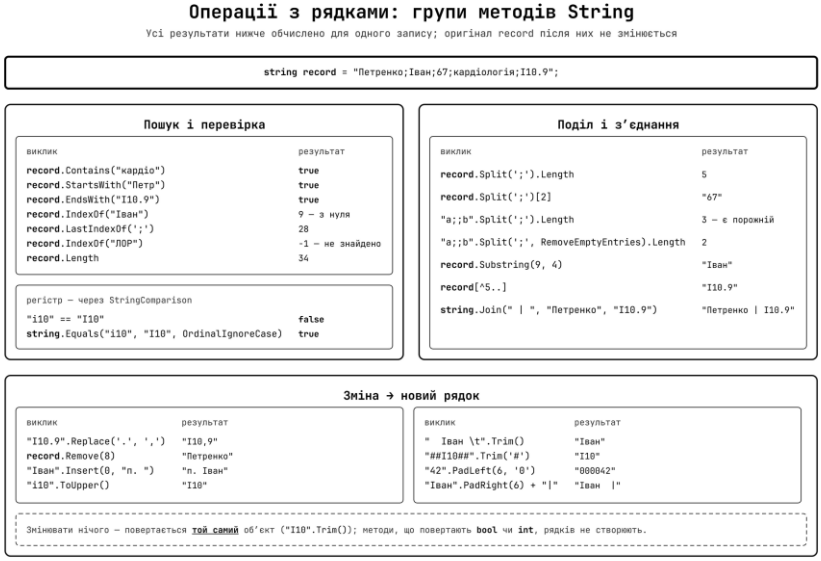
Console.WriteLine("=== Символи коду МКХ-10 ===");
string code = "I10.9";
for (int i = 0; i < code.Length; i++)
{
    char c = code[i];
    Console.WriteLine($"  [{i}] '{c}'  Letter={char.IsLetter(c)} Digit={char.IsDigit(c)}");
}

Console.WriteLine("\n=== IsNullOrWhiteSpace ===");
string[] inputs = { "I10", "", "   ", "J45.0" };
foreach (var s in inputs)
    Console.WriteLine($"  '{s}' -> порожній: {string.IsNullOrWhiteSpace(s)}");

Console.WriteLine("\n=== Конструктори String ===");
string separator = new string('-', 30);
string fromChars = new string(new char[] { 'I', '1', '0' });
Console.WriteLine(separator);
Console.WriteLine($"Із масиву символів: {fromChars}");
Console.WriteLine($"OrdinalIgnoreCase: {string.Equals("i10", "I10", StringComparison.OrdinalIgnoreCase)}");
```

11.2. Операції з рядками

Клас String надає широкий набір методів для пошуку, поділу, об'єднання, зміни та аналізу рядків. Усі вони дотримуються принципу незмінності (immutability) з розділу 11.1: жоден метод не модифікує оригінальний рядок — кожен повертає **новий об'єкт** у heap. Розуміння цього факту критично важливе для оцінки продуктивності коду, особливо у циклах.



Операції з рядками — групи методів класу String

Об'єднання рядків

Конкатенація виконується оператором + або статичним методом Concat:

```
C#
1  string firstName = "Іван";
2  string lastName  = "Петренко";
3
4  string fullName  = lastName + " " + firstName;             // оператор +
5  string fullName2 = string.Concat(lastName, " ", firstName); // метод Concat
```

Оператор + для рядків — це синтаксичний цукор над string.Concat. Компілятор Roslyn оптимізує кількість алокацій: якщо всі операнди є рядковими **літералами**, компілятор об'єднує їх ще на етапі компіляції — у IL потрапляє вже один готовий рядок, без жодного виклику Concat. Але якщо хоч один операнд є змінною або виразом, відбувається виклик string.Concat з відповідною кількістю аргументів.

string.Concat має перевантаження до 4 аргументів, де він може уникнути проміжних алокацій. При більшій кількості частин або у циклі — продуктивнішим є StringBuilder (розд. 11.4).

Для з'єднання **масиву або колекції** рядків через роздільник використовується Join:

```
C#
1  string[] parts = { "Петренко", "Іван", "67", "кардіологія", "I10.9" };
2  string record  = string.Join(";", parts); //
   "Петренко;Іван;67;кардіологія;I10.9"
```

Join є статичним методом і приймає також IEnumerable<T> (де T може бути будь-яким типом — кожен елемент перетворюється через ToString()). Це дуже зручно при формуванні CSV-рядків або протоколів зі списків об'єктів. Join внутрішньо виділяє пам'ять лише один раз (попередньо обчислюючи сумарну довжину), тому є ефективнішим за ручну конкатенацію у циклі.

Пошук у рядку

Contains перевіряє наявність підрядка. За замовчуванням пошук чутливий до регістру; для нечутливого пошуку передається StringComparison:

```
C#
1  string diagnosis = "Гіпертензія артеріальна";
2
3  bool found1 = diagnosis.Contains("артеріальна");
   // true
4  bool found2 = diagnosis.Contains("АРТЕРІАЛЬНА");
   // false
5  bool found3 = diagnosis.Contains("АРТЕРІАЛЬНА",
   StringComparison.OrdinalIgnoreCase); // true
```

Порівняння через StringComparison.OrdinalIgnoreCase є **надійнішим і швидшим** за diagnosis.ToLower().Contains(...), бо не виділяє проміжний рядок у heap. Крім того, ToLower() залежить від поточної культури операційної системи — в деяких локалях перетворення регістру відрізняється від очікуваного для ASCII-символів. OrdinalIgnoreCase порівнює рядки посимвольно за кодами UTF-16, попередньо зводячи обидва символи до одного регістру за інваріантними правилами Unicode, і тому є **незалежним від культури**. Для кирилиці це теж працює: "Гіпертензія артеріальна".Contains("АРТЕРІАЛЬНА", StringComparison.OrdinalIgnoreCase) дає true.

IndexOf та LastIndexOf повертають позицію першого / останнього входження (0-based). Якщо підрядок не знайдено, повертається -1. Перевірка на -1 є стандартним патерном:

```
C#
1  string text = "Діагноз: I10.9. Примітка: I10.9 підтверджено.";
2
3  int idx = text.IndexOf("I10.9");
4  if (idx >= 0)
5      Console.WriteLine($"Знайдено на позиції {idx}");
```

IndexOf також приймає startIndex — початок пошуку, що дозволяє знаходити **наступні** входження у тексті за допомогою циклу:

```
C#
 1  string notes = "Прийом: 09:00. Виписка: 14:00. Повторний прийом: 17:00";
 2
 3  int pos = 0;
 4  while (true)
 5  {
 6      int found = notes.IndexOf("прийом", pos,
    StringComparison.OrdinalIgnoreCase);
 7      if (found < 0) break;
 8      Console.WriteLine($"  знайдено на [{found}]: '{notes.Substring(found,
    6)}'");
 9      pos = found + 1; // пересуваємося на 1 символ вперед від поточного збігу
10  }
```

Метод LastIndexOf виконує пошук з кінця рядка — корисний, наприклад, для знаходження останньої крапки в іменах файлів або останнього роздільника в шляху.

StartsWith та EndsWith перевіряють початок і кінець рядка — зручно для перевірки форматів:

```
C#
1  string icdCode = "I10.9";
2  bool isCardio = icdCode.StartsWith("I");  // true — серцево-судинні хвороби
3  bool hasSpec  = icdCode.EndsWith(".9");   // true — неуточнена форма
```

Обидва методи також приймають StringComparison для нечутливого до регістру пошуку.

Поділ та витягування підрядків

Split розбиває рядок на масив підрядків за роздільником. Це один із найбільш завантажених роботою методів класу String — він має численні перевантаження:

```
C#
 1  string record = "Петренко;Іван;;67;кардіологія";
 2
 3  // 1) Роздільник — один символ
 4  string[] fields1 = record.Split(';');
 5  // ["Петренко", "Іван", "", "67", "кардіологія"] — порожній елемент присутній
```

```
 6
 7  // 2) Видалити порожні елементи
 8  string[] fields2 = record.Split(';', StringSplitOptions.RemoveEmptyEntries);
 9  // ["Петренко", "Іван", "67", "кардіологія"]
10
11  // 3) Обмежити кількість частин
12  string[] parts = record.Split(';', 3);
13  // ["Петренко", "Іван", ";67;кардіологія"] — від 3-ї частини — весь залишок
14
15  // 4) Масив символів-роздільників (ділимо за кількома)
16  string notes = "АТ: 140/90\nЧСС: 78\rSpO2: 97%";
17  string[] lines = notes.Split(new char[] { '\n', '\r' },
    StringSplitOptions.RemoveEmptyEntries);
```

Параметр StringSplitOptions.RemoveEmptyEntries — дуже важлива опція: без неї подвійні роздільники (наприклад, "a;;b") дають порожній рядок у масиві результату. Починаючи з .NET 5, є також StringSplitOptions.TrimEntries — він автоматично прибирає пробіли з кожної отриманої частини. Обидва значення можна комбінувати через побітове АБО: StringSplitOptions.RemoveEmptyEntries | StringSplitOptions.TrimEntries.

Substring витягує підрядок за позицією. Є два перевантаження:

```
C#
1  string icd  = "I10.9 — Гіпертензія артеріальна";
2
3  string code = icd.Substring(0, 5);  // від 0, 5 символів: "I10.9"
4  string desc = icd.Substring(8);     // від 8 до кінця: "Гіпертензія артеріальна"
```

У C# 8+ з'явився синтаксис **Range** через оператор .. та тип Range:

```
C#
1  string code = icd[0..5]; // те саме що Substring(0, 5)
2  string desc = icd[8..];  // те саме що Substring(8)
3  string last = icd[^3..]; // 3 символи з кінця
```

Оператор ^ (hat) позначає індекс відносно кінця рядка: ^1 — останній символ, ^3.. — три останніх. Це суттєво скорочує код при роботі з суфіксами та закінченнями.

Зміна рядка

Replace замінює **всі** входження підрядка або символу на інший. Оскільки метод повертає новий рядок, виклики можна **ланцюгувати** — це зручний патерн для заповнення шаблонів:

```
C#
1  string template = "Пацієнт {NAME} прийнятий {DATE}";
2  string filled   = template
```

```
3      .Replace("{NAME}", "Петренко І.С.")
4      .Replace("{DATE}", "10.06.2026");
5  // "Пацієнт Петренко І.С. прийнятий 10.06.2026"
```

Ланцюгування стає можливим саме завдяки незмінності: кожен Replace повертає новий рядок, на якому можна одразу викликати наступний метод. Кількість ланцюгових викликів обмежена лише читабельністю коду.

Replace також має перевантаження для заміни одного символу: "code".Replace('I', 'J'). Символьна версія є швидшою, ніж рядкова, оскільки порівнює одне значення, а не підрядок.

Insert(index, value) вставляє підрядок на вказану позицію:

```
C#
1  string name    = "Петренко Іван";
2  string updated = name.Insert(9, "Степанович ");
3  // "Петренко Степанович Іван"
```

Важливо: index не може перевищувати Length рядка. При index == Length — вставка відбувається в кінець, що еквівалентно Append.

Remove має два перевантаження:

— Remove(startIndex) — видаляє всі символи від startIndex до кінця рядка.

— Remove(startIndex, count) — видаляє рівно count символів, починаючи з startIndex:

```
C#
1  string record = "ID:00042 Петренко";
2  string name   = record.Remove(0, 9);  // видалити перші 9 символів → "Петренко"
3  string prefix = record.Remove(7);     // залишити перші 7 символів → "ID:0004"
```

Обрізка рядка

Trim() видаляє пробільні символи (пробіл, табуляція, \r, \n) на початку та в кінці рядка. TrimStart() і TrimEnd() обрізають тільки один бік відповідно:

```
C#
1  string raw     = "  Петренко Іван  \t";
2  string cleaned = raw.Trim();      // "Петренко Іван"
3  string left    = raw.TrimStart(); // "Петренко Іван  \t"
4  string right   = raw.TrimEnd();   // "  Петренко Іван"
```

Якщо відомо, що зайві символи можуть бути лише з одного боку (наприклад, читаємо рядок з файлу, де пробіли завжди є в кінці) — TrimEnd() є ефективнішим за Trim(), оскільки аналізує менше символів.

Можна також задати **довільні символи** для обрізання замість пробілу:

```
C#
1  string code  = "###I10.9###";
2  string clean = code.Trim('#'); // "I10.9"
3
4  // Або масив символів:
5  string messy = "--**I10.9**--";
6  string result = messy.Trim('-', '*', ' '); // "I10.9"
```

PadLeft(totalWidth) та **PadRight(totalWidth)** доповнюють рядок пробілами зліва або справа до заданої загальної ширини. Якщо рядок вже довший за totalWidth — він не обрізається, повертається без змін. Другий параметр дозволяє замінити пробіл на довільний символ:

```
C#
1  Console.WriteLine("Петренко".PadRight(20) + "| I10.9");
2  Console.WriteLine("Коваль".PadRight(20)   + "| J45.0");
3  // Петренко            | I10.9
4  // Коваль              | J45.0
5
6  // PadLeft з '0' для ідентифікаторів:
7  Console.WriteLine(42.ToString().PadLeft(6, '0')); // "000042"
```

PadLeft і PadRight є особливо цінними при формуванні текстових звітів з фіксованою шириною колонок, де необхідно вирівнювати дані по лівому або правому краю.

Зміна регістру та порівняння

ToUpper() і ToLower() нормалізують регістр. Але для **порівняння** без урахування регістру краще передавати StringComparison напряму — без проміжної алокації:

```
C#
1  string input = "кардіологія";
2
3  // НЕ оптимально: виділяє новий рядок у heap перед порівнянням
4  bool match1 = input.ToLower() == "кардіологія";
5
6  // Краще: без зайвої алокації, незалежно від культури
7  bool match2 = string.Equals(input, "Кардіологія",
   StringComparison.OrdinalIgnoreCase);
```

Статичний метод string.Compare(s1, s2) повертає від'ємне число якщо s1 < s2, нуль якщо рядки рівні, додатне якщо s1 > s2. Важлива деталь: **без** параметра StringComparison цей метод порівнює рядки **з урахуванням поточної культури** (CurrentCulture) — за правилами алфавіту мови, а не за кодами символів. Посимвольне порівняння кодів UTF-16 (так зване ординальне) виконується лише з StringComparison.Ordinal. Різницю легко побачити на латиниці: за кодами 'B' (66) менше за 'a' (97), тож string.Compare("a", "B", StringComparison.Ordinal) дає додатне число, а культурне порівняння — від'ємне, бо в алфавіті «а» йде перед «В». Для українських прізвищ нижче обидва способи дають однаковий порядок ('І' — U+0406 — і за кодом, і за алфавітом стоїть перед 'П'):

```
C#
1  int cmp1 = string.Compare("Іваненко", "Петренко"); // < 0 — 'І' перед 'П'
2  int cmp2 = string.Compare("Коваль", "Коваль");     // == 0
3  int cmp3 = string.Compare("Петренко", "Іваненко"); // > 0
```

string.Compare також приймає StringComparison для культурно-незалежного порівняння. Метод екземпляра s.CompareTo(other) є аналогом string.Compare(s, other) без можливості вказати StringComparison — тому для точного контролю над режимом порівняння перевага надається статичному Compare.

Парсинг медичної картки — runnable приклад

Розбираємо рядок медичного запису на поля, перевіряємо формат і формуємо листа:

```
C#
 1  using System;
 2
 3  string record = "Петренко;Іван;67;кардіологія;I10.9";
 4
 5  string[] fields  = record.Split(';');
 6  string lastName  = fields[0];
 7  string firstName = fields[1];
 8  int    age       = int.Parse(fields[2]);
 9  string dept      = fields[3];
10  string icd       = fields[4];
11
12  Console.WriteLine("=== Картка пацієнта ===");
13  Console.WriteLine($"ПІБ:        {lastName} {firstName}");
14  Console.WriteLine($"Вік:        {age} р.");
15  Console.WriteLine($"Відділення: {dept}");
16  Console.WriteLine($"МКХ-10:     {icd}");
17
18  bool isCardio = dept.Contains("кардіо", StringComparison.OrdinalIgnoreCase);
```

```
bool isHypert = icd.StartsWith("I10");
Console.WriteLine($"\nКардіологія: {isCardio}");
Console.WriteLine($"Гіпертензія: {isHypert}");

string letter = "Шановний пацієнте {NAME}, ваш код діагнозу: {ICD}."
    .Replace("{NAME}", $"{firstName} {lastName}")
    .Replace("{ICD}",  icd);
Console.WriteLine($"\n{letter}");

string rebuilt = string.Join(" | ", lastName, firstName, icd);
Console.WriteLine($"\nЗапис: {rebuilt}");
```

Форматування звіту відділення — runnable приклад

Вирівнювання колонок через PadRight, аналіз кодів МКХ через StartsWith:

C#

```
using System;

string[] patients = {
    "Петренко Іван;I10.9;кардіологія",
    "Коваль Марія;J45.0;пульмонологія",
    "Сидоренко Олег;K25.3;гастроентерологія",
    "Мельник Ганна;M54.5;неврологія",
};

Console.WriteLine("=== Список пацієнтів ===");
Console.WriteLine("Ім'я".PadRight(22) + "МКХ".PadRight(8) + "Відділення");
Console.WriteLine(new string('-', 55));

foreach (var row in patients)
{
    string[] f   = row.Split(';');
    string name  = f[0];
    string icd   = f[1];
    string dept  = f[2];

    char system = icd[0];
    string flag = system == 'I' ? "[серце]"  :
                  system == 'J' ? "[легені]" :
                  system == 'K' ? "[ШКТ]"   : "[інше]";

    Console.WriteLine($"{name.PadRight(22)}{icd.PadRight(8)}{dept}  {flag}");
}

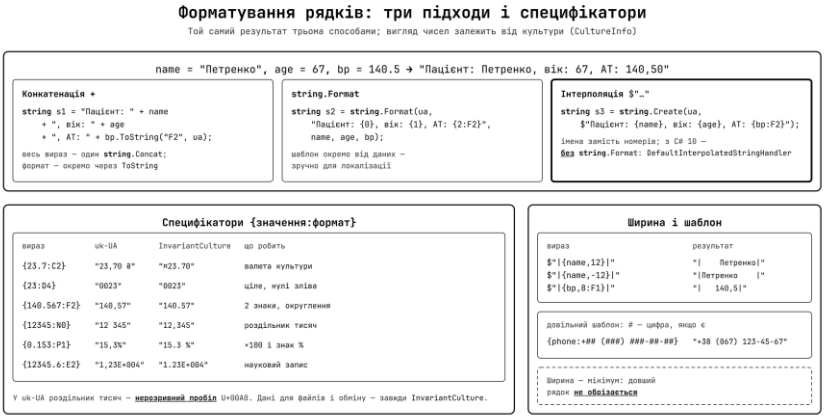
Console.WriteLine(new string('-', 55));

int cardioCount = 0;
foreach (var row in patients)
    if (row.Contains("кардіологія", StringComparison.OrdinalIgnoreCase))
        cardioCount++;

Console.WriteLine($"Кардіологія: {cardioCount} з {patients.Length} пацієнтів");
```

11.3. Форматування та інтерполяція рядків

Виведення числових, датових та складних рядкових значень вимагає контролю над форматом: кількість знаків після коми, вирівнювання колонок у звіті, символ валюти. C# надає три підходи до вставки значень у рядки, кожен зі своїми сценаріями застосування.



Форматування рядків у C# — три підходи та специфікатори

Конкатенація (+)

Найпростіший спосіб — об'єднання через +. Числові та інші типи автоматично перетворюються через ToString():

```
C#
1  string name = "Петренко";
2  int    age  = 67;
3  double bp   = 140.5;
4
5  string s = "Пацієнт: " + name + ", вік: " + age + ", AT: " + bp.ToString("F2") +
   " мм рт.ст.";
```

Недоліки: кожен **окремий** вираз конкатенації створює новий об'єкт-рядок (хоча весь ланцюжок a + b + c + d в одному виразі компілятор перетворює на єдиний виклик string.Concat і одну алокацію результату — проблема виникає при += у циклі); при більш ніж трьох значеннях код стає важкочитаним; числові специфікатори потрібно задавати через ToString() окремо, що розриває читабельний потік рядка.

Конкатенація + доцільна лише для об'єднання двох-трьох частин без специфікаторів.

Важливий нюанс компілятора: якщо **всі** операнди є рядковими літералами ("a" + "b" + "c"), Roslyn розраховує результат ще під час компіляції і поміщає в IL єдиний рядок "abc" — жодної алокації під час виконання. Проте як тільки з'являється змінна або вираз — рядок будується динамічно.

string.Format()

Метод string.Format приймає рядок-шаблон із плейсхолдерами {0}, {1}, ... і набір аргументів:

```
C#
1  string s = string.Format(
2      "Пацієнт: {0}, вік: {1}, АТ: {2:F2} мм рт.ст.",
3      name, age, bp);
```

Переваги string.Format:

— Шаблон відокремлено від даних — зручно для локалізації (зберігати шаблони в ресурсних файлах).

— Один аргумент можна повторити кілька разів: "{0}...{0}".

— Плейсхолдер {index:specifier} поєднує позицію і формат в одному місці.

Метод string.Format (і клас StringBuilder.AppendFormat) також приймає необов'язковий перший параметр типу IFormatProvider — зазвичай CultureInfo. Він визначає культурно-залежне форматування: який символ використовується як десятковий роздільник, розділювач тисяч, символ валюти. За замовчуванням використовується культура поточного потоку (CultureInfo.CurrentCulture):

```
C#
1  using System.Globalization;
2
3  double bp = 140.5;
4  // В українській культурі десятковий роздільник — кома
5  string ua = string.Format(CultureInfo.GetCultureInfo("uk-UA"), "{0:F2}", bp); //
   "140,50"
6  // В американській — крапка
7  string us = string.Format(CultureInfo.InvariantCulture, "{0:F2}", bp);       //
   "140.50"
```

Це критично важливо при записі числових даних у файли або передачі між сервісами: якщо на сервері культура uk-UA, а клієнт

очікує InvariantCulture — парсинг поверне помилку. Для серіалізації даних завжди слід явно передавати CultureInfo.InvariantCulture.

Інтерполяція рядків ($"...")

Інтерполяція — найчитабельніший і найчастіше вживаний підхід. Рядок починається зі знака $; всередині фігурних дужок — будь-який вираз C#:

```
C#
1  string s = $"Пацієнт: {name}, вік: {age}, АТ: {bp:F2} мм рт.ст.";
```

До C# 10 компілятор перетворював $"..." у виклик string.Format() (або string.Concat, якщо специфікаторів немає) — тобто на рівні IL підходи були ідентичні. Переваги $"...": імена змінних замість номерів, компілятор перевіряє типи на етапі компіляції, будь-які вирази прямо у {}:

```
C#
1  int systolic  = 140;
2  int diastolic = 90;
3  Console.WriteLine($"АТ: {systolic}/{diastolic} — {(systolic > 130 ? "підвищений"
   : "норма")}");
4  // АТ: 140/90 — підвищений
```

Починаючи з **C# 10** (.NET 6), компілятор робить інакше: замість виклику string.Format він використовує DefaultInterpolatedStringHandler — структуру, яка дописує частини в позичений буфер (Span<char>), форматує числа без проміжних рядків і без упаковки (boxing) значущих типів, а наприкінці створює лише один результуючий рядок. Тож твердження «$"..." — це те саме, що string.Format» для сучасного .NET уже неточне: інтерполяція ефективніша. Культуру для інтерполяції задають через string.Create(CultureInfo.InvariantCulture, $"...") або FormattableString. Ця оптимізація вмикається автоматично і не потребує змін у коді.

Інтерполяцію можна поєднати з **verbatim-рядком** (@) для тексту з зворотними слешами або переносами рядків без \n:

```
C#
1  // @$ або $@ — обидва порядки допустимі з C# 8
2  string path = $@"C:\Patients\{lastName}\record.txt";
```

Специфікатори форматування

Специфікатори використовуються однаково у string.Format та $"..." через двокрапку після виразу: {value:specifier}.

Числові специфікатори

Специфікатор	Назва	Приклад
C2	Валюта	{23.7:C2} → 23,70 ₴
D4	Ціле (з нулями)	{23:D4} → 0023
F2	Дробове	{140.567:F2} → 140,57
N0	Тисячний роздільник	{12345:N0} → 12 345 (нерозривний пробіл)
P1	Відсоток	{0.153:P1} → 15,3%
E2	Науковий формат	{12345.6:E2} → 1,23E+004

Число після літери специфікатора задає кількість знаків після коми (або мінімальну кількість цифр для D). Приклади в таблиці наведено для культури uk-UA; в InvariantCulture той самий код дасть ¤23.70, 140.57, 12,345, 15.3 %, 1.23E+004 — роздільники й символ валюти визначає культура, а не специфікатор.

Детальніше про кожен:

— **C (Currency)** — додає символ валюти поточної культури. Кількість знаків після коми визначається культурою за замовчуванням (зазвичай 2). Для uk-UA сучасний .NET (дані ICU) дає 23,70 ₴; старий .NET Framework на Windows (дані NLS) міг давати 23,70 грн. — тому в звітах, де важливий точний вигляд, суму краще форматувати як N2 і дописувати назву валюти явно.

— **D (Decimal)** — тільки для цілочисельних типів (int, long). Заповнює нулями зліва до мінімальної кількості цифр. {42:D6} → "000042". Ідеальний для номерів медичних карток і ідентифікаторів пацієнтів.

— **F (Fixed-point)** — дробове число з фіксованою кількістю знаків після коми. Використовується для медичних показників: артеріальний тиск, глюкоза, температура.

— **N (Number)** — додає тисячний роздільник. {12345678:N0} → "12 345 678". Зручний для сум у фінансових звітах лікарні. Зверніть увагу: в uk-UA роздільник тисяч — не звичайний пробіл, а **нерозривний** (U+00A0); порівняння такого результату з рядком, набраним зі звичайними пробілами, дасть false.

— **P (Percent)** — множить значення на 100 і додає знак %. Значення 0.847 дасть "84,7%". Корисний для статистики успішності лікування.

— **E (Exponential)** — науковий запис. Використовується рідко — лише для дуже великих або малих чисел у лабораторних дослідженнях.

Довільний формат через

Символ # дозволяє задати довільний шаблон. Наприклад, телефон пацієнта:

```
C#
long phone = 380671234567;
Console.WriteLine($"{phone:+## (###) ###-##-##}"); // +38 (067) 123-45-67
```

У шаблонах # означає «необов'язкова цифра» (нічого не виводиться якщо цифри немає), а 0 — «обов'язкова цифра» (виводиться 0). Шаблон обробляється справа наліво від десяткової крапки — тому цифри з кінця числа розподіляються по позиціях шаблону.

Вирівнювання

Після виразу і специфікатора можна вказати ширину поля через кому. Від'ємне значення — вирівнювання ліворуч, додатне — праворуч:

```
C#
string name = "Петренко";
double value = 140.5;

Console.WriteLine($"|{"Ім'я",15}|{"AT",8}|");
Console.WriteLine($"|{name,15}|{value,8:F1}|");
Console.WriteLine($"|{name,-15}|{value,-8:F1}|");
```

Вивід:

```
|          Ім'я|      AT|
|       Петренко|   140,5|
```

```
3  |Петренко       |140,5   |
```

Ширина поля — це **мінімальна** ширина: якщо значення довше, воно виводиться повністю і не обрізається ($"{"Петренко",3}" дасть "Петренко"), тож таблиця «поїде». Механізм вирівнювання є надзвичайно зручним для побудови текстових таблиць: кожна колонка має фіксовану ширину, а значення автоматично вирівнюються. Правостороннє вирівнювання ({value,8}) стандартне для чисел; лівостороннє ({name,-20}) — для текстових полів.

Raw string literals (C# 11)

Починаючи з C# 11, рядки можна записувати у потрійних лапках """..."""". Такий рядок може містити лапки та переноси рядків без екранування:

```
C#
1  string template = """
2      Виписка з лікарні
3      Пацієнт: Петренко І.С.
4      Діагноз: "Гіпертензія артеріальна"
5      Рекомендації: прийом "Лізиноприл" 10 мг
6      """;
7  Console.WriteLine(template);
```

Raw string literals особливо зручні для JSON-шаблонів, SQL-запитів або HTML-фрагментів у медичних застосунках. Відступ, що збігається з відступом закриваючих """, автоматично видаляється компілятором — рядок зберігатиме відносні відступи правильно.

Крім того, raw string literals можна поєднувати з $ для інтерполяції: $"""..."""". Якщо всередині потрібна фігурна дужка як літерал, вона записується подвоєно: {{ i }}.

Метод ToString() зі специфікатором

Будь-який числовий тип підтримує ToString(specifier) — ті самі специфікатори, що й у Format:

```
C#
1  double bp     = 140.567;
2  int    patId  = 42;
3  long   phone  = 380671234567;
4
5  Console.WriteLine(bp.ToString("F1"));               // 140,6
6  Console.WriteLine(patId.ToString("D6"));            // 000042
7  Console.WriteLine(phone.ToString("+## (###) ###-##-##")); // +38 (067) 123-45-67
```

ToString(specifier) також приймає другий параметр IFormatProvider: bp.ToString("F2", CultureInfo.InvariantCulture). Це той самий механізм, що й у string.Format — реалізується через інтерфейс IFormattable, який реалізований усіма числовими типами .NET.

Форматований звіт пацієнта — runnable приклад

Демонстрація специфікаторів C/F/N/P у клінічному звіті:

```
C#
using System;

string name    = "Петренко Іван Степанович";
int    age     = 67;
double bp      = 140.5;
double glucose = 6.83;
double cost    = 1250.5;
double recover = 0.847;

Console.WriteLine("=== Медична картка ===");
Console.WriteLine($"Пацієнт: {name}");
Console.WriteLine($"Вік:     {age} р.");
Console.WriteLine($"АТ:      {bp:F1} мм рт.ст.  ({(bp > 130 ? "підвищений" :
"норма")})");
Console.WriteLine($"Глюкоза: {glucose:F2} ммоль/л");
Console.WriteLine($"Вартість лікування: {cost:N2} грн.");
Console.WriteLine($"Ймовірність одужання: {recover:P1}");

Console.WriteLine("\n=== Ідентифікатори ===");
int patId = 42;
Console.WriteLine($"ID пацієнта: {patId:D6}");

long phone = 380671234567;
Console.WriteLine($"Телефон: {phone:+## (###) ###-##-##}");

Console.WriteLine("\n=== string.Format (шаблон) ===");
string report = string.Format(
    "Пацієнт {0}, {1} р. | АТ: {2:F0}/{3:F0} | Глюкоза: {4:F1}",
    name, age, bp, 90.0, glucose);
Console.WriteLine(report);
```

Таблиця відділення з вирівнюванням — runnable приклад

Форматована таблиця через вирівнювання у $"...":

```
C#
using System;

Console.WriteLine($"{"Пацієнт",-24}{"Вік",5}{"АТ",8}{"МКХ",8}");
Console.WriteLine(new string('-', 48));

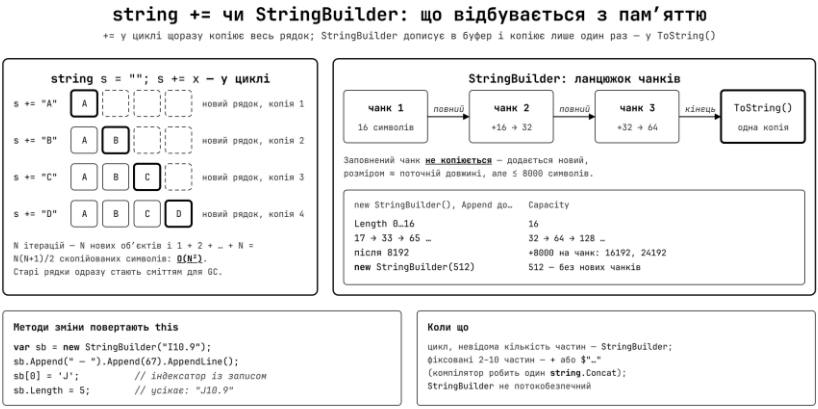
var patients = new[]
{
    ("Петренко Іван",     67, 140.0, "I10.9"),
```

```
 9      ("Коваль Марія",      45, 118.0, "J45.0"),
10      ("Сидоренко Олег",    71, 158.0, "I10.9"),
11      ("Мельник Ганна",     38, 112.0, "M54.5"),
12  };
13
14  foreach (var (name, age, bp, icd) in patients)
15  {
16      string flag = bp > 130 ? "(!)" : "   ";
17      Console.WriteLine($"{name,-24}{age,5}{bp,7:F0} {flag}  {icd}");
18  }
19
20  Console.WriteLine(new string('-', 48));
21
22  double avgBp = (140.0 + 118.0 + 158.0 + 112.0) / 4;
23  Console.WriteLine($"{"Середній АТ:",-24}{avgBp,12:F1} мм рт.ст.");
```

11.4. Клас StringBuilder

З розділу 11.1 відомо: рядок string є незмінним — кожна операція створює новий об'єкт у heap. При одиничних операціях це не проблема. Але якщо потрібно побудувати рядок із 50 фрагментів (наприклад, виписку пацієнта по рядку за рядком), конкатенація через + виконає **50 алокацій**, де кожен новий об'єкт копіює весь попередній вміст. Складність такого підходу — $O(N^2)$.

Клас StringBuilder з простору імен System.Text вирішує цю задачу: він зберігає **мутабельний буфер** символів і додає нові символи на місці, без копіювання попереднього вмісту. Алокація відбувається лише тоді, коли буфер переповнюється, — і тоді до нього додається новий шматок (чанк) пам'яті, приблизно такий самий за розміром, як уже накопичений текст, тож загальна ємність фактично **подвоюється**.



StringBuilder vs string — алокація пам'яті при побудові рядка у циклі

Чому string у циклі — це O(N²)

Розглянемо приклад: потрібно зібрати рядок зі 100 частин. При конкатенації через +:

```
1  Ітерація 1: новий рядок на 1 символ, скопіювати 1 символ
2  Ітерація 2: новий рядок на 2 символи, скопіювати 2 символи (1 + 1)
3  Ітерація 3: новий рядок на 3 символи, скопіювати 3 символи (2 + 1)
4  ...
5  Ітерація N: новий рядок на N символів, скопіювати N символів
```

Загальна кількість скопійованих символів: 1 + 2 + 3 + ... + N = N(N+1)/2 ≈ N²/2.

При N=1000 — близько **500 000 скопійованих символів** і 1000 алокацій нових рядків (кожен наступний довший за попередній, тож сумарно виділяється близько 1 МБ пам'яті під символи, які одразу стають сміттям). Кожен такий об'єкт у кінцевому рахунку стає сміттям для збирача — тиск на GC зростає, і в певний момент він зупиняє виконання для збирання. У медичних застосунках, де будуються тисячі виписок або журналів — це помітна деградація продуктивності.

Алгоритм подвоєння ємності

StringBuilder виділяє початковий буфер на 16 символів (якщо рядок у конструкторі не перевищує 16 символів). Далі важливо знати, як саме влаштовано розширення в сучасному .NET (починаючи з .NET Framework 4.0). StringBuilder — це не один масив, який подвоюють і копіюють, як у List<T>, а **ланцюжок чанків** (шматків пам'яті). Коли

поточний чанк заповнено, виділяється **новий** чанк розміром приблизно з уже накопичений текст (але не більше 8000 символів) і приєднується до ланцюжка; **старий вміст не копіюється взагалі**. Тому Capacity спершу зростає вдвічі — 16 → 32 → 64 → … → 8192, — а далі на 8000 за кожен новий чанк: 16 192, 24 192, … (обмеження в 8000 символів тримає кожен чанк поза купою великих об'єктів, LOH).

Математично: дописування кожного символу — O(1), нових чанків при N символах — близько $\log_2(N)$ (плюс по одному на кожні 8000 символів після межі), а символи копіюються лише **один раз** — у ToString(), коли всі чанки збираються в підсумковий рядок. Тобто O(N) замість $O(N^2)$ — лінійна складність.

```
C#
1  using System.Text;
2
3  var sb = new StringBuilder("Виписка:");
4  Console.WriteLine($"Length={sb.Length}, Capacity={sb.Capacity}"); // Length=8,
   Capacity=16
5
6  sb.Append(" Петренко");
7  Console.WriteLine($"Length={sb.Length}, Capacity={sb.Capacity}"); // Length=17,
   Capacity=32  <-- подвоєно
```

Якщо кінцевий розмір рядка відомий наперед, ємність можна задати явно в конструкторі — тоді реалокацій буде нуль:

```
C#
1  var sb = new StringBuilder(capacity: 512); // одразу виділяємо 512 символів
```

Властивість Capacity завжди показує поточний розмір виділеного буфера, а Length — кількість реально записаних символів. Різниця між ними — незайнята ємність, у яку можна дописувати без реалокації. Властивість MaxCapacity повертає максимально допустимий розмір буфера — за замовчуванням int.MaxValue (≈ 2,1 млрд символів); фактично ж підсумковий рядок обмежений максимальною довжиною string (≈ 1 млрд символів), тож на практиці ліміт недосяжний. Задати менший ліміт можна конструктором new StringBuilder(capacity, maxCapacity) — тоді спроба його перевищити кине ArgumentOutOfRangeException.

Важлива деталь: StringBuilder не є потокобезпечним

На відміну від string (незмінний — тому безпечний між потоками), StringBuilder **не є потокобезпечним**. Одночасний запис у один StringBuilder з кількох потоків призведе до непередбачуваних

результатів. Для багатопотокових сценаріїв кожен потік повинен мати **власний** екземпляр StringBuilder, або доступ до спільного слід синхронізувати через lock.

У типових навчальних та прикладних сценаріях StringBuilder використовується в рамках одного потоку — тому потокобезпека зазвичай не є проблемою.

Індексатор StringBuilder

На відміну від string, де індексатор доступний лише для **читання**, StringBuilder надає індексатор з можливістю **читання і запису**:

```
C#
1  var sb = new StringBuilder("I10.9");
2  char c = sb[0];     // 'I' — читання
3  sb[0] = 'J';        // заміна першого символу — запис!
4  Console.WriteLine(sb.ToString()); // "J10.9"
```

Це ще одне підтвердження мутабельності StringBuilder на противагу незмінному string. Крім того, властивості Length можна **присвоїти значення** — це усікає буфер до вказаної довжини:

```
C#
1  var sb = new StringBuilder("I10.9 — Гіпертензія");
2  sb.Length = 5; // Усікаємо до "I10.9"
3  Console.WriteLine(sb.ToString()); // "I10.9"
```

Створення StringBuilder

```
C#
 1  using System.Text;
 2
 3  // Порожній буфер (Capacity=16)
 4  var sb1 = new StringBuilder();
 5
 6  // З початковим рядком
 7  var sb2 = new StringBuilder("Виписка пацієнта:");
 8
 9  // З заданою ємністю (уникаємо реалокацій)
10  var sb3 = new StringBuilder(256);
11
12  // З початковим рядком і ємністю
13  var sb4 = new StringBuilder("Виписка пацієнта:", 256);
```

Основні методи

Метод	Що робить
Append(value)	Додати значення в кінець
AppendLine(value)	Додати значення і \n
AppendLine()	Додати порожній рядок
AppendFormat(format, args)	Додати форматований рядок
Insert(index, value)	Вставити за індексом
Remove(start, count)	Видалити символи
Replace(old, new)	Замінити всі входження
Clear()	Очистити буфер (зберігає Capacity)
ToString()	Отримати рядок із буфера

Метод Append(value) є перевантаженим для всіх примітивних типів: Append(bool), Append(int), Append(double), Append(char) — це означає, що значення конвертуються у символи без виклику ToString() і без проміжної алокації рядка. Це ще один аргумент на користь StringBuilder при інтенсивній роботі з числами.

Fluent API — method chaining

Методи, що змінюють вміст (Append, AppendLine, AppendFormat, Insert, Remove, Replace, Clear), повертають **this** — посилання на сам об'єкт (а ToString(), звісно, повертає рядок). Це дозволяє ланцюгувати виклики (fluent API):

```
C#
1  string result = new StringBuilder()
2      .Append("Пацієнт: ")
3      .AppendLine("Петренко І.С.")
4      .Append("Діагноз: ")
5      .AppendLine("I10.9 — Гіпертензія")
6      .Append("Відділення: кардіологія")
7      .ToString();
8
9  Console.WriteLine(result);
```

Fluent API є зручним для побудови структурованих рядків у декларативному стилі — особливо коли потрібно зібрати заголовок,

тіло і підпис у єдиному виразі. Зворотна сторона: дуже довгі ланцюги стають складними для покрокового налагодження у дебаґері.

AppendLine та AppendFormat

AppendLine — найзручніший метод для побудови багаторядкових текстів:

```
C#
1  var sb = new StringBuilder();
2  sb.AppendLine("Виписка з лікарні");
3  sb.AppendLine("==================");
4  sb.AppendLine("Пацієнт: Петренко Іван Степанович");
5  sb.AppendLine("Дата: 10.06.2026");
```

AppendLine() без аргументів додає лише символ нового рядка (\r\n на Windows або \n на Unix) — це зручно для розділення секцій без додавання тексту.

AppendFormat — форматоване додавання з підтримкою специфікаторів (розд. 11.3):

```
C#
1  sb.AppendFormat("АТ: {0:F0}/{1:F0} мм рт.ст.\n", 140.5, 90.0);
2  sb.AppendFormat("Вартість: {0:C2}\n", 1250.50);
```

AppendFormat використовує ті самі специфікатори, що й string.Format. Комбінація AppendLine і AppendFormat покриває переважну більшість завдань із побудовою текстових звітів.

Clear() для повторного використання

Clear() обнуляє Length до нуля, але **зберігає виділений буфер**. Це дозволяє повторно використовувати StringBuilder без повторного виділення пам'яті:

```
C#
1  var sb = new StringBuilder(256);
2
3  for (int i = 1; i <= 3; i++)
4  {
5      sb.Clear();                          // скидаємо вміст, буфер лишається
6      sb.AppendLine($"Звіт #{i}");
7      sb.AppendLine($"Пацієнт: Петренко");
8      Console.WriteLine(sb.ToString());
9  }
```

Це типовий патерн у серверному коді: StringBuilder з достатньою ємністю створюється один раз і очищується між ітераціями циклу. Це ефективніше за new StringBuilder() щоразу — розподіл пам'яті для великого буфера відбувається тільки одного разу.

Коли string, а коли StringBuilder?

Ситуація	Рекомендація
До ~10 операцій конкатенації	string — компілятор може оптимізувати
Цикл з невідомою кількістю ітерацій	StringBuilder — O(N) vs O(N²)
Побудова багаторядкового звіту	StringBuilder.AppendLine
Шаблонне форматування	string.Format або $"..."
Пошук (IndexOf, Contains)	string — у StringBuilder їх немає

StringBuilder не має методів IndexOf, Contains, StartsWith. Якщо потрібен пошук під час побудови — зберігайте проміжні string, або завершіть побудову через ToString() і шукайте вже у звичайному рядку.

Побудова виписки — runnable приклад

Демонстрація Capacity, AppendLine, AppendFormat, Clear:

```
C#
 1  using System;
 2  using System.Text;
 3
 4  var sb = new StringBuilder(128);
 5  Console.WriteLine($"Початок: Length={sb.Length}, Capacity={sb.Capacity}");
 6
 7  sb.AppendLine("================================");
 8  sb.AppendLine("     ВИПИСКА З ЛІКАРНІ");
 9  sb.AppendLine("================================");
10  sb.AppendFormat("Пацієнт:    {0}\n", "Петренко Іван Степанович");
11  sb.AppendFormat("Вік:        {0} р.\n", 67);
12  sb.AppendFormat("Відділення: {0}\n", "кардіологія");
13  sb.AppendFormat("МКХ-10:     {0}\n", "I10.9");
14  sb.AppendLine("--------------------------------");
15  sb.AppendFormat("АТ при вступі:   {0:F0}/{1:F0} мм рт.ст.\n", 158.0, 100.0);
16  sb.AppendFormat("АТ при виписці:  {0:F0}/{1:F0} мм рт.ст.\n", 130.0, 85.0);
17  sb.AppendLine("--------------------------------");
18  sb.AppendFormat("Вартість:   {0:N2} грн.\n", 4350.75);
19  sb.AppendLine("================================");
```

```
20
21  Console.WriteLine($"Після заповнення: Length={sb.Length},
    Capacity={sb.Capacity}");
22  Console.WriteLine("");
23  Console.WriteLine(sb.ToString());
```

Fluent chaining та повторне використання — runnable приклад

Ланцюг викликів та Clear() між ітераціями:

```
C#
 1  using System;
 2  using System.Text;
 3
 4  string[] names = { "Петренко Іван", "Коваль Марія", "Сидоренко Олег" };
 5  string[] icds  = { "I10.9", "J45.0", "K25.3" };
 6
 7  var sb = new StringBuilder(256);
 8
 9  for (int i = 0; i < names.Length; i++)
10  {
11      sb.Clear();
12
13      string card = sb
14          .Append($"[{i+1}] ")
15          .Append(names[i].PadRight(20))
16          .Append(" | ")
17          .Append(icds[i])
18          .ToString();
19
20      Console.WriteLine(card);
21  }
22
23  Console.WriteLine("");
24  Console.WriteLine($"Capacity буфера: {sb.Capacity} (без реалокацій після
    Clear)");
25
26  // Replace у StringBuilder
27  sb.Clear();
28  sb.Append("Шановний {NAME}, ваш прийом {DATE} підтверджено.");
29  sb.Replace("{NAME}", "Петренко І.С.");
30  sb.Replace("{DATE}", "12.06.2026 о 10:00");
31  Console.WriteLine(sb.ToString());
```

Вимірювання різниці — Stopwatch

Теоретичне співвідношення $O(N^2)$ проти $O(N)$ добре підтверджується на практиці. Нижче — вимірювання часу побудови рядка з 5 000 фрагментів двома способами: конкатенацією через + і через StringBuilder:

```
C#
 1  using System;
 2  using System.Diagnostics;
```

```
using System.Text;

const int N = 5_000;
const string fragment = "Пацієнт:Петренко|";

// --- Метод 1: конкатенація через += (O(N²) алокацій) ---
var sw1 = Stopwatch.StartNew();
string result1 = "";
for (int i = 0; i < N; i++)
    result1 += fragment;
sw1.Stop();

// --- Метод 2: StringBuilder (O(N) — амортизована O(1) вставка) ---
var sw2 = Stopwatch.StartNew();
var sb = new StringBuilder(N * fragment.Length); // виділяємо одразу
for (int i = 0; i < N; i++)
    sb.Append(fragment);
string result2 = sb.ToString();
sw2.Stop();

Console.WriteLine($"N = {N} фрагментів по {fragment.Length} символів");
Console.WriteLine($"string +=      : {sw1.ElapsedMilliseconds,6} мс  (довжина {result1.Length})");
Console.WriteLine($"StringBuilder  : {sw2.ElapsedMilliseconds,6} мс  (довжина {result2.Length})");
Console.WriteLine($"Прискорення    : ~{(double)sw1.ElapsedMilliseconds / Math.Max(1, sw2.ElapsedMilliseconds):F0}×");
```

На типовій машині при N=5 000 конкатенація через += займає сотні мілісекунд, тоді як StringBuilder — менше 1 мс. Різниця стає ще помітнішою при N=50 000 і вище, оскільки складність конкатенації зростає квадратично — кожна нова ітерація копіює весь попередній вміст рядка.

Важливо розуміти **виняток**: компілятор C# оптимізує конкатенацію через + **поза циклом**, коли всі операнди відомі на етапі компіляції або кількість операндів мала і фіксована. Вираз a + b + c + d компілятор перетворює на один виклик string.Concat(a, b, c, d) — без проміжних алокацій. Ця оптимізація не поширюється на цикли і на += у загальному випадку:

```
C#
using System;

// Компілятор сам застосує string.Concat — StringBuilder тут зайвий і надлишковий
string name    = "Петренко Іван";
string code    = "I10.9";
string section = "кардіологія";

// Один виклик Concat, нуль зайвих алокацій — компілятор вирішив це за нас
string card = name + " | " + code + " | " + section;
Console.WriteLine(card);

```

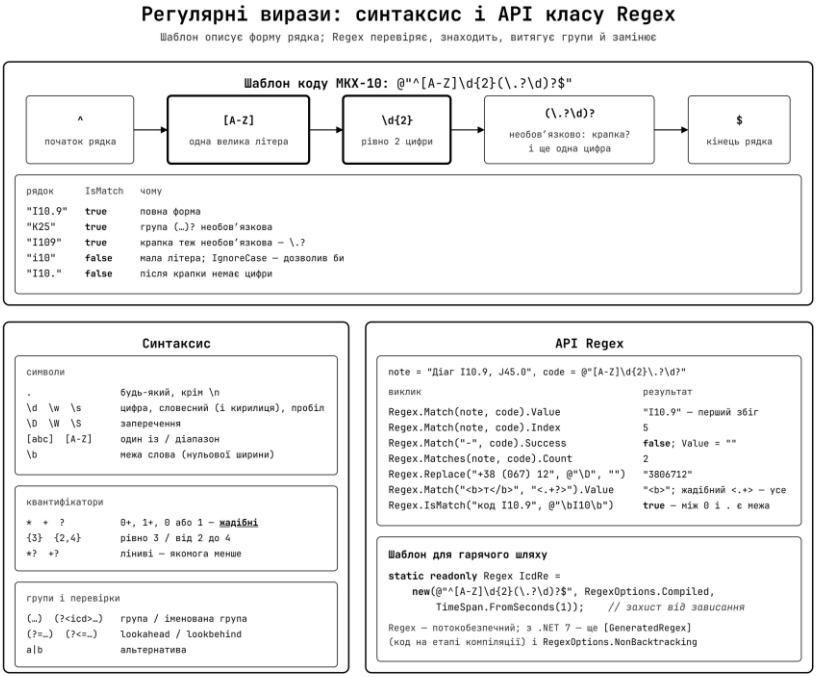
```
12  // Інтерполяція компілюється у DefaultInterpolatedStringHandler (net6+) — теж
    ефективно
13  string card2 = $"{name} | {code} | {section}";
14  Console.WriteLine(card2);
```

Практичне правило: якщо рядок будується у **циклі** або з **невідомої наперед кількості частин** — StringBuilder. Якщо кількість операндів фіксована і мала (умовно до 10) — string + або $"..." є повністю прийнятним і читабельнішим варіантом. Не треба замінювати кожен + на StringBuilder заради «продуктивності» — це надмірна оптимізація, яка ускладнює код без реальної вигоди.

11.5. Регулярні вирази

Методи Contains, IndexOf, Split — ефективні для пошуку фіксованих підрядків. Але коли потрібно знайти рядок за **шаблоном** — наприклад, будь-який код МКХ-10 формату [Літера][2 цифри][.цифра]?, номер телефону або email — ці методи вимагають складного ручного коду. Для таких задач існують **регулярні вирази** (regular expressions, regex).

Основна функціональність у .NET зосереджена у просторі імен System.Text.RegularExpressions. Центральний клас — Regex.



Регулярні вирази у C# — синтаксис та API класу Regex

Як працює двигун регулярних виразів

За замовчуванням .NET використовує **двигун із поверненнями** (backtracking), який у літературі традиційно називають NFA-двигуном (Non-deterministic Finite Automaton — недетермінований скінченний автомат). Принцип роботи:

1. Двигун починає порівнювати шаблон з вхідним рядком з поточної позиції.
2. При збігу він просувається далі; при невдачі повертається назад (backtracking) і пробує альтернативні шляхи.
3. Якщо всі альтернативи вичерпано без збігу — позиція зсувається на 1 символ і процес повторюється.

Повернення (backtracking) — ключова властивість NFA: двигун може «відкатуватися» до попередньої точки вибору і спробувати інший шлях. Саме це дозволяє описувати складні шаблони з альтернативами | і квантифікаторами. Але воно ж може стати пасткою: погано написаний

шаблон з багатьма вкладеними квантифікаторами може генерувати **експоненційну кількість спроб** — явище, відоме як «катастрофічне повернення» (catastrophic backtracking). Для продуктивного коду шаблони мають бути специфічними і уникати надмірного перекриття між альтернативами. Класичний приклад — (a+)+$ на рядку з сорока літер a і знаком ! у кінці: двигун перебирає всі способи розбити літери між вкладеними +, і перевірка триває практично вічно. .NET дає два засоби захисту. Перший — **тайм-аут**: конструктор new Regex(pattern, options, TimeSpan.FromSeconds(1)) (і статичні методи з параметром matchTimeout) кидає RegexMatchTimeoutException, якщо пошук триває довше. Другий — з .NET 7 параметр RegexOptions.NonBacktracking: інший двигун, що гарантує лінійний час від довжини вхідного рядка, але не підтримує зворотні посилання й lookaround-перевірки. Для шаблонів, які застосовуються до даних від користувача, хоча б один із цих засобів обов'язковий.

Жадібні та ліниві квантифікатори

За замовчуванням квантифікатори (*, +, ?, {n,m}) є **жадібними** (greedy): вони намагаються захопити якомога більше символів, а потім повертаються назад, поступаючись символами лише якщо це необхідно для продовження збігу:

```
C#
1  // Жадібний: захоплює все між першим < і останнім >
2  Match m1 = Regex.Match("<b>текст</b>", "<.+>");
3  Console.WriteLine(m1.Value); // "<b>текст</b>" — увесь рядок
4
5  // Лінивий (lazy): захоплює мінімально можливу частину
6  Match m2 = Regex.Match("<b>текст</b>", "<.+?>");
7  Console.WriteLine(m2.Value); // "<b>" — тільки перший тег
```

Ледачий (lazy) квантифікатор записується як *?, +?, ??, {n,m}?. Він означає «якомога менше, але стільки, скільки треба для збігу». Для HTML і XML-подібного тексту ліниві квантифікатори зазвичай дають правильний результат, тоді як жадібні захоплюють занадто багато.

У медичних застосунках цей нюанс важливий при парсингу XML-виписок або HL7 FHIR-повідомлень, де між тегами можуть бути вкладені структури.

Синтаксис регулярних виразів

Регулярний вираз — це рядок-шаблон, де більшість символів означають себе, а спеціальні символи мають особливе значення:

Елемент	Значення		
^ / $	Початок / кінець рядка		
.	Будь-який один символ (крім \n)		
*	Попередній елемент 0 або більше разів		
+	Попередній елемент 1 або більше разів		
?	Попередній елемент 0 або 1 раз		
{n}	Рівно n разів		
{n,m}	Від n до m разів		
\d / \D	Цифра (будь-яка Unicode-цифра, не лише 0–9) / не цифра		
\w / \W	Словесний символ (літера, цифра,) / решта		
\s / \S	Пробільний / не пробільний символ		
\b	Межа слова (word boundary)		
[abc]	Один із символів a, b або c		
[A-Z]	Символ у діапазоні		
(...)	Група збігу		

Елемент	Значення		
(?<name>...)	**Іменована** група		
(?=...)	Позитивний lookahead		
(?!...)	Негативний lookahead		
(?<=...)	Позитивний lookbehind		
`\	`	Альтернатива: `abc\	def`
\.	Екранування: літеральна крапка		

У C# шаблони зазвичай записують як **verbatim-рядки** @"...", щоб уникнути подвійного екранування зворотних слешів: @"\d+" замість "\\d+".

Межа слова (\b)

Метасимвол \b позначає **межу слова** — позицію між символом \w і символом \W (або початком/кінцем рядка). Він дозволяє шукати слово як ціле, а не підрядок:

```
C#
 1  // Без \b — знаходить "I10" у будь-якому місці
 2  var re1 = new Regex("I10");
 3  Console.WriteLine(re1.IsMatch("код I10.9"));   // true
 4  Console.WriteLine(re1.IsMatch("XI10Y"));        // true — помилковий збіг!
 5
 6  // З \b — тільки якщо I10 є окремим словом
 7  var re2 = new Regex(@"\bI10\b");
 8  Console.WriteLine(re2.IsMatch("код I10.9"));   // true — між '0' (\w) і '.'
    (\W) є межа слова
 9  Console.WriteLine(re2.IsMatch("код I105"));    // false — після '0' йде '5',
    межі немає
10  Console.WriteLine(re2.IsMatch("код I10 є"));   // true — I10 між пробілами
```

Зверніть увагу на перший виклик з \b: крапка не є «словесним» символом, тому між 0 і . межа слова є, і "код I10.9" дає збіг — \b відсікає лише ті випадки, коли код продовжується літерою, цифрою чи _

(XI10Y, I105). Якщо потрібно відкинути саме I10 з уточненням після крапки, додайте негативний lookahead: @"\bI10\b(?!\.)".

Межі слова є важливими при пошуку кодів у клінічних текстах, де код може бути частиною більшого ідентифікатора. Ще одна .NET-особливість: \d і \w за замовчуванням працюють з Unicode — \d збігається, наприклад, і з арабсько-індійською цифрою ٣, а \w — з кирилицею. Якщо потрібні лише ASCII-цифри, пишіть [0-9] або додайте RegexOptions.ECMAScript.

Lookahead та lookbehind

Lookahead (?=...) і **lookbehind** (?<=...) дозволяють описувати умови оточення збігу, **не включаючи** це оточення у результат. Це так звані «нульові твердження» (zero-width assertions) — вони перевіряють контекст, але не «з'їдають» символи:

```
C#
1  // Знайти число, що стоїть перед " мм рт.ст."
2  var reBP = new Regex(@"\d+(?= мм рт\.ст\.)");
3  Match m = reBP.Match("Тиск 140 мм рт.ст. норма");
4  Console.WriteLine(m.Value); // "140" — без " мм рт.ст."
5
6  // Знайти число після "ЧСС: "
7  var reHR = new Regex(@"(?<=ЧСС: )\d+");
8  Match m2 = reHR.Match("АТ: 140, ЧСС: 78, SpO2: 97");
9  Console.WriteLine(m2.Value); // "78" — без "ЧСС: "
```

Negative lookahead (?!...) та negative lookbehind (?<!...) — інвертовані версії: збіг відбувається лише якщо **не** виконується умова оточення.

Клас Regex та основні методи

```
C#
 1  using System.Text.RegularExpressions;
 2
 3  var regex = new Regex(@"[A-Z]\d{2}(\.?\d)?");
 4
 5  // IsMatch — bool: чи є хоч один збіг
 6  bool ok = regex.IsMatch("I10.9");   // true
 7  bool no = regex.IsMatch("999");     // false
 8
 9  // Match — перший збіг
10  Match m = regex.Match("код: I10.9, примітка");
11  Console.WriteLine(m.Value);  // "I10.9"
12  Console.WriteLine(m.Index);  // 5 — позиція у рядку
13  Console.WriteLine(m.Success); // true — перевіряємо перед використанням
14
15  // Matches — всі збіги
```

```
16  MatchCollection all = regex.Matches("I10.9 і J45.0");
17  foreach (Match match in all)
18      Console.WriteLine(match.Value); // "I10.9", "J45.0"
```

Властивість m.Success — обов'язкова перевірка при роботі з Match. Якщо збігу немає, m.Value та m.Index повернуть порожній рядок та 0 відповідно — без виключень. Тому код, який звертається до m.Value без перевірки m.Success, може давати хибні результати замість очевидної помилки.

RegexOptions

Конструктор Regex приймає параметр RegexOptions, який змінює поведінку шаблону:

Параметр	Ефект
IgnoreCase	Нечутливість до регістру
Multiline	^ та $ — початок/кінець кожного рядка
Singleline	. відповідає і \n
Compiled	Компіляція у IL для швидшого виконання при багаторазовому використанні
CultureInvariant	Ігнорувати регіональні відмінності

```
C#
1  var re = new Regex(@"гіпертензія", RegexOptions.IgnoreCase |
   RegexOptions.Compiled);
```

RegexOptions.Compiled заслуговує на особливу увагу: без цього параметра .NET розбирає шаблон у внутрішнє представлення й інтерпретує його при кожному виклику IsMatch/Match. З Compiled — при створенні об'єкта Regex шаблон компілюється у IL (проміжну мову CLR, яку далі JIT перетворює на машинний код), і всі наступні виклики виконуються значно швидше. Платою є затримка і більше пам'яті при створенні. Починаючи з .NET 7 є ще кращий варіант — **генератор вихідного коду** [GeneratedRegex]: атрибут на partial-методі змушує компілятор C# згенерувати код пошуку ще на етапі збірки, тож немає ні затримки при запуску, ні рефлексії:

```C#
partial class IcdValidator
{
    [GeneratedRegex(@"^[A-Z]\d{2}(\.?\d)?$")]
    private static partial Regex IcdRegex();

    public static bool IsValid(string code) => IcdRegex().IsMatch(code);
}
```

Іменовані групи

Групи (...) дозволяють витягати підчастини збігу. **Іменовані групи** (?<name>...) зручніші за числові: не потрібно пам'ятати порядок:

```C#
var re = new Regex(
    @"(?<last>\w+);(?<first>\w+);(?<age>\d+);(?<icd>[A-Z]\d{2}\.?\d?)");

Match m = re.Match("Петренко;Іван;67;I10.9");
if (m.Success)
{
    Console.WriteLine(m.Groups["last"].Value);  // Петренко
    Console.WriteLine(m.Groups["age"].Value);   // 67
    Console.WriteLine(m.Groups["icd"].Value);   // I10.9
}
```

Якщо поля в рядку змінять порядок — треба змінити лише шаблон, а не весь код читання груп.

Числові групи m.Groups[1] доступні поряд з іменованими. Нульова група m.Groups[0] завжди дорівнює m.Value — тобто весь збіг цілком. Групи можуть бути вкладеними, а група під квантифікатором може захопити кілька фрагментів в одному збігу (наприклад, (\w+;)+). Тоді m.Groups[1].Value містить лише **останнє** захоплення, а m.Groups[1].Captures — усі.

Replace та Split

Replace замінює всі збіги шаблону на заданий рядок:

```C#
// Видалити всі не-цифри з телефону
string phone  = "+38 (067) 123-45-67";
string digits = Regex.Replace(phone, @"\D", ""); // "380671234567"

// Нормалізувати множинні пробіли
string notes  = "пацієнт   прийнятий   10.06.2026";
string clean  = Regex.Replace(notes, @"\s{2,}", " "); // "пацієнт прийнятий
  10.06.2026"
```

Replace також підтримує **підстановки** (back-references) у рядку заміни. Наприклад, $1 вставляє вміст першої групи, ${name} — вміст іменованої групи. Це дозволяє трансформувати формат без повторного парсинга:

```
C#
1  // Перетворити "Петренко Іван" → "Іван Петренко"
2  string name    = "Петренко Іван";
3  string swapped = Regex.Replace(name, @"(?<last>\w+) (?<first>\w+)", "${first}
   ${last}");
4  Console.WriteLine(swapped); // "Іван Петренко"
```

Regex.Split розбиває рядок по збігах шаблону — потужніший аналог string.Split:

```
C#
1  // Розбити по будь-якій кількості пробілів або крапок з комою
2  string[] parts = Regex.Split("Петренко ; Іван ;; 67", @"\s*;\s*");
3  // ["Петренко", "Іван", "", "67"]
```

Статичні методи Regex

Клас Regex дозволяє викликати методи статично, без створення об'єкта. Зручно для разових перевірок:

```
C#
1  bool valid    = Regex.IsMatch("I10.9", @"^[A-Z]\d{2}(\.?\d)?$");
2  string digits = Regex.Replace("+38 067 123-45-67", @"\D", "");
```

Але для повторного використання в циклі або гарячому шляху краще створити об'єкт Regex **один раз** і зберегти його у полі класу — об'єкт Regex є **потокобезпечним** (thread-safe) для читання. Статичні методи внутрішньо кешують останні 15 шаблонів, але при багатьох різних шаблонах кеш переповнюється, і компіляція відбувається щоразу заново.

Оптимальний паттерн для сервісних класів:

```
C#
1  class PatientValidator
2  {
3      // Компілюється один раз при першому виклику ValidateIcd
4      private static readonly Regex IcdRegex =
5          new Regex(@"^[A-Z]\d{2}(\.?\d)?$", RegexOptions.Compiled);
6
7      public bool ValidateIcd(string code) => IcdRegex.IsMatch(code);
8  }
```

readonly static поле гарантує, що об'єкт Regex створюється лише раз і безпечно використовується з будь-яких потоків.

Валідація клінічних даних — runnable приклад

Перевірка формату коду МКХ-10 та телефону пацієнта:

```C#
using System;
using System.Text.RegularExpressions;

// Шаблони
var icdRe   = new Regex(@"^[A-Z]\d{2}(\.?\d)?$");
var phoneRe = new Regex(@"^\+?(\d[\s\-]?){10,13}$");

// Тестові дані
string[] icds = { "I10.9", "J45.0", "999", "i10", "K25", "Z00.0" };
Console.WriteLine("=== Валідація МКХ-10 ===");
foreach (var code in icds)
    Console.WriteLine($"  {code,-8} -> {(icdRe.IsMatch(code) ? "OK" : "INVALID")}");

string[] phones = {
    "+380671234567",
    "0671234567",
    "+38 067 123-45-67",
    "abc",
    "123"
};
Console.WriteLine("\n=== Валідація телефону ===");
foreach (var p in phones)
    Console.WriteLine($"  {p,-22} -> {(phoneRe.IsMatch(p) ? "OK" : "INVALID")}");

// Витягування всіх кодів МКХ з клінічної нотатки
string notes = "Основний: I10.9. Супутні: J45.0, E11.9. Перенесений: K25.";
var allCodes = new Regex(@"[A-Z]\d{2}\.?\d?").Matches(notes);
Console.WriteLine("\n=== Коди МКХ у нотатці ===");
foreach (Match m in allCodes)
    Console.WriteLine($"  {m.Value} (позиція {m.Index})");
```

Парсинг медичного запису через іменовані групи — runnable приклад

```C#
using System;
using System.Text.RegularExpressions;

string[] records = {
    "Петренко;Іван;67;I10.9;кардіологія",
    "Коваль;Марія;45;J45.0;пульмонологія",
    "Сидоренко;Олег;71;K25.3;гастроентерологія",
};

var re = new Regex(
```

```
    @"(?<last>\w+);(?<first>\w+);(?<age>\d+);(?<icd>[A-Z]\d{2}\.?\d?);(?<dept>\w+)");

Console.WriteLine($"{"ПІБ",-22}{"Вік",5}{"МКХ",8}  Відділення");
Console.WriteLine(new string('-', 55));

foreach (var row in records)
{
    Match m = re.Match(row);
    if (!m.Success) { Console.WriteLine($"  Помилка: {row}"); continue; }

    string name = $"{m.Groups["last"].Value} {m.Groups["first"].Value}";
    int    age  = int.Parse(m.Groups["age"].Value);
    string icd  = m.Groups["icd"].Value;
    string dept = m.Groups["dept"].Value;

    Console.WriteLine($"{name,-22}{age,5}{icd,8}  {dept}");
}

Console.WriteLine(new string('-', 55));

// Replace: видалити все крім цифр з телефону
string rawPhone = "+38 (067) 123-45-67";
string digits   = Regex.Replace(rawPhone, @"\D", "");
Console.WriteLine($"\nТелефон: {rawPhone} -> {digits}");

// Нормалізація пробілів у нотатці
string note  = "пацієнт   Петренко   прийнятий   10.06.2026";
string clean = Regex.Replace(note, @"\s{2,}", " ");
Console.WriteLine($"Нотатка: {clean}");
```

Питання для самоконтролю — Розділ 11. Робота з рядками

11.1. Рядки та клас String

1. Що означає **незмінність** (immutability) рядків у C#? Що насправді відбувається при виклику diagnosis.ToUpper() — повертається змінений оригінал чи новий об'єкт?
2. Що виведе наступний код і чому?

```
C#
string a = "I10";
string b = "I10";
string c = new string(new char[] { 'I', '1', '0' });
Console.WriteLine(a == b);
Console.WriteLine(a == c);
Console.WriteLine(ReferenceEquals(a, b));
Console.WriteLine(ReferenceEquals(a, c));
```

3. Що таке **string interning** і навіщо він потрібен? Чому рядки, створені через конкатенацію або new string(...), за замовчуванням не потрапляють у пул?
4. Поясніть різницю між типами char і string на рівні пам'яті. Де зберігається char — в heap чи на стеку? А string?
5. Чим string.IsNullOrWhiteSpace відрізняється від string.IsNullOrEmpty? Наведіть приклад рядка, для якого перший метод поверне true, а другий — false.
6. Поясніть, чому s1 == s2 для рядків є порівнянням **значень**, а не посилань, хоча для більшості класів == порівнює посилання. Що для цього потрібно на рівні мови?
7. Що поверне icd.Length для рядка "I10.9"? Чому для деяких Unicode-символів Length може бути більшим за кількість видимих символів?
8. Для порівняння медичних кодів без урахування регістру який метод є **кращим** з точки зору продуктивності та коректності: code.ToLower() == "i10" чи string.Equals(code, "I10", StringComparison.OrdinalIgnoreCase)? Обґрунтуйте відповідь.

11.2. Операції з рядками

9. Що поверне "I10.9".Substring(0, 3) і "I10.9"[^2..]? Запишіть еквівалентний синтаксис Range для обох варіантів.
10. Що виведе наступний код? Знайдіть потенційну проблему та поясніть як її уникнути.

```
C#
1  string notes = "Прийом: 09:00. Виписка: 14:00. Повторний прийом: 17:00";
2  int pos = notes.IndexOf("прийом");
3  Console.WriteLine(pos);
4  int pos2 = notes.IndexOf("прийом", 0, StringComparison.OrdinalIgnoreCase);
5  Console.WriteLine(pos2);
```

11. Поясніть роль параметра StringSplitOptions.RemoveEmptyEntries. Що поверне "a;;b".Split(';') без цього параметра і з ним?
12. Поясніть оператор ^ (hat) та синтаксис Range (..) у C# 8+. Що означають icd[^1], icd[1..3], icd[^2..]?
13. Чому **ланцюгування** Replace є можливим і зручним:

```
C#
```

```
string result = template
    .Replace("{NAME}", name)
    .Replace("{DATE}", date);
```

Як це пов'язане з незмінністю рядків?

14. Напишіть код, який із рядка " Петренко Іван Степанович \t" видаляє пробіли з обох кінців, перетворює на верхній регістр і замінює всі пробіли на дефіси.
15. Порівняйте string.Compare(s1, s2) і s1.CompareTo(s2). Які значення вони повертають і коли? Чому перевага надається string.Compare при необхідності контролю над режимом порівняння?
16. Поясніть, чому diagnosis.Contains("АРТЕРІАЛЬНА", StringComparison.OrdinalIgnoreCase) є кращим за diagnosis.ToLower().Contains("артеріальна") не лише з точки зору продуктивності, але й коректності.

11.3. Форматування та інтерполяція рядків

17. Перелічіть три підходи до вставки значень у рядки в C#. Для якого сценарію кожен є найбільш доречним?
18. Що виведе наступний код? Поясніть, що виводить кожен рядок.

```
C#
double bp = 140.567;
int id = 42;
Console.WriteLine($"{bp:F1}");
Console.WriteLine($"{bp:F2}");
Console.WriteLine($"{id:D6}");
Console.WriteLine($"{0.847:P1}");
Console.WriteLine($"{12345.6:N0}");
```

19. Поясніть різницю між специфікаторами F, N і C. Де для кожного найкраще застосування в медичному контексті?
20. Що таке **вирівнювання** у інтерполяції рядків? Поясніть синтаксис {value,8} і {name,-20}. В якому напрямку вирівнює кожна форма?
21. Поясніть чому $"..." на рівні IL є еквівалентом string.Format(). В чому переваги $"..." перед string.Format для розробника?
22. Що таке **raw string literals** (C# 11) і які проблеми вони вирішують? Наведіть приклад рядка з лапками всередині у старому і новому синтаксисі.

23. Для чого потрібен параметр IFormatProvider (або CultureInfo) у string.Format? Чому при серіалізації числових даних у файли завжди слід передавати CultureInfo.InvariantCulture?
24. Напишіть код, який форматує таблицю зі стовпцями "Пацієнт" (ширина 24, ліворуч), "Вік" (ширина 5, праворуч), "АТ" (ширина 8, одна десяткова, праворуч) для трьох пацієнтів.

11.4. Клас StringBuilder

25. Чому конкатенація рядків у циклі через += має складність $O(N^2)$? Поясніть причину через механізм незмінності рядків і підрахуйте загальну кількість операцій копіювання для N=100.
26. Що виведе наступний код? Поясніть значення Length і Capacity після кожного кроку.

```
C#
1  var sb = new StringBuilder("Виписка:");
2  Console.WriteLine($"L={sb.Length}, C={sb.Capacity}");
3  sb.Append(" Петренко Іван Степанович");
4  Console.WriteLine($"L={sb.Length}, C={sb.Capacity}");
5  sb.Length = 8;
6  Console.WriteLine(sb.ToString());
```

27. Поясніть алгоритм **подвоєння ємності** StringBuilder. Чому загальна складність вставки N символів є O(N), а не O(N log N)?
28. Чим Clear() відрізняється від new StringBuilder()? Коли Clear() є ефективнішим варіантом?
29. Поясніть, що означає **fluent API** у контексті StringBuilder. Що повертає кожен метод (Append, AppendLine) і чому це уможливлює ланцюгування?
30. Напишіть метод string BuildReport(Patient[] patients), який через StringBuilder формує багаторядковий текстовий звіт: заголовок, рядок для кожного пацієнта (ім'я, вік, МКХ), підсумковий рядок з кількістю пацієнтів.
31. Порівняйте StringBuilder і string для наступних сценаріїв: (а) об'єднання 5 рядків поза циклом; (б) побудова рядка з 1000 фрагментів у циклі; (в) пошук підрядка у готовому тексті. Який варіант кращий у кожному випадку?
32. Знайдіть проблему та поясніть як її виправити:

```
C#
1  var sb = new StringBuilder();
```

```
2  for (int i = 0; i < 10_000; i++)
3      sb = new StringBuilder(); // reset
4      sb.Append($"Запис {i}\n");
5  Console.WriteLine(sb.ToString());
```

11.5. Регулярні вирази

33. Яку задачу вирішують регулярні вирази, яку не вирішують стандартні методи класу String (Contains, IndexOf, Split)? Наведіть конкретний приклад патерну, який неможливо виразити через ці методи.
34. Що виведе наступний код і чому?

```
C#
1  var re = new Regex(@"^[A-Z]\d{2}(\.?\d)?$");
2  string[] codes = { "I10.9", "J45.0", "999", "i10", "K25", "AB12.3" };
3  foreach (var c in codes)
4      Console.WriteLine($"{c}: {re.IsMatch(c)}");
```

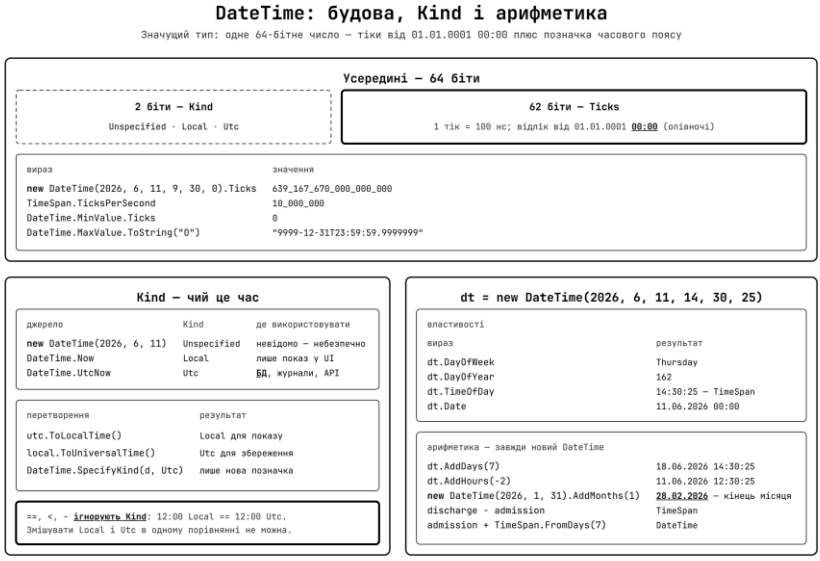
35. Поясніть різницю між **жадібним** і **лінивим** квантифікатором. Що поверне <.+> і <.+?> для рядка "текст"?
36. Поясніть метасимвол \b (межа слова). Чим результат IsMatch("код I10.9", @"I10") відрізняється від IsMatch("код I10.9", @"\bI10\b")?
37. Що таке **іменовані групи** (?<name>...) і яку перевагу вони мають над числовими групами? Напишіть шаблон для парсингу рядка "Петренко;Іван;67" з іменованими групами last, first, age.
38. Поясніть різницю між lookahead (?=...) і lookbehind (?<=...). Напишіть вираз, який витягує **число** зі строки "АТ: 140 мм рт.ст." без включення одиниць виміру у результат.
39. Порівняйте підходи: (а) статичний метод Regex.IsMatch(str, pattern) у циклі; (б) new Regex(pattern, RegexOptions.Compiled) як поле класу. В якому випадку кожен підхід є оптимальним?
40. Спроєктуйте клас PatientValidator зі статичними readonly полями Regex для валідації: коду МКХ-10 (формат [A-Z]\d{2} або [A-Z]\d{2}.\d), номера телефону (10-13 цифр, можливо з +, пробілами, дефісами). Реалізуйте методи ValidateIcd(string) і ValidatePhone(string).

Розділ 12. Робота з датами та часом

12.1. Структура DateTime

Дата і час — невід'ємна частина медичних інформаційних систем. Час прийому пацієнта, дата виписки, вік пацієнта в роках, тривалість госпіталізації, терміни зберігання аналізів — усе це вимагає роботи з датами. Центральний тип у .NET для цього завдання — структура DateTime з простору імен System.

DateTime — це **структура** (struct), тобто **значущий тип** (value type). На відміну від класів, екземпляр DateTime зберігається безпосередньо на стеку або всередині об'єкта і копіюється при присвоєнні — так само як int або double. Порівняння двох DateTime через == порівнює їхні **значення**, а не посилання.



Структура DateTime — внутрішня будова та ключові властивості

Внутрішнє представлення: тіки

Внутрішньо DateTime зберігає єдине 64-бітне ціле число. Його молодші 62 біти — кількість **тіків** (ticks), а старші 2 біти — значення Kind (про

нього нижче). Один тік дорівнює 100 наносекундам, тобто в секунді 10 000 000 тіків (TimeSpan.TicksPerSecond). Відлік іде від **опівночі** 1 січня 0001 року (00:00:00.0000000) за григоріанським календарем; DateTime.MinValue.Ticks дорівнює 0. Доступ до тіків — через властивість Ticks:

```
C#
1  DateTime admission = new DateTime(2026, 6, 11, 9, 30, 0);
2  Console.WriteLine($"{admission.Ticks}"); // 639167670000000000
```

Знання про тіки важливе для двох причин: по-перше, всі операції порівняння і арифметика дат зводяться до порівняння та арифметики цих цілих чисел, що є дуже швидким; по-друге, TimeSpan (розд. 12.4) також зберігає тіки — тому два типи легко взаємодіють.

DateTimeKind: прив'язка до часового поясу

Кожен об'єкт DateTime має властивість Kind типу DateTimeKind — вона визначає, до якого часового поясу прив'язане значення:

Kind	**Значення**	**Коли використовується**
Unspecified	Невідомо (за замовчуванням)	Локальні розрахунки без прив'язки до поясу
Local	Локальний час ОС	Відображення в UI поточного сервера/комп'ютера
Utc	Час UTC (Coordinated Universal Time)	Зберігання у БД, API, аудит-журнали

```
C#
1  DateTime local = DateTime.Now;        // Kind = Local
2  DateTime utc   = DateTime.UtcNow;     // Kind = Utc
3  DateTime unsp  = new DateTime(2026, 6, 11); // Kind = Unspecified
```

Зверніть увагу: оператори порівняння й віднімання DateTime (==, <, -) **не враховують Kind** — вони порівнюють лише тіки. Тому 12:00 Local == 12:00 Utc дає true, хоча це різні моменти часу (на сервері в Києві між ними 3 години). Саме тому змішувати значення з різним Kind в одному порівнянні не можна — це і показує приклад нижче.

Чому це критично в медичних системах: якщо лікарня або телемедичний сервіс працює в кількох часових поясах, зберігання DateTime.Now (Local) у базі даних призводить до помилок при агрегації даних. Правило: **завжди зберігати у БД UTC** (DateTime.UtcNow), а конвертацію у локальний час виконувати на рівні UI.

UTC vs Local: практична пастка та конвертація

Порушення правила «зберігати UTC, відображати локальний час» призводить до важко діагностованих помилок:

```
C#
 1  using System;
 2
 3  // === ПРОБЛЕМА: зберігаємо Local, порівнюємо неправильно ===
 4  Console.WriteLine("=== Антипатерн: змішування Local і Utc ===");
 5
 6  DateTime localAdmission = DateTime.Now;          // Kind = Local
 7  DateTime utcAuditLog    = DateTime.UtcNow;       // Kind = Utc
 8
 9  // Ці два значення в один момент відрізняються на offset часового поясу!
10  Console.WriteLine($"Local: {localAdmission:HH:mm:ss}
    Kind={localAdmission.Kind}");
11  Console.WriteLine($"Utc:   {utcAuditLog:HH:mm:ss}  Kind={utcAuditLog.Kind}");
12
13  // Порівняння Local з Utc дає НЕПРАВИЛЬНИЙ результат:
14  bool wrongComparison = localAdmission > utcAuditLog; // на UTC+2/UTC+3 буде
    true!
15  Console.WriteLine($"\nlocalAdmission > utcAuditLog = {wrongComparison}  ←
    НЕПРАВИЛЬНО");
16  Console.WriteLine("(DateTime порівнює тіки без урахування Kind)");
17
18  // === ПРАВИЛЬНО: завжди одна «мова» — UTC ===
19  Console.WriteLine("\n=== Правильно: конвертація та порівняння в UTC ===");
20
21  DateTime utcAdmission = DateTime.UtcNow;  // зберігаємо в БД UTC
22  DateTime utcEvent     = DateTime.UtcNow.AddMinutes(5);
23
24  bool correctComparison = utcAdmission < utcEvent; // true — правильно
25  Console.WriteLine($"utcAdmission < utcEvent = {correctComparison}  ←
    ПРАВИЛЬНО");
26
27  // Конвертація UTC -> Local для відображення в UI
28  DateTime displayTime = utcAdmission.ToLocalTime();
29  Console.WriteLine($"\nДля відображення в UI: {displayTime:dd.MM.yyyy HH:mm:ss}
    (local)");
30  Console.WriteLine($"Зберігаємо в БД:       {utcAdmission:dd.MM.yyyy HH:mm:ss}
    (utc)");
```

Методи конвертації:

Метод	Що робить
.ToUniversalTime()	Local → UTC (або Unspecified → UTC припускаючи Local)
.ToLocalTime()	UTC → Local
DateTime.SpecifyKind(dt, kind)	Перепозначити Kind без зміни значення

```C#
using System;

// SpecifyKind: якщо з БД прийшов DateTime без Kind (Unspecified),
// але ми знаємо, що це UTC:
DateTime fromDb = new DateTime(2026, 6, 11, 7, 30, 0); // Kind = Unspecified
Console.WriteLine($"З БД (Unspecified): {fromDb.Kind}");

DateTime asUtc = DateTime.SpecifyKind(fromDb, DateTimeKind.Utc);
Console.WriteLine($"Після SpecifyKind:  {asUtc.Kind}");

DateTime localForDisplay = asUtc.ToLocalTime();
Console.WriteLine($"Для UI (Local):     {localForDisplay:HH:mm}
({localForDisplay.Kind})");
```

Правило для медичних систем: **все, що стосується аудиту, журналів, зберігання у БД, API** — тільки UTC. **Відображення у UI або при звітуванні** — конвертувати у Local прямо перед виводом.

Конструктори DateTime

```C#
// Мінімальна дата (01.01.0001 00:00:00)
DateTime zero = new DateTime();
Console.WriteLine(DateTime.MinValue); // 01.01.0001 0:00:00

// Рік, місяць, день
DateTime birthDate = new DateTime(1958, 4, 15); // 15 квітня 1958 р.

// Рік, місяць, день, година, хвилина, секунда
DateTime admission = new DateTime(2026, 6, 11, 9, 30, 0); // 09:30:00

// Повна точність: + мілісекунди
DateTime precise  = new DateTime(2026, 6, 11, 9, 30, 0, 123); // 09:30:00.123

// З явним Kind
DateTime utc = new DateTime(2026, 6, 11, 7, 30, 0, DateTimeKind.Utc);
```

Статичні властивості

Властивість	Що повертає
DateTime.Now	Поточна дата і час (Kind=Local)
DateTime.UtcNow	Поточна дата і час (Kind=Utc)
DateTime.Today	Поточна дата, час = 00:00:00
DateTime.MinValue	01.01.0001 00:00:00
DateTime.MaxValue	31.12.9999 23:59:59.9999999

```
C#
1  Console.WriteLine(DateTime.Now);    // 11.06.2026 11:43:33
2  Console.WriteLine(DateTime.UtcNow); // 11.06.2026  8:43:33 (UTC+3 → -3 год)
3  Console.WriteLine(DateTime.Today);  // 11.06.2026  0:00:00
```

Властивості екземпляра

Об'єкт DateTime надає доступ до кожного компоненту окремо:

Властивість	Тип	Опис
Year / Month / Day	int	Рік, місяць (1-12), день (1-31)
Hour / Minute / Second	int	Година (0-23), хвилина (0-59), секунда (0-59)
Millisecond	int	Мілісекунди (0-999)
DayOfWeek	DayOfWeek	День тижня (DayOfWeek.Monday тощо)
DayOfYear	int	Порядковий номер дня у році (1-366)
TimeOfDay	TimeSpan	Час доби від початку дня
Date	DateTime	Лише дата, час = 00:00:00
Kind	DateTimeKind	Тип часової прив'язки

Властивість	Тип	Опис
Ticks	long	Кількість тіків

```
C#
1  DateTime dt = new DateTime(2026, 6, 11, 14, 30, 25);
2
3  Console.WriteLine(dt.Year);        // 2026
4  Console.WriteLine(dt.DayOfWeek);   // Thursday
5  Console.WriteLine(dt.DayOfYear);   // 162
6  Console.WriteLine(dt.TimeOfDay);   // 14:30:25
7  Console.WriteLine(dt.Date);        // 11.06.2026 0:00:00
```

*Арифметика дат: методи Add**

Усі методи Add* повертають **новий** об'єкт DateTime — не змінюють оригінал (структура є незмінним значенням у цьому сенсі):

Метод	Опис
AddYears(n)	Додати/відняти роки
AddMonths(n)	Додати/відняти місяці
AddDays(n)	Додати/відняти дні
AddHours(n)	Додати/відняти години
AddMinutes(n)	Додати/відняти хвилини
AddSeconds(n)	Додати/відняти секунди
Add(TimeSpan)	Додати довільний інтервал

Для віднімання передають **від'ємне число**:

```
C#
1  DateTime admission = new DateTime(2026, 6, 11, 9, 30, 0);
2
3  DateTime discharge = admission.AddDays(7);        // виписка через 7 днів
4  DateTime nextVisit = discharge.AddMonths(1);       // повторний огляд за місяць
5  DateTime before    = admission.AddHours(-2);       // 2 години до прийому
6
7  Console.WriteLine(discharge);  // 18.06.2026 9:30:00
8  Console.WriteLine(nextVisit);  // 18.07.2026 9:30:00
```

Різниця між датами: Subtract і оператор –

Метод Subtract(DateTime) і оператор – між двома датами повертають **TimeSpan** — об'єкт, що представляє тривалість інтервалу (розд. 12.4):

```
C#
1  DateTime admission  = new DateTime(2026, 6, 1, 10, 0, 0);
2  DateTime discharge  = new DateTime(2026, 6, 8, 14, 30, 0);
3
4  TimeSpan stay = discharge - admission;          // або
   discharge.Subtract(admission)
5  Console.WriteLine(stay.Days);                   // 7
6  Console.WriteLine(stay.TotalHours);             // 172.5
```

Порівняння дат

Оператори >, <, >=, <=, ==, != для DateTime порівнюють значення хронологічно:

```
C#
1  DateTime d1 = new DateTime(2026, 6, 1);
2  DateTime d2 = new DateTime(2026, 7, 1);
3
4  bool earlier = d1 < d2;  // true — червень раніше за липень
5  bool equal   = d1 == d1; // true
6
7  int cmp = DateTime.Compare(d1, d2); // -1 (d1 < d2), 0 (рівні), +1 (d1 > d2)
```

Для перевірки чи дата потрапляє у діапазон достатньо двох умов:

```
C#
1  DateTime start = new DateTime(2026, 6, 1);
2  DateTime end   = new DateTime(2026, 6, 30);
3  DateTime today = DateTime.Today;
4
5  bool inRange = today >= start && today <= end;
```

Корисні статичні методи

```
C#
1  // Перевірка високосного року
2  bool leap = DateTime.IsLeapYear(2024); // true
3
4  // Кількість днів у місяці
5  int days = DateTime.DaysInMonth(2024, 2); // 29 (2024 — високосний)
```

Дата народження та вік пацієнта — runnable приклад

Розрахунок віку пацієнта та основні операції з датами прийому:

```C#
using System;

DateTime birthDate = new DateTime(1958, 4, 15);
DateTime admission = new DateTime(2026, 6, 11, 9, 30, 0);
DateTime discharge = admission.AddDays(7);

Console.WriteLine("=== Картка пацієнта ===");
Console.WriteLine($"Дата народження: {birthDate:dd.MM.yyyy}");
Console.WriteLine($"День народження: {birthDate.DayOfWeek}");

int age = admission.Year - birthDate.Year;
if (birthDate.Date > admission.AddYears(-age).Date) age--;
Console.WriteLine($"Вік при вступі: {age} р.");

Console.WriteLine($"\n=== Госпіталізація ===");
Console.WriteLine($"Прийом:  {admission:dd.MM.yyyy HH:mm}");
Console.WriteLine($"Виписка: {discharge:dd.MM.yyyy HH:mm}");

TimeSpan stay = discharge - admission;
Console.WriteLine($"Тривалість: {stay.Days} днів");

DateTime nextVisit = discharge.AddMonths(1);
Console.WriteLine($"Повторний огляд: {nextVisit:dd.MM.yyyy}");

Console.WriteLine($"\n=== Властивості DateTime ===");
Console.WriteLine($"DayOfYear: {admission.DayOfYear}");
Console.WriteLine($"Kind:      {DateTime.UtcNow.Kind}");
Console.WriteLine($"IsLeapYear(2026): {DateTime.IsLeapYear(2026)}");
Console.WriteLine($"DaysInMonth(2026,2): {DateTime.DaysInMonth(2026, 2)}");
```

Розклад прийомів у відділенні — runnable приклад

Робота з датами та часом в контексті розкладу пацієнтів:

```C#
using System;

Console.WriteLine("=== Розклад прийомів кардіологічного відділення ===");
Console.WriteLine($"{"Пацієнт",-22} {"Прийом",16} {"Виписка",16} {"Днів",6}");
Console.WriteLine(new string('-', 65));

var records = new[]
{
    ("Петренко Іван",     new DateTime(2026,6,1,  9,  0, 0), new DateTime(2026,6, 8, 14, 0, 0)),
    ("Коваль Марія",      new DateTime(2026,6,3,  11, 30, 0), new DateTime(2026,6,10, 10, 0, 0)),
    ("Сидоренко Олег",    new DateTime(2026,6,10, 8,  0, 0), new DateTime(2026,6,15, 16, 0, 0)),
};

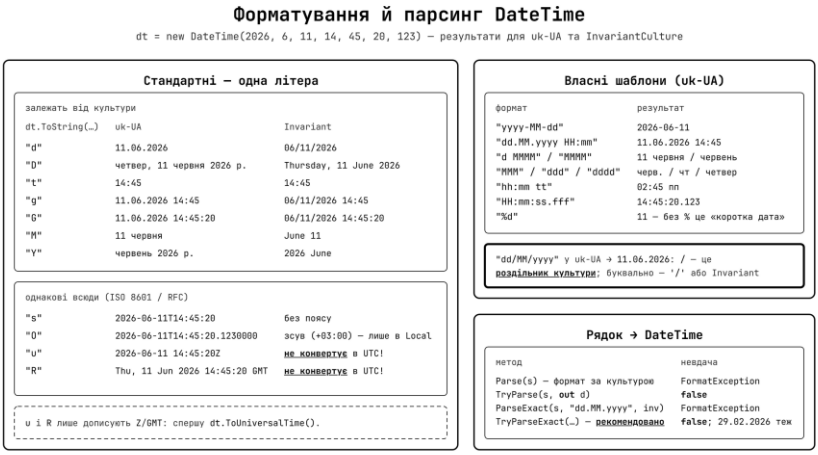
TimeSpan totalStay = TimeSpan.Zero;
foreach (var (name, adm, dis) in records)
{
    TimeSpan stay = dis - adm;
    totalStay += stay;
```

```
19      Console.WriteLine($"{name,-22} {adm,16:dd.MM.yy HH:mm} {dis,16:dd.MM.yy
    HH:mm} {stay.Days,6}");
20  }
21
22  Console.WriteLine(new string('-', 65));
23  Console.WriteLine($"{"Середня тривалість:",-22} {totalStay.TotalDays /
    records.Length,39:F1} днів");
24
25  Console.WriteLine($"\n=== Порівняння дат ===");
26  DateTime today  = new DateTime(2026, 6, 11);
27  DateTime future = today.AddDays(30);
28  Console.WriteLine($"Сьогодні: {today:dd.MM.yyyy}");
29  Console.WriteLine($"За 30 днів: {future:dd.MM.yyyy}  Day={future.DayOfWeek}");
30  Console.WriteLine($"Compare(today, future) = {DateTime.Compare(today,
    future)}");
```

12.2. Налаштування формату часу та дати

Об'єкт DateTime зберігає дату і час як точне числове значення — але виводити його на екран потрібно в людиночитному форматі. Проблема: у різних країнах ці формати різні. У США пишуть 06/11/2026, в Україні — 11.06.2026, у міжнародних стандартах — 2026-06-11. Медичні системи мають підтримувати чітко визначені формати для обміну даними між установами, для друку документів та для зберігання в базах даних.

Клас DateTime вирішує це через метод ToString(format), де format — рядок із специфікаторами форматування.



Форматування DateTime — стандартні та кастомні специфікатори

Стандартні специфікатори форматування

Стандартні специфікатори — це однолітерні коди, що дають готові локалізовані формати. Їхній вивід залежить від поточної культури (CultureInfo) операційної системи — тому "d" в Україні дає 11.06.2026, а в США — 6/11/2026:

```
C#
 1  DateTime now = new DateTime(2026, 6, 11, 14, 45, 20);
 2
 3  Console.WriteLine(now.ToString("d")); // 11.06.2026       – коротка дата
 4  Console.WriteLine(now.ToString("D")); // четвер, 11 червня 2026 р. – довга дата
 5  Console.WriteLine(now.ToString("t")); // 14:45            – короткий час
 6  Console.WriteLine(now.ToString("T")); // 14:45:20         – довгий час
 7  Console.WriteLine(now.ToString("f")); // четвер, 11 червня 2026 р. 14:45
 8  Console.WriteLine(now.ToString("F")); // четвер, 11 червня 2026 р. 14:45:20
 9  Console.WriteLine(now.ToString("g")); // 11.06.2026 14:45
10  Console.WriteLine(now.ToString("G")); // 11.06.2026 14:45:20
```

Важливо розуміти різницю між великою і малою літерою: "d" — коротка дата, "D" — довга; "t" — короткий час, "T" — довгий. Аналогічно для f/F і g/G.

Повна таблиця стандартних специфікаторів:

Специфікатор	Приклад	Опис
d	11.06.2026	Коротка дата
D	четвер, 11 червня 2026 р.	Довга дата (з днем тижня)
t	14:45	Короткий час
T	14:45:20	Довгий час
f	четвер, 11 червня 2026 р. 14:45	Довга дата + короткий час
F	четвер, 11 червня 2026 р. 14:45:20	Довга дата + довгий час
g	11.06.2026 14:45	Коротка дата + короткий час
G	11.06.2026 14:45:20	Коротка дата + довгий час (за замовчуванням)
M або m	11 червня	Місяць і день

Специфікатор	Приклад	Опис
Y або y	червень 2026 р.	Рік і місяць
s	2026-06-11T14:45:20	ISO 8601 (без часового поясу)
O або o	2026-06-11T14:45:20.0000000	ISO 8601 round-trip (7 знаків дробу; зсув +03:00 — лише для Kind=Local, Z — для Utc)
R або r	Thu, 11 Jun 2026 14:45:20 GMT	RFC 1123 (HTTP-заголовки)
u	2026-06-11 14:45:20Z	Універсальний сортований

Тут дві пастки. По-перше, приклади в таблиці наведено для культури uk-UA, а символи між числами (наприклад, нерозривний пробіл перед «р.») теж задає культура. По-друге, специфікатори "u" і "R" для DateTime **не перетворюють час в UTC** — вони лише дописують Z або GMT до того значення, яке є. Для локального часу спершу потрібно викликати ToUniversalTime(), інакше рядок «бреше» на величину зсуву поясу. (DateTimeOffset у цих форматах конвертується в UTC сам.)

Специфікатор "s" (ISO 8601) є особливо важливим у медичній інформатиці: стандарт HL7 FHIR, що використовується для обміну медичними даними між системами, вимагає саме ISO 8601 для дат. Специфікатор "O" дає повний ISO 8601 з усіма 7 знаками дробу секунди, а для Kind=Local чи Utc — ще й зсув (+03:00) або Z; саме цей формат гарантує точне відновлення значення при зворотному парсингу.

Кастомні специфікатори форматування

Кастомні специфікатори дають повний контроль над форматом. Вони комбінуються довільно в рядку формату:

```
C#
1  DateTime now = new DateTime(2026, 6, 11, 14, 45, 20, 123);
2
3  Console.WriteLine(now.ToString("dd.MM.yyyy"));          // 11.06.2026
4  Console.WriteLine(now.ToString("dd/MM/yyyy HH:mm:ss")); // 11.06.2026 14:45:20
   (!) y uk-UA
```

```
5  Console.WriteLine(now.ToString("yyyy-MM-dd"));          // 2026-06-11 (ISO дата)
6  Console.WriteLine(now.ToString("d MMMM yyyy р."));      // 11 червня 2026 р.
7  Console.WriteLine(now.ToString("dddd, d MMMM"));        // четвер, 11 червня
8  Console.WriteLine(now.ToString("HH:mm:ss.fff"));        // 14:45:20.123
```

Повна таблиця кастомних специфікаторів:

Специфікатор	Опис	Приклад
yyyy	Рік 4 цифри	2026
yy	Рік 2 цифри	26
MM	Місяць 2 цифри	06
MMM	Місяць коротко	черв.
MMMM	Місяць повністю	червень
dd	День 2 цифри	11
d	День без нуля	11 або 1
ddd	День тижня коротко	чт
dddd	День тижня повністю	четвер
HH	Година 24h (00-23)	14
hh	Година 12h (01-12)	02
mm	Хвилина (00-59)	45
ss	Секунда (00-59)	20
fff	Мілісекунди	123
tt	AM/PM (за культурою)	пп (uk-UA), PM (en-US)
zzz	Зсув часового поясу	+02:00

Символ / у шаблоні — не буквальна скісна риска, а **роздільник дати поточної культури**, так само як : — роздільник часу. Тому now.ToString("dd/MM/yyyy") в українській культурі дає 11.06.2026, а не 11/06/2026. Щоб отримати саме скісну риску, її екранують апострофами ("dd'/'MM'/'yyyy") або форматують з CultureInfo.InvariantCulture.

Якщо рядок формату містить лише один символ — він інтерпретується як **стандартний** специфікатор. Щоб рядок з одним

кастомним символом розпізнався правильно, ставте перед ним %: now.ToString("%d") → 11 (день місяця), а не "d" — коротка дата 11.06.2026.

Ще одна часта помилка в інтерполяції — порядок ширини і формату. Правильно {dt,12:dd.MM.yyyy}: спершу ширина через кому, потім формат після двокрапки. Запис {dt:dd.MM.yyyy,12} не дає помилки компіляції — ,12 просто стає частиною рядка формату й виводиться буквально: 11.06.2026,12.

Культура та CultureInfo

За замовчуванням ToString(format) використовує культуру поточного потоку (Thread.CurrentThread.CurrentCulture). Щоб форматувати незалежно від регіональних налаштувань, передається другий параметр IFormatProvider:

```
C#
 1  using System.Globalization;
 2
 3  DateTime dt = new DateTime(2026, 6, 11, 14, 45, 20);
 4
 5  // Явна культура — незалежна від ОС
 6  string ua  = dt.ToString("dd.MM.yyyy", CultureInfo.GetCultureInfo("uk-UA")); //
    11.06.2026
 7  string us  = dt.ToString("MM/dd/yyyy", CultureInfo.GetCultureInfo("en-US")); //
    06/11/2026
 8  string iso = dt.ToString("yyyy-MM-dd", CultureInfo.InvariantCulture);        //
    2026-06-11
 9
10  Console.WriteLine(ua);
11  Console.WriteLine(us);
12  Console.WriteLine(iso);
```

CultureInfo.InvariantCulture — культура, що не прив'язана до жодної конкретної мови чи регіону. Її використовують для **зберігання і передачі** дат між системами (файли, API, бази даних), де однакове форматування є критичним. Для **відображення** в UI — краще використовувати культуру користувача.

Парсинг: перетворення рядка у DateTime

Зворотна операція — перетворення рядка у DateTime — не менш важлива. Наприклад, читаємо дату народження пацієнта з поля форми або з CSV-файлу.

DateTime.Parse(s) — найпростіший варіант. Розпізнає багато поширених форматів автоматично, але залежить від культури і кидає FormatException при невдачі:

```
C#
DateTime dt = DateTime.Parse("11.06.2026 14:45"); // OK для uk-UA культури
```

DateTime.TryParse(s, out DateTime result) — безпечний варіант: не кидає виняток, а повертає false при невдачі. Рекомендований для обробки введення від користувача:

```
C#
string input = "11.06.2026";
if (DateTime.TryParse(input, out DateTime result))
    Console.WriteLine($"Розпізнано: {result:dd.MM.yyyy}");
else
    Console.WriteLine("Некоректна дата");
```

DateTime.ParseExact(s, format, provider) — парсинг з точно вказаним форматом. Якщо формат відомий наперед (наприклад, дані приходять із зовнішньої системи у фіксованому вигляді), цей метод є найнадійнішим:

```
C#
using System.Globalization;

// Формат з медичної інформаційної системи: "рр/ММ/дд"
DateTime dt = DateTime.ParseExact(
    "26/06/11",
    "yy/MM/dd",
    CultureInfo.InvariantCulture);
Console.WriteLine(dt); // 11.06.2026 0:00:00
```

DateTime.TryParseExact(s, format, provider, style, out result) — комбінація точного формату і безпечного повернення bool. Найбільш рекомендований варіант для парсингу в медичних системах:

```
C#
using System.Globalization;

string raw = "11.06.2026 14:30";
bool ok = DateTime.TryParseExact(
    raw,
    "dd.MM.yyyy HH:mm",
    CultureInfo.InvariantCulture,
    DateTimeStyles.None,
    out DateTime admission);

if (ok)
    Console.WriteLine($"Прийом: {admission:F}");
```

Форматування дат у медичних документах — runnable приклад

Використання стандартних і кастомних специфікаторів для різних форматів виводу:

```C#
using System;

DateTime admission = new DateTime(2026, 6, 11, 9, 30, 0);
DateTime discharge = new DateTime(2026, 6, 18, 14, 0, 0);
DateTime birth     = new DateTime(1958, 4, 15);

Console.WriteLine("=== Виписний епікриз ===");
Console.WriteLine($"Дата народження:  {birth:dd MMMM yyyy р.}");
Console.WriteLine($"Дата вступу:      {admission:dd.MM.yyyy HH:mm}");
Console.WriteLine($"Дата виписки:     {discharge:dd.MM.yyyy HH:mm}");
Console.WriteLine($"День тижня вступу: {admission:dddd}");

Console.WriteLine("\n=== Формати для різних цілей ===");
Console.WriteLine($"Для звіту (UA):   {discharge:D}");
Console.WriteLine($"Для бази даних:   {discharge:yyyy-MM-ddTHH:mm:ss}");
Console.WriteLine($"ISO 8601:         {discharge:s}");
Console.WriteLine($"Тільки час:       {admission:HH:mm:ss}");
Console.WriteLine($"Місяць і день:    {admission:d MMMM}");

Console.WriteLine("\n=== Розрахунки ===");
TimeSpan stay = discharge - admission;
Console.WriteLine($"Тривалість: {stay.Days} днів {stay.Hours} годин");

int age = DateTime.Today.Year - birth.Year;
if (birth > DateTime.Today.AddYears(-age)) age--;
Console.WriteLine($"Вік пацієнта: {age} р.");
```

Парсинг та валідація введення — runnable приклад

Безпечний парсинг дат із різними форматами введення:

```C#
using System;
using System.Globalization;

Console.WriteLine("=== TryParse: гнучкий парсинг ===");
string[] inputs = { "11.06.2026", "2026-06-11", "11/06/2026", "abc",
"32.13.2026" };

foreach (var s in inputs)
{
    if (DateTime.TryParse(s, out DateTime dt))
        Console.WriteLine($"  '{s}' -> OK: {dt:dd.MM.yyyy}");
    else
        Console.WriteLine($"  '{s}' -> ПОМИЛКА: некоректна дата");
}

Console.WriteLine("\n=== TryParseExact: точний формат ===");
string[] records = {
    "15.04.1958",
```

```
    "11.06.2026",
    "1958/04/15",   // неправильний формат
    "29.02.2026",   // 2026 не високосний
};

foreach (var s in records)
{
    bool ok = DateTime.TryParseExact(
        s, "dd.MM.yyyy",
        CultureInfo.InvariantCulture,
        DateTimeStyles.None,
        out DateTime dt);

    Console.WriteLine(ok
        ? $"  '{s}' -> {dt:D}"
        : $"  '{s}' -> ПОМИЛКА формату або дата не існує");
}

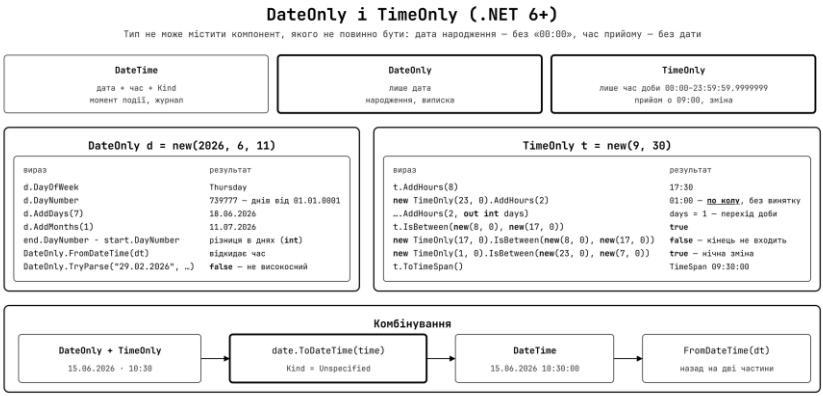
Console.WriteLine("\n=== ParseExact: ISO 8601 з зовнішньої системи ===");
string isoDate = "2026-06-11T09:30:00";
DateTime parsed = DateTime.ParseExact(isoDate, "yyyy-MM-ddTHH:mm:ss",
CultureInfo.InvariantCulture);
Console.WriteLine($"ISO вхід:  {isoDate}");
Console.WriteLine($"Результат: {parsed:F}");
```

12.3. DateOnly та TimeOnly

Починаючи з .NET 6, мова отримала два спеціалізовані типи: DateOnly — лише дата без часу, і TimeOnly — лише час без дати. Обидва є **структурами** (struct) і розташовані в просторі імен System.

Навіщо вони потрібні, якщо є DateTime? Причина принципова: DateTime завжди містить обидва компоненти — і дату, і час. Для ситуацій, де час не має значення, використання DateTime породжує небажані питання: а що означає 00:00:00 у полі «дата народження»? Це північ UTC? Місцевий час? Відсутність часу? При передачі таких значень між серверами різних часових поясів виникають помилки конвертації.

DateOnly і TimeOnly усувають цю двозначність: вони фізично не можуть містити компонент, якого не повинно бути.



DateOnly та TimeOnly — спеціалізовані типи для дати та часу

DateOnly — тільки дата

Структура DateOnly представляє календарну дату без будь-якої прив'язки до часу. Це ідеальний тип для: дати народження пацієнта, дати госпіталізації або виписки (якщо точний час не важливий), дати проведення аналізу, дати складання документа.

Конструктори

```
C#
1  // Рік, місяць, день
2  DateOnly birth  = new DateOnly(1958, 4, 15);
3  DateOnly today  = DateOnly.FromDateTime(DateTime.Today);  // з DateTime
4
5  // Із рядка
6  DateOnly parsed = DateOnly.Parse("11.06.2026");
7
8  // Із порядкового номера дня від 01.01.0001
9  DateOnly fromN  = DateOnly.FromDayNumber(738920);
```

При використанні конструктора без параметрів структура ініціалізується датою 01.01.0001:

```
C#
1  DateOnly empty = new DateOnly();
2  Console.WriteLine(empty); // 01.01.0001
```

Конструктор також приймає об'єкт календаря — System.Globalization.Calendar. Наприклад, розрахунок за юліанським календарем:

```
C#
1  using System.Globalization;
2
3  DateOnly julianDate = new DateOnly(2026, 1, 6, new JulianCalendar());
4  Console.WriteLine(julianDate); // 19.01.2026 — той самий день за григоріанським
```

Це корисно при роботі зі старими медичними архівами або документами, де дати записані за іншою системою числення.

Властивості DateOnly

Властивість	Опис
Year	Рік
Month	Місяць (1-12)
Day	День (1-31)
DayOfWeek	День тижня (DayOfWeek.Monday тощо)
DayOfYear	Порядковий номер дня у році
DayNumber	Кількість днів від 01.01.0001

```
C#
1  DateOnly date = new DateOnly(2026, 6, 11);
2
3  Console.WriteLine(date.Year);       // 2026
4  Console.WriteLine(date.Month);      // 6
5  Console.WriteLine(date.Day);        // 11
6  Console.WriteLine(date.DayOfWeek);  // Thursday
7  Console.WriteLine(date.DayOfYear);  // 162
8  Console.WriteLine(date.DayNumber);  // 739777
```

Методи DateOnly

Арифметика дат:

```
C#
1  DateOnly admission = new DateOnly(2026, 6, 11);
2
3  DateOnly discharge  = admission.AddDays(7);   // 18.06.2026
4  DateOnly nextVisit  = discharge.AddMonths(1); // 18.07.2026
5  DateOnly nextYear   = admission.AddYears(1);  // 11.06.2027
6  DateOnly yesterday  = admission.AddDays(-1);  // 10.06.2026
```

Різниця між датами — через DayNumber:

```C#
DateOnly start = new DateOnly(2026, 6, 1);
DateOnly end   = new DateOnly(2026, 6, 18);
int daysBetween = end.DayNumber - start.DayNumber; // 17
```

Форматування та конвертація:

```C#
DateOnly date = new DateOnly(2026, 6, 11);

Console.WriteLine(date.ToShortDateString()); // 11.06.2026
Console.WriteLine(date.ToLongDateString());  // четвер, 11 червня 2026 р.
Console.WriteLine(date.ToString("yyyy-MM-dd")); // 2026-06-11

// Комбінування з часом → DateTime
TimeOnly time = new TimeOnly(9, 30);
DateTime appointment = date.ToDateTime(time); // 11.06.2026 09:30:00
```

Порівняння:

```C#
DateOnly d1 = new DateOnly(2026, 6, 1);
DateOnly d2 = new DateOnly(2026, 7, 1);

bool earlier = d1 < d2;  // true
bool equal   = d1 == d1; // true
```

TimeOnly — тільки час

Структура TimeOnly представляє час доби в діапазоні від 00:00:00.0000000 до 23:59:59.9999999, без прив'язки до конкретної дати. Це ідеальний тип для: часу прийому пацієнта (понеділок 09:00), робочих годин відділення, розкладу змін медперсоналу.

Важлива відмінність від TimeSpan (розд. 12.4): TimeOnly завжди представляє **час на годиннику** (0 – 23:59), тоді як TimeSpan може бути більшим за 24 години або від'ємним — це **тривалість**, а не час доби.

Конструктори

```C#
// Різна точність
TimeOnly t1 = new TimeOnly(9, 30);         // 09:30:00
TimeOnly t2 = new TimeOnly(14, 45, 20);    // 14:45:20
TimeOnly t3 = new TimeOnly(8, 0, 0, 500);  // 08:00:00.500

// Із DateTime
TimeOnly fromDt = TimeOnly.FromDateTime(DateTime.Now);

// Із рядка
TimeOnly parsed = TimeOnly.Parse("14:30");

```

```
12  // Із TimeSpan
13  TimeOnly fromTs = TimeOnly.FromTimeSpan(new TimeSpan(9, 30, 0));
```

При використанні конструктора без параметрів — 0:00:

```
C#
1  TimeOnly empty = new TimeOnly();
2  Console.WriteLine(empty); // 0:00
```

Властивості TimeOnly

Властивість	Опис
Hour	Година (0-23)
Minute	Хвилини (0-59)
Second	Секунди (0-59)
Millisecond	Мілісекунди (0-999)
Ticks	Тіки від початку дня

```
C#
1  TimeOnly time = new TimeOnly(14, 23, 30);
2
3  Console.WriteLine(time.Hour);   // 14
4  Console.WriteLine(time.Minute); // 23
5  Console.WriteLine(time.Second); // 30
```

Методи TimeOnly

Арифметика:

```
C#
 1  TimeOnly start = new TimeOnly(9, 0);
 2
 3  TimeOnly end  = start.AddHours(8);     // 17:00 (завершення зміни)
 4  TimeOnly mid  = start.AddMinutes(30);  // 09:30
 5
 6  // AddHours з переповненням через 24:00 не кидає виняток — обертається
 7  TimeOnly night = new TimeOnly(23, 0);
 8  TimeOnly next  = night.AddHours(2);    // 01:00 (наступний день)
 9
10  // Скільки разів перейшли через північ — через out-параметр
11  TimeOnly next2 = night.AddHours(2, out int wrappedDays); // 01:00, wrappedDays
    = 1
```

IsBetween — перевірка, чи потрапляє час у проміжок. Корисно для перевірки, чи прийом проводиться в межах робочого часу:

```
C#
1  TimeOnly workStart = new TimeOnly(8, 0);
2  TimeOnly workEnd   = new TimeOnly(17, 0);
3  TimeOnly appoint   = new TimeOnly(10, 30);
4
5  bool inWork = appoint.IsBetween(workStart, workEnd); // true
```

Інтервал у IsBetween **півідкритий**: початок входить, кінець — ні. new TimeOnly(17, 0).IsBetween(workStart, workEnd) дає false, тож сусідні зміни 07:00–15:00 і 15:00–23:00 не перетинаються — рівно 15:00 належить лише другій. Якщо початок більший за кінець, інтервал вважається таким, що проходить через північ: new TimeOnly(1, 0).IsBetween(new TimeOnly(23, 0), new TimeOnly(7, 0)) — true (нічна зміна).

Конвертація:

```
C#
1  TimeOnly time = new TimeOnly(14, 30, 0);
2
3  TimeSpan ts = time.ToTimeSpan(); // 14:30:00 як TimeSpan
4  Console.WriteLine(time.ToShortTimeString()); // 14:30
5  Console.WriteLine(time.ToLongTimeString());  // 14:30:00
```

Комбінування DateOnly і TimeOnly

Найпоширеніший сценарій — призначення прийому: є дата і є час, потрібно отримати повний DateTime:

```
C#
1  DateOnly date = new DateOnly(2026, 6, 15);   // дата прийому
2  TimeOnly time = new TimeOnly(10, 30);         // час прийому
3
4  DateTime appointment = date.ToDateTime(time); // 15.06.2026 10:30:00
5  Console.WriteLine(appointment);
```

Зворотня операція — розкладання DateTime на компоненти:

```
C#
1  DateTime dt = new DateTime(2026, 6, 15, 10, 30, 0);
2
3  DateOnly d = DateOnly.FromDateTime(dt); // 15.06.2026
4  TimeOnly t = TimeOnly.FromDateTime(dt); // 10:30:00
```

Цей підхід особливо зручний при зберіганні розкладу в базі даних: дата і час прийому зберігаються в окремих колонках (типи SQL date і time), і при читанні легко об'єднуються назад.

Розклад прийомів та дати народження — runnable приклад

```C#
using System;

Console.WriteLine("=== Список пацієнтів — дата народження ===");
var patients = new[]
{
    ("Петренко Іван",   new DateOnly(1958, 4, 15)),
    ("Коваль Марія",    new DateOnly(1978, 11, 3)),
    ("Сидоренко Олег",  new DateOnly(1951, 7, 22)),
};

DateOnly today = DateOnly.FromDateTime(DateTime.Today);
Console.WriteLine($"{"Пацієнт",-22} {"Народження",12} {"Вік",5}");
Console.WriteLine(new string('-', 42));

foreach (var (name, birth) in patients)
{
    int age = today.Year - birth.Year;
    if (birth > today.AddYears(-age)) age--;
    Console.WriteLine($"{name,-22} {birth,12:dd.MM.yyyy} {age,5}");
}

Console.WriteLine("\n=== Розклад прийомів — комбінування DateOnly + TimeOnly ===");
var schedule = new[]
{
    (new DateOnly(2026, 6, 15), new TimeOnly(9,  0)),
    (new DateOnly(2026, 6, 15), new TimeOnly(10, 30)),
    (new DateOnly(2026, 6, 16), new TimeOnly(14, 0)),
};

TimeOnly workStart = new TimeOnly(8, 0);
TimeOnly workEnd   = new TimeOnly(17, 0);

Console.WriteLine($"{"Дата",-14}{"Час",-8}{"У робочий час"}");
Console.WriteLine(new string('-', 35));
foreach (var (d, t) in schedule)
{
    bool inWork = t.IsBetween(workStart, workEnd);
    DateTime appoint = d.ToDateTime(t);
    Console.WriteLine($"{d,-14:dd.MM.yyyy}{t,-8:HH:mm}{(inWork ? "Так" : "Ні")}");
}
```

Парсинг і статичні методи — runnable приклад

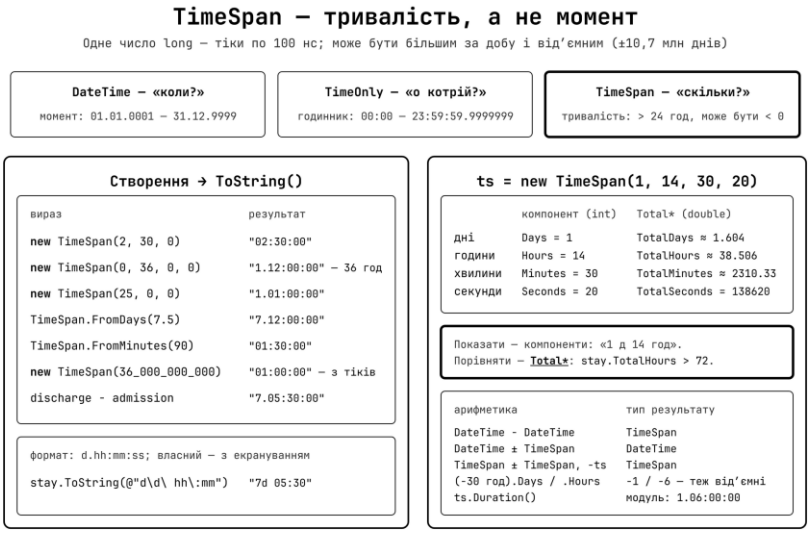
```C#
using System;
using System.Globalization;
```

```
 3
 4  Console.WriteLine("=== DateOnly — Parse та статичні методи ===");
 5  DateOnly d1 = DateOnly.Parse("11.06.2026");
 6  Console.WriteLine($"Parse: {d1}");
 7  Console.WriteLine($"DayNumber: {d1.DayNumber}");
 8  Console.WriteLine($"FromDayNumber: {DateOnly.FromDayNumber(d1.DayNumber)}");
 9
10  DateOnly d2 = DateOnly.ParseExact("2026-06-11", "yyyy-MM-dd",
    CultureInfo.InvariantCulture);
11  Console.WriteLine($"ParseExact ISO: {d2:dd.MM.yyyy}");
12
13  bool ok = DateOnly.TryParse("29.02.2026", out DateOnly d3); // 2026 не
    високосний
14  Console.WriteLine($"TryParse 29.02.2026: ok={ok}");
15
16  Console.WriteLine("\n=== TimeOnly — IsBetween і AddHours ===");
17  TimeOnly shift1Start = new TimeOnly(7, 0);
18  TimeOnly shift1End   = new TimeOnly(15, 0);
19  TimeOnly shift2Start = new TimeOnly(15, 0);
20  TimeOnly shift2End   = new TimeOnly(23, 0);
21
22  TimeOnly[] calls = { new TimeOnly(9, 0), new TimeOnly(14, 30), new TimeOnly(18,
    0), new TimeOnly(6, 0) };
23  foreach (var call in calls)
24  {
25      string shift = call.IsBetween(shift1Start, shift1End) ? "Зміна 1 (07-15)" :
26                     call.IsBetween(shift2Start, shift2End) ? "Зміна 2 (15-23)" :
    "Поза змінами";
27      Console.WriteLine($"  {call:HH:mm} -> {shift}");
28  }
29
30  Console.WriteLine("\n=== ToDateTime: об'єднання DateOnly + TimeOnly ===");
31  DateOnly date = new DateOnly(2026, 6, 20);
32  TimeOnly time = new TimeOnly(11, 15);
33  DateTime dt   = date.ToDateTime(time);
34  Console.WriteLine($"DateOnly: {date}  TimeOnly: {time}  -> DateTime:
    {dt:dd.MM.yyyy HH:mm}");
```

12.4. Структура TimeSpan

Якщо DateTime відповідає на питання «**коли?**» (конкретна точка в часі), то TimeSpan відповідає на питання «**скільки?**» (тривалість інтервалу). Наприклад: скільки тривала госпіталізація? Через скільки годин призначити повторний прийом? Яка різниця між часом надходження пацієнта і виконанням аналізу?

TimeSpan — це структура (value type) з простору імен System, що внутрішньо зберігає одне long-значення — кількість **тіків** (тих самих 100-наносекундних тіків, що й DateTime). Тому TimeSpan може представляти тривалість від мінусових тисячоліть до позитивних — або, точніше, від TimeSpan.MinValue (близько -10 мільйонів днів) до TimeSpan.MaxValue (близько +10 мільйонів днів).



TimeSpan — представлення тривалості часового інтервалу

TimeSpan vs DateTime vs TimeOnly

Ці три типи часто плутають. Принципова різниця:

Тип	Що представляє	Діапазон	Може бути від'ємним?
DateTime	Момент в часі	01.01.0001 – 31.12.9999	Ні
TimeOnly	Час на годиннику	00:00:00 – 23:59:59	Ні
TimeSpan	Тривалість	±10 млн днів	Так

TimeSpan є результатом **різниці** двох DateTime: dt1 - dt2 = TimeSpan. Але він може також виражати довільну тривалість: «через 72 години», «за 21 день», «відняти 30 хвилин».

Створення TimeSpan

Є кілька способів створити TimeSpan:

```
C#
 1  // 1. Конструктори
 2  var ts1 = new TimeSpan(2, 30, 0);          // 2 год 30 хв 0 сек = 02:30:00
 3  var ts2 = new TimeSpan(7, 0, 0, 0);        // 7 днів
 4  var ts3 = new TimeSpan(0, 36, 0, 0);       // 36 годин (= 1 день 12 год)
 5  var ts4 = new TimeSpan(1, 14, 30, 20, 500); // 1 дн 14 год 30 хв 20 сек 500 мс
 6
 7  // 2. Статичні методи From*
 8  TimeSpan week   = TimeSpan.FromDays(7);       // 7 днів (7.5 = 7 днів 12 год)
 9  TimeSpan day    = TimeSpan.FromHours(24);     // 1 день
10  TimeSpan hour   = TimeSpan.FromMinutes(60);   // 1 година
11  TimeSpan minute = TimeSpan.FromSeconds(60);   // 1 хвилина
12  TimeSpan zero   = TimeSpan.Zero;              // 00:00:00
13
14  // 3. З тіків
15  TimeSpan fromTicks = new TimeSpan(36_000_000_000L); // 1 година = 10^7
    тіків/сек * 3600
16
17  // 4. Різниця двох DateTime (найчастіший спосіб)
18  DateTime admission = new DateTime(2026, 6, 1, 9, 0, 0);
19  DateTime discharge = new DateTime(2026, 6, 8, 14, 30, 0);
20  TimeSpan stay = discharge - admission; // 7 днів 5 год 30 хв
```

*Властивості TimeSpan: компоненти та Total**

Клас TimeSpan має два набори властивостей, які легко переплутати:

Компонентні властивості — цілочисельна частина відповідного компонента:

```
C#
1  TimeSpan ts = new TimeSpan(1, 14, 30, 20); // 1 день 14 год 30 хв 20 сек
2
3  Console.WriteLine(ts.Days);         // 1   — тільки дні
4  Console.WriteLine(ts.Hours);        // 14  — тільки години (0-23)
5  Console.WriteLine(ts.Minutes);      // 30  — тільки хвилини (0-59)
6  Console.WriteLine(ts.Seconds);      // 20  — тільки секунди (0-59)
7  Console.WriteLine(ts.Milliseconds); // 0
```

Total-властивості — загальна тривалість, виражена у відповідній одиниці як double:

```
C#
1  Console.WriteLine(ts.TotalDays);    // 1.604...  — усі дні включно із годинами
2  Console.WriteLine(ts.TotalHours);   // 38.50...  — усього годин (1*24 + 14 = 38)
3  Console.WriteLine(ts.TotalMinutes); // 2310.333  — усього хвилин
4  Console.WriteLine(ts.TotalSeconds); // 138620.0  — усього секунд
```

Правило: для відображення («госпіталізація тривала 7 днів і 5 годин») — використовуйте компонентні властивості Days + Hours; для порівняння або арифметики («чи пройшло більше 72 годин?») — TotalHours:

```
C#
1  if (stay.TotalHours > 72)
2      Console.WriteLine("Пацієнт перебуває понад 3 дні");
```

Арифметика TimeSpan

TimeSpan підтримує такі операції:

```
C#
 1  TimeSpan t1 = TimeSpan.FromDays(3);
 2  TimeSpan t2 = TimeSpan.FromHours(12);
 3
 4  TimeSpan sum  = t1 + t2;           // 3 дні 12 годин
 5  TimeSpan diff = t1 - t2;           // 2 дні 12 годин
 6  TimeSpan neg  = -t1;               // -3 дні (від'ємний)
 7  TimeSpan abs  = t1.Duration();     // завжди невід'ємний (як Math.Abs)
 8
 9  // DateTime + TimeSpan → DateTime
10  DateTime admission  = new DateTime(2026, 6, 1, 9, 0, 0);
11  DateTime scheduledD = admission + TimeSpan.FromDays(21); // плановий повторний
    огляд
12  DateTime reminderT  = scheduledD - TimeSpan.FromHours(24); // нагадування за
    добу
13
14  // DateTime - DateTime → TimeSpan
15  TimeSpan elapsed    = DateTime.Now - admission; // скільки часу пройшло
```

Від'ємний TimeSpan виникає, коли перша дата пізніша за другу: dt1 - dt2 де dt1 < dt2. Для отримання абсолютного значення — метод Duration() або статичний TimeSpan.FromTicks(Math.Abs(ts.Ticks)).

Властивість Duration

ts.Duration() завжди повертає невід'ємний TimeSpan — аналог Math.Abs() для тривалості:

```
C#
1  DateTime d1 = new DateTime(2026, 6, 1);
2  DateTime d2 = new DateTime(2026, 6, 10);
3
4  TimeSpan diff1 = d2 - d1;          // +9 днів
5  TimeSpan diff2 = d1 - d2;          // -9 днів
6
7  Console.WriteLine(diff2.Duration()); // 9.00:00:00 (без знаку)
```

Це зручно, коли порядок дат невідомий заздалегідь — наприклад, при обчисленні відстані між двома датами в медичній статистиці без знання, яка з них раніша.

Форматування TimeSpan

TimeSpan можна форматувати через ToString() або $"...":

```
C#
1  TimeSpan stay = new TimeSpan(7, 5, 30, 0);
2
3  Console.WriteLine(stay);              // 7.05:30:00  — формат за замовчуванням
4  Console.WriteLine(stay.ToString(@"d\d\ hh\:mm")); // 7d 05:30
5  Console.WriteLine($"{(int)stay.TotalDays} днів {stay.Hours} годин"); // 7 днів 5
   годин
```

Формат за замовчуванням d.hh:mm:ss — 7.05:30:00 — де перша цифра є кількістю днів. Для виводу в людиночитному вигляді зазвичай будують рядок вручну через компонентні властивості.

Медична картка: тривалість госпіталізації — runnable приклад

```
C#
 1  using System;
 2
 3  Console.WriteLine("=== Тривалість госпіталізації ===");
 4  Console.WriteLine($"{"Пацієнт",-22} {"Вступ",12} {"Виписка",12}
    {"Тривалість"}");
 5  Console.WriteLine(new string('-', 65));
 6
 7  var records = new[]
 8  {
 9      ("Петренко Іван",    new DateTime(2026,6,1,  9,  0, 0), new
    DateTime(2026,6, 8, 14, 30, 0)),
10      ("Коваль Марія",     new DateTime(2026,6,3,  11, 30, 0), new
    DateTime(2026,6,10, 10,  0, 0)),
11      ("Сидоренко Олег",   new DateTime(2026,6,10, 8,  0, 0), new
    DateTime(2026,6,15, 16,  0, 0)),
12  };
13
14  TimeSpan total = TimeSpan.Zero;
15  foreach (var (name, adm, dis) in records)
16  {
17      TimeSpan stay = dis - adm;
18      total += stay;
19      string duration = $"{stay.Days} д. {stay.Hours} год.";
20      Console.WriteLine($"{name,-22} {adm,12:dd.MM.yy} {dis,12:dd.MM.yy}
    {duration}");
21  }
22
23  Console.WriteLine(new string('-', 65));
24  double avgDays = total.TotalDays / records.Length;
25  Console.WriteLine($"Середня тривалість: {avgDays:F1} днів");
26
27  Console.WriteLine("\n=== Перевірка тривалості лікування ===");
28  var (_, admission, _) = records[0];
29  DateTime now = new DateTime(2026, 6, 5, 12, 0, 0);
30  TimeSpan elapsed = now - admission;
31
```

```
Console.WriteLine($"Пацієнт Петренко: у лікарні {elapsed.Days} дн. {elapsed.Hours} год.");
if (elapsed.TotalHours > 72)
    Console.WriteLine("  → Пройшло більше 3 діб. Потрібен повторний аналіз крові.");
```

Планування лікування та цикли — runnable приклад

Розрахунок дат повторних прийомів і перевірка відповідності розкладу:

```C#
using System;

DateTime firstVisit = new DateTime(2026, 6, 11, 10, 30, 0);
TimeSpan treatmentCycle = TimeSpan.FromDays(21); // цикл лікування — 21 день
int totalCycles = 6;

Console.WriteLine("=== Розклад хіміотерапії ===");
Console.WriteLine($"Перший прийом: {firstVisit:dd.MM.yyyy HH:mm}");
Console.WriteLine($"Цикл:          {(int)treatmentCycle.TotalDays} днів");
Console.WriteLine("");

Console.WriteLine($"{"Цикл",-6} {"Дата",12} {"День тижня",-12}");
Console.WriteLine(new string('-', 32));

DateTime current = firstVisit;
for (int i = 1; i <= totalCycles; i++)
{
    Console.WriteLine($"{i,-6} {current,12:dd.MM.yyyy} {current.DayOfWeek,-12}");
    current += treatmentCycle;
}

Console.WriteLine("\n=== Оцінка часу з останнього візиту ===");
DateTime lastVisit = new DateTime(2026, 5, 21, 10, 30, 0);
DateTime today     = new DateTime(2026, 6, 11, 9,  0, 0);

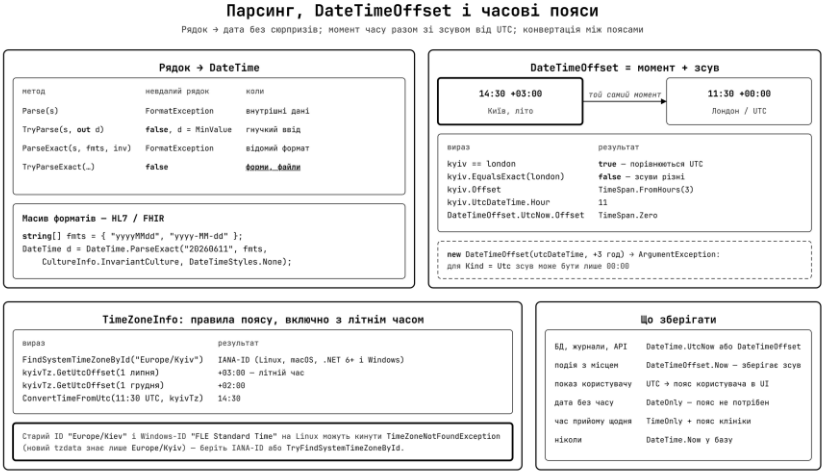
TimeSpan sinceVisit = today - lastVisit;
Console.WriteLine($"Минуло: {sinceVisit.Days} днів {sinceVisit.Hours} год. ({sinceVisit.TotalHours:F0} год. усього)");

TimeSpan expectedInterval = TimeSpan.FromDays(21);
TimeSpan diff = sinceVisit - expectedInterval;

if (diff.Duration().TotalHours < 48)
    Console.WriteLine("  → Пацієнт прийшов вчасно (±2 дні від плану)");
else if (diff.TotalHours > 0)
    Console.WriteLine($"  → Запізнення на {(int)diff.TotalDays} днів");
else
    Console.WriteLine($"  → Прийшов на {(int)diff.Duration().TotalDays} днів раніше");
```

12.5. DateTimeOffset та часові пояси

Медичні інформаційні системи дедалі частіше є розподіленими: сервери можуть знаходитися в різних часових поясах, пацієнти проходять лікування в різних країнах, телемедичні консультації перетинають кордони. У такому середовищі DateTime вже недостатньо — він не містить інформацію про те, до якого поясу прив'язаний час. Для цього призначені DateTimeOffset і TimeZoneInfo.



Парсинг DateTime та DateTimeOffset — безпечне перетворення рядків

Типи парсингу DateTime

Перетворення рядка у DateTime — одна з найчастіших операцій при читанні даних з форм, файлів, API. Є чотири основних методи, що відрізняються гнучкістю і безпекою:

DateTime.Parse — гнучко, але небезпечно

Parse розпізнає багато поширених форматів, але залежить від поточної культури і кидає FormatException якщо рядок некоректний. Підходить для контрольованих внутрішніх сценаріїв, але **не для вводу від користувача**:

```
C#
1  DateTime dt = DateTime.Parse("11.06.2026 14:45"); // OK для uk-UA
2  // DateTime.Parse("abc") → FormatException !!!
```

DateTime.TryParse — безпечний варіант

TryParse ніколи не кидає виняток — при невдачі повертає false, а out-параметр отримує DateTime.MinValue. Рекомендований для обробки будь-якого введення від користувача:

```
C#
1  string input = "11.06.2026";
2  if (DateTime.TryParse(input, out DateTime result))
3      Console.WriteLine($"OK: {result:dd.MM.yyyy}");
4  else
5      Console.WriteLine("Некоректна дата");
```

DateTime.ParseExact — точний формат

ParseExact вимагає точного відповідного формату. Якщо рядок не відповідає — FormatException. Використовується коли формат гарантований (дані з відомої зовнішньої системи):

```
C#
1  using System.Globalization;
2
3  // Дата з HL7-повідомлення: формат YYYYMMDD
4  DateTime dt = DateTime.ParseExact("20260611", "yyyyMMdd",
   CultureInfo.InvariantCulture);
5  Console.WriteLine(dt); // 11.06.2026 0:00:00
```

Важливо: другий параметр може бути **масивом форматів** — тоді метод перебирає їх по черзі:

```
C#
1  string[] formats = { "dd.MM.yyyy", "yyyy-MM-dd", "dd/MM/yyyy" };
2  DateTime dt = DateTime.ParseExact("2026-06-11", formats,
   CultureInfo.InvariantCulture,
3                                    DateTimeStyles.None);
```

DateTime.TryParseExact — найбезпечніший

Поєднує точний формат з безпечним поверненням bool. Найрекомендованіший для медичних систем:

```
C#
 1  using System.Globalization;
 2
 3  bool ok = DateTime.TryParseExact(
 4      "11.06.2026 14:30",
 5      "dd.MM.yyyy HH:mm",
 6      CultureInfo.InvariantCulture,
 7      DateTimeStyles.None,
 8      out DateTime admission);
 9
10  if (ok)
```

```
11      Console.WriteLine($"Прийом: {admission:F}");
12  else
13      Console.WriteLine("Некоректний формат дати/часу");
```

DateTimeOffset — час із зсувом від UTC

DateTimeOffset — це структура, яка зберігає **DateTime разом із зсувом від UTC** (TimeSpan offset). Це дозволяє точно представляти момент часу незалежно від часового поясу сервера, де виконується код.

Порівняння з DateTime:

	DateTime	**DateTimeOffset**
Зберігає	Дата + час	Дата + час + offset від UTC
Kind	Unspecified / Local / Utc	Завжди несе offset
Рівність ==	Порівнює значення (без offset)	Порівнює UTC-еквіваленти
Для UI	Зручно	Зручно
Для БД / API	Ризик помилок часового поясу	Рекомендовано

Створення DateTimeOffset

```
C#
 1  using System;
 2
 3  // Поточний час зі зсувом Local
 4  DateTimeOffset now = DateTimeOffset.Now;        // локальний час + зсув поясу
    сервера
 5  DateTimeOffset utc = DateTimeOffset.UtcNow;     // UTC, offset = +00:00
 6
 7  // Явно задати: час 14:30 зі зсувом +03:00 (Київ, літо)
 8  DateTimeOffset appointment = new DateTimeOffset(
 9      2026, 6, 11, 14, 30, 0,
10      TimeSpan.FromHours(3)); // літній час UTC+3
11
12  // Із DateTime + offset
13  DateTime dt  = new DateTime(2026, 6, 11, 14, 30, 0);
14  TimeSpan off = TimeSpan.FromHours(3);
15  DateTimeOffset dto = new DateTimeOffset(dt, off);
16
17  Console.WriteLine(appointment); // 11.06.2026 14:30:00 +03:00
```

У DateTimeOffset немає властивості Kind — замість неї є явний зсув Offset. Конструктор new DateTimeOffset(dt, offset) перевіряє узгодженість: якщо dt.Kind == DateTimeKind.Utc, зсув мусить бути нульовим, а для Local — дорівнювати зсуву поясу сервера; інакше — ArgumentException.

Властивості DateTimeOffset

Властивість	Опис
Offset	Зсув від UTC як TimeSpan
UtcDateTime	Відповідний DateTime в UTC
LocalDateTime	Відповідний DateTime в локальному часі
DateTime	DateTime без offset-мітки (Kind=Unspecified)
UtcTicks	Тіки в UTC

```
C#
1  DateTimeOffset dto = new DateTimeOffset(2026, 6, 11, 14, 30, 0,
   TimeSpan.FromHours(3));
2
3  Console.WriteLine(dto.Offset);        // +03:00
4  Console.WriteLine(dto.UtcDateTime);   // 11.06.2026 11:30:00
5  Console.WriteLine(dto.LocalDateTime); // залежить від ОС
```

DateTimeOffset та рівність

Ключова відмінність від DateTime: два DateTimeOffset **рівні**, якщо вони представляють один і той самий момент у UTC, навіть якщо зберігають різні зсуви:

```
C#
1  DateTimeOffset kyiv   = new DateTimeOffset(2026, 6, 11, 14, 30, 0,
   TimeSpan.FromHours(3));
2  DateTimeOffset london = new DateTimeOffset(2026, 6, 11, 11, 30, 0,
   TimeSpan.FromHours(0));
3
4  Console.WriteLine(kyiv == london); // true — той самий момент у UTC
5  Console.WriteLine(kyiv.EqualsExact(london)); // false — момент той самий, але
   зсуви різні
```

Якщо ж потрібно, щоб збігалися і момент, і зсув (наприклад, перевірка, що запис не «переїхав» в інший пояс), використовують метод EqualsExact.

Саме це робить DateTimeOffset надійним для зберігання в базі даних: порівняння дат між записами з різних серверів дає коректний результат.

TimeZoneInfo — робота з часовими поясами

Клас TimeZoneInfo дозволяє знаходити часові пояси і конвертувати між ними. Він розташований у System і не потребує додаткових просторів імен.

```
C#
 1  // Поточний часовий пояс ОС
 2  TimeZoneInfo local = TimeZoneInfo.Local;
 3  Console.WriteLine(local.DisplayName); // (UTC+02:00) Helsinki, Kyiv, Riga,
    Sofia...
 4
 5  // UTC
 6  TimeZoneInfo utcZone = TimeZoneInfo.Utc;
 7
 8  // Конкретний пояс за ID (крос-платформений ID IANA або Windows ID)
 9  TimeZoneInfo kyivTz = TimeZoneInfo.FindSystemTimeZoneById("Europe/Kyiv"); //
    IANA
10  // Windows-ID: "FLE Standard Time"
```

Ідентифікатори поясів бувають двох видів: Windows-ID ("FLE Standard Time") і IANA-ID ("Europe/Kyiv"). Починаючи з .NET 6, обидва види працюють на всіх платформах: .NET сам перетворює один на інший за допомогою даних ICU. Але є нюанс із Києвом: 2022 року база IANA перейменувала Europe/Kiev на Europe/Kyiv, а сучасні дистрибутиви Linux (наприклад, Ubuntu 24.04) постачають старі назви лише в окремому пакеті tzdata-legacy. Без нього і "Europe/Kiev", і "FLE Standard Time" (який відображається саме на стару назву) кидають TimeZoneNotFoundException. Надійний вибір — "Europe/Kyiv", а для коду, що має працювати будь-де, — TimeZoneInfo.TryFindSystemTimeZoneById (.NET 8+) із запасним варіантом.

Конвертація між поясами

```
C#
 1  DateTime utcTime = new DateTime(2026, 6, 11, 11, 30, 0, DateTimeKind.Utc);
 2
 3  TimeZoneInfo kyivTz   = TimeZoneInfo.FindSystemTimeZoneById("Europe/Kyiv");
 4  TimeZoneInfo berlinTz = TimeZoneInfo.FindSystemTimeZoneById("Europe/Berlin");
```

```
 5
 6  DateTime kyivTime   = TimeZoneInfo.ConvertTimeFromUtc(utcTime, kyivTz);
 7  DateTime berlinTime = TimeZoneInfo.ConvertTimeFromUtc(utcTime, berlinTz);
 8
 9  Console.WriteLine($"UTC:    {utcTime:HH:mm}");   // 11:30
10  Console.WriteLine($"Київ:   {kyivTime:HH:mm}");  // 14:30 (UTC+3)
11  Console.WriteLine($"Берлін: {berlinTime:HH:mm}"); // 13:30 (UTC+2)
```

Перехід на літній/зимовий час

TimeZoneInfo автоматично враховує перехід на літній і зимовий час (DST — Daylight Saving Time). Метод IsDaylightSavingTime(dt) дозволяє перевірити, чи діє в даний момент літній час:

```
C#
1  TimeZoneInfo kyivTz = TimeZoneInfo.FindSystemTimeZoneById("Europe/Kyiv");
2
3  DateTime summer = new DateTime(2026, 7, 1);   // літо — UTC+3
4  DateTime winter = new DateTime(2026, 12, 1);  // зима — UTC+2
5
6  Console.WriteLine($"Літо DST:  {kyivTz.IsDaylightSavingTime(summer)}"); // True
7  Console.WriteLine($"Зима DST:  {kyivTz.IsDaylightSavingTime(winter)}"); // False
8  Console.WriteLine($"Літо offset: {kyivTz.GetUtcOffset(summer)}"); // +03:00
9  Console.WriteLine($"Зима offset: {kyivTz.GetUtcOffset(winter)}"); // +02:00
```

Рекомендації для медичних систем

Для медичних інформаційних систем, де точність і відтворюваність часових відміток критичні, рекомендуються такі практики:

1. **Зберігати у БД UTC або DateTimeOffset.UtcNow** — ніколи DateTime.Now, оскільки локальний час залежить від сервера і може дати неправильний порядок подій при міграції або масштабуванні.
2. **Для аудит-журналів** (коли пацієнт надійшов, коли прийнятий лікар) — DateTimeOffset. Якщо важливий лише момент — DateTimeOffset.UtcNow (зсув завжди +00:00); якщо потрібно знати ще й місцевий час події — DateTimeOffset.Now, який зберігає зсув поясу, де подію зафіксовано.
3. **Для парсингу вводу від користувача** — завжди TryParseExact з явно вказаним форматом і CultureInfo.InvariantCulture.
4. **Для відображення в UI** — конвертувати UTC у локальний час TimeZoneInfo.ConvertTimeFromUtc безпосередньо перед відображенням.

Парсинг медичних форм — runnable приклад

```C#
using System;
using System.Globalization;

Console.WriteLine("=== Парсинг дат із різних джерел ===");

// 1. Форма введення пацієнта
string[] userInputs = { "15.04.1958", "abc", "32.13.2026", "2026-06-11" };
Console.WriteLine("Введення з форми (TryParseExact, dd.MM.yyyy):");
foreach (var s in userInputs)
{
    bool ok = DateTime.TryParseExact(s, "dd.MM.yyyy",
        CultureInfo.InvariantCulture, DateTimeStyles.None, out DateTime dt);
    Console.WriteLine($"  '{s}' -> {(ok ? dt.ToString("dd.MM.yyyy") : "ПОМИЛКА")}");
}

// 2. HL7 FHIR формат: YYYYMMDD або YYYY-MM-DD
string[] fhirDates = { "20260611", "2026-06-11" };
string[] fhirFormats = { "yyyyMMdd", "yyyy-MM-dd" };
Console.WriteLine("\nFHIR формати (ParseExact із масивом):");
foreach (var s in fhirDates)
{
    DateTime dt = DateTime.ParseExact(s, fhirFormats,
        CultureInfo.InvariantCulture, DateTimeStyles.None);
    Console.WriteLine($"  '{s}' -> {dt:dd.MM.yyyy}");
}

// 3. ISO 8601 з часовим поясом
string iso = "2026-06-11T14:30:00+03:00";
DateTimeOffset dto = DateTimeOffset.Parse(iso, CultureInfo.InvariantCulture);
Console.WriteLine($"\nISO 8601: {iso}");
Console.WriteLine($"  UTC:   {dto.UtcDateTime:dd.MM.yyyy HH:mm}");
Console.WriteLine($"  Local: {dto.LocalDateTime:dd.MM.yyyy HH:mm}");
Console.WriteLine($"  Offset: {dto.Offset}");
```

DateTimeOffset і UTC для аудит-журналу — runnable приклад

```C#
using System;

Console.WriteLine("=== Аудит-журнал медичних подій (DateTimeOffset UTC) ===");

var kyivOffset   = TimeSpan.FromHours(3); // Київ, літній час
var berlinOffset = TimeSpan.FromHours(2); // Берлін, літній час

string[] eventLabels = { "Прийом пацієнта", "Забір аналізу", "Теле-консультація" };
DateTimeOffset[] eventTimes = {
    new DateTimeOffset(2026, 6, 11, 9,  0,  0, kyivOffset),   // Київ
    new DateTimeOffset(2026, 6, 11, 10, 30, 0, kyivOffset),   // Київ
    new DateTimeOffset(2026, 6, 11, 9,  45, 0, berlinOffset), // Берлін
};

Console.WriteLine($"{"Подія",-22} {"Місцевий",10} {"UTC",8}  Offset");
```

```
Console.WriteLine(new string('-', 55));
for (int i = 0; i < eventTimes.Length; i++)
{
    DateTimeOffset ev = eventTimes[i];
    Console.WriteLine($"{eventLabels[i],-22} {ev.DateTime,10:HH:mm}
{ev.UtcDateTime,8:HH:mm}  {ev.ToString("zzz")}");
}

Console.WriteLine("\n=== UTC-момент: порівняння між поясами ===");
// Хто прийшов раніше: Київ 09:00+03 чи Берлін 09:45+02?
DateTimeOffset kyivEvent   = eventTimes[0]; // 09:00 Київ = 06:00 UTC
DateTimeOffset berlinEvent = eventTimes[2]; // 09:45 Берлін = 07:45 UTC
bool kyivFirst = kyivEvent < berlinEvent;
Console.WriteLine($"Київ    09:00+03 → UTC {kyivEvent.UtcDateTime:HH:mm}");
Console.WriteLine($"Берлін  09:45+02 → UTC {berlinEvent.UtcDateTime:HH:mm}");
Console.WriteLine($"Київ раніше: {kyivFirst}");

Console.WriteLine("\n=== DateTimeOffset.UtcNow (рекомендовано для БД) ===");
DateTimeOffset stamp = DateTimeOffset.UtcNow;
Console.WriteLine($"Timestamp UTC: {stamp:yyyy-MM-ddTHH:mm:ssZ}");
Console.WriteLine($"Offset:        {stamp.Offset}");
```

Питання для самоконтролю — Розділ 12. Робота з датами та часом

12.1. Структура DateTime

1. Що таке DateTime з точки зору системи типів .NET — клас чи структура? Які наслідки це має для присвоєння та порівняння значень?
2. Поясніть, що таке «тік» у внутрішньому представленні DateTime. Від якої точки відраховуються тіки і яку одиницю часу вони представляють?
3. В чому різниця між DateTimeKind.Unspecified, DateTimeKind.Local та DateTimeKind.Utc? Чому правильний вибір Kind критично важливий у медичних інформаційних системах?
4. Що виведе наступний код? Поясніть, чому результат може бути несподіваним:

```
C#
DateTime local = DateTime.Now;
DateTime utc   = DateTime.UtcNow;
Console.WriteLine(local > utc);
```

5. Напишіть код, який обчислює точний вік людини в роках на заданий момент часу, використовуючи DateTime. Врахуйте, що день народження може ще не настати в поточному році.
6. Що повертає DateTime.Today, і чим цей результат відрізняється від DateTime.Now? У яких сценаріях краще використовувати Today?
7. Метод AddDays(n) повертає новий DateTime чи змінює поточний? Як відняти 2 тижні від дати прийому?
8. Порівняйте два підходи до обчислення тривалості перебування пацієнта у лікарні:

- TimeSpan stay = discharge - admission; - TimeSpan stay = discharge.Subtract(admission); Яка між ними різниця?

12.2. Налаштування формату часу та дати

9. Що таке «стандартні специфікатори форматування» у DateTime.ToString()? Чим вони відрізняються від «кастомних» специфікаторів?
10. Поясніть різницю між специфікаторами "d" і "D", між "t" і "T"`. Чому важлива різниця між малою та великою літерою?
11. Який специфікатор форматування треба використати, щоб отримати рядок у форматі ISO 8601 без часового поясу (наприклад, 2026-06-11T14:45:20)? Чому ISO 8601 особливо важливий для медичних систем?
12. Що таке CultureInfo.InvariantCulture і в яких ситуаціях його треба обов'язково вказувати при форматуванні дати?
13. Знайдіть помилку в наступному коді та поясніть, чому вона виникає:

```
C#
1  string input = "15.04.1958";
2  DateTime birth = DateTime.Parse(input);
```

14. В чому різниця між DateTime.Parse, DateTime.TryParse, DateTime.ParseExact і DateTime.TryParseExact? Коли слід використовувати кожен із них?
15. Напишіть код, який безпечно зчитує дату народження пацієнта з рядка формату "dd.MM.yyyy" та виводить повідомлення про помилку, якщо рядок некоректний або дата не існує (наприклад, "29.02.2025").

16. Що виведе наступний код і чому?

```C#
DateTime dt = new DateTime(2026, 6, 11, 14, 45, 20, 123);
Console.WriteLine(dt.ToString("HH:mm:ss.fff"));
Console.WriteLine(dt.ToString("hh:mm tt"));
```

12.3. DateOnly та TimeOnly

17. Навіщо в .NET 6 було введено DateOnly і TimeOnly, якщо вже існував DateTime? Яку конкретну проблему вони вирішують?
18. В чому принципова різниця між TimeOnly та TimeSpan? Чому не можна замінити один тип іншим у всіх сценаріях?
19. Як обчислити різницю між двома датами типу DateOnly? Чому у DateOnly немає оператора -, що повертає TimeSpan?
20. Що виведе наступний код?

```C#
TimeOnly night = new TimeOnly(23, 0);
TimeOnly next  = night.AddHours(2);
Console.WriteLine(next);
```

Поясніть поведінку при «переповненні через 24:00».

21. Як комбінувати DateOnly та TimeOnly для отримання DateTime? Напишіть код, який створює DateTime призначення на конкретну дату та час.
22. Напишіть код, який перевіряє, чи потрапляє час прийому пацієнта у робочий час відділення (08:00–17:00), використовуючи TimeOnly.IsBetween.
23. Порівняйте DateOnly.Parse("11.06.2026") та DateTime.Parse("11.06.2026"). Яка різниця у результаті? Де є ризик проблем з часовими поясами?
24. У системі бронювання прийомів дату і час зберігають окремо. Поясніть, чому це архітектурне рішення правильне і яку перевагу дає використання DateOnly + TimeOnly замість DateTime.

12.4. Структура TimeSpan

25. Що представляє TimeSpan — момент у часі чи тривалість? Як він відрізняється від DateTime і TimeOnly?

26. Пояснити різницю між властивостями Days, Hours, Minutes та їхніми аналогами TotalDays, TotalHours, TotalMinutes. Коли використовувати першу групу, а коли другу?
27. Що виведе наступний код? Чи є тут помилка?

```
C#
1  TimeSpan ts = new TimeSpan(1, 14, 30, 0);
2  Console.WriteLine(ts.Days);       // ?
3  Console.WriteLine(ts.TotalDays);  // ?
4  Console.WriteLine(ts.Hours);      // ?
5  Console.WriteLine(ts.TotalHours); // ?
```

28. Напишіть код, який перевіряє, чи пацієнт перебуває у лікарні більше 72 годин, використовуючи TimeSpan.TotalHours.
29. Що таке TimeSpan.Duration() і чим він відрізняється від Math.Abs()? Коли Duration() є необхідним?
30. Як отримати TimeSpan тривалістю рівно 3 доби і 6 годин? Напишіть три різних способи створення цього значення.
31. Порівняйте два способи розрахунку дат наступних прийомів при хіміотерапії з циклом 21 день:

- додавати AddDays(21) в циклі - використовувати TimeSpan.FromDays(21) і оператор + Яку перевагу дає другий підхід?

32. Що відбудеться, якщо від пізнішої дати відняти ранішу? Чи буде TimeSpan від'ємним? Як отримати абсолютну тривалість незалежно від порядку дат?

12.5. DateTimeOffset та часові пояси

33. Що таке DateTimeOffset і яку фундаментальну проблему він вирішує порівняно з DateTime?
34. Порівняйте семантику рівності == для DateTime і DateTimeOffset:

```
C#
1  DateTimeOffset kyiv   = new DateTimeOffset(2026, 6, 11, 14, 30, 0,
   TimeSpan.FromHours(3));
2  DateTimeOffset london = new DateTimeOffset(2026, 6, 11, 11, 30, 0,
   TimeSpan.FromHours(0));
3  Console.WriteLine(kyiv == london); // ?
```

Що виведе цей код і чому?

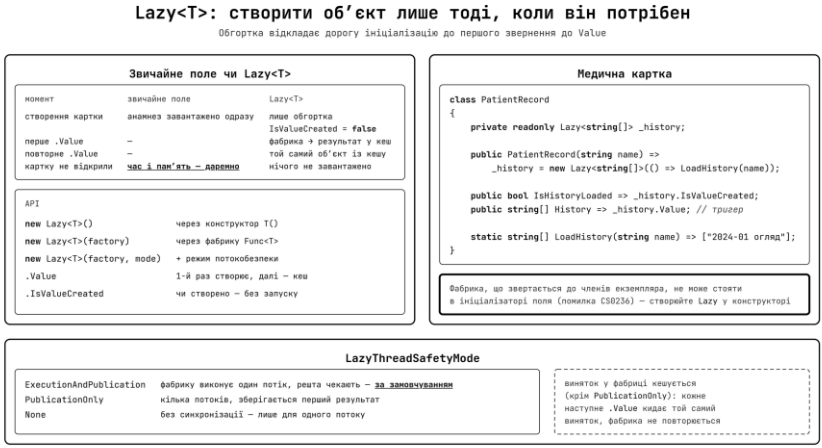
35. Поясніть різницю між властивостями UtcDateTime, LocalDateTime і DateTime у DateTimeOffset. В яких ситуаціях кожна з них корисна?
36. Як TimeZoneInfo враховує перехід на літній/зимовий час? Напишіть код, який перевіряє, чи діє в Києві літній час 1 липня і 1 грудня.
37. Для аудит-журналу медичної системи потрібно зберігати момент події так, щоб пізніше можна було відновити і UTC-час, і локальний час сервера в момент запису. Яку структуру (DateTime, DateTimeOffset) краще використати і чому?
38. Напишіть код, який конвертує UTC-час 2026-06-11 11:30:00 у місцевий час Берліна та Києва, використовуючи TimeZoneInfo.ConvertTimeFromUtc.
39. Що таке DateTime.TryParseExact і чим він відрізняється від DateTimeOffset.Parse? Для яких сценаріїв кожен підходить краще?
40. Сформулюйте три конкретні правила зберігання та відображення дат у медичних інформаційних системах. Обгрунтуйте кожне правило.

Розділ 13. Додаткові класи та структури .NET

13.1. Відкладена ініціалізація та тип Lazy<T>

У реальних застосунках, зокрема медичних інформаційних системах, далеко не кожен об'єкт потрібний при кожному запуску програми. Наприклад, клас PatientRecord може містити посилання на об'єкт, що завантажує повну медичну картку пацієнта з бази даних — але лікар відкриває детальний вигляд лише для деяких пацієнтів. Якщо завантажувати картку завжди при створенні PatientRecord, програма даремно витрачатиме час і пам'ять на дані, які ніхто не переглядатиме.

Це фундаментальний компроміс: **ініціалізувати заздалегідь** (eager) означає завжди бути готовим, але платити вартість навіть за непотрібні об'єкти; **ініціалізувати відкладено** (lazy) означає нести витрати тільки тоді, коли об'єкт справді потрібен.



Lazy<T> — відкладена ініціалізація: Eager vs Lazy

Проблема eager-ініціалізації

Розглянемо клас читача, який зберігає посилання на бібліотеку — об'єкт, що займає значну кількість пам'яті:

```C#
class Reader
{
    Library library = new Library(); // створюється ЗАВЖДИ при new Reader()

    public void ReadBook()
    {
        library.GetBook();
        Console.WriteLine("Читаємо паперову книгу");
    }

    public void ReadEbook()
    {
        Console.WriteLine("Читаємо книгу на комп'ютері");
    }
}

class Library
{
    private string[] books = new string[99]; // займає пам'ять

    public void GetBook()
    {
        Console.WriteLine("Видаємо книгу читачеві");
    }
}
```

Якщо читач використовує лише електронні книги:

```C#
Reader reader = new Reader();
reader.ReadEbook(); // library нікуди не використовується, але вже в пам'яті
```

Об'єкт library вже виділив пам'ять під масив книг, хоча він ніколи не буде використаний. У клінічному контексті це може бути об'єкт, що завантажує тисячі записів аналізів, будує кеш нормативів або відкриває з'єднання з базою даних — усе це відбувається навіть тоді, коли поточний сценарій використання не потребує цих даних.

Клас Lazy<T> — відкладений обгортувач

Для вирішення цієї проблеми .NET надає узагальнений клас Lazy<T>. Він є легким обгортувачем (wrapper): сам по собі Lazy<T> займає мінімум пам'яті і не створює об'єкт T до першого звернення через властивість Value.

Перепишемо клас читача:

```C#
class Reader
{
    Lazy<Library> library = new Lazy<Library>();

```

```
 5      public void ReadBook()
 6      {
 7          library.Value.GetBook(); // .Value — перший виклик створює Library
 8          Console.WriteLine("Читаємо паперову книгу");
 9      }
10
11      public void ReadEbook()
12      {
13          Console.WriteLine("Читаємо книгу на комп'ютері");
14      }
15  }
```

Тепер при new Reader() об'єкт Library **не створюється** — лише легкий Lazy<Library>. Об'єкт Library з'явиться в heap тільки при першому виклику library.Value, тобто тільки якщо хтось викликав ReadBook().

Конструктори Lazy<T>

Lazy<T>() без параметрів вимагає, щоб T мав публічний конструктор без параметрів — він буде викликаний при першому зверненні до Value:

```
C#
1  Lazy<Library> library = new Lazy<Library>(); // new Library() відкладено
```

Якщо потрібна **фабрична функція** (конструктор з параметрами або складна ініціалізація), передається лямбда:

```
C#
1  Lazy<PatientHistory> history = new Lazy<PatientHistory>(
2      () => new PatientHistory(patientId, connectionString));
```

Це найпоширеніший варіант у реальному коді: завдяки замиканню лямбда може захопити будь-які необхідні аргументи, доступні на момент оголошення поля.

Властивості Value та IsValueCreated

Value — єдина точка доступу до обгорнутого об'єкта. При першому зверненні викликає фабрику або конструктор і кешує результат. Усі наступні звернення повертають той самий кешований об'єкт миттєво:

```
C#
1  Lazy<Library> library = new Lazy<Library>();
2
3  // Перше звернення — тригер ініціалізації
4  var lib1 = library.Value; // new Library() викликається тут
5
```

```
6  // Наступні звернення — кеш, без повторного виклику конструктора
7  var lib2 = library.Value; // той самий об'єкт, що й lib1
8  Console.WriteLine(object.ReferenceEquals(lib1, lib2)); // true
```

IsValueCreated — властивість типу bool, яка дозволяє **перевірити** стан без запуску ініціалізації:

```
C#
1  Lazy<Library> library = new Lazy<Library>();
2
3  Console.WriteLine(library.IsValueCreated); // false — ще не створено
4  var lib = library.Value;                   // ініціалізація тут
5  Console.WriteLine(library.IsValueCreated); // true
```

Це корисно в сценаріях, де потрібно з'ясувати, чи вже виконувалася дорога операція — наприклад, для виведення стану кешу в діагностиці або логуванні.

Потокова безпека: LazyThreadSafetyMode

Третій параметр конструктора Lazy<T> — режим потокової безпеки:

```
C#
1  // За замовчуванням: ExecutionAndPublication
2  Lazy<Service> s1 = new Lazy<Service>(
3      () => new Service(),
4      LazyThreadSafetyMode.ExecutionAndPublication);
5
6  // Без синхронізації (однопотоковий код)
7  Lazy<Service> s2 = new Lazy<Service>(
8      () => new Service(),
9      LazyThreadSafetyMode.None);
```

Режим	Поведінка	Коли обирати
ExecutionAndPublication	Тільки один потік виконує ініціалізацію; решта чекають	За замовчуванням, якщо є ризик конкурентного доступу
PublicationOnly	Кілька потоків можуть ініціалізувати паралельно; збережеться перший результат	Якщо ініціалізація ідемпотентна і без побічних ефектів
None	Жодної синхронізації	Тільки для однопотокових сценаріїв

У типових додатках без явного багатопотоку (наприклад, у веб-запитах, де кожен запит має свій потік) достатньо значення за замовчуванням.

Медична картка з відкладеним завантаженням — runnable приклад

Демонструємо IsValueCreated та момент ініціалізації:

```C#
using System;

var p1 = new PatientRecord("Петренко Іван", "I10.9");
var p2 = new PatientRecord("Коваль Марія",  "J45.0");

Console.WriteLine("=== Списку пацієнтів ===");
Console.WriteLine($"{p1.Name} | {p1.IcdCode} | Анамнез завантажено: {p1.IsHistoryLoaded}");
Console.WriteLine($"{p2.Name} | {p2.IcdCode} | Анамнез завантажено: {p2.IsHistoryLoaded}");

Console.WriteLine("\n=== Лікар відкриває картку Петренка ===");
foreach (var item in p1.History)
    Console.WriteLine($"  • {item}");

Console.WriteLine($"\nПетренко: IsHistoryLoaded = {p1.IsHistoryLoaded}");
Console.WriteLine($"Коваль:   IsHistoryLoaded = {p2.IsHistoryLoaded}");
Console.WriteLine("(Анамнез Коваль так і не завантажено — економія ресурсів)");

class PatientRecord
{
    public string Name { get; }
    public string IcdCode { get; }

    // Анамнез — важкий об'єкт, завантажується лише при потребі
    private Lazy<string[]> _history;

    public PatientRecord(string name, string icd)
    {
        Name    = name;
        IcdCode = icd;
        // Фабрика захоплює Name через замикання
        _history = new Lazy<string[]>(() => LoadHistory(name));
    }

    private static string[] LoadHistory(string name)
    {
        Console.WriteLine($"  [ЗАВАНТАЖЕННЯ анамнезу: {name}]");
        return new[] {
            $"2024-01 — первинний огляд",
            $"2024-06 — призначення лікування",
            $"2026-06 — плановий контроль",
        };
    }

    public bool IsHistoryLoaded => _history.IsValueCreated;
```

```
45      public string[] History => _history.Value; // тригер
46  }
```

Сервіс нормативів — runnable приклад

Lazy<T> для відкладеної побудови кешу нормативних значень аналізів:

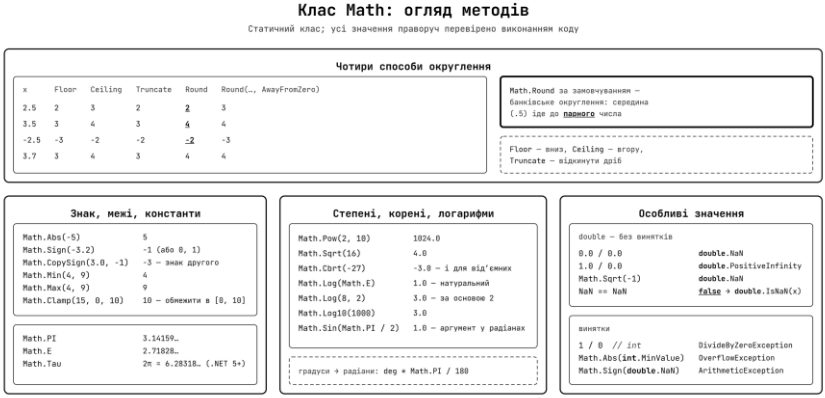
```
C#
 1  using System;
 2  using System.Collections.Generic;
 3
 4  Console.WriteLine("=== Лабораторні показники — Петренко Іван ===");
 5  Console.WriteLine($"IsValueCreated перед першим зверненням: (перевіряємо через
    Check)");
 6  Console.WriteLine();
 7
 8  var results = new[]
 9  {
10      ("glucose",      7.3),
11      ("hemoglobin",   135.0),
12      ("leukocytes",   11.5),
13      ("cholesterol",  4.8),
14      ("erythrocytes", 4.2),
15  };
16
17  foreach (var (test, val) in results)
18  {
19      var (ok, msg) = LabNormService.Check(test, val);
20      string mark = ok ? "✓" : "!";
21      Console.WriteLine($"  [{mark}] {test,-14}: {msg}");
22  }
23
24  class LabNormService
25  {
26      // Таблиця норм будується один раз при першому зверненні
27      private static readonly Lazy<Dictionary<string, (double Min, double Max,
    string Unit)>> _norms
28          = new Lazy<Dictionary<string, (double, double, string)>>(() =>
29          {
30              Console.WriteLine("  [ІНІЦІАЛІЗАЦІЯ таблиці норм — відбувається
    один раз]");
31              return new Dictionary<string, (double, double, string)>
32              {
33                  ["glucose"]      = (3.9,  6.1,  "ммоль/л"),
34                  ["hemoglobin"]   = (120,  160,  "г/л"),
35                  ["erythrocytes"] = (3.8,  5.2,  "10¹²/л"),
36                  ["leukocytes"]   = (4.0,  9.0,  "10⁹/л"),
37                  ["cholesterol"]  = (0,    5.2,  "ммоль/л"),
38              };
39          });
40
41      public static (bool ok, string msg) Check(string test, double value)
42      {
43          var norms = _norms.Value; // тригер при першому виклику
44          if (!norms.TryGetValue(test, out var norm))
45              return (false, $"Невідомий показник: {test}");
46
47          bool ok = value >= norm.Min && value <= norm.Max;
```

```
48          string status = ok ? "НОРМА" : (value < norm.Min ? "НИЖЧЕ НОРМИ" :
    "ВИЩЕ НОРМИ");
49          return (ok, $"{value:F2} {norm.Unit} [{norm.Min}–{norm.Max}] →
    {status}");
50      }
51  }
```

13.2. Клас Math

Обчислення в медичних інформаційних системах пронизують усі рівні застосунку: розрахунок ІМТ пацієнта, дозування препаратів, перетворення одиниць вимірювання, побудова статистичних показників. Для всіх цих потреб .NET надає статичний клас Math із простору імен System — набір математичних функцій і констант, готових до використання без будь-яких додаткових просторів імен.

Math є **запечатаним статичним класом** (static sealed class): неможливо ані успадкувати від нього, ані створити екземпляр — усі члени викликаються безпосередньо через ім'я класу: Math.Round(...), Math.Sqrt(...). Це проектне рішення .NET: математичні операції не потребують стану, тому клас-бібліотека з набором чистих функцій є найпростішим і найефективнішим підходом.



Клас Math — огляд методів

Константи Math.PI та Math.E

Клас Math надає дві фундаментальні математичні константи як public const double:

```
C#
1  Console.WriteLine(Math.PI);  // 3.141592653589793
2  Console.WriteLine(Math.E);   // 2.718281828459045
3  Console.WriteLine(Math.Tau); // 6.283185307179586 = 2 * PI  (.NET 5+)
```

Math.PI — відношення довжини кола до діаметра, використовується в задачах геометрії і тригонометрії. Math.E — основа натурального логарифму, з'являється в формулах неперервного нарахування відсотків, нормального розподілу, ентропії. Math.Tau = 2π — конвенція, що з'явилась у .NET 5: в багатьох формулах зручніше оперувати одним повним обертом τ замість «2π».

Округлення: чотири різні методи

Округлення — одна з найбільш пастяних операцій у C#. Стандарт IEEE 754 для double і стандарт ISO банківської математики дають різні результати на межових значеннях (x.5), тому Math надає кілька різних методів.

Math.Floor та Math.Ceiling

Math.Floor(x) — **округлення вниз** до найближчого цілого, не більшого за x. Математично: ⌊x⌋. Для від'ємних чисел «вниз» означає далі від нуля:

```
C#
1  Console.WriteLine(Math.Floor(2.7));  // 2
2  Console.WriteLine(Math.Floor(2.1));  // 2
3  Console.WriteLine(Math.Floor(-2.7)); // -3  (до -∞, не до 0!)
```

Math.Ceiling(x) — **округлення вгору** до найближчого цілого, не меншого за x. Математично: ⌈x⌉:

```
C#
1  Console.WriteLine(Math.Ceiling(2.1));  // 3
2  Console.WriteLine(Math.Ceiling(2.0));  // 2
3  Console.WriteLine(Math.Ceiling(-2.1)); // -2  (до +∞)
```

У медичних розрахунках Floor і Ceiling з'являються, коли потрібно визначити кількість повних одиниць: «скільки повних днів пройшло», «скільки повних таблеток в упаковці по N мг».

Math.Truncate

Math.Truncate(x) відкидає дробову частину, завжди **до нуля**. На відміну від Floor, для від'ємних чисел вони різняться:

```
C#
1  Console.WriteLine(Math.Truncate(2.9));  // 2
2  Console.WriteLine(Math.Truncate(-2.9)); // -2  (не -3 як Floor!)
```

Math.Round — банківське заокруглення за замовчуванням

Math.Round(x) за замовчуванням застосовує **банківське заокруглення** (MidpointRounding.ToEven): на межовому значенні x.5 округлюється до **найближчого парного**:

```
C#
1  Console.WriteLine(Math.Round(2.5)); // 2  — до парного
2  Console.WriteLine(Math.Round(3.5)); // 4  — до парного
3  Console.WriteLine(Math.Round(4.5)); // 4  — до парного
4  Console.WriteLine(Math.Round(5.5)); // 6  — до парного
```

Це стандарт IEEE 754 і банківської математики: при великих вибірках сума помилок округлення прямує до нуля. Але «шкільне» правило «.5 завжди вгору» виглядає інакше:

```
C#
1  Console.WriteLine(Math.Round(2.5, MidpointRounding.AwayFromZero)); // 3
2  Console.WriteLine(Math.Round(3.5, MidpointRounding.AwayFromZero)); // 4
```

Другий параметр Math.Round(x, n) задає кількість знаків після коми:

```
C#
1  Console.WriteLine(Math.Round(3.14159, 2));  // 3.14
2  Console.WriteLine(Math.Round(3.14159, 4));  // 3.1416
```

Для медичних доз важливо знати, яке саме заокруглення застосовується: помилка заокруглення, що накопичується через сотні пацієнтів, може призводити до систематичних відхилень у статистиці.

Абсолютне значення та знак

Math.Abs(x) повертає модуль числа: Math.Abs(-7.5) == 7.5. Перевантажений для всіх числових типів: int, long, double, decimal тощо.

Math.Sign(x) повертає -1, 0 або 1 залежно від знаку аргументу:

```
C#
1  Console.WriteLine(Math.Sign(-15));  // -1
2  Console.WriteLine(Math.Sign(0));    //  0
3  Console.WriteLine(Math.Sign(3.7));  //  1
```

Це корисно, коли потрібно визначити **напрям відхилення** від норми без знання абсолютної величини. Наприклад: Math.Sign(actual - norm) повертає +1 якщо показник вище норми, -1 якщо нижче, 0 якщо рівний.

Math.CopySign(magnitude, sign) (.NET Core 3.0+) повертає число з абсолютним значенням magnitude і знаком sign:

```
C#
1  Console.WriteLine(Math.CopySign(5.0, -1.0)); // -5
2  Console.WriteLine(Math.CopySign(5.0, 1.0));  // 5
```

Min, Max та Clamp

Math.Min(a, b) і Math.Max(a, b) — найпростіші операції порівняння, перевантажені для всіх числових типів:

```
C#
1  int age   = Math.Max(0, inputAge);  // негативний вік неможливий
2  double bmi = Math.Min(calculatedBmi, 99.9); // обмеження відображення
```

Math.Clamp(value, min, max) (.NET Core 2.0+) — **обмеження значення в діапазон**: якщо value < min, повертає min; якщо value > max, повертає max; інакше повертає value:

```
C#
1  int percent = Math.Clamp(rawValue, 0, 100);
2  double dose = Math.Clamp(calculatedDose, minDose, maxDose);
```

Це надзвичайно корисна операція в медичних системах: кожен показник має допустимий діапазон, і Clamp дозволяє безпечно обмежити значення без громіздкого if:

```
C#
1  // Без Clamp:
2  if (dose < 0.5) dose = 0.5;
3  else if (dose > 10.0) dose = 10.0;
4
5  // З Clamp:
6  dose = Math.Clamp(dose, 0.5, 10.0);
```

Степені та корені

Math.Pow(base, exp) — піднесення до степеня. Обидва аргументи і результат — double:

```
C#
1  Console.WriteLine(Math.Pow(2, 10));   // 1024
```

```
2  Console.WriteLine(Math.Pow(9, 0.5));  // 3.0  (= Math.Sqrt(9))
3  Console.WriteLine(Math.Pow(10, -2));  // 0.01
```

Math.Sqrt(x) — квадратний корінь (square root). Швидший спеціалізований варіант, ніж Math.Pow(x, 0.5):

```
C#
1  double bmi = weight / Math.Pow(heightMeters, 2); // зріст у квадраті
```

Math.Cbrt(x) (.NET Core 2.1+) — кубічний корінь. На відміну від Math.Pow(x, 1.0/3.0), коректно обробляє від'ємні числа:

```
C#
1  Console.WriteLine(Math.Cbrt(27));  // 3
2  Console.WriteLine(Math.Cbrt(-8)); // -2  (Pow(-8, 1.0/3.0) → NaN!)
```

Логарифми:

```
C#
1  Console.WriteLine(Math.Log(Math.E));   // 1.0   — натуральний ln
2  Console.WriteLine(Math.Log(100, 10));  // 2.0   — log₁₀(100)
3  Console.WriteLine(Math.Log2(8));       // 3.0   — log₂(8)
4  Console.WriteLine(Math.Log10(1000));   // 3.0
```

Тригонометрія та радіани

Усі тригонометричні функції Math приймають аргумент **у радіанах**, а не градусах. Перетворення: radians = degrees * Math.PI / 180:

```
C#
1  double degrees = 90;
2  double radians = degrees * Math.PI / 180;
3
4  Console.WriteLine(Math.Sin(radians)); // 1.0
5  Console.WriteLine(Math.Cos(radians)); // 6.12e-17 ≈ 0 (похибка floating-point)
6  Console.WriteLine(Math.Tan(Math.PI / 4)); // 1.0  (tan(45°))
```

Обернені функції повертають радіани:

```
C#
1  double angle = Math.Asin(1.0);  // π/2 ≈ 1.5707...
2  Console.WriteLine(angle * 180 / Math.PI); // 90 градусів
```

Перевірка особливих значень double

При математичних обчисленнях можуть виникати спеціальні значення double:

— 0.0 / 0.0 → double.NaN (Not a Number)

— 1.0 / 0.0 → double.PositiveInfinity

— -1.0 / 0.0 → double.NegativeInfinity

Ці значення не кидають виняток — вони **поширюються**: NaN + 5 == NaN, Infinity * 2 == Infinity. Тому перед використанням результату обчислення слід перевіряти:

```
C#
1  double result = weight / (height * height);
2
3  if (!double.IsFinite(result))
4  {
5      Console.WriteLine("Некоректні вхідні дані");
6      return;
7  }
```

```
C#
1  double.IsNaN(x)       // true якщо NaN
2  double.IsInfinity(x)  // true якщо +∞ або -∞
3  double.IsFinite(x)    // true якщо звичайне скінченне число (не NaN, не Inf)
```

double.IsFinite — найзручніша перевірка «чи придатне це значення для подальших розрахунків».

Розрахунок ІМТ — runnable приклад

Обчислення ІМТ (Індекс маси тіла) за формулою ВООЗ: BMI = вага (кг) / зріст² (м):

```
C#
 1  using System;
 2
 3  static string BmiCategory(double bmi)
 4  {
 5      if (bmi < 18.5) return "Недостатня маса";
 6      if (bmi < 25.0) return "Норма";
 7      if (bmi < 30.0) return "Надмірна вага";
 8      if (bmi < 35.0) return "Ожиріння I ст.";
 9      return "Ожиріння II+ ст.";
10  }
11
12  var patients = new[]
13  {
14      ("Петренко Іван",  72.0, 1.78),
15      ("Коваль Марія",   55.0, 1.62),
16      ("Сидоренко Олег", 105.0, 1.75),
17      ("Бойко Тетяна",   48.0, 1.68),
18  };
19
20  Console.WriteLine($"{"Пацієнт",-22} {"Вага",6} {"Зріст",6} {"ІМТ",6}
    Категорія");
21  Console.WriteLine(new string('-', 65));
22
```

```
double totalBmi = 0;
foreach (var (name, weight, height) in patients)
{
    double bmi = weight / Math.Pow(height, 2);
    double bmiRounded = Math.Round(bmi, 1);
    totalBmi += bmi;

    // Clamp для відображення: візуальний індикатор обмежений 0-100
    int bar = Math.Clamp((int)bmi, 0, 50);
    string category = BmiCategory(bmi);

    Console.WriteLine($"{name,-22} {weight,6:F1} {height,6:F2} {bmiRounded,6:F1}  {category}");
}

double avg = totalBmi / patients.Length;
Console.WriteLine(new string('-', 65));
Console.WriteLine($"Середній ІМТ по групі: {Math.Round(avg, 1):F1}");

// Демонстрація IsFinite: захист від нульового зросту
Console.WriteLine("\n=== Перевірка некоректних даних ===");
double badHeight = 0.0;
double badBmi = 70.0 / Math.Pow(badHeight, 2);
Console.WriteLine($"70кг / 0м² = {badBmi}");
Console.WriteLine($"IsFinite: {double.IsFinite(badBmi)}");
Console.WriteLine($"IsInfinity: {double.IsInfinity(badBmi)}");
```

Округлення та математичні функції — runnable приклад

Демонстрація всіх режимів округлення та функцій степеня/кореня:

```
C#
using System;

Console.WriteLine("=== Порівняння методів округлення ===");
double[] values = { 2.5, 3.5, -2.5, 2.4, 2.6 };

Console.WriteLine($"{"Значення",10} {"Floor",8} {"Ceiling",9} {"Truncate",10} {"Round(ToEven)",15} {"Round(Away0)",13}");
Console.WriteLine(new string('-', 68));

foreach (double v in values)
{
    Console.WriteLine(
        $"{v,10:F1}" +
        $"{Math.Floor(v),8:F0}" +
        $"{Math.Ceiling(v),9:F0}" +
        $"{Math.Truncate(v),10:F0}" +
        $"{Math.Round(v),15:F0}" +
        $"{Math.Round(v, MidpointRounding.AwayFromZero),13:F0}"
    );
}

Console.WriteLine("\n=== Степінь, корінь, логарифм ===");
Console.WriteLine($"Math.Pow(2, 10)  = {Math.Pow(2, 10)}");
Console.WriteLine($"Math.Sqrt(144)   = {Math.Sqrt(144)}");
Console.WriteLine($"Math.Cbrt(125)   = {Math.Cbrt(125)}");
Console.WriteLine($"Math.Log10(1000) = {Math.Log10(1000)}");
```

```
Console.WriteLine($"Math.Log2(64)    = {Math.Log2(64)}");

Console.WriteLine("\n=== Abs, Sign, Min, Max, Clamp ===");
int heartRate = 210; // некоректне значення
int clamped = Math.Clamp(heartRate, 40, 200);
Console.WriteLine($"ЧСС отримано: {heartRate} уд/хв");
Console.WriteLine($"Math.Clamp(210, 40, 200) = {clamped} уд/хв");

double deviation = -4.7;
Console.WriteLine($"\nВідхилення від норми: {deviation}");
Console.WriteLine($"Math.Abs:  {Math.Abs(deviation):F1}");
Console.WriteLine($"Math.Sign: {Math.Sign(deviation)}  (−1 = нижче норми)");

Console.WriteLine("\n=== Тригонометрія: 30, 60, 90 градусів ===");
int[] degrees = { 30, 60, 90 };
foreach (int deg in degrees)
{
    double rad = deg * Math.PI / 180;
    Console.WriteLine($"sin({deg}°) = {Math.Round(Math.Sin(rad), 4),8}
cos({deg}°) = {Math.Round(Math.Cos(rad), 4),8}");
}
```

13.3. Клас Convert

У реальних застосунках дані постійно змінюють форму: рядок із форми стає числом, ціле число перетворюється на рядок для JSON, булеве значення записується як 1 або 0 у базу даних, бінарний вміст файлу кодується у Base64 для передачі через API. Для всіх цих потреб .NET надає статичний клас Convert із простору імен System.

На відміну від оператора явного приведення (int), Convert розроблений для сценаріїв, де тип вхідного значення може бути невідомий на етапі компіляції, де рядок потрібно розпарсити у число, або де важлива семантика заокруглення. Клас реалізує перетворення через інтерфейс IConvertible, який реалізують усі базові типи .NET.



Клас Convert — перетворення типів

Parse та TryParse

Найпростіший шлях перетворити рядок на примітивний тип — методи Parse() та TryParse(), які визначені безпосередньо на кожному числовому типі.

Parse() приймає рядок і повертає значення потрібного типу. Якщо рядок не відповідає очікуваному формату, кидається FormatException:

```
C#
1  using System;
2  using System.Globalization;
3
4  int    age    = int.Parse("67");
5  double weight = double.Parse("82.5", CultureInfo.InvariantCulture); // крапка —
   див. нижче
6  bool   active = bool.Parse("True");
7
8  Console.WriteLine($"Вік: {age}, Вага: {weight}, Активний: {active}");
```

Важливий нюанс: парсинг дробових чисел залежить від поточної культури. На машинах з українською/польською локаллю double.Parse("82.5") може кинути виняток, бо очікується кома. Щоб гарантувати крапку як роздільник — використовуйте CultureInfo.InvariantCulture:

```
C#
1  using System;
2  using System.Globalization;
```

```
3
4  double weight1 = double.Parse("82.5", CultureInfo.InvariantCulture); // завжди
   крапка
5  double weight2 = double.Parse("82,5", new NumberFormatInfo {
   NumberDecimalSeparator = "," });
6
7  Console.WriteLine($"{weight1}  {weight2}");
```

TryParse() — безпечна альтернатива, що не кидає виняток: повертає true якщо перетворення пройшло успішно, false інакше. Значення записується в out-параметр:

```
C#
 1  using System;
 2
 3  string[] inputs = { "67", "abc", "3.14", "" };
 4
 5  foreach (string s in inputs)
 6  {
 7      if (int.TryParse(s, out int value))
 8          Console.WriteLine($"'{s}' -> {value}");
 9      else
10          Console.WriteLine($"'{s}' -> не вдалось розпарсити");
11  }
```

Правило вибору: якщо рядок надходить від користувача або з ненадійного джерела — використовуйте TryParse. Якщо рядок гарантовано правильного формату (з власної БД, відомого API) — Parse. Convert.ToXxx підходить для перетворень між числовими типами та булевими значеннями, а не тільки для рядків.

Cast vs Convert: в чому різниця

Щоб зрозуміти призначення Convert, важливо спочатку побачити, чим він відрізняється від оператора явного приведення типу (T).

Оператор (int) виконує **усічення** (truncation): він просто відкидає дробову частину, не округлюючи:

```
C#
1  double d = 3.9;
2  int a = (int)d;                 // 3  — дробова частина відкинута
3  int b = Convert.ToInt32(d);     // 4  — банківське заокруглення
```

Convert.ToInt32 застосовує **банківське заокруглення** (як Math.Round з MidpointRounding.ToEven), тому 3.5 → 4 і 2.5 → 2. Це важливо для медичних розрахунків, де джерелом числа може бути double, але результат потрібен у цілочисельній формі.

Ще одна принципова відмінність: (int) можна застосувати тільки до сумісних числових типів — компілятор не дозволить (int)"42" або (int)true. Convert же здатен перетворити рядок у число, bool у int і навпаки:

```
C#
1  // Це не компілюється:
2  // int x = (int)"42";   // CS0030
3
4  // Це працює:
5  int x = Convert.ToInt32("42"); // 42
6  int y = Convert.ToInt32(true); // 1
7  int z = Convert.ToInt32(false);// 0
```

Числові перетворення

Методи Convert.ToInt32, Convert.ToInt64, Convert.ToDouble, Convert.ToDecimal тощо приймають object або будь-який тип, що реалізує IConvertible. Якщо передати null — повертається нуль або значення за замовчуванням:

```
C#
1  int fromNull    = Convert.ToInt32(null);   // 0
2  double fromNull = Convert.ToDouble(null);  // 0.0
```

Якщо передати рядок, що не є числом, — кидається FormatException:

```
C#
1  int bad = Convert.ToInt32("abc"); // FormatException!
```

Тому для ненадійних вхідних рядків слід використовувати int.TryParse або double.TryParse — вони повертають bool замість викидання виняткового стану. Convert підходить тоді, коли формат даних гарантований (наприклад, дані з власної бази або відомого API).

При переповненні (overflow) Convert.ToInt32 кидає OverflowException:

```
C#
1  Convert.ToInt32(3_000_000_000L); // OverflowException — не вміщується в int
```

Це **безпечніша** поведінка, ніж явне приведення (int). Якщо значення відоме компілятору (константа), (int)3_000_000_000L просто не скомпілюється (помилка CS0221). Але якщо число приходить у змінній, приведення мовчки дасть некоректний результат:

```
C#
1  long big = 3_000_000_000L;
2  int wrapped = (int)big;   // -1294967296 — старші біти відкинуто, без жодного
   попередження
```

Для дробових чисел поведінка (int) інша: починаючи з .NET 9, значення double поза межами int «насичується» до int.MaxValue або int.MinValue, а NaN стає 0 (раніше результат залежав від процесора). Convert.ToInt32 в обох випадках кидає OverflowException і не дає помилці пройти непоміченою.

Перетворення рядків: Convert.ToString та Convert.ToXxx

Convert.ToString(object value) перетворює будь-яке значення в рядок. Принципова відмінність від виклику .ToString() безпосередньо — коректна обробка null: виклик x.ToString() для x == null кидає NullReferenceException, а Convert.ToString — ні.

Але тут є тонкість: у Convert.ToString **багато перевантажень**, і результат для null залежить від того, яке з них обере компілятор. Перевантаження для object повертає порожній рядок, а перевантаження для string — повертає null як є:

```
C#
 1  object? diagnosis = null;
 2  string? s1 = Convert.ToString(diagnosis);  // "" — перевантаження
    ToString(object)
 3
 4  string? note = null;
 5  string? s2 = Convert.ToString(note);       // null — перевантаження
    ToString(string)!
 6
 7  string? s3 = Convert.ToString(null);       // null — для літерала null
    компілятор
 8                                             // обирає найконкретніше:
    ToString(string)
 9
10  string? s4 = diagnosis?.ToString();        // null — оператор ?. не викликає
    метод для null
```

Якщо поле пацієнта необов'язкове, має тип object і може бути null, Convert.ToString дозволяє отримати рядок для відображення без додаткової перевірки if (x != null). Для змінних типу string надійніше явно задати значення за замовчуванням: note ?? "".

Convert.ToDateTime(string) перетворює рядок у DateTime, орієнтуючись на поточну культуру. Для надійного парсингу з явним форматом краще використовувати DateTime.ParseExact або DateTime.TryParseExact (розд. 12.2).

Булеві перетворення

Convert.ToBoolean розпізнає числові та рядкові представлення булевих значень:

```
C#
1  bool b1 = Convert.ToBoolean(1);       // true
2  bool b2 = Convert.ToBoolean(0);       // false
3  bool b3 = Convert.ToBoolean("True");  // true  (регістронезалежно)
4  bool b4 = Convert.ToBoolean("False"); // false
5  bool b5 = Convert.ToBoolean("yes");   // FormatException! (тільки True/False)
```

Зворотньо — Convert.ToInt32(bool) дає 1 або 0, що зручно при запису в бази даних, де bool зберігається як BIT або TINYINT.

Base64 — кодування бінарних даних

Convert.ToBase64String(byte[]) і Convert.FromBase64String(string) — два методи для кодування бінарних даних у текстовий формат Base64 і назад.

Base64 перетворює довільні байти у рядок із символів ASCII (64 символи: A-Z, a-z, 0-9, +, /). Це стандарт передачі бінарних даних у текстових протоколах:

```
C#
1  byte[] data = { 72, 101, 108, 108, 111 }; // "Hello" у ASCII
2
3  string encoded = Convert.ToBase64String(data);
4  Console.WriteLine(encoded); // "SGVsbG8="
5
6  byte[] decoded = Convert.FromBase64String(encoded);
7  Console.WriteLine(System.Text.Encoding.ASCII.GetString(decoded)); // "Hello"
```

У медичній інформатиці Base64 з'являється у стандарті **HL7 FHIR**: зображення (рентген, УЗД), PDF-документи і файли аналізів передаються у JSON-відповідях API у форматі Base64. При отриманні такого поля Convert.FromBase64String повертає масив байтів, який потім зберігається або відображається.

```
C#
1  // Приклад поля у FHIR-документі:
2  // { "data": "SGVsbG8=" }
3
4  string fhirData = "SGVsbG8=";
5  byte[] documentBytes = Convert.FromBase64String(fhirData);
6  // далі: File.WriteAllBytes("document.pdf", documentBytes)
```

Convert.ChangeType — динамічне перетворення

Convert.ChangeType(object value, Type conversionType) — найгнучкіший метод класу. Він приймає значення і тип як Type, виконуючи перетворення у рантаймі:

```
C#
object result = Convert.ChangeType("42", typeof(int));
int number = (int)result; // 42

object d = Convert.ChangeType("3.14", typeof(double));
```

Це корисно в узагальненому (generic) коді, де тип перетворення стає відомим лише під час виконання: наприклад, при заповненні властивостей об'єкта з рядкового словника (десеріалізація, читання CSV із заголовками, відображення форм).

ChangeType також кидає FormatException якщо рядок некоректний, та OverflowException при переповненні — ті самі виняткові стани, що й конкретні методи ToInt32, ToDouble тощо.

Парсинг даних пацієнта — runnable приклад

Читання клінічних даних пацієнта з рядкового представлення (наприклад, CSV або API-відповідь):

```
C#
using System;
using System.Globalization;

Console.WriteLine("=== Зчитування даних пацієнта з рядків ===");

// Симуляція даних, що прийшли з форми або CSV
string nameRaw     = "Петренко Іван";
string ageRaw      = "67";
string weightRaw   = "82.5";
string heightRaw   = "1.76";
string activeRaw   = "1";
string diagnosisRaw = null; // необов'язкове поле

// Convert для гарантовано коректних рядків
int    age     = Convert.ToInt32(ageRaw);
double weight  = Convert.ToDouble(weightRaw, CultureInfo.InvariantCulture);
double height  = Convert.ToDouble(heightRaw, CultureInfo.InvariantCulture);
bool   active  = Convert.ToBoolean(activeRaw == "1" ? "True" : "False");
string diag    = Convert.ToString(diagnosisRaw); // "" якщо null

double bmi = weight / (height * height);

Console.WriteLine($"Пацієнт:   {nameRaw}");
Console.WriteLine($"Вік:       {age} р.");
Console.WriteLine($"Вага/зріст:{weight} кг / {height} м");
```

```
26  Console.WriteLine($"IMT:       {Math.Round(bmi, 1)}");
27  Console.WriteLine($"Активний:  {active}");
28  Console.WriteLine($"Діагноз:   '{diag}' (пустий рядок, не null)");
29
30  Console.WriteLine("\n=== Різниця: cast vs Convert ===");
31  double d = 3.7;
32  Console.WriteLine($"(int)3.7              = {(int)d}      // Truncate");
33  Console.WriteLine($"Convert.ToInt32(3.7)  = {Convert.ToInt32(d)}      //
    Round");
34
35  double half = 2.5;
36  Console.WriteLine($"(int)2.5              = {(int)half}      // Truncate");
37  Console.WriteLine($"Convert.ToInt32(2.5)  = {Convert.ToInt32(half)}      //
    ToEven = 2");
38
39  Console.WriteLine("\n=== Захист від FormatException ===");
40  string[] inputs = { "45", "abc", "22.5", "" };
41  foreach (var s in inputs)
42  {
43      try
44      {
45          int val = Convert.ToInt32(s);
46          Console.WriteLine($"  '{s}' -> {val}");
47      }
48      catch (FormatException)
49      {
50          Console.WriteLine($"  '{s}' -> FormatException (використовуй
    TryParse!)");
51      }
52      catch (OverflowException)
53      {
54          Console.WriteLine($"  '{s}' -> OverflowException");
55      }
56  }
```

Base64 і ChangeType — runnable приклад

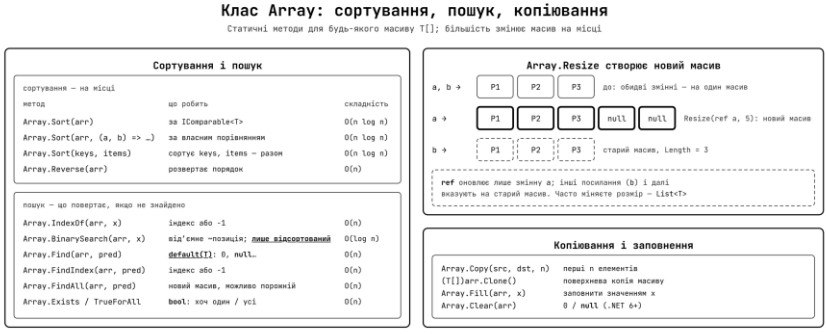
```
C#
 1  using System;
 2  using System.Globalization;
 3  using System.Text;
 4
 5  Console.WriteLine("=== Base64: медичний ідентифікатор ===");
 6
 7  // Унікальний ідентифікатор пацієнта для передачі через API
 8  string patientId = "UA-2026-PETRO-I10";
 9  byte[] idBytes   = Encoding.UTF8.GetBytes(patientId);
10
11  string encoded = Convert.ToBase64String(idBytes);
12  Console.WriteLine($"Оригінал:  {patientId}");
13  Console.WriteLine($"Base64:    {encoded}");
14
15  byte[] decoded  = Convert.FromBase64String(encoded);
16  string restored = Encoding.UTF8.GetString(decoded);
17  Console.WriteLine($"Відновлено:{restored}");
18  Console.WriteLine($"Ідентично: {patientId == restored}");
19
20  Console.WriteLine("\n=== Base64: бінарні дані ===");
```

```
21  byte[] fakeDicom = { 0x44, 0x49, 0x43, 0x4D }; // "DICM" — початок DICOM-файлу
22  string b64 = Convert.ToBase64String(fakeDicom);
23  Console.WriteLine($"DICOM header bytes: {string.Join(", ", fakeDicom)}");
24  Console.WriteLine($"Base64: {b64}");
25  byte[] backBytes = Convert.FromBase64String(b64);
26  Console.WriteLine($"Відновлено байти:   {string.Join(", ", backBytes)}");
27
28  Console.WriteLine("\n=== Convert.ChangeType: динамічне перетворення ===");
29
30  // Словник даних — тип значення невідомий заздалегідь
31  string[] keys   = { "age", "weight", "active", "name" };
32  string[] values = { "67",  "82.5",   "True",   "Петренко Іван" };
33  Type[]   types  = { typeof(int), typeof(double), typeof(bool), typeof(string)
    };
34
35  for (int i = 0; i < keys.Length; i++)
36  {
37      object result = Convert.ChangeType(values[i], types[i],
    CultureInfo.InvariantCulture);
38      Console.WriteLine($"  {keys[i],-8}: \"{values[i],-14}\" -> {result,-16}
    ({result.GetType().Name})");
39  }
```

13.4. Клас Array

Масив — одна з фундаментальних структур даних у C#. Кожен масив, оголошений як int[], string[] або PatientRecord[], насправді є об'єктом класу System.Array — абстрактного базового класу для всіх масивів у .NET. Саме тому масиви мають властивості .Length, .Rank і підтримують інтерфейси IEnumerable<T>, IList<T>, ICloneable.

Поряд із самими масивами, .NET надає **статичний клас Array** із набором методів для пошуку, сортування, копіювання, заповнення та перевірки. Ці методи доповнюють вбудовані можливості масивів і дозволяють виконувати типові операції без написання ручних циклів.



Клас Array — пошук, сортування, копіювання

Властивості масивів

Будь-який масив як екземпляр System.Array має такі властивості:

```
C#
 1  int[] ages = { 67, 45, 52, 71, 38 };
 2
 3  Console.WriteLine(ages.Length);    // 5 — загальна кількість елементів
 4  Console.WriteLine(ages.Rank);      // 1 — одновимірний масив
 5
 6  int[,] matrix = new int[3, 4];
 7  Console.WriteLine(matrix.Length);         // 12 — 3 × 4
 8  Console.WriteLine(matrix.Rank);           // 2
 9  Console.WriteLine(matrix.GetLength(0));   // 3 — рядки
10  Console.WriteLine(matrix.GetLength(1));   // 4 — стовпці
```

Rank завжди 1 для звичайних одновимірних масивів. Для двовимірних (матриць) GetLength(dimension) повертає розмір відповідного виміру.

Сортування: Array.Sort

Array.Sort(arr) виконує сортування **in-place** — змінює сам переданий масив, нічого не повертаючи. Використовує алгоритм introsort (O(n log n)), вимагає, щоб тип T реалізовував IComparable<T>:

```
C#
1  int[] ages = { 67, 45, 52, 71, 38 };
2  Array.Sort(ages);
3  // ages: { 38, 45, 52, 67, 71 }
```

Для сортування масиву об'єктів за конкретним полем передається **лямбда-вираз порівняння**:

```
C#
1  string[] names = { "Сидоренко", "Петренко", "Коваль", "Бойко" };
2  Array.Sort(names, (a, b) => string.Compare(a, b,
   StringComparison.OrdinalIgnoreCase));
3  // names: { "Бойко", "Коваль", "Петренко", "Сидоренко" }
```

Array.Sort(keys, values) дозволяє **паралельно сортувати два масиви**, зберігаючи відповідність між ними. Після сортування елементи values слідують за переміщеними елементами keys:

```
C#
1  string[] ids   = { "P003", "P001", "P002" };
2  int[]    ages  = { 52, 38, 67 };
3
4  Array.Sort(ids, ages); // сортуємо за id, ages рухаються разом
```

```
5  // ids:  { "P001", "P002", "P003" }
6  // ages: { 38, 67, 52 }
```

Array.Reverse(arr) перевертає порядок елементів in-place. Зручно після сортування, якщо потрібний спадний порядок:

```
C#
1  Array.Sort(ages);
2  Array.Reverse(ages); // від більшого до меншого
```

Пошук: IndexOf, BinarySearch, Find

Array.IndexOf(arr, value) виконує **лінійний пошук** (O(n)) і повертає індекс першого знайденого елемента або -1:

```
C#
1  string[] ids = { "P001", "P002", "P003", "P004" };
2  int idx = Array.IndexOf(ids, "P003"); // 2
3  int bad = Array.IndexOf(ids, "P999"); // -1
```

Array.BinarySearch(arr, value) — **бінарний пошук** (O(log n)), але **вимагає відсортованого масиву**. Якщо елемент знайдено, повертає його індекс. Якщо ні — повертає від'ємне число (побітове доповнення позиції вставки):

```
C#
1  int[] sorted = { 38, 45, 52, 67, 71 };
2  int pos  = Array.BinarySearch(sorted, 52);  //  2 — знайдено
3  int miss = Array.BinarySearch(sorted, 50);  // -3 — не знайдено (~позиції 2)
```

Якщо результат від'ємний — елемент відсутній. Щоб отримати позицію, куди можна вставити елемент: ~miss.

Array.Find(arr, predicate) шукає **перший елемент, що задовольняє умову**. Якщо нічого не знайдено — повертає значення за замовчуванням (null для посилальних типів, 0 для int):

```
C#
1  int[] ages = { 67, 45, 52, 71, 38 };
2  int senior = Array.Find(ages, a => a >= 65); // 67
```

Array.FindAll(arr, predicate) повертає **новий масив** з усіма елементами, що відповідають умові:

```
C#
1  int[] seniors = Array.FindAll(ages, a => a >= 65); // { 67, 71 }
```

Array.FindIndex(arr, predicate) — як Find, але повертає **індекс**, а не значення:

```
C#
1  int idx = Array.FindIndex(ages, a => a >= 65); // 0 (перший елемент)
```

Перевірки: Exists та TrueForAll

Array.Exists(arr, predicate) — повертає true, якщо **хоча б один** елемент задовольняє умову:

```
C#
1  bool hasSenior = Array.Exists(ages, a => a >= 65); // true
```

Array.TrueForAll(arr, predicate) — повертає true, якщо **всі** елементи задовольняють умову:

```
C#
1  bool allAdults = Array.TrueForAll(ages, a => a >= 18); // true
2  bool allYoung  = Array.TrueForAll(ages, a => a < 50);  // false
```

Копіювання: Array.Copy та Clone

Array.Copy(sourceArray, destinationArray, length) копіює перші length елементів:

```
C#
1  int[] src = { 1, 2, 3, 4, 5 };
2  int[] dst = new int[5];
3  Array.Copy(src, dst, 3); // dst: { 1, 2, 3, 0, 0 }
```

З вказанням початкових позицій:

```
C#
1  // Array.Copy(src, srcIndex, dst, dstIndex, length)
2  Array.Copy(src, 1, dst, 2, 3); // копіювати src[1..3] у dst[2..4]
3  // src: { 1, 2, 3, 4, 5 }
4  // dst: { 0, 0, 2, 3, 4 }
```

.Clone() повертає **поверхневу копію** (shallow copy) масиву як object. Для масивів значущих типів (int[], double[]) це повна незалежна копія. Для масивів посилальних типів (PatientRecord[]) копіюються **посилання**, не самі об'єкти — зміна поля через dst[0].Name = "..." вплине на обидва масиви:

```
C#
```

```
1  int[] a = { 1, 2, 3 };
2  int[] b = (int[])a.Clone();
3  b[0] = 99;
4  Console.WriteLine(a[0]); // 1 — незалежні копії для int[]
```

Array.Resize — новий масив під капотом

Array.Resize(ref arr, newSize) змінює розмір масиву. Але під капотом це **не розширення існуючого масиву** — .NET виділяє нову ділянку пам'яті, копіює дані і оновлює посилання arr через ref:

```
C#
1  int[] arr = { 1, 2, 3 };
2  Array.Resize(ref arr, 6);
3  // arr: { 1, 2, 3, 0, 0, 0 }
```

Якщо newSize < arr.Length — масив обрізається, зайві елементи губляться.

Важливе наслідство: якщо є інша змінна, що вказувала на старий масив, вона **залишається вказувати на стару ділянку** пам'яті після Resize. Саме тому параметр ref — обов'язковий, щоб оновити хоча б оригінальну змінну. Якщо потрібна динамічна колекція без цих обмежень — краще використовувати List<T>.

Заповнення та очищення

Array.Fill(arr, value) (.NET Core 2.1+) заповнює весь масив заданим значенням:

```
C#
1  int[] slots = new int[5];
2  Array.Fill(slots, -1); // { -1, -1, -1, -1, -1 }
```

З вказанням діапазону:

```
C#
1  Array.Fill(slots, 0, 2, 3); // заповнити 3 елементи починаючи з індексу 2
2  // slots: { -1, -1, 0, 0, 0 }
```

Array.Clear(arr, index, length) заповнює елементи **значенням за замовчуванням**: 0 для числових, false для bool, null для посилальних типів:

```
C#
1  int[] data = { 10, 20, 30, 40, 50 };
2  Array.Clear(data, 1, 3);
```

```
3  // data: { 10, 0, 0, 0, 50 }
```

В .NET 6+ з'явився Array.Clear(arr) без параметрів — очищення всього масиву.

Сортування та пошук пацієнтів — runnable приклад

```
C#
 1  using System;
 2
 3  var patients = new Patient[]
 4  {
 5      new Patient { Id="P001", Name="Петренко Іван",   Age=67, Bmi=27.3,
    Icd="I10.9" },
 6      new Patient { Id="P002", Name="Коваль Марія",    Age=45, Bmi=23.1,
    Icd="J45.0" },
 7      new Patient { Id="P003", Name="Сидоренко Олег",  Age=52, Bmi=31.8,
    Icd="E11.9" },
 8      new Patient { Id="P004", Name="Бойко Тетяна",    Age=71, Bmi=19.4,
    Icd="I10.9" },
 9      new Patient { Id="P005", Name="Мороз Василь",    Age=38, Bmi=25.6,
    Icd="J45.0" },
10  };
11
12  Console.WriteLine("=== Сортування за віком (спадно) ===");
13  Array.Sort(patients, (a, b) => b.Age.CompareTo(a.Age));
14  Console.WriteLine($"{"ID",-6} {"Пацієнт",-22} {"Вік",4} {"IMT",6}  ICD");
15  Console.WriteLine(new string('-', 48));
16  foreach (var p in patients)
17      Console.WriteLine($"{p.Id,-6} {p.Name,-22} {p.Age,4} {p.Bmi,6:F1}
    {p.Icd}");
18
19  Console.WriteLine("\n=== Сортування за IMT (зростаючи) ===");
20  Array.Sort(patients, (a, b) => a.Bmi.CompareTo(b.Bmi));
21  foreach (var p in patients)
22      Console.WriteLine($"{p.Id,-6} {p.Name,-22} IMT={p.Bmi:F1}");
23
24  Console.WriteLine("\n=== Пошук: FindAll за діагнозом I10.9 ===");
25  var hypertensive = Array.FindAll(patients, p => p.Icd == "I10.9");
26  Console.WriteLine($"Гіпертонія (I10.9): {hypertensive.Length} пацієнтів");
27  foreach (var p in hypertensive)
28      Console.WriteLine($"  {p.Name}, {p.Age} р.");
29
30  Console.WriteLine("\n=== Exists / TrueForAll ===");
31  bool hasObesity  = Array.Exists(patients, p => p.Bmi >= 30);
32  bool allAdults   = Array.TrueForAll(patients, p => p.Age >= 18);
33  Console.WriteLine($"Є ожиріння (IMT≥30): {hasObesity}");
34  Console.WriteLine($"Усі повнолітні:       {allAdults}");
35
36  Console.WriteLine("\n=== Find + FindIndex ===");
37  var firstSenior = Array.Find(patients, p => p.Age >= 65);
38  int seniorIdx   = Array.FindIndex(patients, p => p.Age >= 65);
39  Console.WriteLine($"Перший пацієнт 65+: {firstSenior.Name} (індекс
    {seniorIdx})");
40
41  struct Patient
42  {
```

```
43      public string Id;
44      public string Name;
45      public int    Age;
46      public double Bmi;
47      public string Icd;
48  }
```

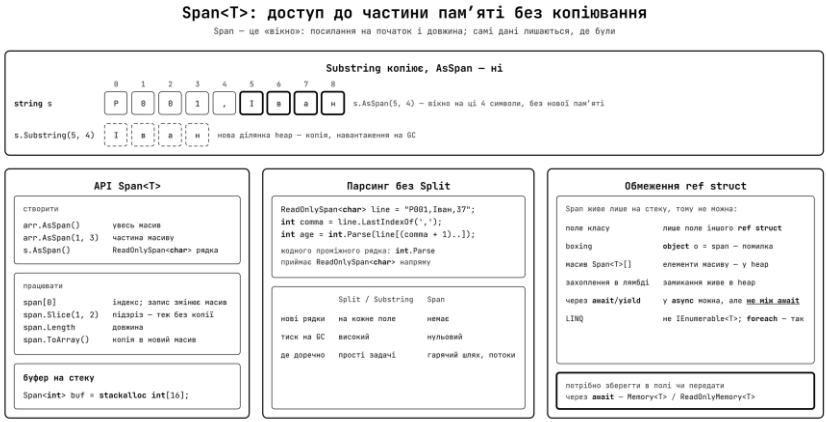
Копіювання, Fill та IndexOf — runnable приклад

```
C#
 1  using System;
 2
 3  Console.WriteLine("=== Array.Copy ===");
 4  string[] source = { "P001", "P002", "P003", "P004", "P005" };
 5  string[] backup = new string[5];
 6
 7  Array.Copy(source, backup, 3);
 8  Console.WriteLine("Source: " + string.Join(", ", source));
 9  Console.WriteLine("Backup: " + string.Join(", ", backup));
10
11  Console.WriteLine("\n=== Array.IndexOf ===");
12  int idx = Array.IndexOf(source, "P003");
13  int bad = Array.IndexOf(source, "P999");
14  Console.WriteLine($"IndexOf(\"P003\") = {idx}");
15  Console.WriteLine($"IndexOf(\"P999\") = {bad}  // не знайдено");
16
17  Console.WriteLine("\n=== Array.BinarySearch (потрібен відсортований масив)
    ===");
18  int[] ages = { 38, 45, 52, 67, 71 }; // вже відсортований
19  int found   = Array.BinarySearch(ages, 52);
20  int missing = Array.BinarySearch(ages, 50);
21  Console.WriteLine($"BinarySearch(52) = {found}   // знайдено на індексі
    {found}");
22  Console.WriteLine($"BinarySearch(50) = {missing}  // не знайдено, позиція
    вставки = {~missing}");
23
24  Console.WriteLine("\n=== Array.Fill ===");
25  int[] slots = new int[6];
26  Array.Fill(slots, -1);
27  Console.WriteLine("Fill(-1):   " + string.Join(", ", slots));
28  Array.Fill(slots, 0, 2, 3);
29  Console.WriteLine("Fill(0,2,3):" + string.Join(", ", slots));
30
31  Console.WriteLine("\n=== Array.Clear ===");
32  int[] data = { 10, 20, 30, 40, 50 };
33  Console.WriteLine("До Clear:   " + string.Join(", ", data));
34  Array.Clear(data, 1, 3);
35  Console.WriteLine("Після Clear:" + string.Join(", ", data));
36
37  Console.WriteLine("\n=== Array.Resize ===");
38  string[] ids = { "P001", "P002", "P003" };
39  Console.WriteLine($"До Resize:   Length={ids.Length}, дані={string.Join(", ",
    ids)}");
40  Array.Resize(ref ids, 5);
41  Console.WriteLine($"Після Resize:Length={ids.Length}, дані={string.Join(", ",
    ids ?? new string[0])}");
42  ids[3] = "P004"; ids[4] = "P005";
43  Console.WriteLine($"Після заповн:дані={string.Join(", ", ids)}");
```

13.5. Span<T> та ReadOnlySpan<T>

Коли медична інформаційна система обробляє тисячі CSV-рядків із результатами аналізів або парсить бінарний потік HL7 MLLP, кожен виклик string.Substring() або arr[2..5] виділяє нову ділянку пам'яті у heap і копіює туди дані. При великих обсягах — мільйони дрібних об'єктів, надмірне навантаження на збирач сміття (GC), непередбачувані паузи.

Span<T> вирішує цю проблему принципово: він не виділяє пам'ять і не копіює дані — він лише **вказує на вже існуючу ділянку** пам'яті (масив, рядок, stackalloc-буфер) і зберігає довжину. Весь доступ до даних відбувається через цей вказівник без жодного копіювання.



Span<T> — безкопійний доступ до пам'яті

Проблема копіювання: Substring vs Span

Розглянемо рядок "P001,Іван,67,I10.9". Щоб отримати поле "Іван" традиційним способом:

```
C#
1  string line = "P001,Іван,67,I10.9";
2  string name = line.Substring(5, 4); // нова ділянка heap + копіювання 4 символів
```

Substring **завжди** створює новий рядок на heap. При парсингу 100 000 рядків — 100 000 дрібних рядків у heap, щоразу виділяються і збираються GC.

З ReadOnlySpan<char>:

```
C#
1  ReadOnlySpan<char> span = line.AsSpan(5, 4);
2  // heap не виділяється — span вказує на символи [5..8] оригінального рядка
```

span живе на **стеку** і містить лише два значення: вказівник на пам'ять і довжину. Оригінальний рядок не змінюється, копіювання не відбувається.

Що таке Span<T>

Span<T> — це ref struct, тобто структура, яка **завжди знаходиться на стеку**. Вона може вказувати на:

- масив: new int[]{1,2,3}.AsSpan()
- частину масиву: arr.AsSpan(2, 5)
- рядок: str.AsSpan()
- буфер на стеку: stackalloc int[16]
- неупорядковану пам'ять (через MemoryMarshal)

```
C#
1  int[] data = { 10, 20, 30, 40, 50 };
2
3  Span<int> all    = data.AsSpan();         // весь масив
4  Span<int> middle = data.AsSpan(1, 3);     // { 20, 30, 40 } без копії
5
6  middle[0] = 99; // змінює data[1]!
7  Console.WriteLine(data[1]); // 99 — Span<T> є вікном у оригінальні дані
```

Span<T> — **змінний**: через нього можна не лише читати, але й записувати дані. Запис через Span змінює оригінальну ділянку пам'яті.

ReadOnlySpan<T> — незмінний варіант

ReadOnlySpan<T> — аналог Span<T>, але тільки для читання. Рядки C# незмінні, тому string.AsSpan() повертає саме ReadOnlySpan<char>:

```
C#
1  string s = "Петренко Іван";
2  ReadOnlySpan<char> span = s.AsSpan();
3
4  char first = span[0];   // 'П'
5  // span[0] = 'X';       // помилка компіляції — ReadOnly
```

ReadOnlySpan<char> є найпоширенішим застосуванням Span: парсинг рядків без виділення підрядків.

API Span<T>

Індексація та довжина:

```
C#
1  Span<int> span = new int[]{ 10, 20, 30, 40, 50 }.AsSpan();
2
3  int val    = span[2];    // 30
4  span[2]    = 99;         // запис
5  int length = span.Length; // 5
```

Slice — безкопійний підзріз:

```
C#
1  Span<int> sub = span.Slice(1, 3); // { 20, 99, 40 } — вказівник, не копія
```

CopyTo — копіювання у масив або інший Span:

```
C#
1  int[] dest = new int[3];
2  span.Slice(1, 3).CopyTo(dest); // копіювати 3 елементи у dest
```

Fill та Clear:

```
C#
1  span.Fill(0);  // заповнити нулями
2  span.Clear();  // теж 0 (для числових), false для bool, null для ref
```

SequenceEqual — порівняння вмісту:

```
C#
1  ReadOnlySpan<char> a = "hello".AsSpan();
2  ReadOnlySpan<char> b = "hello".AsSpan();
3  Console.WriteLine(a.SequenceEqual(b)); // true
```

Парсинг рядків через ReadOnlySpan<char>

Ключова перевага ReadOnlySpan<char> — методи int.TryParse, double.TryParse та інші приймають ReadOnlySpan<char> як аргумент, дозволяючи парсити **без виділення підрядка**:

```
C#
1  ReadOnlySpan<char> numSpan = "67".AsSpan();
2  int.TryParse(numSpan, out int age); // age = 67, heap не виділявся
```

Для парсингу CSV-рядка поле за полем:

```
C#
 1  ReadOnlySpan<char> line = "P001,Іван,67,I10.9".AsSpan();
 2
 3  int comma1 = line.IndexOf(',');                          // 4
 4  ReadOnlySpan<char> id   = line.Slice(0, comma1);         // "P001"
 5  ReadOnlySpan<char> rest = line.Slice(comma1 + 1);        // "Іван,67,I10.9"
 6
 7  int comma2 = rest.IndexOf(',');
 8  ReadOnlySpan<char> name = rest.Slice(0, comma2);         // "Іван"
 9  rest = rest.Slice(comma2 + 1);                           // "67,I10.9"
10
11  int comma3 = rest.IndexOf(',');
12  ReadOnlySpan<char> ageSpan = rest.Slice(0, comma3);      // "67"
13  int.TryParse(ageSpan, out int age);                      // 67
```

Жоден новий рядок на heap не виділяється — лише вказівники на ділянки оригінального буфера.

Обмеження ref struct

Span<T> є ref struct — це технічне обмеження, що забезпечує гарантію перебування на стеку. Наслідки:

- **Не можна зберігати в полі класу**: ref struct не може бути частиною heap-об'єкта. Полем він може бути лише в іншому ref struct.
- **Не можна тримати через await або yield**: локальні змінні async-методу та ітератора після await/yield переносяться в об'єкт стану на heap, а ref struct туди потрапити не може. До C# 12 Span<T> узагалі не можна було оголосити в async-методі. Починаючи з C# 13, це дозволено, але лише на ділянці коду **між** await: використати Span<T> до await можна, а звернутися до нього після — помилка компіляції CS4007.
- **Не можна боксувати**: object o = span — помилка компіляції. З тієї ж причини Span<T> не можна привести до інтерфейсу.
- **Не можна в масиві**: Span<T>[] неможливо — елементи масиву лежать у heap.
- **Не можна захопити в лямбді**: замикання компілятор перетворює на об'єкт у heap.
- **LINQ не підтримується**: більшість LINQ-методів приймають IEnumerable<T>, але Span<T> не реалізує цей інтерфейс. foreach — працює.

Якщо потрібно **зберегти посилання на дані між викликами методів, у полі класу або через await** — використовується Memory<T> (змінний) або ReadOnlyMemory<T> (незмінний). Memory<T> — звичайна структура (не ref struct), яка зберігає посилання на масив (або інший буфер), зсув і довжину. Сам Span<T> вона всередині не тримає — це було б неможливо, — а створює його на вимогу через властивість .Span саме там, де з даними треба попрацювати.

Парсинг CSV без Split — runnable приклад

Демонстрація ReadOnlySpan<char> для парсингу рядка без виділення підрядків:

```C#
using System;

static void ParsePatientLine(ReadOnlySpan<char> line)
{
    int c1 = line.IndexOf(',');
    ReadOnlySpan<char> id = line.Slice(0, c1);
    line = line.Slice(c1 + 1);

    int c2 = line.IndexOf(',');
    ReadOnlySpan<char> name = line.Slice(0, c2);
    line = line.Slice(c2 + 1);

    int c3 = line.IndexOf(',');
    ReadOnlySpan<char> ageSpan = line.Slice(0, c3);
    ReadOnlySpan<char> icd = line.Slice(c3 + 1);

    int.TryParse(ageSpan, out int age);

    Console.WriteLine($"  ID:   {id.ToString()}");
    Console.WriteLine($"  Ім'я: {name.ToString()}");
    Console.WriteLine($"  Вік:  {age}");
    Console.WriteLine($"  ICD:  {icd.ToString()}");
}

string[] csvLines = {
    "P001,Петренко Іван,67,I10.9",
    "P002,Коваль Марія,45,J45.0",
    "P003,Сидоренко Олег,52,E11.9",
};

Console.WriteLine("=== Парсинг CSV через ReadOnlySpan<char> ===");
Console.WriteLine("(підрядки НЕ виділяються у heap)\n");

foreach (string line in csvLines)
{
    Console.WriteLine($"Рядок: \"{line}\"");
    ParsePatientLine(line.AsSpan());
    Console.WriteLine();
}

```

```
Console.WriteLine("=== SequenceEqual та IndexOf ===");
ReadOnlySpan<char> icd1 = "I10.9".AsSpan();
ReadOnlySpan<char> icd2 = "I10.9".AsSpan();
ReadOnlySpan<char> icd3 = "J45.0".AsSpan();

Console.WriteLine($"icd1 == icd2: {icd1.SequenceEqual(icd2)}");
Console.WriteLine($"icd1 == icd3: {icd1.SequenceEqual(icd3)}");
Console.WriteLine($"IndexOf('.') у \"I10.9\": {icd1.IndexOf('.')}");
```

Span<double> для підмасиву вимірювань — runnable приклад

```
C#
using System;

Console.WriteLine("=== Span<double>: робота з підмасивом ===");

double[] pressures = { 120, 135, 128, 142, 118, 155, 130, 122, 145, 138 };

Console.WriteLine("Всі вимірювання (мм рт. ст.):");
Console.WriteLine("  " + string.Join(", ", pressures));

Span<double> lastFive = pressures.AsSpan(5, 5);
Console.WriteLine("\nОстанні 5 вимірювань (Span без копії):");
Console.Write("  ");
for (int i = 0; i < lastFive.Length; i++)
    Console.Write($"{lastFive[i]}{(i < lastFive.Length - 1 ? ", " : "\n")}");

double sum = 0;
for (int i = 0; i < lastFive.Length; i++)
    sum += lastFive[i];
Console.WriteLine($"Середній тиск (останні 5): {sum / lastFive.Length:F1} мм
рт. ст.");

Console.WriteLine("\n=== Fill та запис через Span ===");
int[] slots = new int[8];
Span<int> span = slots.AsSpan();
span.Fill(-1);
Console.WriteLine("Після Fill(-1): " + string.Join(", ", slots));

Span<int> middle = span.Slice(2, 4);
middle.Fill(0);
Console.WriteLine("Після Slice(2,4).Fill(0): " + string.Join(", ", slots));

int[] copy = new int[4];
middle.CopyTo(copy);
Console.WriteLine("CopyTo нового масиву: " + string.Join(", ", copy));

Console.WriteLine("\n=== Slice не копіює — перевірка ===");
int[] src = { 1, 2, 3, 4, 5 };
Span<int> sliced = src.AsSpan(1, 3);
sliced[0] = 99;
Console.WriteLine($"Після sliced[0]=99: src = [{string.Join(", ", src)}]");
Console.WriteLine("src[1] змінився — Span є вікном, не копією");
```

13.6. Індекси та діапазони (Index та Range)

До C# 8 отримати останній елемент масиву виглядало так: arr[arr.Length - 1]. Передостанній — arr[arr.Length - 2]. Взяти підмасив від 2-го до 5-го: arr.Skip(2).Take(3).ToArray() або ручний цикл. Ці операції прості концептуально, але громіздкі в записі і схильні до помилок «на одиницю».

C# 8 (.NET Core 3.0+) ввів два нових типи — System.Index і System.Range — та відповідний синтаксис: оператор ^ для індексування від кінця і оператор .. для діапазонів. Разом вони роблять роботу з послідовностями значно лаконічнішою.



Index та Range — індексування від кінця та діапазони

Тип Index та оператор ^

System.Index — структура, що представляє позицію в послідовності. Вона може бути **прямою** (від початку, як звичайний int) або **зворотньою** (від кінця, через оператор ^):

```
C#
1  Index i0 = 2;    // третій елемент (звичайний індекс)
2  Index i1 = ^1;   // останній елемент
3  Index i2 = ^2;   // передостанній
```

Оператор ^n означає «n-й від кінця», де **^1 — останній**, ^2 — передостанній і т. д. ^0 рівний .Length — це позиція **за межею масиву**, тому arr[^0] дасть IndexOutOfRangeException:

```
C#
1  string[] names = { "Петренко", "Коваль", "Бойко", "Сидоренко" };
2
3  Console.WriteLine(names[^1]); // "Сидоренко" — останній
4  Console.WriteLine(names[^2]); // "Бойко"     — передостанній
5  Console.WriteLine(names[0]);  // "Петренко"  — перший (прямий індекс)
```

Index можна зберігати у змінній і передавати у методи:

```
C#
1  Index last = ^1;
2  Console.WriteLine(names[last]); // "Сидоренко"
```

Index.GetOffset(length) перетворює ^-індекс у числовий: ^1.GetOffset(4) == 3.

Тип Range та оператор ..

System.Range — структура, що представляє діапазон індексів. Синтаксис a..b означає «від a включно до b **виключно**». Обидві межі можуть бути звичайними індексами або ^-індексами:

```
C#
1  double[] pressures = { 120, 135, 128, 142, 118, 155, 130 };
2
3  double[] first3 = pressures[0..3];    // { 120, 135, 128 } — індекси 0,1,2
4  double[] mid    = pressures[2..5];    // { 128, 142, 118 } — індекси 2,3,4
5  double[] last3  = pressures[^3..];    // { 155, 130, ? } — від ^3 до кінця
6  double[] all    = pressures[..];      // весь масив
```

Межі .. є необов'язковими: arr[..n] — перші n елементів (від 0), arr[n..] — від n до кінця, arr[..] — весь масив.

Поєднання ^ і .. робить типові операції дуже виразними:

```
C#
1  double[] withoutFirst  = pressures[1..];    // відкинути перший
2  double[] withoutLast   = pressures[..^1];   // відкинути останній
3  double[] inner         = pressures[1..^1];  // без першого і останнього
4  double[] lastFive      = pressures[^5..];   // останні 5
```

Range можна зберігати у змінній

Range — це значущий тип (struct), його можна зберігати і передавати:

```
C#
1  Range recent = ^5..;       // останні 5
2  Range inner  = 1..^1;      // без крайніх
3
4  double[] recentReadings = pressures[recent];
```

```
5  double[] innerReadings  = pressures[inner];
```

Range.GetOffsetAndLength(length) повертає кортеж (offset, length) — числові значення для ручного використання, якщо API не підтримує Range безпосередньо:

```
C#
1  Range r = 1..^1;
2  var (offset, length) = r.GetOffsetAndLength(pressures.Length);
3  // offset=1, length=5 (для масиву з 7 елементів)
```

Застосування до рядків

Оператор [..] працює зі string так само, як Substring, але значно лаконічніше:

```
C#
1  string icd = "I10.9";
2
3  char   first  = icd[0];       // 'I'
4  char   last   = icd[^1];      // '9'
5  string code   = icd[1..];     // "10.9" — без першого символу
6  string digits = icd[1..^2];   // "10"   — код без літери і дрібниці
7  string ext    = icd[^3..];    // ".9" — три символи з кінця
```

Результат str[a..b] — це **новий рядок** (heap-копія), тобто семантично еквівалентний str.Substring(a, b - a). Для безкопійного варіанту — str.AsSpan()[a..b], що повертає ReadOnlySpan<char>.

Range та масиви: завжди нова копія

На відміну від Span, при застосуванні Range до масиву (T[]) завжди створюється **новий масив**:

```
C#
1  int[] src  = { 1, 2, 3, 4, 5 };
2  int[] copy = src[1..4]; // новий масив { 2, 3, 4 }
3
4  copy[0] = 99;
5  Console.WriteLine(src[1]); // 2 — src не змінився
```

Якщо потрібна безкопійна вибірка — використовуйте Span<T> або ReadOnlySpan<T>:

```
C#
1  Span<int> slice = src.AsSpan()[1..4]; // вказівник, не копія
2  slice[0] = 99;
3  Console.WriteLine(src[1]); // 99 — оригінал змінився
```

Вибір між arr[a..b] (нова копія) і span[a..b] (без копії) залежить від сценарію: якщо результат зберігається надовго або передається куди-небудь — копія доречна; якщо обробляється тут-і-зараз у межах методу — Span ефективніший.

List<T> та інші колекції

Index і Range підтримуються **не всіма** колекціями. T[] і string підтримують повністю. Span<T> і ReadOnlySpan<T> — повністю. List<T> підтримує Index (list[^1]) і, починаючи з .NET 8, також Range: list[1..3] компілюється, бо в List<T> з'явився метод Slice(start, length), який компілятор викликає для діапазонів. Результат — **новий** List<T> (копія), як і для масивів. У старіших версіях .NET для зрізів використовують .GetRange(index, count) — він робить те саме.

Підтримку визначає не «тип колекції», а наявність потрібних членів. Для ^ потрібні властивість Length або Count та індексатор за int. Для .. потрібен метод Slice(int, int). Тому через інтерфейс IList<T> індекс від кінця (list[^1]) працює, а діапазон — ні: методу Slice в інтерфейсі немає.

Вимірювання тиску: Index та Range — runnable приклад

```
C#
 1  using System;
 2
 3  Console.WriteLine("=== Вимірювання артеріального тиску ===");
 4  double[] systolic = { 145, 138, 152, 130, 147, 135, 128, 142, 155, 133 };
 5
 6  Console.WriteLine("Всі вимірювання:");
 7  Console.WriteLine("  " + string.Join(", ", systolic));
 8
 9  Console.WriteLine($"\nПерше:        {systolic[0]}");
10  Console.WriteLine($"Останнє:      {systolic[^1]}");
11  Console.WriteLine($"Передостаннє: {systolic[^2]}");
12
13  Console.WriteLine("\nОстанні 3 вимірювання (^3..):");
14  double[] last3 = systolic[^3..];
15  Console.WriteLine("  " + string.Join(", ", last3));
16
17  Console.WriteLine("\nБез крайніх (1..^1) — прибираємо першу і останню точку:");
18  double[] inner = systolic[1..^1];
19  Console.WriteLine("  " + string.Join(", ", inner));
20
21  double sum = 0;
22  for (int i = 0; i < inner.Length; i++) sum += inner[i];
23  Console.WriteLine($"  Середнє (без крайніх): {sum / inner.Length:F1}");
24
25  Console.WriteLine("\n=== Рядки: ICD-10 ===");
26  string[] icds = { "I10.9", "J45.0", "E11.9", "K21.0" };
27  foreach (string icd in icds)
```

```
{
    char   letter = icd[0];
    string num    = icd[1..^2];
    string sub    = icd[^1..];
    Console.WriteLine($"  {icd}  → клас={letter}  код={num}  підклас={sub}");
}

Console.WriteLine("\n=== Range у змінній ===");
Range recentRange = ^5..;
double[] recent5 = systolic[recentRange];
Console.WriteLine($"recent5 = [{string.Join(", ", recent5)}]");

var (offset, length) = recentRange.GetOffsetAndLength(systolic.Length);
Console.WriteLine($"GetOffsetAndLength: offset={offset}, length={length}");
```

Range зі Span та GetOffsetAndLength — runnable приклад

```
C#
using System;

Console.WriteLine("=== Span<double> з Range — без копіювання ===");
double[] glucose = { 5.1, 6.3, 7.8, 5.9, 6.1, 8.2, 5.4, 6.7 };

Console.WriteLine("Всі вимірювання глюкози:");
Console.WriteLine("  " + string.Join(", ", glucose));

// Span-зріз без копії
Span<double> spanAll   = glucose.AsSpan();
Span<double> spanInner = spanAll[1..^1]; // без першого і останнього

Console.WriteLine($"\nSpan[1..^1] (без крайніх), довжина =
{spanInner.Length}:");
Console.Write("  ");
for (int i = 0; i < spanInner.Length; i++)
    Console.Write($"{spanInner[i]}{(i < spanInner.Length - 1 ? ", " : "\n")}");

// Зміна через Span змінює оригінальний масив
spanInner[0] = 6.0;
Console.WriteLine($"Після spanInner[0]=6.0: glucose[1] = {glucose[1]}");

Console.WriteLine("\n=== Масив vs Span: копія чи вказівник ===");
int[] data = { 10, 20, 30, 40, 50 };

int[] arrSlice  = data[1..4];      // нова копія
Span<int> spanSlice = data.AsSpan()[1..4]; // вказівник

arrSlice[0]  = 99;
spanSlice[1] = 88;

Console.WriteLine($"data після arrSlice[0]=99:   [{string.Join(", ", data)}]");
Console.WriteLine($"  data[1] = {data[1]} (не змінився — копія)");
Console.WriteLine($"data після spanSlice[1]=88:  [{string.Join(", ", data)}]");
Console.WriteLine($"  data[2] = {data[2]} (змінився — вказівник)");

Console.WriteLine("\n=== Index: GetOffset ===");
Index last  = ^1;
Index third = ^3;
Console.WriteLine($"^1 у масиві з 5 елементів → індекс {last.GetOffset(5)}");
```

```
40  Console.WriteLine($"^3 у масиві з 5 елементів → індекс {third.GetOffset(5)}");
41
42  Console.WriteLine("\n=== Range: GetOffsetAndLength ===");
43  Range[] ranges = { .., 1..^1, ^3.., 2..4 };
44  string[] labels = { "..", "1..^1", "^3..", "2..4" };
45  for (int i = 0; i < ranges.Length; i++)
46  {
47      var (off, len) = ranges[i].GetOffsetAndLength(data.Length);
48      Console.WriteLine($"  {labels[i],-8} → offset={off}, length={len}");
49  }
```

Питання для самоконтролю — Розділ 13. Додаткові класи та структури .NET

13.1. Відкладена ініціалізація та тип Lazy<T>

1. Що таке «відкладена ініціалізація» (lazy initialization) і яку проблему вона вирішує порівняно з «ранньою ініціалізацією» (eager initialization)?
2. Що таке Lazy<T>? Коли саме відбувається ініціалізація обгорнутого об'єкта — при створенні Lazy<T> чи при першому зверненні до .Value?
3. Що виведе наступний код? Поясніть порядок виконання:

```
C#
1  Lazy<Library> lib = new Lazy<Library>();
2  Console.WriteLine(lib.IsValueCreated);
3  var x = lib.Value;
4  Console.WriteLine(lib.IsValueCreated);
5  var y = lib.Value;
6  Console.WriteLine(object.ReferenceEquals(x, y));
```

4. Як передати параметри в конструктор об'єкта, що обгортається Lazy<T>? Напишіть приклад зі лямбда-виразом, де потрібний параметр patientId.
5. Для чого призначена властивість IsValueCreated? Назвіть практичний сценарій, де важливо перевірити стан без запуску ініціалізації.
6. Поясніть три режими потокової безпеки LazyThreadSafetyMode: ExecutionAndPublication, PublicationOnly і None. В яких умовах кожен із них є правильним вибором?
7. Спроєктуйте клас DiagnosticService, у якому словник нормативних значень лабораторних показників завантажується

лише при першому зверненні. Використайте Lazy<T> зі статичним полем.

8. Порівняйте Lazy<T> і ручну реалізацію відкладеної ініціалізації через null-перевірку:

```
C#
1  private Library? _lib;
2  public Library Lib => _lib ??= new Library();
```

Яка різниця в поведінці при конкурентному доступі з кількох потоків?

13.2. Клас Math

9. Чому Math є static sealed class? Які архітектурні переваги дає таке рішення?
10. Пояснити різницю між Math.Floor, Math.Ceiling і Math.Truncate для від'ємних чисел. Що виведе наступний код?

```
C#
1  Console.WriteLine(Math.Floor(-2.7));
2  Console.WriteLine(Math.Ceiling(-2.1));
3  Console.WriteLine(Math.Truncate(-2.9));
```

11. Що таке «банківське заокруглення» і чому Math.Round(2.5) повертає 2, а не 3? Як отримати «шкільне» заокруглення для значень x.5?
12. Що виведе цей код і чому результат може здивувати?

```
C#
1  Console.WriteLine(Math.Round(2.5));
2  Console.WriteLine(Math.Round(3.5));
3  Console.WriteLine(Math.Round(4.5));
```

13. Що таке Math.Clamp(value, min, max) і яку задачу він вирішує? Перепишіть наступний код з використанням Clamp:

```
C#
1  if (heartRate < 40) heartRate = 40;
2  else if (heartRate > 200) heartRate = 200;
```

14. Напишіть функцію обчислення ІМТ (BMI = вага / зріст²). Додайте захист від ділення на нуль або некоректних вхідних даних за допомогою double.IsFinite.

15. Чому Math.Cbrt(-8) повертає -2, а Math.Pow(-8, 1.0/3.0) повертає NaN? Що таке NaN і як воно «поширюється» у виразах?
16. Усі тригонометричні функції Math приймають аргументи в радіанах. Напишіть вираз для перетворення 45 градусів у радіани і обчислення sin(45°). Яким повинен бути результат?

13.3. Клас Convert

17. В чому різниця між int.Parse(s) і Convert.ToInt32(s)? В яких сценаріях кожен метод є кращим вибором?
18. Що виведе наступний код? Поясніть результат:

```
C#
1  double d = 3.9;
2  Console.WriteLine((int)d);
3  Console.WriteLine(Convert.ToInt32(d));
4
5  double half = 2.5;
6  Console.WriteLine((int)half);
7  Console.WriteLine(Convert.ToInt32(half));
```

19. Чому Convert.ToInt32(null) повертає 0, а не кидає NullReferenceException? В якому сценарії ця поведінка корисна?
20. Що кине цей код і як правильно його переписати?

```
C#
1  string s = "abc";
2  int val = Convert.ToInt32(s);
```

21. Що таке Base64-кодування і де воно застосовується у медичній інформатиці (HL7 FHIR)? Напишіть код, який кодує рядок у Base64 і декодує назад.
22. Порівняйте Convert.ToString(null) і null?.ToString(). Яка різниця в результаті? В яких умовах перша форма безпечніша?
23. Що таке Convert.ChangeType(value, type) і коли він є незамінним? Наведіть приклад сценарію, де тип перетворення відомий лише під час виконання.
24. Який із методів слід використати в кожному сценарії і чому?

- Зчитати вік пацієнта з поля форми (рядок, введений користувачем) - Конвертувати bool у int для збереження у БД - Отримати DateTime з рядка невідомого формату - Перетворити long у int з перевіркою переповнення

13.4. Клас Array

25. Чому кожен масив int[] або string[] є об'єктом класу System.Array? Які властивості це надає будь-якому масиву?
26. Array.Sort(arr) виконує сортування на місці (in-place) — що це означає для вихідного масиву? Як відсортувати масив об'єктів за полем Age у спадному порядку?
27. В чому різниця між Array.IndexOf і Array.BinarySearch? Що повернає BinarySearch, якщо елемент не знайдено, і як це значення інтерпретувати?
28. Що виведе наступний код?

```
C#
1  int[] ages = { 38, 45, 52, 67, 71 }; // вже відсортований
2  int result = Array.BinarySearch(ages, 50);
3  Console.WriteLine(result);
4  Console.WriteLine(~result);
```

29. В чому різниця між Array.Find і Array.FindAll? Що повертає Find, якщо умова не виконується жодним елементом?
30. Що таке «поверхнева копія» (shallow copy)? Чим Array.Copy відрізняється від .Clone() при роботі з масивами посилальних типів?
31. Що відбувається «під капотом» при виклику Array.Resize(ref arr, newSize)? Чому параметр обов'язково ref? Які ризики при наявності кількох змінних, що вказують на один масив?
32. Спроєктуйте функцію, яка приймає масив пацієнтів і повертає тільки тих, хто старший за 65 років (масив нащадків). Використайте Array.FindAll. Потім перевірте через Array.TrueForAll, чи всі знайдені пацієнти дійсно старші за 65.

13.5. Span<T> та ReadOnlySpan<T>

33. Яку проблему вирішує Span<T>? Чому string.Substring() у великих циклах може бути проблематичним?
34. Що означає, що Span<T> є ref struct? Які конкретні обмеження накладає цей факт і чому вони введені?
35. Що виведе наступний код? Чи зміниться оригінальний масив?

```
C#
1  int[] data = { 1, 2, 3, 4, 5 };
2  Span<int> slice = data.AsSpan(1, 3);
```

```
3  slice[0] = 99;
4  Console.WriteLine(data[1]);
```

36. В чому різниця між Span<T> (змінний) і ReadOnlySpan<T> (незмінний)? Чому string.AsSpan() повертає саме ReadOnlySpan<char>?
37. Поясніть, як int.TryParse(ReadOnlySpan<char>, out int) дозволяє парсити число без виділення підрядка в heap. Чому це важливо при обробці великих CSV-файлів?
38. Порівняйте парсинг CSV-рядка через string.Split(",") і через ручне ітерування з ReadOnlySpan<char> та IndexOf. Яка різниця в алокаціях пам'яті?
39. Коли слід використовувати Memory<T> замість Span<T>? В яких ситуаціях Span технічно неможливо застосувати?
40. Напишіть метод static double Average(ReadOnlySpan<double> values), який обчислює середнє значення без виділення нових об'єктів. Поясніть, чому параметр ReadOnlySpan<double> є кращим вибором ніж double[].

13.6. Індекси та діапазони (Index та Range)

41. Що таке оператор ^ у контексті System.Index? Що означає ^1, ^2 і ^0? Що станеться при arr[^0]?
42. Що означає оператор .. при роботі з масивами? Чи включає права межа останній елемент?
43. Що виведе наступний код?

```
C#
1  int[] arr = { 10, 20, 30, 40, 50 };
2  int[] a = arr[1..^1];
3  int[] b = arr[^3..];
4  Console.WriteLine(string.Join(", ", a));
5  Console.WriteLine(string.Join(", ", b));
```

44. В чому різниця між arr[1..4] (Range на масиві) і arr.AsSpan()[1..4] (Range на Span)? Що відбувається з оригінальним масивом при зміні елемента зрізу в кожному випадку?
45. Чому list[1..3] не компілюється для List<T>, але list[^1] — компілюється? Як отримати підмножину List<T> за аналогічною логікою?

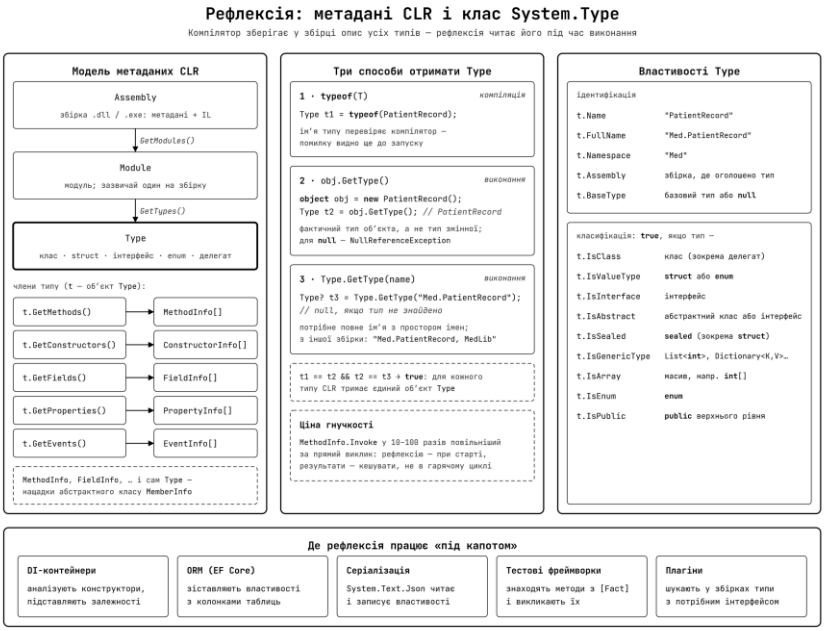
46. Як зберегти Range у змінній і передати у метод? Напишіть приклад з Range recentRange = ^5..; і покажіть, як через GetOffsetAndLength отримати числові значення.
47. Напишіть код, який розкладає ICD-10 код (наприклад, "I10.9") на три частини: перша літера, числовий код, і підклас після крапки — використовуючи тільки синтаксис індексів і діапазонів.
48. Порівняйте синтаксис str[1..^2] та str.Substring(1, str.Length - 3). Чи еквівалентні вони? Яка версія більш читабельна і менш схильна до помилок «на одиницю»?

Розділ 14. Рефлексія

14.1. Введення у рефлексію. Клас System.Type

Рефлексія є процесом виявлення типів під час виконання програми. Кожна програма містить набір використовуваних класів, інтерфейсів, а також їх методів, властивостей та інших цеглинок, з яких складається програма. І рефлексія дозволяє визначити всі ці складові елементи застосунку — причому не статично під час написання коду, а **динамічно під час виконання**.

Щоб зрозуміти, чому рефлексія взагалі можлива, потрібно усвідомити ключову особливість .NET: компілятор зберігає у скомпільованій збірці (.dll або .exe) не лише виконуваний код, а й **метадані** — детальний опис усіх типів, їхніх членів, параметрів методів, атрибутів. Це так звані **метадані CLR** (Common Language Runtime). Саме цей опис і читає рефлексія, відповідаючи на запитання: «які поля є у цього класу?», «які методи він має?», «які інтерфейси реалізує?».



Рефлексія: модель метаданих CLR та клас System.Type

Модель метаданих CLR

Збірка .dll або .exe — це не просто скомпільований байт-код. Вона має дворівневу структуру:

— **Метадані** — таблиці з описом усіх типів, методів, полів, властивостей, їх модифікаторів доступу, параметрів і атрибутів. CLR завантажує ці таблиці в пам'ять і надає до них доступ через System.Reflection.

— **IL-код** (Intermediate Language) — інструкції, що виконуються після JIT-компіляції. Метадані й IL зберігаються разом, тому рефлексія може не лише досліджувати тип, але й через MethodInfo.Invoke викликати його методи.

Ієрархія об'єктів рефлексії відображає структуру збірки:

```
Assembly (.dll)
  └── Module (зазвичай один на збірку)
        └── Type (клас, struct, інтерфейс, enum…)
              ├── MethodInfo[]      — методи
              ├── ConstructorInfo[] — конструктори
              ├── FieldInfo[]       — поля
              ├── PropertyInfo[]    — властивості
```

```
8                └── EventInfo[]       — події
```

Усі класи MethodInfo, FieldInfo, PropertyInfo, ConstructorInfo, EventInfo є нащадками абстрактного класу MemberInfo, який визначає загальний функціонал: ім'я члена, тип, що його оголошує, і набір атрибутів.

Де застосовується рефлексія

Рефлексія — це фундамент, на якому побудовані ключові інструменти сучасного .NET-розробника:

- **DI-контейнери** (Autofac, Microsoft.Extensions.DependencyInjection): сканують збірку на наявність типів, аналізують конструктори, визначають залежності і автоматично їх інжектують.
- **ORM** (Entity Framework Core): відображає властивості класу на колонки таблиці бази даних, генерує SQL-запити, читає і записує значення полів через рефлексію.
- **Серіалізація** (System.Text.Json, Newtonsoft.Json): перетворює об'єкт у JSON/XML, читаючи властивості типу через GetProperties().
- **Тестові фреймворки** (NUnit, xUnit): знаходять усі методи з атрибутом [Test] або [Fact] і викликають їх через MethodInfo.Invoke.
- **Плагінна архітектура**: завантаження сторонніх збірок у рантаймі, пошук у них типів, що реалізують певний інтерфейс, і їх динамічне підключення.

Це пояснює, чому рефлексія є важливою темою, навіть якщо у власному коді ви рідко пишете typeof(...) або GetMethods() напряму: **інструменти, якими ви користуєтесь щодня, використовують рефлексію під капотом**.

Важливе застереження: продуктивність

Рефлексія надає виняткову гнучкість, але **коштує продуктивністю**. Виклик методу через MethodInfo.Invoke у 10–100 разів повільніший за прямий виклик, оскільки кожного разу виконуються перевірки видимості, типів параметрів і розпакування object. Виклик GetProperties() теж не безкоштовний — він алокує масив PropertyInfo[].

Практичне правило: рефлексія доречна в ініціалізаційному коді (один раз при старті застосунку), в інструментах і фреймворках — але не у гарячому шляху виконання (цикли, обробка запитів). У критично важливих сценаріях результати кешуються або замінюються Expression<T> / source generators.

Простір імен System.Reflection

Основний функціонал рефлексії зосереджений у просторі імен System.Reflection. Ключові класи:

- Assembly — представляє збірку і дозволяє завантажувати, досліджувати її типи
- AssemblyName — зберігає ідентифікаційну інформацію збірки (ім'я, версію, культуру)
- MemberInfo — абстрактний базовий клас для всіх членів типу
- EventInfo — подія типу
- FieldInfo — поле типу (включно з приватними)
- MethodInfo — метод (включно з getters/setters властивостей)
- PropertyInfo — властивість
- ConstructorInfo — конструктор
- ParameterInfo — параметр методу або конструктора
- Module — модуль всередині збірки

Усі ці класи надають не лише інформацію про член, але й можливість діяти: MethodInfo.Invoke — викликати метод, FieldInfo.SetValue — змінити значення поля, ConstructorInfo.Invoke — створити екземпляр.

Клас System.Type — центральний об'єкт рефлексії

Клас System.Type є точкою входу до всієї інформації про тип. Він інкапсулює повний опис класу, структури, інтерфейсу або переліку і надає методи для отримання всіх його складових.

Ключові властивості Type:

Властивість	**Повертає**
Name	Коротке ім'я: "PatientRecord"

Властивість	Повертає
FullName	Повне ім'я з namespace: "Med.PatientRecord"
Namespace	Простір імен
Assembly	Збірка, де визначено тип
BaseType	Базовий клас (Type?, null для object)
IsClass	true якщо клас (reference type)
IsValueType	true якщо struct або enum
IsInterface	true якщо інтерфейс
IsAbstract	true якщо abstract (або інтерфейс)
IsSealed	true якщо sealed
IsGenericType	true якщо List<T>, Dictionary<K,V> тощо
IsArray	true якщо масив
IsEnum	true якщо enum
IsPublic	true якщо публічний тип

Методи Type для отримання членів:

Метод	Повертає
GetMembers()	Всі публічні члени + успадковані
GetMethods()	Методи як MethodInfo[]
GetConstructors()	Конструктори як ConstructorInfo[]
GetFields()	Поля як FieldInfo[]
GetProperties()	Властивості як PropertyInfo[]
GetEvents()	Події як EventInfo[]
GetInterfaces()	Реалізовані інтерфейси як Type[]
GetCustomAttributes()	Атрибути типу

Кожен метод також має перевантаження з BindingFlags для тонкого контролю — детальніше у розд. 14.2.

Три способи отримати об'єкт Type

Спосіб 1: typeof(T) — compile-time

Оператор typeof є найшвидшим і найбезпечнішим способом: тип T відомий на етапі компіляції, компілятор перевіряє коректність, ніяких рантайм-помилок:

```
C#
1  Type t = typeof(PatientRecord);
2  Console.WriteLine(t.Name);    // PatientRecord
3  Console.WriteLine(t.IsClass); // True
```

Якщо передати неіснуючий тип — отримаємо помилку компіляції, що значно краще за рантайм-виняток. Це рекомендований спосіб для більшості сценаріїв.

Спосіб 2: obj.GetType() — runtime, є екземпляр

Метод GetType() успадкований від object і доступний для будь-якого об'єкта. Повертає **фактичний** тип об'єкта, а не тип змінної-посилання:

```
C#
1  PatientRecord p = new PatientRecord("P001", "Петренко", "I10.9");
2  Type t = p.GetType(); // PatientRecord
3
4  // Поліморфний приклад:
5  object obj = new PatientRecord("P001", "Петренко", "I10.9");
6  Console.WriteLine(obj.GetType().Name); // PatientRecord, НЕ object
```

Саме ця поліморфна поведінка робить GetType() незамінним у колекціях гетерогенних об'єктів.

Спосіб 3: Type.GetType(string) — runtime, є рядок імені

Статичний метод Type.GetType дозволяє отримати тип за рядком з його повним іменем. Це єдиний спосіб, коли тип взагалі невідомий під час написання коду — наприклад, ім'я береться з конфігурації або бази даних:

```
C#
1  // Перший параметр — повне ім'я з namespace
2  // Другий — кидати TypeLoadException якщо не знайдено (false = повернути null)
```

```
3  // Третій — ігнорувати регістр
4  Type? t = Type.GetType("Med.PatientRecord", false, true);
5
6  if (t is not null)
7      Console.WriteLine(t.FullName);
8  else
9      Console.WriteLine("Тип не знайдено");
```

Якщо тип знаходиться в **іншій збірці**, після повного імені через кому вказується ім'я збірки:

```
C#
1  Type? t = Type.GetType("Med.PatientRecord, MedLibrary", false, true);
```

Пошук реалізованих інтерфейсів

Метод GetInterfaces() повертає масив Type[] усіх інтерфейсів, що реалізує тип, — включно з успадкованими:

```
C#
 1  Type t = typeof(HospitalDoctor);
 2
 3  Console.WriteLine("Реалізовані інтерфейси:");
 4  foreach (Type i in t.GetInterfaces())
 5  {
 6      Console.WriteLine($"  {i.Name}");
 7  }
 8
 9  interface ISchedulable  { void ScheduleAppointment(string patientId); }
10  interface IReportWriter { void WriteReport(); }
11
12  class HospitalDoctor : ISchedulable, IReportWriter
13  {
14      public string Name { get; }
15      public HospitalDoctor(string name) => Name = name;
16      public void ScheduleAppointment(string pid) => Console.WriteLine($"Записую
    {pid} до {Name}");
17      public void WriteReport() => Console.WriteLine($"Щоденний звіт лікаря
    {Name}");
18  }
```

Оскільки кожен інтерфейс представлений об'єктом Type, для нього також можна застосувати GetMethods(), GetProperties() тощо — отже, рефлексія однаково добре описує як конкретні класи, так і абстрактні інтерфейси.

Дослідження типу PatientRecord — runnable приклад

```
C#
 1  using System;
 2  using System.Reflection;
```

```
3
4  Type t = typeof(PatientRecord);
5
6  Console.WriteLine("=== Загальна інформація ===");
7  Console.WriteLine($"Name:        {t.Name}");
8  Console.WriteLine($"FullName:    {t.FullName}");
9  Console.WriteLine($"Namespace:   {t.Namespace ?? "(global)"}");
10 Console.WriteLine($"IsClass:     {t.IsClass}");
11 Console.WriteLine($"IsValueType: {t.IsValueType}");
12 Console.WriteLine($"IsAbstract:  {t.IsAbstract}");
13 Console.WriteLine($"BaseType:    {t.BaseType?.Name}");
14
15 Console.WriteLine("\n=== Реалізовані інтерфейси ===");
16 Type[] ifaces = t.GetInterfaces();
17 Console.WriteLine(ifaces.Length == 0 ? "  (немає)" : string.Join(", ",
   Array.ConvertAll(ifaces, i => i.Name)));
18
19 Console.WriteLine("\n=== 3 способи отримати Type ===");
20 // 1. typeof
21 Type t1 = typeof(PatientRecord);
22 Console.WriteLine($"typeof:     {t1.Name}");
23
24 // 2. GetType()
25 var p = new PatientRecord("P001", "Петренко", "I10.9", 67);
26 Type t2 = p.GetType();
27 Console.WriteLine($"GetType():  {t2.Name}");
28
29 // 3. Type.GetType з рядка
30 Type? t3 = Type.GetType("PatientRecord");
31 Console.WriteLine($"GetType(s): {(t3 != null ? t3.Name : "null (немає
   namespace)")}");
32
33 Console.WriteLine($"\nВсі три однакові: {t1 == t2}");
34
35 class PatientRecord
36 {
37     public string  Id      { get; }
38     public string  Name    { get; set; }
39     public string  IcdCode { get; set; }
40     public int     Age     { get; set; }
41     private double _bmi;
42
43     public PatientRecord(string id, string name, string icd, int age)
44     {
45         Id = id; Name = name; IcdCode = icd; Age = age;
46     }
47
48     public string GetSummary()    => $"{Name} ({Age} p.) — {IcdCode}";
49     private bool  IsHighRisk()    => IcdCode.StartsWith("I") ||
   IcdCode.StartsWith("C");
50 }
```

Порівняння typeof і GetType() для поліморфних об'єктів — runnable приклад

```
C#
1  using System;
2
```

```
MedicalRecord[] records = {
    new InpatientRecord("R001", 7),
    new OutpatientRecord("R002", "Кардіологія"),
    new InpatientRecord("R003", 3),
    new OutpatientRecord("R004", "Неврологія"),
};

Console.WriteLine("=== Поліморфне GetType() ===");
Console.WriteLine($"{"ID",-6} {"RecordType",-14} {"ActualType",-20} {"BaseType"}");
Console.WriteLine(new string('-', 58));

foreach (var rec in records)
{
    Type actual = rec.GetType();          // фактичний тип
    Type declared = typeof(MedicalRecord); // тип змінної — завжди MedicalRecord

    Console.WriteLine($"{rec.Id,-6} {rec.RecordType,-14} {actual.Name,-20} {actual.BaseType?.Name}");
}

Console.WriteLine("\n=== IsAssignableFrom — перевірка ієрархії ===");
Type baseType   = typeof(MedicalRecord);
Type inpatient  = typeof(InpatientRecord);
Type outpatient = typeof(OutpatientRecord);

Console.WriteLine($"MedicalRecord.IsAssignableFrom(InpatientRecord): {baseType.IsAssignableFrom(inpatient)}");
Console.WriteLine($"MedicalRecord.IsAssignableFrom(OutpatientRecord): {baseType.IsAssignableFrom(outpatient)}");
Console.WriteLine($"InpatientRecord.IsAssignableFrom(MedicalRecord): {inpatient.IsAssignableFrom(baseType)}");

Console.WriteLine("\n=== Підрахунок за типами ===");
int inCount  = 0; int outCount = 0;
foreach (var rec in records)
{
    if (rec.GetType() == typeof(InpatientRecord))  inCount++;
    if (rec.GetType() == typeof(OutpatientRecord)) outCount++;
}
Console.WriteLine($"Стаціонарних: {inCount}, Амбулаторних: {outCount}");

class MedicalRecord
{
    public string Id { get; }
    public MedicalRecord(string id) => Id = id;
    public virtual string RecordType => "Base";
}

class InpatientRecord : MedicalRecord
{
    public int StayDays { get; }
    public InpatientRecord(string id, int days) : base(id) => StayDays = days;
    public override string RecordType => "Inpatient";
}

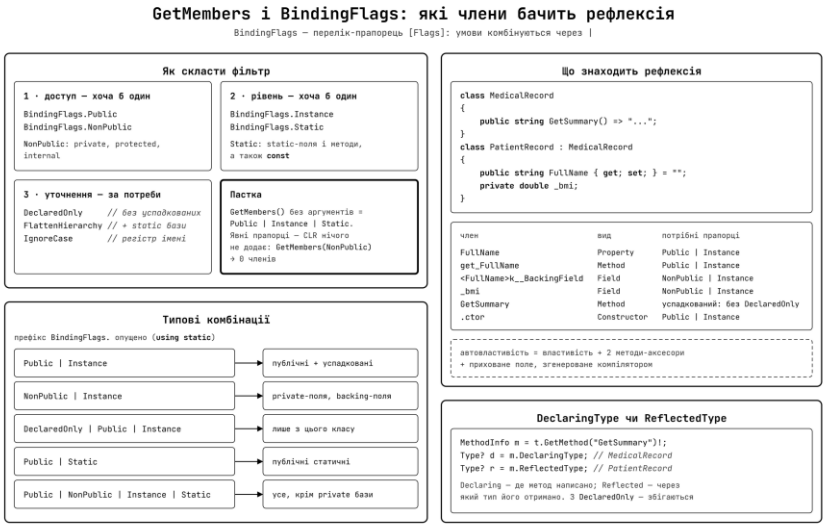
class OutpatientRecord : MedicalRecord
{
    public string Clinic { get; }
```

```
57      public OutpatientRecord(string id, string clinic) : base(id) => Clinic =
    clinic;
58      public override string RecordType => "Outpatient";
59  }
```

14.2. Застосування рефлексії та дослідження типів

Після того як ми отримали об'єкт Type, наступний крок — дослідити його **членів** (members): методи, поля, властивості, конструктори, події. Саме це і є серцем рефлексії: не просто знати, що тип існує, а отримати структурований опис кожного його елемента і, якщо потрібно, взаємодіяти з ним.

Базовим класом для всіх членів типу є абстрактний MemberInfo з простору імен System.Reflection. Від нього успадковують: MethodInfo, ConstructorInfo, FieldInfo, PropertyInfo, EventInfo. Кожен із них надає специфічний для свого виду API, але всі мають спільний базовий функціонал MemberInfo.



GetMembers та BindingFlags — дослідження членів типу

Клас MemberInfo та його спільні властивості

Оскільки GetMembers() повертає MemberInfo[], варто знати, що доступно через базовий клас для будь-якого члена незалежно від його виду:

Властивість / Метод	Що повертає
.Name	Ім'я члена: "GetSummary", "_bmi", "Age"
.DeclaringType	Тип, де **оголошено** цей член
.ReflectedType	Тип, через який **отримано** цей член
.MemberType	MemberTypes.Method, .Field, .Property, .Constructor, .Event
.GetCustomAttributes(bool)	Масив атрибутів, декорованих на члені
.IsDefined(Type, bool)	true якщо атрибут певного типу присутній

DeclaringType vs ReflectedType

Ця різниця важлива при роботі з ієрархіями класів. DeclaringType — тип, де метод або поле **фізично написане**. ReflectedType — тип, через typeof(...) якого ми дісталися до цього члена.

```
C#
1  class MedicalRecord
2  {
3      public string GetSummary() => "Base summary";
4  }
5
6  class PatientRecord : MedicalRecord { }
```

```
C#
1  // Отримуємо GetSummary через нащадка PatientRecord
2  MethodInfo m = typeof(PatientRecord).GetMethod("GetSummary");
3
4  Console.WriteLine(m.DeclaringType.Name);  // MedicalRecord — де оголошено
5  Console.WriteLine(m.ReflectedType.Name);  // PatientRecord  — через кого
   отримано
```

DeclaringType відповідає на питання «де це написано?», ReflectedType — «через який клас ми запитали?». При використанні DeclaredOnly обидва збігаються.

Отримання всіх компонентів типу

Метод GetMembers() повертає всі доступні компоненти типу як об'єкт MemberInfo. Цей об'єкт дозволяє отримати інформацію про компонент типу. Зокрема, деякі його властивості:

— DeclaringType: повертає повну назву типу

— MemberType: повертає значення з переліку MemberTypes:

- MemberTypes.Constructor - MemberTypes.Method - MemberTypes.Field - MemberTypes.Event - MemberTypes.Property - MemberTypes.NestedType

— Name: повертає назву компонента

Застосуємо метод GetMembers і виведемо всі доступні елементи типу:

```
C#
using System.Reflection; // Підключаємо функціонал рефлексії.

Type myType = typeof(Person);

foreach (MemberInfo member in myType.GetMembers())
{
    Console.WriteLine($"{member.DeclaringType} {member.MemberType}
{member.Name}");
}

public class Person
{
    string name;
    public int Age { get; set; }
    public Person(string name, int age)
    {
        this.name = name;
        this.Age = age;
    }
    public void Print() => Console.WriteLine($"Name: {name} Age: {Age}");
}
```

В даному випадку ми отримаємо всі доступні члени класу Person.

```
Person Method get_Age
Person Method set_Age
Person Method Print

System.Object Method GetType
System.Object Method ToString
System.Object Method Equals
System.Object Method GetHashCode
Person Constructor .ctor
```

```
10  Person Property Age
```

Зверніть увагу, що в даному випадку ми отримуємо тільки всі публічні компоненти класу, і нам не виводиться інформація про приватну змінну name.

З іншого боку, властивості виводяться методи доступу - геттер (тут get_Age) і сеттер (тут set_Age).

Третій момент, який слід зазначити, що за умовчанням ми отримуємо весь функціонал, у тому числі успадкований від базових класів (у цьому випадку функціонал базового класу Object).

BindingFlags — бітова маска фільтрації

У прикладі вище використовувалася проста форма GetMembers() без аргументів. Вона повертає лише **публічні** члени типу, включаючи успадковані від базових класів. Щоб отримати непублічні, статичні або тільки власні члени, використовується перевантаження GetMembers(BindingFlags).

BindingFlags — це перелік-прапорець ([Flags] enum), значення якого комбінуються через побітове АБО (|). Кожен прапорець вмикає або вимикає певну категорію членів. Важлива особливість: якщо не передати жодного прапорця, CLR застосовує внутрішній дефолт — еквівалент Public | Instance | Static з включеними успадкованими. Щойно ви передаєте хоча б один прапорець явно, CLR починає вимагати _всі_ потрібні категорії.

Ключові прапорці:

Прапорець	Що вмикає
Public	публічні члени (public)
NonPublic	непублічні (private, protected, internal)
Instance	члени екземпляра (нестатичні)
Static	статичні члени (static)
DeclaredOnly	тільки члени _цього_ класу, успадковані виключаються
FlattenHierarchy	включити статичні члени базових класів (актуально при Static)
IgnoreCase	ігнорувати регістр у GetMethod(name) / GetProperty(name)

Основна пастка: при явній передачі прапорців CLR не додає нічого автоматично. Якщо написати лише NonPublic, без Instance — результат буде порожнім, бо жоден член не задовольняє комбінацію «непублічний і не-екземпляр і не-статичний».

Типові практичні комбінації:

```
C#
 1  // Усі публічні методи екземпляра (стандартний запит для DI)
 2  BindingFlags.Public | BindingFlags.Instance
 3
 4  // Усі приватні поля об'єкта (для серіалізаторів, ORM)
 5  BindingFlags.NonPublic | BindingFlags.Instance
 6
 7  // Тільки власні члени класу без успадкованих (для документації)
 8  BindingFlags.DeclaredOnly | BindingFlags.Public | BindingFlags.Instance
 9
10  // Абсолютно всі члени: public/private, instance/static
11  BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.Instance |
    BindingFlags.Static
12
13  // Тільки публічні статичні (константи, factory-методи)
14  BindingFlags.Public | BindingFlags.Static
```

BackingField — прихована ціна автовластивостей

Коли компілятор C# обробляє автоматично реалізовану властивість (public string Name { get; set; }), він **генерує** приватне поле зберігання за ім'ям <Name>k__BackingField. Це поле існує в IL-коді збірки і повністю видиме через рефлексію.

```
C#
1  public class PatientRecord
2  {
3      public string FullName { get; set; }   // → <FullName>k__BackingField
4      public DateTime BirthDate { get; set; } // → <BirthDate>k__BackingField
5      private decimal _bmi;                  // → явне поле _bmi
6  }
```

При виклику GetFields() без NonPublic — backing-поля **не видно**, бо вони private. Але щойно додається NonPublic | Instance, вони з'являються поруч із явними приватними полями:

```
1  Field  <FullName>k__BackingField
2  Field  <BirthDate>k__BackingField
3  Field  _bmi
```

На практиці це має значення для:

— **Серіалізаторів**, які обходять властивості і читають поля напряму — вони повинні відфільтровувати k__BackingField або знати, що він є.

— **ORM-маперів**, де треба розрізнити «бізнесове» поле від backing-поля.

— **Клонування об'єктів** через рефлексію — необхідно копіювати або поле, або властивість, але не обидва одночасно.

Для надійної фільтрації backing-полів перевіряють field.Name.StartsWith("<") та наявність атрибута [CompilerGenerated]:

```
C#
1  bool isBackingField =
   field.IsDefined(typeof(System.Runtime.CompilerServices.CompilerGeneratedAttribut
   e), false);
```

GetMember — отримання члена за ім'ям

Метод GetMember(string name) повертає MemberInfo[], а не один об'єкт. Це свідоме рішення: у класі може бути _кілька_ членів з однаковим ім'ям — перевантажені методи:

```
C#
1  public class PatientRecord
2  {
3      public void AddNote(string text) { }
4      public void AddNote(string text, DateTime timestamp) { }
5      public void AddNote(NoteDto dto) { }
6  }
```

```
C#
1  Type t = typeof(PatientRecord);
2  MemberInfo[] notes = t.GetMember("AddNote", BindingFlags.Instance |
   BindingFlags.Public);
3  Console.WriteLine(notes.Length); // 3 — три перевантаження
```

Якщо потрібен _конкретний_ перевантажений варіант, використовують GetMethod(name, Type[]) з масивом типів параметрів:

```
C#
1  MethodInfo specific = t.GetMethod("AddNote", new[] { typeof(string),
   typeof(DateTime) });
```

Приклад 1 — PatientRecord: порівняння GetMembers з різними BindingFlags

```
C#
 1  using System;
```

```
using System.Linq;
using System.Reflection;
using System.Runtime.CompilerServices;

Type t = typeof(PatientRecord);

Console.WriteLine("=== GetMembers() — публічні + успадковані ===");
foreach (var m in t.GetMembers())
    Console.WriteLine($"  [{m.MemberType,-11}] {m.Name,-30} (з
{m.DeclaringType!.Name})");

Console.WriteLine();
Console.WriteLine("=== GetFields(NonPublic|Instance) — приватні поля +
BackingField ===");
var allFields = t.GetFields(BindingFlags.NonPublic | BindingFlags.Instance);
foreach (var f in allFields)
{
    bool isBacking = f.IsDefined(typeof(CompilerGeneratedAttribute), false);
    string tag = isBacking ? " [backing]" : "";
    Console.WriteLine($"  {f.Name}{tag}  ({f.FieldType.Name})");
}

Console.WriteLine();
Console.WriteLine("=== DeclaredOnly — тільки власні члени ===");
var declared = t.GetMembers(BindingFlags.DeclaredOnly | BindingFlags.Public
                          | BindingFlags.NonPublic | BindingFlags.Instance);
foreach (var m in declared.OrderBy(m => m.MemberType.ToString()))
    Console.WriteLine($"  [{m.MemberType,-11}] {m.Name}");

public class MedicalRecord
{
    public int RecordId { get; set; }
    public virtual string GetSummary() => $"Record #{RecordId}";
}

public class PatientRecord : MedicalRecord
{
    public string FullName { get; set; } = "";
    public DateTime BirthDate { get; set; }
    private decimal _bmi;
    private string? _notes;

    public PatientRecord(string fullName, DateTime birthDate)
    {
        FullName = fullName;
        BirthDate = birthDate;
    }

    public void SetBmi(decimal bmi) => _bmi = bmi;
    public decimal GetBmi() => _bmi;

    public void AddNote(string text) => _notes = text;
    public void AddNote(string text, DateTime timestamp)
        => _notes = $"[{timestamp:d}] {text}";

    public override string GetSummary() =>
        $"Пацієнт: {FullName}, ІМТ: {_bmi:F1}";
}
```

Очікуваний вивід ілюструє три рівні видимості: публічні (включно з успадкованими від MedicalRecord та Object), приватні поля зі згенерованими backing-полями, і лише власні задекларовані члени PatientRecord.

Приклад 2 — TypeExplorer: структурований звіт про тип

```
C#
 1  using System;
 2  using System.Linq;
 3  using System.Reflection;
 4  using System.Runtime.CompilerServices;
 5  using System.Text;
 6
 7  Console.WriteLine(TypeExplorer.Report(typeof(PatientRecord)));
 8
 9  public class PatientRecord
10  {
11      public string FullName { get; set; } = "";
12      public DateTime BirthDate { get; set; }
13      public int RecordId { get; init; }
14      private decimal _bmi;
15      private readonly string _ward = "Cardiology";
16
17      public PatientRecord() { }
18      public PatientRecord(string fullName) => FullName = fullName;
19
20      public void SetBmi(decimal v) => _bmi = v;
21      public decimal GetBmi() => _bmi;
22      public static PatientRecord CreateAnonymous() => new();
23      public override string ToString() => $"Patient: {FullName}";
24  }
25
26  public static class TypeExplorer
27  {
28      public static string Report(Type t)
29      {
30          var sb = new StringBuilder();
31          var flags = BindingFlags.Public | BindingFlags.NonPublic
32                    | BindingFlags.Instance | BindingFlags.Static
33                    | BindingFlags.DeclaredOnly;
34
35          sb.AppendLine($"╔══ Тип: {t.FullName} ══╗");
36          sb.AppendLine($"  Base   : {t.BaseType?.Name ?? "—"}");
37          sb.AppendLine($"  IsClass: {t.IsClass}  IsAbstract: {t.IsAbstract}");
38
39          // Конструктори
40          var ctors = t.GetConstructors(BindingFlags.Public |
    BindingFlags.NonPublic | BindingFlags.Instance);
41          sb.AppendLine($"\n  ── Конструктори ({ctors.Length}) ──");
42          foreach (var c in ctors)
43          {
44              var ps = string.Join(", ", c.GetParameters().Select(p =>
    $"{p.ParameterType.Name} {p.Name}"));
45              sb.AppendLine($"    .ctor({ps})");
46          }
```

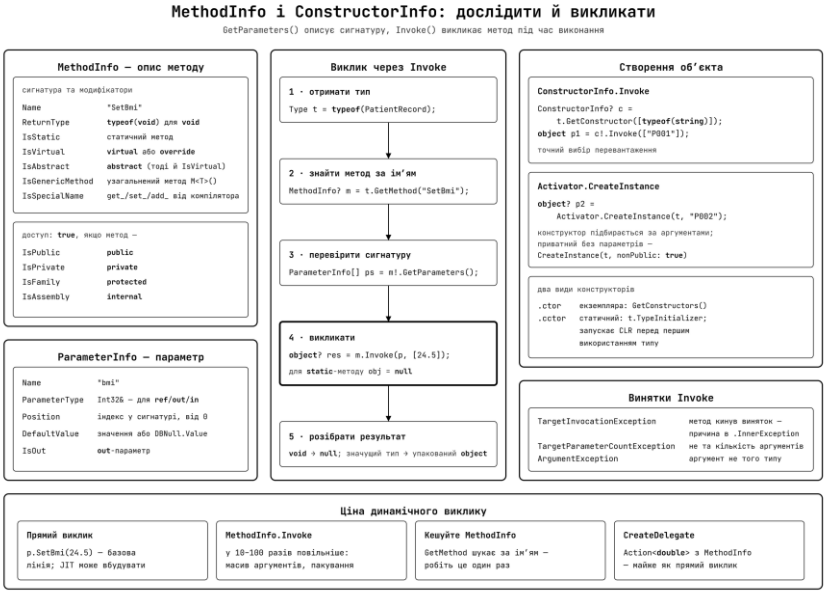
```
47
48          // Властивості
49          var props = t.GetProperties(flags);
50          sb.AppendLine($"\n  ── Властивості ({props.Length}) ──");
51          foreach (var p in props)
52          {
53              string rw = (p.CanRead ? "get" : "") + (p.CanWrite ? ";set" : "");
54              sb.AppendLine($"    {p.PropertyType.Name,-15} {p.Name,-20}
    [{rw}]");
55          }
56
57          // Поля (без backing)
58          var fields = t.GetFields(flags);
59          var realFields  = fields.Where(f =>
    !f.IsDefined(typeof(CompilerGeneratedAttribute), false)).ToArray();
60          var backings    = fields.Where(f =>
    f.IsDefined(typeof(CompilerGeneratedAttribute), false)).ToArray();
61
62          sb.AppendLine($"\n  ── Явні поля ({realFields.Length}) ──");
63          foreach (var f in realFields)
64          {
65              string mod = f.IsStatic ? "static " : "";
66              string ro  = f.IsInitOnly ? "readonly " : "";
67              sb.AppendLine($"    {mod}{ro}{f.FieldType.Name,-15} {f.Name}");
68          }
69
70          sb.AppendLine($"\n  ── BackingFields ({backings.Length}) ──");
71          foreach (var f in backings)
72              sb.AppendLine($"    {f.Name}  ({f.FieldType.Name})");
73
74          // Методи
75          var methods = t.GetMethods(flags)
76                         .Where(m => !m.IsSpecialName)  // прибираємо get_*/set_*
77                         .ToArray();
78          sb.AppendLine($"\n  ── Методи ({methods.Length}) ──");
79          foreach (var m in methods)
80          {
81              string mod = m.IsStatic ? "static " : "";
82              var ps = string.Join(", ", m.GetParameters().Select(p =>
    p.ParameterType.Name));
83              sb.AppendLine($"    {mod}{m.ReturnType.Name,-10} {m.Name}({ps})");
84          }
85
86          sb.AppendLine("╚══════════════════════╝");
87          return sb.ToString();
88      }
89  }
```

TypeExplorer.Report демонструє реальну практику: конструктори через окремий GetConstructors, властивості з CanRead/CanWrite, явні поля відокремлені від k__BackingField через атрибут [CompilerGenerated], методи без IsSpecialName (щоб не показувати get_FullName). Такий підхід використовується в документаційних генераторах, Coverage-аналізаторах та ORM-інспекторах.

14.3. Дослідження методів та конструкторів за допомогою рефлексії

Якщо System.Type — це паспорт типу, то MethodInfo — це докладна анкета кожного його методу. MethodInfo успадковує від MemberInfo і доповнює базові властивості (.Name, .DeclaringType, .MemberType) специфічним для методів API: типом повернення, параметрами, модифікаторами, а головне — можливістю **викликати метод під час виконання** через Invoke.

Саме ця пара — дослідження (GetParameters) і виклик (Invoke) — лежить в основі ін'єкції залежностей, тестових фреймворків, генераторів документації та протоколів серіалізації.



MethodInfo та ConstructorInfo — виклик через рефлексію

Клас MethodInfo та його властивості

MethodInfo надає повний опис одного методу. Його ключові властивості:

Властивість	Що повертає
.Name	ім'я методу: "SetBmi", "GetSummary"
.ReturnType	Type типу, що повертається; typeof(void) для void
.IsPublic	true якщо public
.IsPrivate	true якщо private
.IsFamily	true якщо protected
.IsAssembly	true якщо internal
.IsStatic	true якщо статичний метод
.IsVirtual	true якщо virtual або override
.IsAbstract	true якщо abstract (обов'язково є IsVirtual = true)
.IsFinal	true якщо sealed override
.IsGenericMethod	true якщо метод generic
.IsSpecialName	true для get_*, set_*, add_* — обслуговуючих методів
.GetParameters()	ParameterInfo[] — масив параметрів
.GetGenericArguments()	Type[] — типові параметри generic-методу
.MakeGenericMethod(types)	конкретизація generic-методу для певних типів
.Invoke(obj, args)	динамічний виклик методу

IsSpecialName — чому get_Age з'являється серед методів

При виклику GetMethods() без фільтрів у списку опиняються методи get_FullName, set_BirthDate, add_StatusChanged, remove_StatusChanged. Це **обслуговуючі методи** (accessor methods), автоматично згенеровані компілятором для властивостей та подій.

Щоб відфільтрувати їх, перевіряємо IsSpecialName:

```
C#
```

```
1  var methods = typeof(PatientRecord)
2      .GetMethods(BindingFlags.Public | BindingFlags.Instance |
   BindingFlags.DeclaredOnly)
3      .Where(m => !m.IsSpecialName);
```

IsSpecialName = true не означає, що метод недоступний — він цілком придатний для Invoke. Прапорець лише сигналізує: «цей метод генерується компілятором, а не написаний вручну».

Клас ParameterInfo — деталі сигнатури

GetParameters() повертає ParameterInfo[] — масив дескрипторів параметрів у порядку їхнього оголошення в сигнатурі:

Властивість	Що повертає
.Name	ім'я параметра: "bmi", "timestamp"
.ParameterType	Type параметра
.Position	індекс у сигнатурі (від 0)
.IsOptional	true якщо параметр має значення за замовчуванням
.HasDefaultValue	true якщо .DefaultValue доступний
.DefaultValue	значення за замовчуванням або DBNull.Value
.IsOut	true якщо out-параметр
.IsIn	true якщо in-параметр
.GetCustomAttributes()	атрибути на параметрі (наприклад, [FromBody])

Якщо параметр передається через ref, in або out, то .ParameterType.Name отримує суфікс &: "String&", "Int32&". Для out додатково IsOut = true, для in — IsIn = true.

При побудові документаційних генераторів або IoC-контейнерів IsOptional і DefaultValue дозволяють визначити, які аргументи можна не передавати:

```
C#
1  void Register(PatientRecord record, bool urgent = false, string ward =
   "General")
```

```
C#
1  MethodInfo m = typeof(PatientService).GetMethod("Register")!;
2  foreach (var p in m.GetParameters())
3  {
4      string defaultStr = p.HasDefaultValue ? $" = {p.DefaultValue}" : "";
5      Console.WriteLine($"  {p.ParameterType.Name} {p.Name}{defaultStr}");
6  }
7  // PatientRecord record
8  // Boolean urgent = False
9  // String ward = General
```

Invoke — динамічний виклик методу

Invoke — найпотужніша операція MethodInfo. Її сигнатура:

```
C#
1  public object? Invoke(object? obj, object?[]? parameters);
```

— **obj** — екземпляр, для якого викликається метод. Для static передається null.

— **parameters** — масив аргументів у порядку оголошення. Якщо метод безпараметровий — передати null або [].

— **Результат** — object?. Для void-методів завжди null. Для value-типів відбувається **boxing**.

Повний workflow виклику відповідає п'яти крокам з діаграми:

```
C#
 1  // 1. Отримати Type
 2  Type t = typeof(PatientRecord);
 3
 4  // 2. Знайти метод
 5  MethodInfo? mi = t.GetMethod("SetBmi",
 6      BindingFlags.Public | BindingFlags.Instance);
 7
 8  // 3. Перевірити параметри (опціонально)
 9  foreach (var p in mi!.GetParameters())
10      Console.WriteLine($"  {p.ParameterType.Name} {p.Name}");
11
12  // 4. Викликати Invoke
13  var patient = new PatientRecord("Олена Петренко", new DateTime(1985, 3, 12));
14  mi.Invoke(patient, new object[] { 24.7m });  // аргументи у масиві
15
16  // 5. Отримати і розкастувати результат
17  MethodInfo? getBmi = t.GetMethod("GetBmi");
18  object? raw = getBmi!.Invoke(patient, null);
19  decimal bmi = (decimal)raw!;
20  Console.WriteLine($"ІМТ: {bmi}");
```

При передачі аргументів через object[] відбувається **boxing** value-типів (decimal, int, bool). Саме тут криється основна ціна

продуктивності: CLR упаковує значення в heap-об'єкти, а під час кожного виклику перевіряє кількість і типи аргументів і розпаковує їх назад у параметри методу. (Перевірки безпеки коду — Code Access Security — у .NET Core і новіших версіях більше немає, тож старі пояснення про «security-check при Invoke» до сучасного .NET не стосуються.)

TargetInvocationException — обгортка справжнього виключення

Якщо метод, викликаний через Invoke, кидає виключення, CLR **перехоплює його** і перепакує у TargetInvocationException. Причина — захистити call stack від змішування рефлексійного і прикладного стеків.

```C#
try
{
    mi.Invoke(patient, new object[] { -5.0m });  // SetBmi(-5) — кидає
}
catch (TargetInvocationException ex)
{
    // ex.Message — технічний опис обгортки
    // ex.InnerException — справжнє виключення з методу
    Console.WriteLine($"Помилка: {ex.InnerException?.Message}");
}
```

Якщо треба перекинути справжнє виключення зі збереженням оригінального стеку, використовують ExceptionDispatchInfo:

```C#
catch (TargetInvocationException ex) when (ex.InnerException is not null)
{
    System.Runtime.ExceptionServices.ExceptionDispatchInfo
        .Capture(ex.InnerException)
        .Throw();
}
```

Це стандартний патерн у фреймворках: тестові раннери (xUnit, NUnit) завжди «розпаковують» TargetInvocationException, щоб показати оригінальний AssertionException, а не обгортку.

ConstructorInfo та Activator

.ctor vs .cctor

Рефлексія розрізняє два типи конструкторів:

— **.ctor** — звичайний конструктор екземпляра. Присутній у GetConstructors().

— **.cctor** — статичний конструктор (static PatientRecord() { ... }). Виконується CLR автоматично перед першим використанням типу. Видимий через GetConstructors(BindingFlags.Static | BindingFlags.NonPublic) або зручніше — через властивість Type.TypeInitializer. Запускає його **тільки CLR і лише один раз**: ручний виклик t.TypeInitializer.Invoke(null, null) компілюється і не кидає винятку, але вже виконаний статичний конструктор повторно не запускає. Тому статичний конструктор досліджують через рефлексію, але не викликають.

Invoke vs Activator.CreateInstance

Два способи динамічно створити об'єкт:

```
C#
1  // Спосіб 1 — через ConstructorInfo.Invoke
2  ConstructorInfo? ctor = typeof(PatientRecord)
3      .GetConstructor(new[] { typeof(string), typeof(DateTime) });
4  object? instance = ctor?.Invoke(new object[] { "Іван Коваль", new DateTime(1972,
   7, 1) });
5
6  // Спосіб 2 — через Activator (спрощений API)
7  object? instance2 = Activator.CreateInstance(
8      typeof(PatientRecord), "Іван Коваль", new DateTime(1972, 7, 1));
```

Activator.CreateInstance — зручна обгортка, що всередині так само знаходить конструктор за типами аргументів і викликає Invoke. Різниця:

Критерій	ConstructorInfo.Invoke	Activator.CreateInstance
Контроль	явний вибір конструктора	CLR шукає сам за аргументами
Перевантаження	повний контроль	може знайти не той, якщо є неоднозначність
Безпараметровий	Invoke(null)	CreateInstance(type)
Приватний конструктор	потрібен NonPublic у BindingFlags	лише безпараметровий: CreateInstance(type, nonPublic: true); з параметрами — MissingMethodException
Рекомендація	для складних сценаріїв	для простого factory за типом

Зверніть увагу на рядок про приватний конструктор: звичайний виклик Activator.CreateInstance(type) для класу з приватним конструктором кидає MissingMethodException, але перевантаження з другим параметром nonPublic: true дозволяє використати приватний **безпараметровий** конструктор. Для приватного конструктора з параметрами потрібен ConstructorInfo з BindingFlags.NonPublic | BindingFlags.Instance.

Існує ще й третій спосіб — RuntimeHelpers.GetUninitializedObject(type) (простір імен System.Runtime.CompilerServices). Він обходить **будь-який** конструктор і повертає «сирий» об'єкт з нульовими полями. Використовується в десеріалізаторах, де конструктор викликається окремо або взагалі не викликається. У старому коді ви побачите FormatterServices.GetUninitializedObject — цей клас позначено застарілим (попередження SYSLIB0050) разом з усією formatter-серіалізацією.

Продуктивність: чотири рівні

Рефлексійний виклик через Invoke — не безкоштовний. Абсолютні числа сильно залежать від версії .NET (починаючи з .NET 7 Invoke генерує спеціалізовані заглушки і став у рази швидшим) та від сигнатури методу, тому корисніше пам'ятати **відносний порядок** і міряти свій випадок інструментом BenchmarkDotNet:

Спосіб	Відносна вартість	Звідки береться
Прямий виклик	1× — базова лінія	JIT може вбудувати метод
MethodInfo.Invoke	у 10–100 разів повільніше	масив object[], boxing value-типів, перевірка аргументів при кожному виклику
GetMethod() + Invoke у циклі	ще повільніше	пошук у метаданих за ім'ям на кожній ітерації — кешуйте MethodInfo

Спосіб	Відносна вартість	Звідки береться
CreateDelegate / скомпільований Expression	майже як прямий виклик	дорого один раз побудувати, далі — звичайний виклик делегата

Кешування MethodInfo

Якщо метод викликається у циклі або на «гарячому шляху», GetMethod() не можна викликати всередині циклу:

```
C#
 1  // Неефективно — пошук по метаданих на кожній ітерації
 2  foreach (var patient in patients)
 3      typeof(PatientRecord).GetMethod("Validate")!.Invoke(patient, null);
 4
 5  // Ефективно — кешуємо одноразово
 6  static readonly MethodInfo ValidateMethod =
 7      typeof(PatientRecord).GetMethod("Validate")!;
 8
 9  foreach (var patient in patients)
10      ValidateMethod.Invoke(patient, null);
```

CreateDelegate — майже пряма швидкість

MethodInfo.CreateDelegate<TDelegate>() компілює обгортку навколо методу, що по продуктивності близька до прямого виклику:

```
C#
 1  // Один раз будуємо делегат
 2  static readonly Action<PatientRecord, decimal> SetBmiDelegate =
 3      (Action<PatientRecord, decimal>)
 4      typeof(PatientRecord)
 5          .GetMethod("SetBmi")!
 6          .CreateDelegate(typeof(Action<PatientRecord, decimal>));
 7
 8  // Гарячий шлях — майже без overhead
 9  foreach (var patient in patients)
10      SetBmiDelegate(patient, 22.5m);
```

Цей підхід використовує Entity Framework для виклику методів властивостей, ASP.NET для прив'язки параметрів запиту, AutoMapper для маппінгу значень.

Приклад 1 — PatientRecord: дослідження сигнатур та виклик через Invoke

```C#
using System;
using System.Linq;
using System.Reflection;

Type t = typeof(PatientRecord);

Console.WriteLine("=== Методи PatientRecord (власні, без IsSpecialName) ===\n");

var flags = BindingFlags.Public | BindingFlags.NonPublic
          | BindingFlags.Instance | BindingFlags.Static
          | BindingFlags.DeclaredOnly;

foreach (MethodInfo m in t.GetMethods(flags).Where(m => !m.IsSpecialName))
{
    string access = m.IsPublic ? "public" : m.IsFamily ? "protected" : "private";
    string modifier = m.IsStatic ? " static" : m.IsAbstract ? " abstract"
                    : m.IsVirtual ? " virtual" : "";

    string ret = m.ReturnType == typeof(void) ? "void" : m.ReturnType.Name;

    var pars = m.GetParameters().Select(p =>
    {
        string mod = p.IsOut ? "out " : p.IsIn ? "in " : "";
        string def = p.HasDefaultValue ? $" = {p.DefaultValue}" : "";
        return $"{mod}{p.ParameterType.Name} {p.Name}{def}";
    });

    Console.WriteLine($"  {access}{modifier} {ret} {m.Name}({string.Join(", ", pars)})");
}

Console.WriteLine("\n=== Invoke: SetBmi → GetBmi ===\n");

var patient = new PatientRecord("Олена Петренко", new DateTime(1985, 3, 12));

MethodInfo setBmi = t.GetMethod("SetBmi", flags)!;
setBmi.Invoke(patient, new object[] { 26.3m });

MethodInfo getBmi = t.GetMethod("GetBmi", flags)!;
object? result = getBmi.Invoke(patient, null);
Console.WriteLine($"ІМТ після Invoke: {result}");

Console.WriteLine("\n=== TargetInvocationException ===\n");

try
{
    setBmi.Invoke(patient, new object[] { -1.0m });
}
catch (TargetInvocationException ex)
{
    Console.WriteLine($"TargetInvocationException: {ex.Message}");
```

```
51      Console.WriteLine($"InnerException: {ex.InnerException?.GetType().Name} —
    {ex.InnerException?.Message}");
52  }
53
54  Console.WriteLine("\n=== Конструктори PatientRecord ===\n");
55
56  foreach (ConstructorInfo ctor in t.GetConstructors(
57      BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic))
58  {
59      string access = ctor.IsPublic ? "public" : ctor.IsPrivate ? "private" :
    "protected";
60      var ps = ctor.GetParameters()
61                   .Select(p => $"{p.ParameterType.Name} {p.Name}");
62      Console.WriteLine($"  {access} PatientRecord({string.Join(", ", ps)})");
63  }
64
65  public class PatientRecord
66  {
67      public string FullName { get; set; }
68      public DateTime BirthDate { get; set; }
69      private decimal _bmi;
70      private string? _notes;
71
72      public PatientRecord(string fullName, DateTime birthDate)
73      {
74          FullName = fullName;
75          BirthDate = birthDate;
76      }
77
78      private PatientRecord() : this("Анонімний", DateTime.MinValue) { }
79
80      public void SetBmi(decimal bmi)
81      {
82          if (bmi <= 0) throw new ArgumentOutOfRangeException(nameof(bmi), "IMT
    має бути додатнім");
83          _bmi = bmi;
84      }
85
86      public decimal GetBmi() => _bmi;
87
88      public void AddNote(string text) => _notes = text;
89      public void AddNote(string text, DateTime timestamp)
90          => _notes = $"[{timestamp:d}] {text}";
91
92      public string GetSummary() => $"Пацієнт: {FullName}, IMT: {_bmi:F1}";
93      public static PatientRecord CreateAnonymous() => new();
94
95      protected virtual string FormatForExport() =>
96          $"{FullName};{BirthDate:yyyy-MM-dd};{_bmi}";
97  }
```

Приклад 2 — MethodDispatcher<T>: кешування та CreateDelegate

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Reflection;
```

```

// Демонстрація кешування MethodInfo і CreateDelegate для гарячого шляху.
var patients = new[]
{
    new PatientRecord("Іван Коваль",   new DateTime(1980, 5, 15)),
    new PatientRecord("Марія Гончар",  new DateTime(1992, 11, 3)),
    new PatientRecord("Петро Мельник", new DateTime(1965, 8, 22)),
};

// Стандартний MethodDispatcher через кешований MethodInfo
var dispatcher = new MethodDispatcher<PatientRecord>();

Console.WriteLine("=== Invoke через кешований MethodInfo ===\n");

dispatcher.Invoke(patients[0], "SetBmi", 23.1m);
dispatcher.Invoke(patients[1], "SetBmi", 30.4m);
dispatcher.Invoke(patients[2], "SetBmi", 26.7m);

foreach (var p in patients)
{
    object? summary = dispatcher.Invoke(p, "GetSummary");
    Console.WriteLine($"  {summary}");
}

Console.WriteLine("\n=== CreateDelegate — гарячий шлях ===\n");

// Будуємо делегат одноразово
Action<PatientRecord, decimal> fastSetBmi =
    (Action<PatientRecord, decimal>)
    typeof(PatientRecord)
        .GetMethod("SetBmi")!
        .CreateDelegate(typeof(Action<PatientRecord, decimal>));

Func<PatientRecord, string> fastGetSummary =
    (Func<PatientRecord, string>)
    typeof(PatientRecord)
        .GetMethod("GetSummary")!
        .CreateDelegate(typeof(Func<PatientRecord, string>));

// Гарячий шлях: майже без overhead рефлексії
foreach (var p in patients)
{
    fastSetBmi(p, 25.0m);
    Console.WriteLine($"  {fastGetSummary(p)}");
}

Console.WriteLine("\n=== Activator.CreateInstance ===\n");

// Динамічне створення за Type (наприклад, з рядка конфігурації)
Type recordType = Type.GetType("PatientRecord") ?? typeof(PatientRecord);
object? newRecord = Activator.CreateInstance(
    recordType, "Оксана Лисенко", new DateTime(1990, 2, 14));
Console.WriteLine(dispatcher.Invoke(newRecord!, "GetSummary"));

public class PatientRecord
{
    public string FullName { get; set; }
    public DateTime BirthDate { get; set; }
    private decimal _bmi;

```

```
    public PatientRecord(string fullName, DateTime birthDate)
    {
        FullName = fullName;
        BirthDate = birthDate;
    }

    public void SetBmi(decimal bmi) => _bmi = Math.Max(0, bmi);
    public decimal GetBmi() => _bmi;
    public string GetSummary() => $"Пацієнт: {FullName}, ІМТ: {_bmi:F1}";
}

public class MethodDispatcher<T>
{
    private readonly Dictionary<string, MethodInfo> _cache = new();

    private MethodInfo Resolve(string name)
    {
        if (!_cache.TryGetValue(name, out var mi))
        {
            mi = typeof(T).GetMethod(name,
                    BindingFlags.Public | BindingFlags.NonPublic |
BindingFlags.Instance)
                ?? throw new MissingMethodException(typeof(T).Name, name);
            _cache[name] = mi;
        }
        return mi;
    }

    public object? Invoke(object target, string methodName, params object[]
args)
    {
        try
        {
            return Resolve(methodName).Invoke(target, args.Length == 0 ? null
: args);
        }
        catch (TargetInvocationException ex) when (ex.InnerException is not
null)
        {
            System.Runtime.ExceptionServices.ExceptionDispatchInfo
                .Capture(ex.InnerException).Throw();
            return null; // недосяжний код
        }
    }

    public int CacheSize => _cache.Count;
}
```

MethodDispatcher<T> демонструє три речі: кешування MethodInfo у словнику за ім'ям, «розпакування» TargetInvocationException через ExceptionDispatchInfo, і параметр params object[] для зручного синтаксису виклику. Поруч — делегати через CreateDelegate як найефективніший варіант для повторюваних викликів на одному типі.

14.4. Дослідження полів та властивостей за допомогою рефлексії

У попередньому розділі ми досліджували методи та конструктори через MethodInfo і ConstructorInfo. Тепер розглянемо, як рефлексія дозволяє читати і змінювати **дані** типу — поля через FieldInfo та властивості через PropertyInfo.

Ці два класи виглядають схожими (обидва мають GetValue і SetValue), але представляють різні рівні абстракції. **Поле** — це пряма ділянка пам'яті об'єкта. **Властивість** — абстракція з get/set-аксесорами, які можуть містити довільну логіку, кидати виключення і бути реалізовані через backing-field або взагалі обчислюватись динамічно.



FieldInfo та PropertyInfo — читання та запис значень через рефлексію

Клас FieldInfo

FieldInfo описує одне поле класу або структури. Ключові властивості:

Властивість	Що повертає
.Name	ім'я поля: "_bmi", "<FullName>k__BackingField"
.FieldType	Type поля

Властивість	Що повертає
.IsPublic	true якщо public
.IsPrivate	true якщо private
.IsFamily	true якщо protected
.IsAssembly	true якщо internal
.IsStatic	true якщо static
.IsInitOnly	true якщо readonly
.IsLiteral	true якщо const (значення вбудовано на етапі компіляції)
.IsNotSerialized	true якщо є [NonSerialized]
.GetValue(obj)	прочитати значення поля у об'єкта obj
.SetValue(obj, val)	записати значення val у поле об'єкта obj

const vs readonly через рефлексію

Обидва ключових слова задають поля, значення яких не змінюється — але механізм різний, і рефлексія це чітко розрізняє:

Модифікатор	IsLiteral	IsInitOnly	SetValue	Примітки
const	true	false	FieldAccessException	значення вбудоване в IL; завжди static логічно
static readonly	false	true + IsStatic=true	FieldAccessException	після ініціалізації типу змінити не можна
readonly (екземпляра)	false	true	**працює**	readonly — обмеження компілятора, не рефлексії

Модифікатор	IsLiteral	IsInitOnly	SetValue	Примітки
Звичайне поле	false	false	працює	—

Для **const**-поля GetValue(null) повертає вбудоване значення (аргумент obj ігнорується — поле не прив'язане до екземпляра). Записати його неможливо в принципі: значення const компілятор **копіює** в IL кожного місця використання, тож у пам'яті немає комірки, яку можна було б змінити. Тому SetValue на const кидає FieldAccessException:

```
C#
1  FieldInfo maxAgeField = typeof(PatientRecord)
2      .GetField("MaxAge", BindingFlags.Public | BindingFlags.Static)!;
3
4  // const — читаємо через null
5  object? val = maxAgeField.GetValue(null);
6  Console.WriteLine($"MaxAge = {val}"); // MaxAge = 150
7
8  // SetValue → FieldAccessException
9  maxAgeField.SetValue(null, 200); // кидає!
```

З readonly-полями ситуація інша, і вона часто дивує. Ключове слово readonly — це обіцянка, яку перевіряє **компілятор C#**: він не дозволить написати присвоєння поза конструктором. Але рефлексія працює вже на рівні CLR, тож для readonly-поля **екземпляра** SetValue спрацьовує без жодних хаків:

```
C#
1  // у класі PatientRecord: private readonly double _weight;
2  FieldInfo weight = typeof(PatientRecord)
3      .GetField("_weight", BindingFlags.NonPublic | BindingFlags.Instance)!;
4
5  Console.WriteLine(weight.IsInitOnly);   // True — поле readonly
6  weight.SetValue(patient, 82.5);         // працює: значення змінено
```

Винятком є **static readonly**-поля: починаючи з .NET Core 3.0, після того як статичний конструктор типу відпрацював, CLR забороняє їх змінювати — SetValue кидає FieldAccessException. Причина в оптимізації: JIT має право вбудувати значення static readonly у машинний код як константу, і зміна поля після цього призвела б до неузгодженого стану.

Практичний висновок: readonly захищає від помилок у **вашому** коді, але не від рефлексії. Якщо бібліотека чи серіалізатор має доступ до об'єкта через рефлексію, він технічно може змінити його readonly-

поля. Робити так у продакшн-коді не варто — це руйнує інваріанти класу; такий прийом трапляється хіба що в тестових інструментах і десеріалізаторах.

Клас PropertyInfo

PropertyInfo описує одну властивість. Ключові члени:

Властивість / Метод	Що повертає
.Name	ім'я властивості: "FullName", "BirthDate"
.PropertyType	Type значення властивості
.CanRead	true якщо є getter
.CanWrite	true якщо є setter або init-accessor
.GetMethod	MethodInfo? — getter як метод
.SetMethod	MethodInfo? — setter як метод
.GetValue(obj)	прочитати значення властивості
.SetValue(obj, val)	записати значення властивості
.GetIndexParameters()	ParameterInfo[] — для індексаторів (this[int])
.GetAccessors()	масив MethodInfo — get + set разом

Ключовий момент: .GetMethod і .SetMethod — це справжні MethodInfo. Через них можна перевірити модифікатор доступу аксесора:

```
C#
1  PropertyInfo prop = typeof(PatientRecord).GetProperty("RecordId")!;
2
3  Console.WriteLine(prop.CanRead);               // true
4  Console.WriteLine(prop.CanWrite);              // true (init)
5  Console.WriteLine(prop.GetMethod!.IsPublic);   // true
6  Console.WriteLine(prop.SetMethod!.IsPublic);   // true (але init-only)
```

init-only setter (C# 9) — обмеження лише для компілятора

C# 9 ввів init-аксесор: public string Name { get; init; }. Такий аксесор — це звичайний метод-setter (set_Name), у сигнатурі якого компілятор

залишає спеціальну позначку — обов'язковий модифікатор IsExternalInit. Компілятор C#, побачивши цю позначку, дозволяє викликати setter лише з конструктора або ініціалізатора об'єкта (new PatientRecord { Name = "…" }). А от для рефлексії й CLR це звичайний setter.

Тому рефлексія показує CanWrite = true, і SetValue **справді записує** значення — навіть після того, як об'єкт уже створено:

```
C#
1  PropertyInfo nameProp = typeof(PatientRecord).GetProperty("FullName")!;
2  Console.WriteLine(nameProp.CanWrite); // True — setter існує
3
4  var patient = new PatientRecord("Іван Коваль", new DateTime(1980, 1, 1));
5
6  nameProp.SetValue(patient, "Петро Мельник"); // працює
7  Console.WriteLine(patient.FullName);         // Петро Мельник
```

Так само працює і запис у приховане backing-поле властивості (<FullName>k__BackingField): компілятор позначає його як readonly (IsInitOnly = true), але, як ми бачили вище, для полів екземпляра це не заважає FieldInfo.SetValue.

Як відрізнити init від звичайного set, якщо це потрібно (наприклад, генератору документації)? Перевірити модифікатор у сигнатурі setter-а:

```
C#
1  bool isInit = nameProp.SetMethod?.ReturnParameter
2      .GetRequiredCustomModifiers()
3      .Any(m => m.Name == "IsExternalInit") ?? false;
```

Отже, init, як і readonly, — це гарантія **компілятора** для звичайного коду, а не захист від рефлексії. Серіалізатори (наприклад, System.Text.Json) якраз користуються цим, щоб заповнювати init-властивості під час десеріалізації. У прикладному коді змінювати init-властивості через рефлексію після створення об'єкта — антипатерн: це порушує задум автора класу, який розраховував на незмінність.

Workflow GetValue і SetValue

Порядок дій при роботі з полями і властивостями через рефлексію відповідає трьом крокам з діаграми (знайти → прочитати → записати):

```
C#
 1  Type t = typeof(PatientRecord);
 2  var patient = new PatientRecord("Олена Петренко", new DateTime(1985, 3, 12));
 3
 4  // 1. Отримати FieldInfo / PropertyInfo з BindingFlags
```

```
 5  FieldInfo? bmiField = t.GetField("_bmi",
 6      BindingFlags.NonPublic | BindingFlags.Instance);
 7
 8  PropertyInfo? fullNameProp = t.GetProperty("FullName",
 9      BindingFlags.Public | BindingFlags.Instance);
10
11  // 2. Перевірити CanRead / CanWrite (для Property)
12  if (fullNameProp is { CanRead: true, CanWrite: true })
13  {
14      // 3. GetValue → object? (boxing для value-типів)
15      object? name = fullNameProp.GetValue(patient);
16      Console.WriteLine(name); // Олена Петренко
17
18      // 4. SetValue — val приводиться до PropertyType
19      fullNameProp.SetValue(patient, "Марія Гончар");
20
21      // 5. Розкастити результат
22      string updated = (string)fullNameProp.GetValue(patient)!;
23      Console.WriteLine(updated); // Марія Гончар
24  }
25
26  // Читання приватного поля через FieldInfo
27  object? rawBmi = bmiField?.GetValue(patient);
28  decimal bmi = rawBmi is null ? 0m : (decimal)rawBmi;
```

Для **статичних** полів і властивостей аргумент obj у GetValue/SetValue передається як null:

```
C#
1  FieldInfo? maxAge = t.GetField("MaxAge", BindingFlags.Public |
   BindingFlags.Static);
2  object? val = maxAge?.GetValue(null);  // null для static
```

Практичне застосування

Серіалізація — JSON/XML-серіалізатори проходять по всіх публічних властивостях типу і перетворюють значення в рядок. Базова схема:

```
C#
1  var props = t.GetProperties(BindingFlags.Public | BindingFlags.Instance)
2               .Where(p => p.CanRead);
3  foreach (var p in props)
4      json[p.Name] = p.GetValue(obj)?.ToString() ?? "null";
```

ORM (Entity Framework Core) — при матеріалізації запиту EF Core записує значення кожної колонки в відповідну властивість об'єкта через SetValue або через скомпільований Expression. Рефлексія тут — фундамент маппінгу колонка↔властивість.

AutoMapper — при конфігурації маппінгу бібліотека одноразово будує словник PropertyInfo для джерела і цілі, потім кожен маппінг читає GetValue і пише SetValue. Кешування PropertyInfo є обов'язковим.

Клонування об'єктів — deep copy через перебір GetFields з копіюванням GetValue→SetValue для кожного поля нового примірника. Правильна реалізація пропускає IsLiteral-поля і backing-поля, які обробляються через властивості.

Валідатори — перевірка атрибутів на властивостях [Required], [Range], [MaxLength] через GetCustomAttributes() + GetValue() для отримання фактичного значення.

Приклад 1 — PatientRecord: інспекція полів і маніпуляції значеннями

```C#
using System;
using System.Linq;
using System.Reflection;
using System.Runtime.CompilerServices;

Type t = typeof(PatientRecord);
var patient = new PatientRecord("Олена Петренко", new DateTime(1985, 3, 12)) { RecordId = 42 };

Console.WriteLine("=== Усі поля (з категоризацією) ===\n");

var allFields = t.GetFields(
    BindingFlags.Public | BindingFlags.NonPublic |
    BindingFlags.Instance | BindingFlags.Static);

foreach (var f in allFields)
{
    bool isBacking = f.IsDefined(typeof(CompilerGeneratedAttribute), false);
    string category = f.IsLiteral ? "const" : f.IsInitOnly ? "readonly"
                    : isBacking ? "backing" : "field";
    string access   = f.IsPublic ? "public" : f.IsFamily ? "protected" : "private";
    string stat     = f.IsStatic ? " static" : "";

    object? val = f.IsStatic ? f.GetValue(null) : f.GetValue(patient);
    Console.WriteLine($"  [{category}] {access}{stat} {f.FieldType.Name,-12} {f.Name,-30} = {val}");
}

Console.WriteLine("\n=== Властивості з CanRead/CanWrite ===\n");

foreach (var p in t.GetProperties(BindingFlags.Public | BindingFlags.Instance))
{
    string rw = (p.CanRead ? "get" : "") + (p.CanWrite ? "; set" : "");
    bool isInit = p.SetMethod?.ReturnParameter
                    .GetRequiredCustomModifiers()
                    .Any(m => m.Name == "IsExternalInit") ?? false;
    if (isInit) rw += " [init-only]";

    object? val = p.CanRead ? p.GetValue(patient) : "—";
    Console.WriteLine($"  {p.PropertyType.Name,-12} {p.Name,-15} [{rw}]  = {val}");
```

```
}

Console.WriteLine("\n=== SetValue через PropertyInfo ===\n");

var ageProp = t.GetProperty("BirthDate");
ageProp?.SetValue(patient, new DateTime(1990, 7, 22));
Console.WriteLine($"  BirthDate після SetValue: {patient.BirthDate:yyyy-MM-dd}");

Console.WriteLine("\n=== SetValue через FieldInfo (приватне поле _bmi) ===\n");

var bmiField = t.GetField("_bmi", BindingFlags.NonPublic | BindingFlags.Instance);
bmiField?.SetValue(patient, 27.4m);
Console.WriteLine($"  _bmi через GetValue: {bmiField?.GetValue(patient)}");
Console.WriteLine($"  GetBmi(): {patient.GetBmi()}");

Console.WriteLine("\n=== const-поле через GetValue(null) ===\n");

var maxAgeField = t.GetField("MaxAge", BindingFlags.Public | BindingFlags.Static);
Console.WriteLine($"  MaxAge = {maxAgeField?.GetValue(null)}");

public class PatientRecord
{
    public const int MaxAge = 150;
    public static readonly string DefaultWard = "Загальна терапія";

    public string FullName { get; set; }
    public DateTime BirthDate { get; set; }
    public int RecordId { get; init; }

    private decimal _bmi;
    private string? _notes;

    public PatientRecord(string fullName, DateTime birthDate)
    {
        FullName = fullName;
        BirthDate = birthDate;
    }

    public void SetBmi(decimal bmi) => _bmi = Math.Max(0, bmi);
    public decimal GetBmi() => _bmi;
    public void AddNote(string text) => _notes = text;
    public string GetSummary() => $"Пацієнт: {FullName}, ІМТ: {_bmi:F1}";
}
```

Приклад 2 — ObjectCloner<T>: deep copy через FieldInfo

```
C#
using System;
using System.Linq;
using System.Reflection;
using System.Runtime.CompilerServices;

var original = new PatientRecord("Іван Коваль", new DateTime(1975, 5, 20)) { RecordId = 7 };
original.SetBmi(25.1m);
```

```
original.AddNote("Перший візит");

var clone = ObjectCloner<PatientRecord>.Clone(original);

Console.WriteLine("=== Оригінал ===");
Console.WriteLine(original.GetSummary());
Console.WriteLine($"RecordId: {original.RecordId}");

Console.WriteLine("\n=== Клон (після Clone) ===");
Console.WriteLine(clone.GetSummary());
Console.WriteLine($"RecordId: {clone.RecordId}");

// Перевірка незалежності
original.SetBmi(30.0m);
Console.WriteLine("\n=== Після зміни оригіналу ===");
Console.WriteLine($"Оригінал ІМТ: {original.GetBmi()}");
Console.WriteLine($"Клон ІМТ:     {clone.GetBmi()}");

public class PatientRecord
{
    public const int MaxAge = 150;

    public string FullName { get; set; }
    public DateTime BirthDate { get; set; }
    public int RecordId { get; init; }

    private decimal _bmi;
    private string? _notes;

    public PatientRecord(string fullName, DateTime birthDate)
    {
        FullName = fullName;
        BirthDate = birthDate;
    }

    public void SetBmi(decimal bmi) => _bmi = Math.Max(0, bmi);
    public decimal GetBmi() => _bmi;
    public void AddNote(string text) => _notes = text;
    public string GetSummary() => $"Пацієнт: {FullName}, ІМТ: {_bmi:F1},
Нотатка: {_notes}";
}

public static class ObjectCloner<T> where T : class
{
    private static readonly FieldInfo[] Fields = typeof(T)
        .GetFields(BindingFlags.Public | BindingFlags.NonPublic |
BindingFlags.Instance)
        .Where(f => !f.IsLiteral)  // пропускаємо const (IsLiteral)
        .ToArray();

    public static T Clone(T source)
    {
        // Створюємо порожній об'єкт без виклику конструктора
        T target = (T)RuntimeHelpers.GetUninitializedObject(typeof(T));

        foreach (var field in Fields)
        {
            object? val = field.GetValue(source);

```

```
65              // readonly-поля і backing-поля init-властивостей мають
    IsInitOnly=true,
66              // але для полів екземпляра SetValue все одно працює
67              // Для value-типів та рядків shallow copy достатньо
68              field.SetValue(target, val);
69          }
70
71          return target;
72      }
73
74      public static void PrintCopiedFields()
75      {
76          Console.WriteLine($"\nObjectCloner<{typeof(T).Name}> копіює
    {Fields.Length} полів:");
77          foreach (var f in Fields)
78          {
79              bool isBacking = f.IsDefined(typeof(CompilerGeneratedAttribute),
    false);
80              bool isReadonly = f.IsInitOnly;
81              string tags = string.Join(", ", new[]
82              {
83                  isBacking  ? "backing" : null,
84                  isReadonly ? "readonly/init" : null,
85                  f.IsStatic ? "static" : null,
86              }.Where(x => x is not null));
87
88              Console.WriteLine($"  {f.FieldType.Name,-15} {f.Name}" +
89                                (tags.Length > 0 ? $"  [{tags}]" : ""));
90          }
91      }
92  }
```

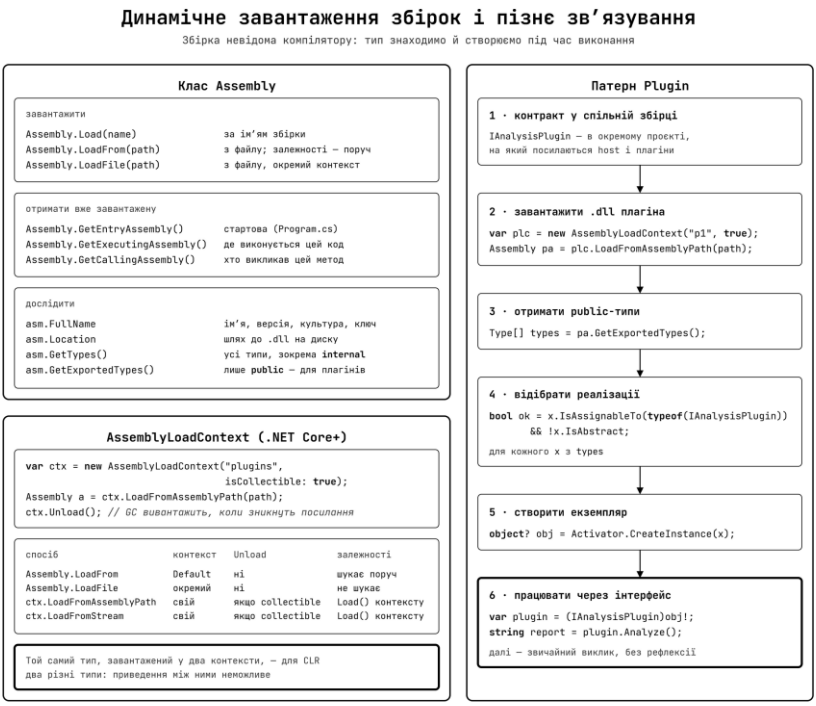
ObjectCloner<T> демонструє реальну техніку клонування через рефлексію: RuntimeHelpers.GetUninitializedObject створює об'єкт в обхід конструктора (не потрібно знати, які аргументи йому передати; старий аналог FormatterServices.GetUninitializedObject позначено застарілим), IsLiteral фільтрує const-поля (вони не копіюються, бо належать типу, а не екземпляру), а копіювання на рівні полів автоматично охоплює і backing-поля init-властивостей: хоча вони мають IsInitOnly = true, для полів екземпляра FieldInfo.SetValue працює. Копіювати саме поля, а не властивості, тут правильно: так ми не викликаємо логіку setter-ів і отримуємо точну копію стану.

14.5. Динамічне завантаження збірок та пізнє зв'язування

У попередніх розділах ми досліджували типи збірок, вже присутніх у програмі на момент старту. Але рефлексія дозволяє йти далі: **завантажувати збірки динамічно** — з файлової системи, з потоку байтів, або навіть з коду, згенерованого під час виконання. Саме це

відкриває двері до плагінних архітектур, hot reload і DI-контейнерів, що сканують цілі папки з .dll.

Ключова різниця між статичним і динамічним завантаженням: при статичному посилання на збірку визначається в .csproj і CLR завантажує її автоматично перед запуском. При динамічному — збірка невідома компілятору, і вся відповідальність за пошук типів, перевірку сумісності і виклик методів лежить на коді рефлексії.



Динамічне завантаження збірок — Assembly та AssemblyLoadContext

Клас Assembly

Assembly з простору імен System.Reflection — центральний клас для роботи зі збірками. Він надає як методи **завантаження**, так і методи **дослідження** вмісту.

Методи завантаження

Метод	Коли використовувати
Assembly.Load(name)	завантаження за ім'ям збірки (AssemblyName або рядок) через контекст за замовчуванням
Assembly.LoadFrom(path)	завантаження з файлу в контекст за замовчуванням; залежності CLR шукає поруч
Assembly.LoadFile(path)	завантаження у **окремий** контекст, без резолюції залежностей
Assembly.GetEntryAssembly()	стартова збірка (Program.cs)
Assembly.GetCallingAssembly()	збірка, що викликала поточний метод
Assembly.GetExecutingAssembly()	збірка, де виконується поточний код

Де шукає Assembly.Load. У .NET Framework за ім'ям збірки CLR шукав також у глобальному кеші збірок (GAC). У .NET Core і .NET 5+ **GAC немає**: Assembly.Load(name) шукає серед збірок застосунку (перелік у файлі .deps.json) і спільних бібліотек платформи, тобто в контексті завантаження за замовчуванням.

LoadFrom vs LoadFile: LoadFrom реєструє шлях у CLR-завантажувачі і CLR може знайти залежності автоматично. LoadFile завантажує файл **ізольовано** — якщо збірка залежить від SomeLib.dll, CLR її не знайде автоматично. Тому LoadFile зазвичай використовують лише для інспекції метаданих.

Властивості для дослідження

Властивість / Метод	Що повертає
.FullName	рядок "MyMedApp, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
.GetName().Version	об'єкт Version (Major, Minor, Build, Revision)
.Location	повний шлях до .dll на диску
.GetTypes()	**всі** типи збірки, включаючи internal і compiler-generated

Властивість / Метод	Що повертає
.GetExportedTypes()	тільки public типи — безпечніше для плагінного сканування
.GetType(name)	Type? — знайти конкретний тип за повним ім'ям
.GetCustomAttributes()	атрибути рівня збірки ([assembly: AssemblyVersion(...)])

Різниця між GetTypes() і GetExportedTypes() важлива: GetTypes() може кинути ReflectionTypeLoadException, якщо не вдається завантажити якийсь тип (наприклад, відсутня залежність). GetExportedTypes() надійніший у цьому сенсі, бо повертає тільки типи, доступні ззовні.

```
C#
 1  // Безпечна версія GetTypes з обробкою ReflectionTypeLoadException
 2  static IEnumerable<Type> SafeGetTypes(Assembly asm)
 3  {
 4      try { return asm.GetExportedTypes(); }
 5      catch (ReflectionTypeLoadException ex)
 6      {
 7          // Повертаємо ті типи, що вдалося завантажити
 8          return ex.Types.Where(t => t is not null)!;
 9      }
10  }
```

AssemblyLoadContext — ізоляція та Unload

У .NET Framework завантажені збірки не можна було вивантажити без перезапуску AppDomain (що фактично означало зупинку процесу). .NET Core і .NET 5+ вирішили цю проблему через AssemblyLoadContext (ALC).

ALC — це ізольований контекст завантаження. Кожен ALC має власний набір завантажених збірок. Один і той самий тип, завантажений у двох різних ALC, вважається **різними типами** з точки зору CLR.

```
C#
 1  // Стандартний (незмінний) контекст — містить усі збірки програми
 2  AssemblyLoadContext defaultCtx = AssemblyLoadContext.Default;
 3
 4  // Ізольований колектибельний контекст для плагіна
 5  var pluginCtx = new AssemblyLoadContext("PluginCtx", isCollectible: true);
 6  Assembly pluginAsm = pluginCtx.LoadFromAssemblyPath(fullPath);
```

```
 7
 8  // ... використання плагіна ...
 9
10  // Вивантаження: CLR виконає збірку сміття після звільнення всіх посилань
11  pluginCtx.Unload();
```

Умови для успішного Unload:

1. ALC створений з isCollectible: true.
2. Немає жодних живих посилань на типи або екземпляри з цього ALC (у стеку, статичних полях, обробниках подій, делегатах).
3. Unload() — це сигнал, а не миттєве вивантаження. GC вирішує, коли фактично звільнити пам'ять.

Метод	Ізоляція	Unload	Залежності
LoadFrom(path)	у Default ALC	ні	CLR вирішує авто
LoadFile(path)	часткова	ні	не вирішує
ALC.LoadFromAssemblyPath	повна	так (якщо collectible)	треба реалізувати Load()
ALC.LoadFromStream	повна	так	з байтового потоку

Реалізація PluginLoadContext

Для коректного завантаження плагіна з власними залежностями необхідно перевизначити Load(AssemblyName):

```
C#
 1  class PluginLoadContext : AssemblyLoadContext
 2  {
 3      private readonly AssemblyDependencyResolver _resolver;
 4
 5      public PluginLoadContext(string pluginPath) : base(isCollectible: true)
 6      {
 7          _resolver = new AssemblyDependencyResolver(pluginPath);
 8      }
 9
10      protected override Assembly? Load(AssemblyName name)
11      {
12          string? path = _resolver.ResolveAssemblyToPath(name);
13          return path is not null ? LoadFromAssemblyPath(path) : null;
14          // null → CLR шукатиме у Default ALC (shared типи: System.*,
    Microsoft.*)
15      }
```

```
16  }
```

AssemblyDependencyResolver читає .deps.json файл поруч з .dll і знає, де лежать залежності плагіна.

Патерн Plugin через рефлексію

Плагінна архітектура — найпопулярніший сценарій динамічного завантаження. Вона складається з 6 кроків:

Крок 1. Визначити інтерфейс у **host-збірці** (окремий проєкт, на який посилається і host, і плагіни):

```
C#
1  // Збірка MedAnalysis.Contracts.dll
2  namespace MedAnalysis.Contracts;
3
4  public interface IAnalysisPlugin
5  {
6      string Name { get; }
7      string Description { get; }
8      AnalysisResult Analyze(PatientRecord[] records);
9  }
```

Крок 2. Завантажити .dll плагіна через ALC:

```
C#
1  var ctx = new PluginLoadContext(pluginPath);
2  Assembly asm = ctx.LoadFromAssemblyPath(pluginPath);
```

Крок 3. Отримати всі public типи:

```
C#
1  IEnumerable<Type> types = SafeGetTypes(asm);
```

Крок 4. Відфільтрувати по інтерфейсу — тільки конкретні, не абстрактні класи:

```
C#
1  Type pluginInterface = typeof(IAnalysisPlugin);
2  IEnumerable<Type> pluginTypes = types
3      .Where(t => pluginInterface.IsAssignableFrom(t)
4               && !t.IsAbstract
5               && !t.IsInterface);
```

IsAssignableFrom(t) — це «зворотний» IsAssignableTo: pluginInterface.IsAssignableFrom(t) означає «t реалізує pluginInterface». Аналогічно t.IsAssignableTo(pluginInterface) (з'явилось у .NET 5).

Крок 5. Створити екземпляри через Activator:

```
C#
1  var plugins = pluginTypes
2      .Select(t => (IAnalysisPlugin)Activator.CreateInstance(t)!)
3      .ToList();
```

Крок 6. Використовувати через інтерфейс:

```
C#
1  foreach (var plugin in plugins)
2      Console.WriteLine($"[{plugin.Name}] {plugin.Analyze(records)}");
```

Перевірка відповідності типу

При скануванні збірок часто потрібно перевірити різні умови відповідності:

Перевірка	Що означає
t.IsAssignableTo(typeof(IPlugin))	t реалізує інтерфейс або успадковує тип
t.IsSubclassOf(typeof(BaseRecord))	t є нащадком конкретного класу
t.GetInterface("IAnalysisPlugin") != null	реалізує інтерфейс за ім'ям (для cross-ALC)
!t.IsAbstract && !t.IsInterface	лише конкретні інстанційовані типи

Приклад 1 — Дослідження поточної збірки та пізнє зв'язування

```
C#
 1  using System;
 2  using System.Linq;
 3  using System.Reflection;
 4
 5  // Досліджуємо власну збірку програми
 6  Assembly self = Assembly.GetExecutingAssembly();
 7
 8  Console.WriteLine("=== Інформація про збірку ===\n");
 9  Console.WriteLine($"  FullName  : {self.FullName}");
10  Console.WriteLine($"  Version   : {self.GetName().Version}");
11  Console.WriteLine($"  Location  : {self.Location}");
12
13  Console.WriteLine("\n=== Усі public типи ===\n");
14  foreach (Type t in self.GetExportedTypes())
15  {
16      string kind = t.IsInterface ? "interface" : t.IsAbstract ? "abstract" :
    "class";
```

```
    Console.WriteLine($"  [{kind}] {t.Name}");
}

Console.WriteLine("\n=== Пізнє зв'язування: IAnalysisPlugin ===\n");

Type pluginInterface = typeof(IAnalysisPlugin);

// Знаходимо всі реалізації IAnalysisPlugin у поточній збірці
var plugins = self.GetExportedTypes()
    .Where(t => pluginInterface.IsAssignableFrom(t) && !t.IsAbstract &&
!t.IsInterface)
    .Select(t => (IAnalysisPlugin)Activator.CreateInstance(t)!)
    .ToList();

// Тестові записи
var records = new[]
{
    new PatientRecord("Іван Коваль",   28.5m, 72),
    new PatientRecord("Марія Гончар",  32.1m, 88),
    new PatientRecord("Петро Мельник", 22.3m, 65),
};

foreach (var plugin in plugins)
{
    Console.WriteLine($"Плагін: {plugin.Name} — {plugin.Description}");
    AnalysisResult result = plugin.Analyze(records);
    Console.WriteLine($"  Результат: {result.Summary}");
    Console.WriteLine();
}

// ==================== Контракти ====================
public interface IAnalysisPlugin
{
    string Name { get; }
    string Description { get; }
    AnalysisResult Analyze(PatientRecord[] records);
}

public record AnalysisResult(string Summary);
public record PatientRecord(string FullName, decimal Bmi, int HeartRate);

// ==================== Плагіни ====================
public class BmiAnalysisPlugin : IAnalysisPlugin
{
    public string Name => "BMI Analyzer";
    public string Description => "Аналіз ІМТ пацієнтів";

    public AnalysisResult Analyze(PatientRecord[] records)
    {
        int overweight = records.Count(r => r.Bmi >= 30);
        double avg = records.Average(r => (double)r.Bmi);
        return new AnalysisResult($"Середній ІМТ: {avg:F1}, з надмірною вагою:
{overweight}/{records.Length}");
    }
}

public class HeartRatePlugin : IAnalysisPlugin
{
    public string Name => "HeartRate Analyzer";
    public string Description => "Аналіз частоти серцевих скорочень";
```

```
75
76      public AnalysisResult Analyze(PatientRecord[] records)
77      {
78          int elevated = records.Count(r => r.HeartRate > 80);
79          int min = records.Min(r => r.HeartRate);
80          int max = records.Max(r => r.HeartRate);
81          return new AnalysisResult($"ЧСС: min={min}, max={max}, підвищена у
    {elevated}/{records.Length}");
82      }
83  }
```

Приклад 2 — PluginScanner: динамічне сканування збірок у папці

```
C#
  1  using System;
  2  using System.Collections.Generic;
  3  using System.IO;
  4  using System.Linq;
  5  using System.Reflection;
  6  using System.Runtime.Loader;
  7
  8  // Демонстрація PluginScanner без реального запису на диск:
  9  // симулюємо завантаження з поточної збірки як "зовнішнього" плагіна.
 10  var scanner = new PluginScanner<IReportPlugin>();
 11  scanner.ScanAssembly(Assembly.GetExecutingAssembly());
 12
 13  Console.WriteLine($"Знайдено плагінів: {scanner.Plugins.Count}\n");
 14
 15  var patients = new[]
 16  {
 17      new MedRecord("Олена Петренко", new DateTime(1985, 3, 12), 25.4m),
 18      new MedRecord("Сергій Бондар",  new DateTime(1970, 8, 5),  31.2m),
 19      new MedRecord("Тетяна Кравець", new DateTime(1995, 1, 20), 19.8m),
 20  };
 21
 22  foreach (var plugin in scanner.Plugins)
 23  {
 24      Console.WriteLine($"=== {plugin.ReportName} ===");
 25      plugin.GenerateReport(patients);
 26      Console.WriteLine();
 27  }
 28
 29  Console.WriteLine("=== PluginScanner статистика ===");
 30  scanner.PrintStats();
 31
 32  // ==================== Контракт ====================
 33  public interface IReportPlugin
 34  {
 35      string ReportName { get; }
 36      void GenerateReport(MedRecord[] records);
 37  }
 38
 39  public record MedRecord(string FullName, DateTime BirthDate, decimal Bmi);
 40
 41  // ==================== Реалізації ====================
 42  public class DemographicsReport : IReportPlugin
 43  {
 44      public string ReportName => "Демографічний звіт";
```

```
45
46      public void GenerateReport(MedRecord[] records)
47      {
48          var today = DateTime.Today;
49          foreach (var r in records)
50          {
51              int age = today.Year - r.BirthDate.Year;
52              if (r.BirthDate.Date > today.AddYears(-age)) age--;
53              Console.WriteLine($"  {r.FullName}: {age} років");
54          }
55          Console.WriteLine($"  Середній вік: {records.Average(r => today.Year -
    r.BirthDate.Year):F0}");
56      }
57  }
58
59  public class BmiCategoryReport : IReportPlugin
60  {
61      public string ReportName => "Звіт за категоріями ІМТ";
62
63      public void GenerateReport(MedRecord[] records)
64      {
65          foreach (var r in records)
66          {
67              string cat = r.Bmi switch
68              {
69                  < 18.5m => "Недостатня вага",
70                  < 25.0m => "Норма",
71                  < 30.0m => "Надмірна вага",
72                  _       => "Ожиріння"
73              };
74              Console.WriteLine($"  {r.FullName}: ІМТ {r.Bmi:F1} — {cat}");
75          }
76      }
77  }
78
79  // ==================== Scanner ====================
80  public class PluginScanner<TPlugin> where TPlugin : class
81  {
82      private readonly List<TPlugin> _plugins = new();
83      private readonly List<(string Assembly, string TypeName)> _found = new();
84
85      public IReadOnlyList<TPlugin> Plugins => _plugins;
86
87      public void ScanAssembly(Assembly asm)
88      {
89          Type contractType = typeof(TPlugin);
90          IEnumerable<Type> candidates;
91
92          try { candidates = asm.GetExportedTypes(); }
93          catch (ReflectionTypeLoadException ex)
94          { candidates = ex.Types.Where(t => t is not null)!; }
95
96          foreach (var type in candidates.Where(t =>
97              contractType.IsAssignableFrom(t) && !t.IsAbstract &&
    !t.IsInterface))
98          {
99              try
100             {
101                 var instance = (TPlugin)Activator.CreateInstance(type)!;
102                 _plugins.Add(instance);
```

```
                _found.Add((asm.GetName().Name ?? "?", type.Name));
            }
            catch (Exception ex)
            {
                Console.WriteLine($"  [!] Не вдалося створити {type.Name}:
{ex.InnerException?.Message ?? ex.Message}");
            }
        }
    }

    public void ScanDirectory(string directory)
    {
        foreach (string dll in Directory.GetFiles(directory, "*.dll"))
        {
            try
            {
                var ctx = new
AssemblyLoadContext($"Plugin_{Path.GetFileNameWithoutExtension(dll)}",
isCollectible: true);
                Assembly asm =
ctx.LoadFromAssemblyPath(Path.GetFullPath(dll));
                ScanAssembly(asm);
            }
            catch (Exception ex)
            {
                Console.WriteLine($"  [!] Не вдалося завантажити {dll}:
{ex.Message}");
            }
        }
    }

    public void PrintStats()
    {
        Console.WriteLine($"  Контракт  : {typeof(TPlugin).FullName}");
        Console.WriteLine($"  Знайдено  : {_plugins.Count} плагінів");
        foreach (var (asm, typeName) in _found)
            Console.WriteLine($"    [{asm}] {typeName}");
    }
}
```

PluginScanner<TPlugin> показує повний цикл: GetExportedTypes() з безпечним перехопленням ReflectionTypeLoadException, фільтрація за контрактним інтерфейсом через IsAssignableFrom, Activator.CreateInstance із перехопленням помилок ініціалізації, і AssemblyLoadContext з isCollectible: true для кожної папкової збірки — щоб пам'ять можна було звільнити після деактивації плагіна.

14.6. Атрибути у .NET

Атрибути — це спосіб додати **декларативні метадані** до елементів коду: класів, методів, властивостей, параметрів, полів, або навіть до цілої збірки. На відміну від коментарів, атрибути вбудовуються в IL-

код збірки і доступні через рефлексію під час виконання. Саме це робить їх потужним інструментом: фреймворки читають атрибути і автоматично змінюють поведінку — валідують, серіалізують, маплять на БД, реєструють у DI.

Усі атрибути успадковують від абстрактного класу System.Attribute. Синтаксично атрибут — це клас із конструктором; значення, передані у квадратних дужках, — це аргументи конструктора або іменовані властивості.



Атрибути — Attribute, AttributeUsage, читання через рефлексію

Вбудовані атрибути .NET

.NET постачає велику кількість стандартних атрибутів для різних потреб:

Атрибут	Де застосовується	Призначення
[Obsolete]	метод, клас	попереджає про застарілий API, опціонально — помилка компіляції
[Serializable]	клас	дозволяє серіалізацію (BinaryFormatter)
[NonSerialized]	поле	виключає поле з серіалізації

Атрибут	Де застосовується	Призначення
[DllImport]	метод	P/Invoke — виклик нативного коду
[Flags]	enum	дозволяє бітові операції над значеннями
[AttributeUsage]	клас атрибута	обмежує де і як може застосовуватись атрибут
[CallerMemberName]	параметр	компілятор підставляє ім'я викликача (для INotifyPropertyChanged)
[Required]	властивість	валідація — поле обов'язкове (DataAnnotations)
[JsonPropertyName]	властивість	ім'я ключа при JSON-серіалізації
[CompilerGenerated]	будь-що	позначає згенерований компілятором код (backing fields і т.д.)

Суфікс Attribute — конвенція

За конвенцією .NET класи атрибутів мають суфікс Attribute. При застосуванні суфікс можна **опустити**: [Obsolete] і [ObsoleteAttribute] — повністю еквівалентні записи. Компілятор шукає обидва варіанти. Якщо є обидва класи (Foo і FooAttribute), використовується повне ім'я з Attribute.

Власний атрибут — анатомія

Власний атрибут — це звичайний клас, успадкований від System.Attribute. Йому можна додати конструктор з позиційними параметрами і публічні властивості для іменованих параметрів.

```
C#
 1  [AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited =
    true)]
 2  public sealed class ValidationRangeAttribute : Attribute
 3  {
 4      public double Min { get; }
 5      public double Max { get; }
```

```
 6      public string? Message { get; set; }
 7
 8      public ValidationRangeAttribute(double min, double max)
 9      {
10          Min = min;
11          Max = max;
12      }
13  }
```

Застосування з позиційними і іменованими параметрами:

```
C#
1  public class PatientRecord
2  {
3      [ValidationRange(0, 300, Message = "ІМТ поза нормою")]
4      public decimal Bmi { get; set; }
5
6      [ValidationRange(30, 220)]
7      public int HeartRate { get; set; }
8  }
```

— 0, 300 — позиційні аргументи → передаються в конструктор

— Message = "..." — іменований аргумент → встановлюється як властивість після виклику конструктора

Клас атрибута рекомендується оголошувати sealed — атрибути рідко потребують успадкування, а sealed запобігає випадковому розширенню і прискорює перевірку CLR.

[AttributeUsage] — обмеження застосування

[AttributeUsage] декорує сам клас атрибута і задає три речі:

1. AttributeTargets — де може застосовуватись атрибут:

```
C#
1  [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
2  // Тільки на класах і структурах
3
4  [AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
5  // Тільки на властивостях і полях
6
7  [AttributeUsage(AttributeTargets.All)]
8  // Будь-який елемент коду
```

Повний список цілей: Assembly, Module, Class, Struct, Enum, Constructor, Method, Property, Field, Event, Interface, Parameter, Delegate, ReturnValue, GenericParameter, All.

2. AllowMultiple — чи можна застосувати кілька разів:

```
C#
```

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
public sealed class RouteAttribute : Attribute
{
    public string Path { get; }
    public RouteAttribute(string path) => Path = path;
}

// Тепер можна:
[Route("/patients")]
[Route("/records")]
public class PatientController { }
```

За замовчуванням AllowMultiple = false — компілятор заборонить дублювання.

3. Inherited — чи передається атрибут нащадкам:

```
C#
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
// Атрибут на BaseClass НЕ видно через
typeof(DerivedClass).GetCustomAttributes(true)

[AttributeUsage(AttributeTargets.Class, Inherited = true)]  // за замовчуванням
// Атрибут успадковується нащадками
```

Важливо: для **інтерфейсів** Inherited завжди false незалежно від налаштування — це рішення CLR. Клас, що реалізує інтерфейс, не «успадковує» атрибути цього інтерфейсу.

Читання атрибутів через рефлексію

Один і той самий API читання атрибутів підтримують усі нащадки MemberInfo — Type, MethodInfo, ConstructorInfo, PropertyInfo, FieldInfo, EventInfo, — а також класи, що **не** є MemberInfo, але теж можуть нести атрибути: ParameterInfo (атрибути параметрів), Assembly і Module (атрибути рівня збірки, [assembly: …]). Об'єднує їх спільний інтерфейс ICustomAttributeProvider:

Метод	Що повертає
IsDefined(Type, bool)	true/false — найшвидша перевірка наявності
GetCustomAttribute<T>()	T? — один атрибут або null
GetCustomAttributes<T>()	IEnumerable<T> — усі атрибути типу T
GetCustomAttributes(bool inherit)	object[] — усі атрибути

Метод	Що повертає
GetCustomAttributesData()	IList<CustomAttributeData> — сирі дані без інстанціювання

Параметр inherit у GetCustomAttributes(bool):

— true — шукати атрибут і в базових класах (для методів і класів)

— false — тільки на цьому конкретному елементі

Дві пастки параметра inherit:

4. Для **PropertyInfo і EventInfo** методи екземпляра GetCustomAttributes(inherit) та IsDefined(type, inherit) **ігнорують** inherit: true — атрибут з virtual-властивості базового класу на override-властивості нащадка не знайдеться. Щоб урахувати успадкування, використовують статичні методи класу Attribute:

```
C#
1  PropertyInfo prop = typeof(InpatientRecord).GetProperty("Bmi")!;
2  prop.GetCustomAttributes(typeof(ValidationRangeAttribute), true);           //
   [] — inherit ігнорується
3  Attribute.GetCustomAttributes(prop, typeof(ValidationRangeAttribute), true); //
   [ValidationRange] — знайдено
```

5. Атрибути, поставлені на **інтерфейс**, не переходять на клас, що його реалізує, — незалежно від inherit і Inherited (див. вище).

```
C#
 1  Type t = typeof(PatientRecord);
 2
 3  // Перевірити наявність (без виділення пам'яті)
 4  bool hasRange = t.GetProperty("Bmi")!
 5      .IsDefined(typeof(ValidationRangeAttribute), false);
 6
 7  // Отримати один конкретний атрибут
 8  ValidationRangeAttribute? attr = t.GetProperty("Bmi")!
 9      .GetCustomAttribute<ValidationRangeAttribute>();
10
11  if (attr is not null)
12      Console.WriteLine($"IMT: [{attr.Min}..{attr.Max}]");
13
14  // Отримати всі атрибути методу
15  MethodInfo m = t.GetMethod("GetSummary")!;
16  foreach (var a in m.GetCustomAttributes(inherit: false))
17      Console.WriteLine(a.GetType().Name);
```

GetCustomAttributesData — без інстанціювання атрибута

GetCustomAttributesData() повертає CustomAttributeData — опис атрибута через його конструктор і аргументи **без** виклику new. Це корисно для інструментів аналізу коду та генераторів, де інстанціювати атрибут може бути небажаним:

```
C#
1  foreach (var data in typeof(PatientRecord).GetCustomAttributesData())
2  {
3      Console.WriteLine($"Атрибут: {data.AttributeType.Name}");
4      foreach (var arg in data.ConstructorArguments)
5          Console.WriteLine($"  arg: {arg.Value}");
6      foreach (var named in data.NamedArguments)
7          Console.WriteLine($"  {named.MemberName} = {named.TypedValue.Value}");
8  }
```

Приклад 1 — PatientRecord: власні атрибути та рефлексійний валідатор

```
C#
  1  using System;
  2  using System.Collections.Generic;
  3  using System.Linq;
  4  using System.Reflection;
  5
  6  // Валідація через атрибути
  7  var patient1 = new PatientRecord("Іван Коваль",   26.5m, 72,  38.6f);
  8  var patient2 = new PatientRecord("Марія Гончар",  35.0m, 110, 36.6f);
  9  var patient3 = new PatientRecord("Петро Мельник", 15.0m, 55,  40.2f);
 10
 11  var validator = new AttributeValidator<PatientRecord>();
 12
 13  foreach (var patient in new[] { patient1, patient2, patient3 })
 14  {
 15      var errors = validator.Validate(patient);
 16      if (errors.Count == 0)
 17          Console.WriteLine($"  {patient.FullName}: ✓ валідний");
 18      else
 19      {
 20          Console.WriteLine($"  {patient.FullName}: {errors.Count} помилок");
 21          foreach (var e in errors)
 22              Console.WriteLine($"    [{e.Property}] {e.Message}");
 23      }
 24  }
 25
 26  // ==================== Атрибути ====================
 27  [AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited =
     true)]
 28  public sealed class MedRangeAttribute : Attribute
 29  {
 30      public double Min { get; }
 31      public double Max { get; }
 32      public string Message { get; set; } = "Значення поза допустимим
     діапазоном";
```

```
33
34      public MedRangeAttribute(double min, double max)
35      {
36          Min = min;
37          Max = max;
38      }
39  }
40
41  [AttributeUsage(AttributeTargets.Property)]
42  public sealed class MedRequiredAttribute : Attribute
43  {
44      public string Message { get; set; } = "Поле обов'язкове";
45  }
46
47  // ==================== Модель ====================
48  public class PatientRecord
49  {
50      [MedRequired(Message = "ПІБ пацієнта обов'язкове")]
51      public string FullName { get; }
52
53      [MedRange(10.0, 60.0, Message = "ІМТ має бути від 10 до 60")]
54      public decimal Bmi { get; }
55
56      [MedRange(30, 200, Message = "ЧСС має бути від 30 до 200")]
57      public int HeartRate { get; }
58
59      [MedRange(35.0, 40.0, Message = "Температура поза нормою (35–40°C)")]
60      public float Temperature { get; }
61
62      public PatientRecord(string fullName, decimal bmi, int heartRate, float
    temperature)
63      {
64          FullName = fullName;
65          Bmi = bmi;
66          HeartRate = heartRate;
67          Temperature = temperature;
68      }
69  }
70
71  // ==================== Валідатор ====================
72  public record ValidationError(string Property, string Message);
73
74  public class AttributeValidator<T>
75  {
76      private static readonly PropertyInfo[] Props = typeof(T)
77          .GetProperties(BindingFlags.Public | BindingFlags.Instance);
78
79      public List<ValidationError> Validate(T obj)
80      {
81          var errors = new List<ValidationError>();
82
83          foreach (var prop in Props)
84          {
85              object? value = prop.GetValue(obj);
86
87              // [MedRequired]
88              var required = prop.GetCustomAttribute<MedRequiredAttribute>();
89              if (required is not null)
90              {
```

```
                if (value is null || (value is string s &&
    string.IsNullOrWhiteSpace(s)))
                    errors.Add(new ValidationError(prop.Name,
    required.Message));
            }

            // [MedRange]
            var range = prop.GetCustomAttribute<MedRangeAttribute>();
            if (range is not null && value is not null)
            {
                double numeric = Convert.ToDouble(value);
                if (numeric < range.Min || numeric > range.Max)
                    errors.Add(new ValidationError(prop.Name, range.Message));
            }
        }

        return errors;
    }
}
```

Приклад 2 — AttributeInspector: повна карта атрибутів типу

```
C#
using System;
using System.Linq;
using System.Reflection;

// Виводимо повну карту атрибутів PatientRecord
AttributeInspector.Inspect(typeof(PatientRecord));

// ==================== Атрибути ====================
[AttributeUsage(AttributeTargets.Property, AllowMultiple = true)]
public sealed class TagAttribute : Attribute
{
    public string Value { get; }
    public TagAttribute(string value) => Value = value;
}

[AttributeUsage(AttributeTargets.Method)]
public sealed class AuditLogAttribute : Attribute
{
    public string Action { get; set; } = "read";
    public bool IncludeArgs { get; set; } = false;
}

// ==================== Модель ====================
[Obsolete("Використовуйте ExtendedPatientRecord з версії 2.0", error: false)]
public class PatientRecord
{
    [Tag("pii"), Tag("required")]
    public string FullName { get; set; } = "";

    [Tag("metric")]
    public decimal Bmi { get; set; }

    public int HeartRate { get; set; }

    [AuditLog(Action = "export", IncludeArgs = true)]
```

```
    public string GetSummary() => $"{FullName}, IMT: {Bmi}";

    [AuditLog(Action = "update")]
    public void UpdateBmi(decimal newBmi) => Bmi = newBmi;
}

// ==================== Inspector ====================
public static class AttributeInspector
{
    public static void Inspect(Type t)
    {
        Console.WriteLine($"╔══ Атрибути типу: {t.Name} ══╗\n");

        // Атрибути типу
        var typeAttrs = t.GetCustomAttributes(inherit: false);
        Console.WriteLine($"  [Type] {t.Name}");
        foreach (var a in typeAttrs)
            Console.WriteLine($"    ← [{a.GetType().Name.Replace("Attribute", "")}] {FormatAttr(a)}");

        // Властивості
        Console.WriteLine("\n  [Properties]");
        foreach (var p in t.GetProperties(BindingFlags.Public | BindingFlags.Instance))
        {
            var attrs = p.GetCustomAttributes(inherit: false);
            if (!attrs.Any()) continue;
            Console.WriteLine($"    {p.PropertyType.Name,-12} {p.Name}");
            foreach (var a in attrs)
                Console.WriteLine($"      ← [{a.GetType().Name.Replace("Attribute", "")}] {FormatAttr(a)}");
        }

        // Методи
        Console.WriteLine("\n  [Methods]");
        var flags = BindingFlags.Public | BindingFlags.Instance | BindingFlags.DeclaredOnly;
        foreach (var m in t.GetMethods(flags).Where(x => !x.IsSpecialName))
        {
            var attrs = m.GetCustomAttributes(inherit: false);
            if (!attrs.Any()) continue;
            var ps = string.Join(", ", m.GetParameters().Select(p => p.ParameterType.Name));
            Console.WriteLine($"    {m.ReturnType.Name,-10} {m.Name}({ps})");
            foreach (var a in attrs)
                Console.WriteLine($"      ← [{a.GetType().Name.Replace("Attribute", "")}] {FormatAttr(a)}");
        }

        Console.WriteLine("\n╚══════════════════════════╝");
    }

    private static string FormatAttr(object attr) => attr switch
    {
        TagAttribute t          => $"Value=\"{t.Value}\"",
        AuditLogAttribute al    => $"Action=\"{al.Action}\", IncludeArgs={al.IncludeArgs}",
        ObsoleteAttribute obs   => $"\"{obs.Message}\" IsError={obs.IsError}",
        _                       => attr.ToString() ?? ""
    };
```

```
89  }
```

AttributeInspector демонструє ключові техніки: GetCustomAttributes(inherit: false) на Type, PropertyInfo і MethodInfo, фільтрацію IsSpecialName для пропуску get_*/set_*, підтримку AllowMultiple = true через [Tag] (кілька атрибутів на одній властивості), і роботу зі стандартними атрибутами на кшталт [Obsolete] — всі вони читаються абсолютно так само, як і власні.

Питання для самоконтролю — Розділ 14. Рефлексія

14.1. Введення у рефлексію. Клас System.Type

1. Що таке рефлексія і що робить її можливою у .NET? Чому збірки .dll і .exe містять не лише IL-код, але й метадані?
2. Назвіть п'ять реальних інструментів або фреймворків (DI-контейнери, ORM, тестові фреймворки), які використовують рефлексію «під капотом». Поясніть, для чого саме вона їм потрібна.
3. В чому різниця між трьома способами отримати об'єкт Type:

- typeof(PatientRecord) - patient.GetType() - Type.GetType("PatientRecord") Коли кожен із них є єдиним або кращим вибором?

4. Що виведе наступний код? Поясніть результат:

```
C#
1  object obj = new InpatientRecord("R001", 7);
2  Console.WriteLine(obj.GetType().Name);
3  Console.WriteLine(typeof(MedicalRecord).Name);
4  Console.WriteLine(obj.GetType() == typeof(MedicalRecord));
```

(де InpatientRecord успадковує MedicalRecord)

5. Яке головне застереження щодо продуктивності рефлексії? В яких частинах програми вона доречна, а в яких — ні?
6. Що повертає GetInterfaces() і чому кожний елемент масиву є об'єктом Type, а не рядком?
7. Що повертає Type.IsAssignableFrom(other) і чим це відрізняється від other.IsSubclassOf(type)? Напишіть перевірку, яка визначає, чи реалізує тип t інтерфейс IDisposable.

8. Спроєктуйте функцію PrintTypeInfo(Type t), яка виводить: ім'я типу, чи є він класом/структурою/інтерфейсом, базовий клас і список реалізованих інтерфейсів.

14.2. Застосування рефлексії та дослідження типів

9. Що таке MemberInfo і чому GetMembers() повертає саме цей тип, а не конкретніші MethodInfo чи FieldInfo?
10. Поясніть різницю між DeclaringType та ReflectedType для члена типу. Наведіть конкретний приклад з ієрархією класів, де вони відрізняються.
11. Що таке BindingFlags? Чому при явній передачі прапорців потрібно вказувати всі потрібні категорії (наприклад, і Public, і Instance)?
12. Що виведе наступний код і чому name не з'являється у списку?

```C#
Type t = typeof(Person);
foreach (var m in t.GetMembers())
    Console.WriteLine($"{m.MemberType} {m.Name}");

public class Person
{
    string name;
    public int Age { get; set; }
}
```

13. Що таке «backing field» автовластивості (<Name>k__BackingField)? Як його виявити через рефлексію і чому потрібно вміти відрізняти backing fields від звичайних полів?
14. Напишіть код, який отримує тільки **власні** (не успадковані) публічні та приватні методи-екземпляри класу PatientRecord, використовуючи правильну комбінацію BindingFlags.
15. Чому GetMember("AddNote") може повернути масив з кількох елементів? Як отримати конкретне перевантаження методу, якщо потрібна версія з параметрами (string, DateTime)?
16. Опишіть практичний сценарій, де необхідно динамічно аналізувати структуру невідомого класу (наприклад, у системі звітності або серіалізатора), і поясніть, який набір BindingFlags підходить для отримання повної картини типу.

14.3. Дослідження методів та конструкторів за допомогою рефлексії

17. Що таке MethodInfo? Які властивості дозволяють визначити, є метод public, private, static, virtual або abstract?
18. Що таке IsSpecialName для методів і для чого воно потрібне? Чому при виведенні списку методів типу часто фільтрують !m.IsSpecialName?
19. Що виведе наступний код? Який тип результату і чому?

```
C#
1  var patient = new PatientRecord("Іван", new DateTime(1985, 1, 1));
2  MethodInfo m = typeof(PatientRecord).GetMethod("GetSummary")!;
3  object? result = m.Invoke(patient, null);
4  Console.WriteLine(result?.GetType().Name);
```

20. Що таке TargetInvocationException і чому CLR перегортає реальне виключення в цю обгортку? Як правильно «розпакувати» справжнє виключення?
21. В чому різниця між ConstructorInfo.Invoke і Activator.CreateInstance? Назвіть сценарій, де краще використовувати кожен із підходів.
22. Що таке .cctor (статичний конструктор) з точки зору рефлексії? Чому його можна побачити через GetConstructors, але не можна викликати через Invoke?
23. Напишіть код, який динамічно викликає метод SetBmi(decimal) для масиву пацієнтів, кешуючи MethodInfo у статичне поле, а не шукаючи його щоразу всередині циклу.
24. Що таке MethodInfo.CreateDelegate і яку перевагу він дає порівняно з Invoke? Для якого сценарію використання делегат є кращим вибором: одноразовий виклик чи виклик у циклі на тисячах об'єктів?

14.4. Дослідження полів та властивостей за допомогою рефлексії

25. В чому принципова відмінність між FieldInfo і PropertyInfo? Чому властивість — це абстракція, а не просто «поле з доступом»?
26. Що виведе наступний код і чому?

```
C#
1  public class Patient
2  {
```

```
3       public const int MaxAge = 150;
4       public readonly string Ward = "Cardiology";
5   }
6
7   var fi = typeof(Patient).GetFields(BindingFlags.Public | BindingFlags.Static |
    BindingFlags.Instance);
8   foreach (var f in fi)
9       Console.WriteLine($"{f.Name}: IsLiteral={f.IsLiteral},
    IsInitOnly={f.IsInitOnly}");
```

27. Як прочитати значення приватного поля _bmi об'єкта через рефлексію? Напишіть код із правильним BindingFlags.
28. Що таке init-аксесор у C# 9 і чому PropertyInfo.CanWrite може вводити в оману? Як обійти обмеження init через FieldInfo.SetValue?
29. Як прочитати значення статичного поля через FieldInfo.GetValue(null)? Що означає null як аргумент у цьому контексті?
30. Назвіть три реальні сценарії застосування PropertyInfo.GetValue і PropertyInfo.SetValue у відомих вам фреймворках або бібліотеках.
31. Напишіть метод static void PrintProperties(object obj), який через рефлексію виводить імена і значення всіх публічних властивостей об'єкта. Обробіть випадок, коли CanRead = false.
32. Спроєктуйте спрощений клонувальник ShallowClone<T>(T source), який через FieldInfo копіює всі поля (крім const) у новий об'єкт, створений без конструктора через FormatterServices.GetUninitializedObject. Поясніть, чому const-поля (IsLiteral) потрібно пропускати.

14.5. Динамічне завантаження збірок та пізнє зв'язування

33. В чому різниця між статичним і динамічним завантаженням збірок? Який сценарій вимагає динамічного завантаження?
34. Порівняйте Assembly.LoadFrom(path) і Assembly.LoadFile(path). В яких ситуаціях кожен із них використовується і в чому їхня різниця у вирішенні залежностей?
35. Що таке AssemblyLoadContext (ALC) і яку проблему він вирішує порівняно з AppDomain у .NET Framework? Що означає isCollectible: true?

36. Чому GetTypes() може кинути ReflectionTypeLoadException і як правильно обробити цю ситуацію? Яка альтернатива є надійнішою?
37. Опишіть 6 кроків плагінної архітектури через рефлексію: від визначення інтерфейсу до виклику методів плагіна через цей інтерфейс.
38. Що означає pluginInterface.IsAssignableFrom(t) у контексті сканування типів? Чим це відрізняється від t.IsSubclassOf(pluginInterface)?
39. Що виведе наступний код? Чому перевірка !t.IsAbstract && !t.IsInterface є важливою?

```C#
Type iface = typeof(IAnalysisPlugin);
var candidates = assembly.GetExportedTypes()
    .Where(t => iface.IsAssignableFrom(t));
Console.WriteLine(candidates.Count());
```

(де збірка містить: IAnalysisPlugin (інтерфейс), BmiPlugin : IAnalysisPlugin, AbstractPlugin : IAnalysisPlugin (abstract))

40. Поясніть, що потрібно для коректного Unload() у AssemblyLoadContext. Які три умови мають бути виконані? Що відбувається, якщо хоч одна не виконана?

14.6. Атрибути у .NET

41. Що таке атрибут у .NET? Від якого класу успадковують всі атрибути і де зберігаються їхні дані після компіляції?
42. Поясніть різницю між позиційними аргументами атрибута та іменованими аргументами:

```C#
[MedRange(10.0, 60.0, Message = "ІМТ поза нормою")]
```

Які з цих аргументів потрапляють у конструктор, а які — у властивості?

43. Що таке [AttributeUsage]? Поясніть, що задають параметри AttributeTargets, AllowMultiple і Inherited.
44. Що виведе наступний код і чому?

```C#
[AttributeUsage(AttributeTargets.Class, Inherited = true)]
public sealed class MarkerAttribute : Attribute { }
```

```
3
4  [Marker]
5  public class Base { }
6  public class Derived : Base { }
7
8  Console.WriteLine(typeof(Derived).IsDefined(typeof(MarkerAttribute), true));
9  Console.WriteLine(typeof(Derived).IsDefined(typeof(MarkerAttribute), false));
```

45. Чому рекомендується оголошувати класи атрибутів як sealed? Чому за конвенцією назви атрибутів мають суфікс Attribute, і як це впливає на синтаксис застосування?
46. Назвіть чотири методи для читання атрибутів через рефлексію і поясніть, чим вони відрізняються. Який із них найшвидший для простої перевірки «чи є атрибут»?
47. Напишіть власний атрибут [MedRequired] і клас-валідатор, який через рефлексію перевіряє всі публічні властивості об'єкта і повертає список назв властивостей, що мають цей атрибут, але значення яких є null або порожнім рядком.
48. Порівняйте GetCustomAttribute<T>() і GetCustomAttributesData(). Коли варто використовувати другий метод? Наведіть приклад ситуації, де важливо отримати дані атрибута **без** виклику його конструктора.

Розділ 15. Багатопоточність

15.1. Введення у багатопоточність. Клас Thread

Сучасні програми рідко виконуються суворо послідовно: одна задача за одною. У реальних системах паралельно відбуваються десятки операцій — обробляються запити, зчитуються файли, надсилаються повідомлення, виконуються обчислення. Для ефективного управління такими сценаріями в .NET існує механізм **багатопоточності** — можливість виконувати кілька незалежних послідовностей інструкцій одночасно в межах одного процесу.

Щоб зрозуміти потоки, спочатку варто зрозуміти **процес**. Процес — це екземпляр запущеної програми: він отримує від операційної системи власну ізольовану область пам'яті, набір ресурсів і щонайменше один потік виконання. Потік (thread) — це найменша одиниця виконання всередині процесу. Кілька потоків одного процесу поділяють спільну пам'ять і ресурси процесу, але мають власні стек виклику і лічильник команд. Саме через спільну пам'ять потоки можуть ефективно обмінюватись даними, але саме через неї ж виникають проблеми синхронізації, з якими ми познайомимось у наступних розділах.

Навіщо потрібна багатопоточність: клінічний сценарій

Уявіть реєстратуру медичного центру. При однопотоковій архітектурі все відбувалось би строго по черзі: спочатку зареєструвати пацієнта — потім роздрукувати направлення — потім надіслати SMS — потім завантажити медичну картку з бази даних. Поки база даних відповідає (це може зайняти сотні мілісекунд), програма стоїть і нічого не робить.

Реальна клінічна інформаційна система функціонує паралельно: реєстратура приймає нових пацієнтів, лабораторія обробляє зразки, черговий лікар опитує пацієнтів, система розсилає нагадування — все це одночасно. Саме таку модель і реалізує багатопоточність: замість одного виконавця з довгою чергою задач — кілька паралельних виконавців, кожен зі своїм потоком відповідальності.

Окрім відгуку та паралелізму, багатопоточність дає ще одну перевагу: **ефективне використання процесора**. Поки один потік чекає на відповідь від мережі або диску, інший може виконувати корисну

роботу. На багатоядерних процесорах потоки можуть виконуватись дійсно одночасно на різних ядрах, а не лише чергуватись.

Простір імен System.Threading

Усі класи для роботи з потоками у .NET зосереджені у просторі імен System.Threading. Перед тим як використовувати будь-який клас потоків, необхідно підключити цей простір імен:

```
C#
1  using System.Threading;
```

Основні типи, з якими ми працюватимемо у цьому розділі:

Тип	Призначення
Thread	Основний клас для створення та управління потоком
ThreadStart	Делегат — точка входу потоку без параметрів
ParameterizedThreadStart	Делегат — точка входу потоку з параметром object?
ThreadPriority	Перерахування пріоритетів потоку
ThreadState	Перерахування станів потоку

Клас Thread: властивості

Клас Thread — це центральний інструмент для роботи з потоками у .NET. Кожен запущений потік є об'єктом типу Thread. Щоб отримати посилання на поточний потік (той, у якому виконується код), використовується статична властивість Thread.CurrentThread:

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  // отримати інформацію про головний потік програми
 5  Thread main = Thread.CurrentThread;
 6  main.Name = "MainThread";
 7
 8  Console.WriteLine($"Назва потоку:       {main.Name}");
 9  Console.WriteLine($"ID потоку:          {main.ManagedThreadId.ToString()}");
10  Console.WriteLine($"Стан потоку:        {main.ThreadState.ToString()}");
```

```
11  Console.WriteLine($"Фоновий потік:      {main.IsBackground.ToString()}");
12  Console.WriteLine($"Пріоритет:          {main.Priority.ToString()}");
13  Console.WriteLine($"Активний (живий):   {main.IsAlive.ToString()}");
```

Розглянемо ключові властивості класу Thread:

Name — текстова назва потоку. У .NET Framework назву можна було задати лише один раз (повторне присвоєння кидало InvalidOperationException); починаючи з .NET 6 це обмеження знято, і назву можна змінювати в будь-який момент. Проте змінювати назву без потреби не варто: інструменти діагностики можуть запам'ятати першу. Назва потоку незамінна при налагодженні: у стектрейсах і профайлерах вона дозволяє миттєво розпізнати, який потік що виконував.

ManagedThreadId — унікальний ціло-чисельний ідентифікатор потоку, що присвоюється середовищем виконання (CLR). На відміну від Name, він гарантовано унікальний і незмінний протягом усього часу життя потоку. Корисний для логування і діагностики.

IsAlive — повертає true, якщо потік уже запущений і ще не завершився. Дозволяє перевіряти, чи потік ще активний, не блокуючись у очікуванні.

IsBackground — вказує, чи є потік фоновим. Це ключова властивість, що визначає поведінку при завершенні програми — детально розглянемо нижче.

Priority — задає відносний пріоритет потоку через перерахування ThreadPriority. Детально розглянемо нижче.

ThreadState — поточний стан потоку з перерахування ThreadState. Відображає, чи потік запущений, чи очікує, чи завершений.

Стани потоку

Протягом свого існування потік проходить через кілька станів, описаних у перерахуванні ThreadState. Розуміння цих станів необхідне для правильного управління потоками:

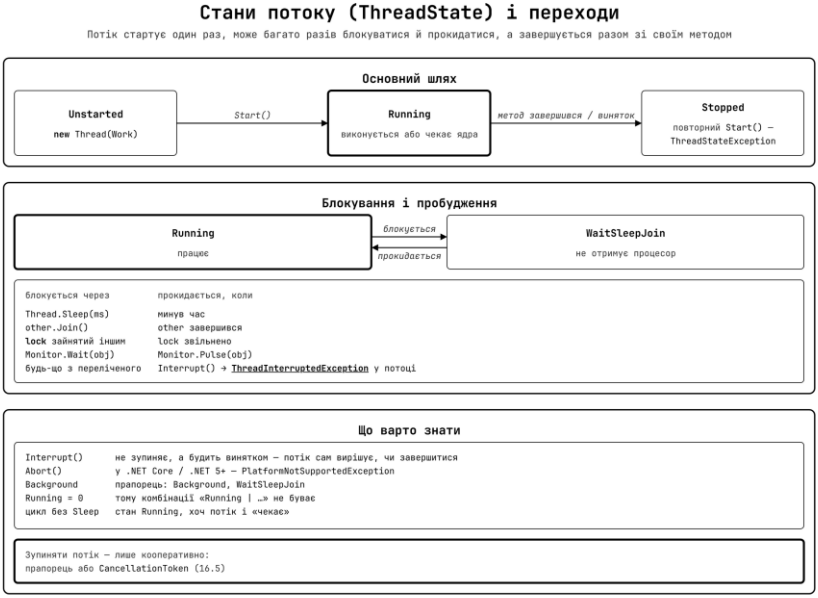
- **Unstarted** — потік створений об'єктом new Thread(...), але метод Start() ще не викликаний. Потік існує як об'єкт, але ще не виконується.
- **Running** — потік виконується або готовий до виконання (чекає виділення процесорного часу планувальником ОС). Числове значення Running дорівнює 0, тому з іншими прапорцями воно не комбінується.
- **WaitSleepJoin** — потік заблокований: через Thread.Sleep() (чекає певний час), Join() (чекає на інший потік), Monitor.Wait()

(чекає сигналу) або тому, що намагається увійти в lock, який тримає інший потік.

— **Stopped** — метод потоку завершився (звичайно або через необроблений виняток). Повторний Start() неможливий — кидає ThreadStateException.

Старі джерела згадують ще метод Abort() і стани AbortRequested/Aborted. У .NET Core і .NET 5+ примусове переривання потоку не підтримується: Abort() кидає PlatformNotSupportedException. Зупиняти потік можна лише кооперативно — сам потік перевіряє прапорець чи CancellationToken (розділ 16.5) і завершує метод.

— **Background** — прапорець, що вказує на фоновий режим потоку. Може поєднуватись з іншими станами.



Стани потоку та переходи між ними

Методи класу Thread

Start — запуск потоку

Метод Start() переводить потік зі стану Unstarted у стан Running. Після виклику Start() планувальник операційної системи вирішує, коли саме

виділити процесорний час цьому потоку — не обов'язково миттєво. Порядок виконання потоків не гарантується: кожен запуск програми може давати різну послідовність.

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  // Створення і запуск потоку для фонової реєстрації пацієнта
 5  Thread registrationThread = new Thread(RegisterPatient);
 6  registrationThread.Name = "RegistrationWorker";
 7  registrationThread.Start();
 8
 9  // Головний потік продовжує роботу паралельно
10  Console.WriteLine($"[{Thread.CurrentThread.Name ?? "Main"}] Головний потік:
    система готова до роботи");
11
12  void RegisterPatient()
13  {
14      Console.WriteLine($"[{Thread.CurrentThread.Name}] Початок реєстрації
    пацієнта...");
15      Thread.Sleep(100); // імітуємо роботу з базою даних
16      Console.WriteLine($"[{Thread.CurrentThread.Name}] Пацієнта зареєстровано
    успішно");
17  }
```

Sleep — призупинення потоку

Метод Thread.Sleep(milliseconds) призупиняє поточний потік на вказану кількість мілісекунд, переводячи його у стан WaitSleepJoin. Поки потік «спить», планувальник ОС може виділяти процесорний час іншим потокам. Thread.Sleep(0) — особливий випадок: він не зупиняє потік, а лише повідомляє планувальнику, що поточний потік готовий поступитись своїм квантом часу.

Sleep широко використовується для:

— Імітації тривалих операцій у прикладах та тестах

— Реалізації простих таймерів і повторних спроб (retry logic)

— Зменшення навантаження від активного очікування (busy-wait)

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  Console.WriteLine("Система моніторингу запущена");
 5
 6  // Потік, що імітує циклічний моніторинг показників
 7  Thread monitorThread = new Thread(MonitorVitals);
 8  monitorThread.Name = "VitalsMonitor";
 9  monitorThread.IsBackground = true; // фоновий — завершиться разом з програмою
10  monitorThread.Start();
11
12  Thread.Sleep(350); // дати монітору попрацювати 350 мс
```

```
13  Console.WriteLine("Сеанс моніторингу завершено");
14
15  void MonitorVitals()
16  {
17      for (int i = 1; i <= 5; i++)
18      {
19          Console.WriteLine($"[{Thread.CurrentThread.Name}] Цикл {i.ToString()}:
    пульс 72 уд/хв, тиск 120/80");
20          Thread.Sleep(100); // перевірка кожні 100 мс
21      }
22  }
```

Join — очікування завершення потоку

Метод Join() блокує викликаючий потік до завершення того потоку, на якому він викликаний. Це основний механізм синхронізації: головний потік може запустити кілька робочих потоків і потім зачекати на завершення всіх через Join().

Існує перевантаження Join(int milliseconds), яке очікує не довше зазначеної кількості мілісекунд і повертає true, якщо потік завершився, або false, якщо час вичерпано.

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  // Клінічний сценарій: паралельний збір аналізів, потім загальний звіт
 5  Thread bloodTest  = new Thread(() => RunTest("Загальний аналіз крові",  400));
 6  Thread urineTest  = new Thread(() => RunTest("Аналіз сечі",             250));
 7  Thread ecgTest    = new Thread(() => RunTest("ЕКГ",                     300));
 8
 9  bloodTest.Name  = "BloodTest";
10  urineTest.Name  = "UrineTest";
11  ecgTest.Name    = "ECG";
12
13  Console.WriteLine("Відправка пацієнта на обстеження...");
14
15  bloodTest.Start();
16  urineTest.Start();
17  ecgTest.Start();
18
19  // Чекаємо завершення всіх трьох потоків
20  bloodTest.Join();
21  urineTest.Join();
22  ecgTest.Join();
23
24  // Цей рядок виконується лише після завершення всіх трьох
25  Console.WriteLine("Всі аналізи отримано. Формую загальний звіт...");
26  Console.WriteLine("Звіт готовий — направляю до лікаря.");
27
28  void RunTest(string testName, int durationMs)
29  {
30      Console.WriteLine($"[{Thread.CurrentThread.Name}] → Початок: {testName}");
31      Thread.Sleep(durationMs);
32      Console.WriteLine($"[{Thread.CurrentThread.Name}] ✓ Готово:  {testName}");
```

```
33  }
```

Якби не було Join(), рядок «Всі аналізи отримано» міг би вивестись ще до того, як всі потоки закінчили роботу. Join() гарантує правильну послідовність: спочатку всі аналізи, потім звіт.

Interrupt — переривання очікування

Метод Interrupt() викидає виняток ThreadInterruptedException у потоці, що знаходиться в стані WaitSleepJoin. Зверніть увагу: Interrupt() не зупиняє потік, а лише «будить» його винятком — якщо потік цей виняток перехопить і продовжить цикл, він працюватиме далі. Якщо потік не перебуває в цьому стані, виняток буде кинуто, як тільки потік у нього перейде. Це дозволяє «розбудити» заблокований потік і попросити його завершити роботу.

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  Thread monitorThread = new Thread(LongMonitoringSession);
 5  monitorThread.Name = "LongMonitor";
 6  monitorThread.IsBackground = true;
 7  monitorThread.Start();
 8
 9  Thread.Sleep(250); // даємо поопрацювати 250 мс
10  Console.WriteLine("[Main] Надзвичайна ситуація — переривання моніторингу");
11  monitorThread.Interrupt(); // надсилаємо переривання
12  monitorThread.Join();       // чекаємо коректного завершення
13  Console.WriteLine("[Main] Моніторинг зупинено коректно");
14
15  void LongMonitoringSession()
16  {
17      try
18      {
19          for (int i = 1; i <= 20; i++)
20          {
21              Console.WriteLine($"[{Thread.CurrentThread.Name}] Цикл моніторингу
    {i.ToString()}");
22              Thread.Sleep(100); // тут може прилетіти ThreadInterruptedException
23          }
24      }
25      catch (ThreadInterruptedException)
26      {
27          // Коректна реакція на переривання: зберегти стан, звільнити ресурси
28          Console.WriteLine($"[{Thread.CurrentThread.Name}] Отримано сигнал
    переривання — завершую сеанс");
29      }
30  }
```

Передній та фоновий потоки

Один із найважливіших концептів управління потоками — розмежування між **переднім** (foreground) і **фоновим** (background) потоком. Ця відмінність визначає, як поведеться програма при завершенні.

Правило просте і критично важливе:

> **Процес .NET завершується, коли завершуються всі передні потоки.** Фонові потоки при цьому примусово зупиняються, незалежно від того, що вони виконують: без винятку і без виконання їхніх блоків finally. Тому фоновий потік, що пише у файл, може бути обірваний посеред запису.

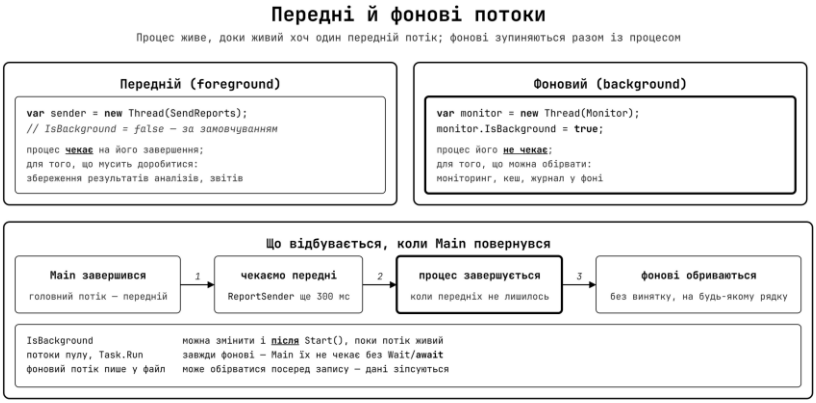
За замовчуванням кожен новий потік є **переднім** (IsBackground = false). Головний потік програми теж є переднім (технічно його можна зробити фоновим, але сенсу в цьому немає: коли Main повертає керування, програма й так завершується, щойно не лишиться інших передніх потоків). Потоки пулу потоків, а отже й задачі Task.Run (розділ 16), — завжди фонові.

Щоб зробити потік фоновим, встановлюють IsBackground = true. Зазвичай це роблять одразу після створення, до Start(), хоча властивість можна змінити й пізніше — поки потік живий:

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  // Передній потік — програма чекатиме його завершення
 5  Thread foregroundWorker = new Thread(SendReports);
 6  foregroundWorker.Name = "ReportSender";
 7  foregroundWorker.IsBackground = false; // значення за замовчуванням
 8
 9  // Фоновий потік — буде зупинений при завершенні головного потоку
10  Thread backgroundMonitor = new Thread(ContinuousMonitor);
11  backgroundMonitor.Name = "BackgroundMonitor";
12  backgroundMonitor.IsBackground = true; // фоновий!
13
14  foregroundWorker.Start();
15  backgroundMonitor.Start();
16
17  Console.WriteLine("[Main] Головний потік завершує роботу");
18  // Програма НЕ завершиться, поки foregroundWorker не закінчить
19  // backgroundMonitor буде зупинений примусово після foregroundWorker
20
21  void SendReports()
22  {
23      Console.WriteLine($"[{Thread.CurrentThread.Name}] Надсилаю звіти...
    (передній потік)");
24      Thread.Sleep(300);
25      Console.WriteLine($"[{Thread.CurrentThread.Name}] Звіти надіслано —
    передній потік завершено");
```

```
}

void ContinuousMonitor()
{
    int cycle = 0;
    while (true) // нескінченний цикл — зупиниться тільки примусово
    {
        cycle++;
        Console.WriteLine($"[{Thread.CurrentThread.Name}] Моніторинг, цикл
{cycle.ToString()}");
        Thread.Sleep(80);
    }
}
```



Передні та фонові потоки: порівняння поведінки

Практичне правило вибору типу потоку:

— **Передній потік** — для критично важливих операцій, які обов'язково мають завершитися: збереження даних, відправлення транзакцій, закриття з'єднань з базою даних. Якщо ця операція не завершиться — цілісність даних буде порушена.

— **Фоновий потік** — для сервісних операцій, які можна безпечно перервати: логування, кешування, моніторинг, виявлення нових файлів. Їх незавершення не призведе до втрати важливих даних.

Пріоритет потоків

Пріоритет потоку (Priority) визначає, яку частину процесорного часу планувальник ОС виділяє потоку відносно інших. Задається через перерахування ThreadPriority:

Значення	Числовий рівень	Клінічне застосування
Lowest	0	Архівація старих записів, генерація звітів
BelowNormal	1	Фонова синхронізація даних
Normal	2	Стандартна обробка запитів *(за замовчуванням)*
AboveNormal	3	Реєстрація пацієнтів в режимі реального часу
Highest	4	Критичні сповіщення: аритмія, зупинка серця

Пріоритет необхідно встановлювати **до** виклику Start(). Після запуску потоку змінити пріоритет технічно можливо, але це рідко доцільно.

```C#
using System;
using System.Threading;

// Потік критичних сповіщень — найвищий пріоритет
Thread alertThread = new Thread(SendCriticalAlert);
alertThread.Name = "CriticalAlert";
alertThread.Priority = ThreadPriority.Highest;

// Потік планових звітів — найнижчий пріоритет
Thread reportThread = new Thread(GenerateReport);
reportThread.Name = "ReportGenerator";
reportThread.Priority = ThreadPriority.Lowest;
reportThread.IsBackground = true;

// Звичайний потік реєстрації — стандартний пріоритет
Thread registrationThread = new Thread(ProcessRegistration);
registrationThread.Name = "Registration";
// Priority.Normal — значення за замовчуванням

alertThread.Start();
reportThread.Start();
registrationThread.Start();

alertThread.Join();
registrationThread.Join();

void SendCriticalAlert()
{
```

```
29      Console.WriteLine($"[{Thread.CurrentThread.Name}] *** КРИТИЧНО: зупинка
    серця — бригада до палати 7! ***");
30      Thread.Sleep(50);
31      Console.WriteLine($"[{Thread.CurrentThread.Name}] Сповіщення доставлено
    всьому персоналу");
32  }
33
34  void ProcessRegistration()
35  {
36      Console.WriteLine($"[{Thread.CurrentThread.Name}] Реєстрація пацієнта
    Петренко О.В.");
37      Thread.Sleep(200);
38      Console.WriteLine($"[{Thread.CurrentThread.Name}] Пацієнта зареєстровано");
39  }
40
41  void GenerateReport()
42  {
43      Console.WriteLine($"[{Thread.CurrentThread.Name}] Генерація місячного звіту
    (фоновий, низький пріоритет)");
44      Thread.Sleep(500);
45      Console.WriteLine($"[{Thread.CurrentThread.Name}] Звіт сформовано");
46  }
```

Важливо розуміти: пріоритет потоку — це лише **підказка** операційній системі, а не жорстка гарантія. ОС може ігнорувати пріоритет або коригувати його динамічно для запобігання «голодуванню» (starvation) — ситуації, коли низькопріоритетний потік ніколи не отримує процесорного часу через постійно активні високопріоритетні потоки. На сучасних багатоядерних системах з невеликим навантаженням різниця між пріоритетами може бути непомітною — потоків вистачить на всіх.

Повний клінічний приклад

Підсумкуємо все вивчене у комплексному прикладі: система первинного прийому пацієнта, де паралельно відбуваються кілька незалежних процесів.

C#

```
1  using System;
2  using System.Threading;
3
4  // --- Запускаємо всі паралельні процеси ---
5  Thread vitalsThread    = CreateThread("VitalsCheck",    MeasureVitals,
   ThreadPriority.AboveNormal, false);
6  Thread historyThread   = CreateThread("HistoryLoad",    LoadMedHistory,
   ThreadPriority.Normal,      false);
7  Thread notifyThread    = CreateThread("NotifySender",   SendNotification,
   ThreadPriority.Normal,      false);
8  Thread archiveThread   = CreateThread("ArchiveWorker",  ArchiveOldData,
   ThreadPriority.Lowest,      true); // фоновий
9
```

```
10  Console.WriteLine("[Прийом] Пацієнт Коваленко Н.В. — починаємо первинний
    прийом");
11  Console.WriteLine($"[Прийом] Головний потік ID:
    {Thread.CurrentThread.ManagedThreadId.ToString()}");
12  Console.WriteLine("");
13
14  vitalsThread.Start();
15  historyThread.Start();
16  notifyThread.Start();
17  archiveThread.Start();
18
19  // Чекаємо обов'язкових процесів (передні потоки)
20  vitalsThread.Join();
21  historyThread.Join();
22  notifyThread.Join();
23
24  Console.WriteLine("");
25  Console.WriteLine("[Прийом] Первинний прийом завершено. Перенаправляємо до
    лікаря.");
26  // archiveThread (фоновий) зупиниться автоматично
27
28  Thread CreateThread(string name, ThreadStart action, ThreadPriority priority,
    bool isBackground)
29  {
30      Thread t = new Thread(action);
31      t.Name         = name;
32      t.Priority     = priority;
33      t.IsBackground = isBackground;
34      return t;
35  }
36
37  void MeasureVitals()
38  {
39      Console.WriteLine($"[{Thread.CurrentThread.Name}] Вимірювання
    показників...");
40      Thread.Sleep(300);
41      Console.WriteLine($"[{Thread.CurrentThread.Name}] Пульс: 78, АТ: 118/76,
    SpO2: 98% — норма");
42  }
43
44  void LoadMedHistory()
45  {
46      Console.WriteLine($"[{Thread.CurrentThread.Name}] Завантаження медичної
    карти...");
47      Thread.Sleep(400);
48      Console.WriteLine($"[{Thread.CurrentThread.Name}] Медична карта завантажена
    (3 попередніх звернення)");
49  }
50
51  void SendNotification()
52  {
53      Console.WriteLine($"[{Thread.CurrentThread.Name}] Надсилання SMS-
    підтвердження...");
54      Thread.Sleep(150);
55      Console.WriteLine($"[{Thread.CurrentThread.Name}] SMS надіслано: 'Ви в
    черзі, орієнтовне очікування 15 хв'");
56  }
57
58  void ArchiveOldData()
59  {
```

```
60      // Фоновий потік — може бути перерваний у будь-який момент
61      for (int i = 1; i <= 10; i++)
62      {
63          Console.WriteLine($"[{Thread.CurrentThread.Name}] Архівація запису
    {i.ToString()}/10...");
64          Thread.Sleep(200);
65      }
66  }
```

У цьому прикладі три передні потоки (VitalsCheck, HistoryLoad, NotifySender) виконуються паралельно та незалежно. Головний потік чекає на всі три через Join(). Фоновий потік ArchiveWorker запущений одночасно, але програма не буде чекати його завершення — він зупиниться, коли завершаться всі передні потоки.

15.2. Створення потоків. ThreadStart та ParameterizedThreadStart

Будь-який потік у .NET починається з делегата — посилання на метод, який потік виконуватиме. Конструктор класу Thread приймає делегат однієї з двох форм: ThreadStart або ParameterizedThreadStart. Розуміння відмінностей між ними та вміння обрати правильний підхід — основа грамотного використання багатопоточності.

Делегат ThreadStart — потік без параметрів

ThreadStart — найпростіша форма: делегат без параметрів і без значення, що повертається:

```
C#
1  public delegate void ThreadStart();
```

Він описує метод, який отримує все необхідне зі свого лексичного оточення (замикання, поля класу, константи) або взагалі не потребує зовнішніх даних. Саме цей делегат підходить для ізольованих завдань, де потік є самодостатнім.

Є три способи передати метод у конструктор Thread: іменований метод, анонімний метод та лямбда-вираз.

Іменований метод

```
C#
 1  using System;
 2  using System.Threading;
 3
```

```
// Класичний спосіб: передаємо ім'я методу
Thread t = new Thread(RunMorningRound);
t.Name = "MorningRound";
t.Start();
t.Join();

void RunMorningRound()
{
    Console.WriteLine($"[{Thread.CurrentThread.Name}] Починаю ранковий обхід палат...");
    Thread.Sleep(200);
    Console.WriteLine($"[{Thread.CurrentThread.Name}] Обхід завершено: 12 пацієнтів, стан стабільний");
}
```

Іменований метод — найзрозуміліший варіант: логіку видно у декларації методу, а не в місці створення потоку. Підходить для складних, об'ємних задач.

Анонімний метод

```
C#
using System;
using System.Threading;

string ward = "Кардіологія";
int patientCount = 8;

// Анонімний метод захоплює ward і patientCount із зовнішньої області
Thread t = new Thread(delegate()
{
    Console.WriteLine($"[Обхід] Починаю обхід відділення '{ward}'");
    Thread.Sleep(150);
    Console.WriteLine($"[Обхід] Оглянуто {patientCount.ToString()} пацієнтів у '{ward}'");
});
t.Start();
t.Join();
```

Анонімний метод зручний, коли логіка проста і тісно пов'язана з контекстом. Він автоматично захоплює змінні зовнішньої функції (замикання).

Лямбда-вираз

```
C#
using System;
using System.Threading;

string ward = "Неврологія";

// Лямбда — найкомпактніший спосіб
Thread t = new Thread(() =>
{
```

```
 9      Console.WriteLine($"[Лямбда] Потік '{Thread.CurrentThread.Name}' обслуговує
   '{ward}'");
10      Thread.Sleep(100);
11      Console.WriteLine($"[Лямбда] Роботу завершено");
12  });
13  t.Name = "NeurologyWorker";
14  t.Start();
15  t.Join();
```

Лямбда-вираз є найпоширенішим сучасним підходом: компактна, читабельна, підтримує замикання. Для простих задач — оптимальний вибір.

Делегат ParameterizedThreadStart — потік з параметром

ParameterizedThreadStart відрізняється від ThreadStart одним параметром типу object?:

```
C#
1  public delegate void ParameterizedThreadStart(object? obj);
```

Відповідно, метод Thread.Start(object? parameter) приймає значення, яке буде передане у тіло потоку. Це дозволяє передавати дані у потік у момент його запуску, не вдаючись до замикань чи спільних полів:

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  // Передаємо ім'я пацієнта як параметр
 5  Thread t1 = new Thread(RegisterPatient);
 6  Thread t2 = new Thread(RegisterPatient);
 7
 8  t1.Name = "RegWorker-1";
 9  t2.Name = "RegWorker-2";
10
11  t1.Start("Олена Коваль");
12  t2.Start("Іван Петренко");
13
14  t1.Join();
15  t2.Join();
16
17  void RegisterPatient(object? param)
18  {
19      // Обов'язково перевіряємо та приводимо тип
20      string name = param as string ?? "Невідомий";
21      Console.WriteLine($"[{Thread.CurrentThread.Name}] Реєстрація: {name}");
22      Thread.Sleep(200);
23      Console.WriteLine($"[{Thread.CurrentThread.Name}] Пацієнта '{name}'
    зареєстровано");
24  }
```

Параметр завжди має тип object?, тому в тілі потоку його необхідно привести до потрібного типу. Якщо тип не збігається — as поверне null, що можна перевірити і обробити. Явне приведення через (T) кине InvalidCastException при невідповідності.

Передача кількох значень через клас-обгортку

Найсуттєве обмеження ParameterizedThreadStart — лише один параметр типу object?. Якщо потоку потрібно передати кілька значень, їх пакують у клас або структуру:

```C#
using System;
using System.Threading;

Thread t1 = new Thread(ProcessAdmission);
Thread t2 = new Thread(ProcessAdmission);

t1.Name = "Admissions-1";
t2.Name = "Admissions-2";

t1.Start(new PatientTask("Марія Шевченко", "Терапія",    1));
t2.Start(new PatientTask("Олег Бондаренко", "Хірургія",  2));

t1.Join();
t2.Join();

void ProcessAdmission(object? param)
{
    if (param is not PatientTask task) return; // pattern matching — безпечне приведення

    Console.WriteLine($"[{Thread.CurrentThread.Name}] Прийом: {task.Name} → {task.Ward} (пріоритет {task.Priority.ToString()})");
    Thread.Sleep(200);
    Console.WriteLine($"[{Thread.CurrentThread.Name}] Завершено: {task.Name} оформлено до відділення '{task.Ward}'");
}

// Клас-обгортка для передачі кількох значень у потік
class PatientTask
{
    public string Name     { get; }
    public string Ward     { get; }
    public int    Priority { get; }

    public PatientTask(string name, string ward, int priority)
    {
        Name     = name;
        Ward     = ward;
        Priority = priority;
    }
}
```

Використання is not PatientTask task — це патерн-матчинг: якщо param не є PatientTask, умова спрацьовує і метод завершується. Це безпечніше і лаконічніше за явне приведення та перевірку на null.

Замикання як альтернатива ParameterizedThreadStart

Коли потрібно передати дані в потік, замикання через лямбда-вираз часто зручніше за ParameterizedThreadStart. Лямбда захоплює змінні безпосередньо, без необхідності приведення типів:

```C#
using System;
using System.Threading;

string[] patients = { "Василь Мороз", "Тетяна Руденко", "Олена Сидоренко" };

Thread[] threads = new Thread[patients.Length];

for (int i = 0; i < patients.Length; i++)
{
    string patient = patients[i]; // важливо: копія для замикання, не сам і
    threads[i] = new Thread(() =>
    {
        Console.WriteLine($"[Worker] Обробка пацієнта: {patient}");
        Thread.Sleep(150);
        Console.WriteLine($"[Worker] Завершено: {patient}");
    });
    threads[i].Name = $"Worker-{(i + 1).ToString()}";
}

foreach (Thread t in threads) t.Start();
foreach (Thread t in threads) t.Join();

Console.WriteLine("Всіх пацієнтів оброблено");
```

Зверніть увагу на рядок string patient = patients[i]: ми копіюємо значення в окрему змінну перед тим, як лямбда її захопить. Це критично важливо. Якщо б лямбда захопила змінну і безпосередньо, всі потоки читали б одне й те саме значення і — те, яке воно буде в момент виконання лямбди, а не в момент її створення. Оскільки цикл завершується швидко, всі потоки могли б прочитати і == 3 і звернутись до одного й того ж елемента масиву (або вийти за межі). Копія виправляє цю проблему.

Порівняння підходів

Усі три підходи — іменований метод, ParameterizedThreadStart і лямбда з замиканням — вирішують одну задачу: зв'язати потік із кодом та даними. Вибір залежить від обсягу логіки та кількості параметрів:

Підхід	Коли обирати
Іменований метод + ThreadStart	Велика, самодостатня логіка потоку, що варта окремого методу
ParameterizedThreadStart	Один або кілька параметрів, що передаються у момент Start()
Лямбда з замиканням	Коротка логіка, що використовує змінні навколишнього контексту

Повний клінічний приклад: паралельна обробка черги

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  LabOrder[] orders =
 5  {
 6      new LabOrder(1, "Коваль М.В.",    "Загальний аналіз крові",   300),
 7      new LabOrder(2, "Петренко І.О.",  "Біохімія крові",            450),
 8      new LabOrder(3, "Бойко О.П.",     "Аналіз сечі",              200),
 9      new LabOrder(4, "Мороз В.К.",     "Коагулограма",             380),
10  };
11
12  Thread[] workers = new Thread[orders.Length];
13
14  for (int i = 0; i < orders.Length; i++)
15  {
16      LabOrder order = orders[i]; // захоплюємо копію для замикання
17      workers[i] = new Thread(() => ProcessLabOrder(order));
18      workers[i].Name = $"Lab-{order.OrderId.ToString()}";
19  }
20
21  Console.WriteLine("=== Лабораторія: запуск паралельної обробки ===");
22
23  foreach (Thread w in workers) w.Start();
24  foreach (Thread w in workers) w.Join();
25
26  Console.WriteLine("=== Всі аналізи оброблено. Результати готові до видачі.
    ===");
27
28  void ProcessLabOrder(LabOrder order)
29  {
30      Console.WriteLine($"[{Thread.CurrentThread.Name}] Початок:
    {order.PatientName} — {order.TestType}");
31      Thread.Sleep(order.ProcessMs);
32      Console.WriteLine($"[{Thread.CurrentThread.Name}] Готово:
    {order.PatientName} — {order.TestType} ({order.ProcessMs.ToString()} мс)");
33  }
34
35  // Завдання для обробки в окремому потоці
36  class LabOrder
```

```
{
    public int    OrderId    { get; }
    public string PatientName { get; }
    public string TestType   { get; }
    public int    ProcessMs  { get; } // час обробки у мс

    public LabOrder(int id, string name, string test, int ms)
    {
        OrderId     = id;
        PatientName = name;
        TestType    = test;
        ProcessMs   = ms;
    }
}
```

Цей приклад демонструє реальну паралельну обробку: чотири лабораторні замовлення виконуються одночасно у чотирьох незалежних потоках. Без багатопоточності загальний час склав би суму всіх затримок (1330 мс); з багатопоточністю — лише час найдовшої операції (450 мс). Метод Join() у фінальному циклі гарантує, що рядок «Всі аналізи оброблено» з'явиться тільки після завершення всіх чотирьох потоків.

15.3. Синхронізація потоків. Оператор lock та клас Lock

Коли кілька потоків одночасно звертаються до спільних даних — читають, модифікують, записують — виникає фундаментальна проблема: **стан перегонів** (race condition). Це ситуація, коли результат виконання програми залежить від непередбачуваного порядку, у якому потоки отримують процесорний час. Стан перегонів є однією з найнебезпечніших помилок у паралельному програмуванні: він не відтворюється стабільно, рідко з'являється на стадії тестування і може призвести до пошкодження даних у виробничій системі.

Демонстрація стану перегонів

Розглянемо класичну проблему: кілька потоків одночасно оновлюють лічильник пацієнтів у черзі:

```
C#
using System;
using System.Threading;

int queueCount = 0; // спільний лічильник

Thread[] threads = new Thread[5];
```

```
 7  for (int i = 0; i < 5; i++)
 8  {
 9      threads[i] = new Thread(() =>
10      {
11          for (int j = 0; j < 1000; j++)
12          {
13              queueCount++; // НЕ є атомарною операцією!
14          }
15      });
16  }
17
18  foreach (Thread t in threads) t.Start();
19  foreach (Thread t in threads) t.Join();
20
21  // Очікуємо 5000, але отримуємо менше через стан перегонів
22  Console.WriteLine($"Очікувано: 5000");
23  Console.WriteLine($"Фактично:  {queueCount.ToString()}");
24  Console.WriteLine($"Різниця:   {(5000 - queueCount).ToString()} — це втрачені
    оновлення!");
```

Результат виявиться меншим за 5000. Чому? Операція queueCount++ виглядає як одна операція, але на рівні процесора це три кроки: **читання** поточного значення, **збільшення** на 1, **запис** нового значення. Якщо два потоки виконують ці кроки одночасно — вони обидва можуть прочитати однакове значення, обидва збільшать його і запишуть однаковий результат. У підсумку два приросту перетворюються на один — оновлення «загублено».

Оператор lock

Найпростіший і найпоширеніший інструмент синхронізації в C# — оператор lock. Він гарантує, що в певний момент часу лише **один** потік може виконувати блок коду, що знаходиться під замком. Решта потоків, що намагаються увійти в цей блок, блокуються і очікують, поки замок не звільниться.

Синтаксис lock:

```
C#
1  lock (об'єкт-замок)
2  {
3      // код, що захищається від паралельного доступу
4  }
```

Об'єктом-замком (lockObject) може бути будь-який **посилальний тип** (reference type) — будь-який клас або object. Важливо: це **не** string (рядкові літерали можуть інтернуватись і бути несподівано спільними) і не значущий тип (int, struct). Стандартна практика — окремий об'єкт-замок, оголошений як приватне статичне або поле екземпляра:

```
C#
1  private readonly object _lock = new object();
```

Виправимо приклад зі стані перегонів за допомогою lock:

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  int queueCount = 0;
 5  object lockObj = new object(); // об'єкт-замок
 6
 7  Thread[] threads = new Thread[5];
 8  for (int i = 0; i < 5; i++)
 9  {
10      threads[i] = new Thread(() =>
11      {
12          for (int j = 0; j < 1000; j++)
13          {
14              lock (lockObj) // лише один потік одночасно
15              {
16                  queueCount++; // тепер безпечно
17              }
18          }
19      });
20  }
21
22  foreach (Thread t in threads) t.Start();
23  foreach (Thread t in threads) t.Join();
24
25  Console.WriteLine($"Очікувано: 5000");
26  Console.WriteLine($"Фактично:  {queueCount.ToString()}");
27  Console.WriteLine($"Різниця:   {(5000 - queueCount).ToString()} — результат
    коректний!");
```

Тепер результат завжди точно 5000. lock є **взаємним виключенням** (mutual exclusion, mutex): поки один потік виконує захищений блок, всі інші чекають у черзі перед вхідними дверима. Коли потік виходить із lock-блоку, замок знімається і один з тих, що чекають, може увійти.

Клінічний приклад: реєстратура

Розглянемо реалістичний клінічний сценарій: реєстратура з кількома операторами, що одночасно записують пацієнтів на прийом.

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  AppointmentScheduler scheduler = new AppointmentScheduler();
 5
 6  // Три оператори паралельно намагаються записати пацієнтів
```

```
string[][] operatorPatients =
{
    new[] { "Коваль М.", "Петренко І.", "Бойко О.", "Мороз В.", "Сидоренко Т.",
            "Руденко Н.", "Кравченко Р.", "Гриценко Л.", "Ткаченко А.",
"Савченко Д." },
    new[] { "Бондаренко Є.", "Шевченко І.", "Яковенко В.", "Романенко О.",
"Захаренко М.",
            "Лисенко Ю.", "Павленко С.", "Олійник В.", "Кириленко Т.",
"Денисенко Н." },
    new[] { "Іваненко А.", "Сергієнко К.", "Тимошенко Б.", "Чорновіл Г.",
"Гнатенко В." }
};

Thread[] operators = new Thread[3];
for (int i = 0; i < 3; i++)
{
    int idx = i; // копія для замикання
    operators[i] = new Thread(() =>
    {
        string opName = $"Реєстратор-{(idx + 1).ToString()}";
        foreach (string patient in operatorPatients[idx])
        {
            scheduler.TryBook(opName, patient);
            Thread.Sleep(20); // невелика затримка між записами
        }
    });
    operators[i].Name = $"Operator-{(i + 1).ToString()}";
}

foreach (Thread t in operators) t.Start();
foreach (Thread t in operators) t.Join();

Console.WriteLine($"\nПідсумок: записано {scheduler.Total.ToString()} пацієнтів
із 20 доступних");

// Розклад лікаря — спільний ресурс для кількох операторів
class AppointmentScheduler
{
    private int _appointmentsToday = 0;
    private readonly int _maxPerDay = 20;
    private readonly object _lock = new object();

    // Повертає true, якщо запис вдався
    public bool TryBook(string operatorName, string patientName)
    {
        lock (_lock) // тільки один оператор одночасно може модифікувати
розклад
        {
            if (_appointmentsToday >= _maxPerDay)
            {
                Console.WriteLine($"[{operatorName}] Відмова: {patientName} —
розклад повний ({_maxPerDay.ToString()}/{_maxPerDay.ToString()})");
                return false;
            }

            _appointmentsToday++;
            Console.WriteLine($"[{operatorName}] Записано: {patientName} —
прийом #{_appointmentsToday.ToString()}");
            return true;
        }
```

```
59      }
60
61      public int Total => _appointmentsToday;
62  }
```

Завдяки lock лічильник _appointmentsToday ніколи не перевищить _maxPerDay, і не виникне ситуації, коли два оператори «одночасно» запишуть 21-го пацієнта. Кожна перевірка-та-запис відбувається атомарно — без можливості втручання іншого потоку між читанням і записом.

Клас Lock (.NET 9)

У .NET 9 з'явився новий клас System.Threading.Lock — спеціалізований тип замку, що замінює lock (object) і забезпечує кращу продуктивність та зручніший API. На відміну від загального object, Lock є явним інструментом синхронізації: компілятор C# 13 для lock над змінною типу Lock генерує виклик EnterScope() замість Monitor.Enter. Але якщо такий об'єкт привести до object і заблокувати вже його, компілятор використає звичайний Monitor (і видасть попередження CS9216) — тому змінну-замок слід оголошувати саме з типом Lock.

Синтаксис залишається майже ідентичним:

```
C#
1  Lock lockObj = new Lock(); // спеціалізований тип
2
3  lock (lockObj) // працює так само
4  {
5      // захищений код
6  }
```

Клас Lock також підтримує явний Enter/Exit через using-патерн:

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  Lock _lock = new Lock();
 5  int _totalBeds = 50;
 6  int _occupiedBeds = 0;
 7
 8  Thread admissions = new Thread(RunAdmissions);
 9  Thread discharges  = new Thread(RunDischarges);
10
11  admissions.Name = "Admissions";
12  discharges.Name  = "Discharges";
13
14  admissions.Start();
15  discharges.Start();
16
```

```
admissions.Join();
discharges.Join();

Console.WriteLine($"Вільних ліжок: {(_totalBeds - _occupiedBeds).ToString()} з
{_totalBeds.ToString()}");

void RunAdmissions()
{
    for (int i = 0; i < 10; i++)
    {
        using (_lock.EnterScope()) // Enter + Exit через using
        {
            if (_occupiedBeds < _totalBeds)
            {
                _occupiedBeds++;
                Console.WriteLine($"[{Thread.CurrentThread.Name}] Пацієнт
госпіталізований. Зайнято: {_occupiedBeds.ToString()}");
            }
        }
        Thread.Sleep(30);
    }
}

void RunDischarges()
{
    Thread.Sleep(50); // небагато затримки, щоб були ліжка для виписки
    for (int i = 0; i < 5; i++)
    {
        using (_lock.EnterScope())
        {
            if (_occupiedBeds > 0)
            {
                _occupiedBeds--;
                Console.WriteLine($"[{Thread.CurrentThread.Name}] Пацієнта
виписано. Зайнято: {_occupiedBeds.ToString()}");
            }
        }
        Thread.Sleep(60);
    }
}
```

EnterScope() повертає об'єкт, який при виклику Dispose() (що відбувається автоматично при виході з using-блоку) знімає замок. Це унеможливлює «забутий» Exit() і гарантує коректне звільнення навіть при виникненні винятку.

Що НЕ варто робити з lock

Не використовуйте this як об'єкт-замок. Якщо зовнішній код також намагається заблокувати той самий об'єкт, це може призвести до непередбачуваної взаємодії:

```
C#
// Погано — this є публічно доступним:
lock (this) { ... }
```

Не використовуйте string як об'єкт-замок. Рядкові літерали в C# можуть інтернуватись — два різних "lockKey" у різних місцях коду можуть посилатись на той самий об'єкт. Це призведе до несподіваної конкуренції між непов'язаними блоками коду.

Не виконуйте довгих операцій усередині lock. Поки один потік виконує довгу операцію під замком (наприклад, запит до бази даних або зчитування файлу), всі інші потоки стоять і чекають. Розміщуйте під замком лише мінімально необхідний код — зазвичай це лише читання-модифікація-запис спільної змінної.

```
C#
 1  // Погано: запит до бази всередині lock тримає замок на весь час запиту
 2  lock (_lock)
 3  {
 4      var result = database.Query("SELECT ...");
 5      _count = result.Count;
 6  }
 7
 8  // Добре: запит виконується поза lock, під замком лише оновлення змінної
 9  var result = database.Query("SELECT ...");
10  lock (_lock)
11  {
12      _count = result.Count;
13  }
```

Взаємне блокування (Deadlock)

Одна з найнебезпечніших пасток при роботі із замками — **взаємне блокування** (deadlock). Воно виникає, коли два або більше потоки блокують один одного: потік A тримає замок X і чекає на замок Y; потік B тримає замок Y і чекає на замок X. Жоден не може продовжити виконання — система «зависла».

```
C#
1  object lockA = new object();
2  object lockB = new object();
3
4  // Потік 1: блокує A, потім намагається заблокувати B
5  lock (lockA) { lock (lockB) { /* ... */ } }
6
7  // Потік 2: блокує B, потім намагається заблокувати A
8  lock (lockB) { lock (lockA) { /* ... */ } } // ← DEADLOCK
```

Основне правило запобігання дедлоку: **завжди захоплюйте кілька замків в одному й тому самому порядку** у всіх потоках. Якщо усі потоки завжди блокують A перед B — дедлок неможливий. Проектування системи синхронізації так, щоб замки захоплювались в

єдиному передбаченому порядку, є важливим аспектом архітектури паралельних систем.

Клас Interlocked — атомарні операції без lock

Для простих операцій над числами (збільшення, зменшення, додавання, обмін) використання lock є надмірним — воно дорожче через захоплення та звільнення монітора. Клас System.Threading.Interlocked надає **атомарні** методи, що виконуються процесором як одна неподільна операція без захоплення замку:

Метод	Опис
Interlocked.Increment(ref n)	Атомарний n++
Interlocked.Decrement(ref n)	Атомарний n--
Interlocked.Add(ref n, value)	Атомарний n += value
Interlocked.Exchange(ref n, value)	Атомарний n = value, повертає старе
Interlocked.CompareExchange(ref n, value, comparand)	Якщо n == comparand, то n = value

```C#
using System;
using System.Threading;

// Лічильник записів пацієнтів — атомарні операції без lock
int appointmentCount = 0;
int cancelledCount   = 0;

Thread[] receptionists = new Thread[4];
for (int i = 0; i < 4; i++)
{
    receptionists[i] = new Thread(() =>
    {
        for (int j = 0; j < 250; j++)
        {
            Interlocked.Increment(ref appointmentCount); // атомарний ++

            if (j % 10 == 0) // кожен десятий скасовуємо
                Interlocked.Increment(ref cancelledCount);
        }
    });
}

foreach (Thread t in receptionists) t.Start();
foreach (Thread t in receptionists) t.Join();

int active = appointmentCount - cancelledCount;
```

```
27  Console.WriteLine($"Всього записів:    {appointmentCount}");   // завжди 1000
28  Console.WriteLine($"Скасовано:         {cancelledCount}");     // завжди 100
29  Console.WriteLine($"Активних записів:  {active}");             // завжди 900
```

Interlocked.CompareExchange — атомарна операція «порівняй і встанови» (CAS, Compare-And-Swap). Вона встановлює нове значення тільки якщо поточне дорівнює очікуваному. Це основа для lock-free алгоритмів:

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  int _status = 0; // 0 = вільно, 1 = зайнято
 5
 6  bool TryAcquire()
 7  {
 8      // Атомарно: якщо _status == 0, встановити 1, повернути старе значення
 9      int previous = Interlocked.CompareExchange(ref _status, 1, 0);
10      return previous == 0; // true якщо вдалось захопити (було 0)
11  }
12
13  void Release()
14  {
15      Interlocked.Exchange(ref _status, 0); // атомарно скинути в 0
16  }
17
18  // Симуляція захоплення ресурсу (наприклад, апарату УЗД)
19  Thread[] technicians = new Thread[3];
20  for (int i = 0; i < 3; i++)
21  {
22      int num = i + 1;
23      technicians[i] = new Thread(() =>
24      {
25          if (TryAcquire())
26          {
27              Console.WriteLine($"Технік-{num}: УЗД-апарат захоплено,
    обстежую...");
28              Thread.Sleep(100);
29              Release();
30              Console.WriteLine($"Технік-{num}: звільнив апарат");
31          }
32          else
33          {
34              Console.WriteLine($"Технік-{num}: апарат зайнятий, спробую
    пізніше");
35          }
36      });
37  }
38
39  foreach (Thread t in technicians) t.Start();
40  foreach (Thread t in technicians) t.Join();
```

Використовуйте Interlocked замість lock для **одиночних операцій над числовими змінними** — це швидше і не може призвести до

дедлоку. Для складніших операцій (перевірка + зміна кількох змінних) все одно потрібен lock.

volatile — видимість змін між потоками

Сучасні процесори та компілятори можуть **кешувати** значення змінних у регістрах або переставляти порядок операцій для оптимізації. Для однопоточного коду це безпечно, але в багатопоточному може призвести до того, що один потік не бачить зміни, зроблені іншим.

Ключове слово volatile повідомляє компілятору і JIT: кожне читання цієї змінної має бути справжнім читанням (без кешування значення в регістрі на весь цикл), а самі читання й записи не можна переставляти з сусідніми операціями пам'яті (читання має семантику acquire, запис — release). Це гарантує **видимість** змін між потоками:

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  var state = new MonitorState(); // прапорець — volatile-поле класу
 5
 6  Thread monitor = new Thread(() =>
 7  {
 8      Console.WriteLine("[Monitor] Старт моніторингу пацієнта...");
 9      int ticks = 0;
10      while (state.IsRunning) // без volatile: JIT міг би прочитати поле один раз
    і крутити цикл вічно
11      {
12          ticks++;
13          Thread.Sleep(50);
14      }
15      Console.WriteLine($"[Monitor] Зупинено після {ticks} тіків моніторингу");
16  });
17
18  monitor.Start();
19  Thread.Sleep(300);
20
21  Console.WriteLine("[Main] Зупиняємо моніторинг...");
22  state.IsRunning = false; // volatile гарантує, що Monitor-потік побачить зміну
23  monitor.Join();
24  Console.WriteLine("[Main] Моніторинг завершено");
25
26  class MonitorState
27  {
28      public volatile bool IsRunning = true; // volatile гарантує видимість між
    потоками
29  }
```

Зверніть увагу: модифікатор volatile можна застосувати **лише до поля** класу чи структури, а не до локальної змінної (volatile bool flag =

true; усередині методу — помилка компіляції CS0106). Тому прапорець винесено в поле класу MonitorState.

volatile гарантує лише **видимість** — що зміна, зроблена одним потоком, буде прочитана іншим. Але не гарантує **атомарності**: volatile int x; x++; все ще є небезпечним, бо ++ — три кроки (читання, збільшення, запис). Для атомарних операцій над числами використовуйте Interlocked.

Засіб	Гарантує	Вартість	Підходить для
volatile	Видимість	Мінімальна	Прапорці bool, статуси — прості поля
Interlocked	Атомарність + видимість	Низька	Лічильники, стани — одиночні числові операції
lock	Атомарність + видимість + взаємне виключення	Помірна	Складні операції над кількома полями

15.4. Клас Monitor

Оператор lock — це зручна синтаксична обгортка над класом System.Threading.Monitor. Кожен lock-блок компілятор перетворює на виклики Monitor.Enter() і Monitor.Exit() у блоці try/finally. Тобто коли ви пишете:

```
C#
1  lock (lockObj)
2  {
3      // код
4  }
```

Компілятор (починаючи з C# 4) генерує приблизно таке — з прапорцем lockTaken, щоб Exit не викликався, якщо виняток стався ще до захоплення замку:

```
C#
 1  bool lockTaken = false;
 2  try
```

```
3  {
4      Monitor.Enter(lockObj, ref lockTaken);
5      // код
6  }
7  finally
8  {
9      if (lockTaken) Monitor.Exit(lockObj);
10 }
```

Monitor надає більш низькорівневий і гнучкий API, ніж lock. Його ключова перевага — можливість **очікування та сигналізації**: потік може добровільно звільнити замок і перейти в режим очікування, а інший потік може «розбудити» його, коли настане відповідний момент. Саме ця можливість дозволяє реалізовувати складні патерни координації між потоками — наприклад, «виробник — споживач».

Методи Monitor

Ключові методи класу Monitor:

Метод	Опис
Monitor.Enter(obj)	Захоплює замок на об'єкті obj. Блокує потік, якщо замок зайнятий
Monitor.Exit(obj)	Звільняє замок. Викликати лише у потоці, що тримає замок
Monitor.TryEnter(obj)	Спроба захопити замок без блокування. Повертає bool
Monitor.TryEnter(obj, ms)	Спроба захопити замок із таймаутом ms мілісекунд
Monitor.Wait(obj)	Звільняє замок і блокує поточний потік до сигналу Pulse/PulseAll. Замок автоматично повертається при пробудженні
Monitor.Pulse(obj)	Сповіщає один потік, що очікує через Wait, про можливість продовження
Monitor.PulseAll(obj)	Сповіщає **всі** потоки, що очікують через Wait

Методи Wait, Pulse і PulseAll можуть викликатись **лише у потоці, що тримає замок** — тобто всередині lock-блоку або між Enter() і Exit(). Порушення цієї умови призведе до SynchronizationLockException.

Monitor.Wait і Monitor.Pulse: механізм координації

Monitor.Wait(obj) виконує три дії атомарно:

1. Звільняє замок на obj
2. Переводить поточний потік у список очікування для obj
3. Блокує поточний потік

Коли інший потік викликає Monitor.Pulse(obj), один потік з цього списку отримує сигнал і може відновити виконання — але спочатку він зобов'язаний знову захопити замок (що відбувається автоматично при виході з Wait).

Ця схема є основою класичного патерну «виробник — споживач»: виробник додає дані до черги і сповіщає споживача через Pulse; споживач чекає через Wait, доки черга не буде заповнена.

Клінічний приклад: черга лабораторних зразків

У клінічній лабораторії зразки надходять від медсестер (виробники) і обробляються лаборантами (споживачі). Черга зразків — це спільний ресурс, доступ до якого потрібно синхронізувати.

```C#
using System;
using System.Collections.Generic;
using System.Threading;

LabQueue queue = new LabQueue();

// Лаборант — споживач (запускаємо першим, він чекатиме на зразки)
Thread labWorker = new Thread(() =>
{
    Console.WriteLine("[Лаборант] Готовий до роботи, очікую зразки...");
    while (true)
    {
        string? sample = queue.TakeSample();
        if (sample == null) break; // черга закрита і порожня — виходимо

        Console.WriteLine($"[Лаборант] Обробляю: '{sample}'");
        Thread.Sleep(150); // час аналізу
        Console.WriteLine($"[Лаборант] Аналіз '{sample}' завершено");
    }
    Console.WriteLine("[Лаборант] Робочий день завершено — всі зразки оброблено");
});
labWorker.Name = "LabWorker";
```

```
labWorker.Start();

// Медсестра — виробник
Thread nurse = new Thread(() =>
{
    string[] samples = { "Кров-Коваль", "Сеча-Петренко", "Кров-Бойко",
"Мокрота-Мороз", "Кров-Сидоренко" };
    foreach (string sample in samples)
    {
        Thread.Sleep(100); // час між надходженнями зразків
        queue.AddSample(sample);
    }
    Console.WriteLine("[Медсестра] Всі зразки здано до лабораторії");
    queue.StopAccepting(); // сигналізуємо, що нових зразків не буде
});
nurse.Name = "Nurse";
nurse.Start();

nurse.Join();
labWorker.Join();
Console.WriteLine("\n=== Лабораторія закрита ===");

// Черга лабораторних зразків — спільний ресурс
class LabQueue
{
    private readonly Queue<string> _samples = new Queue<string>();
    private readonly object _lock = new object();
    private bool _accepting = true; // false = нові зразки більше не надходять

    // Медсестра додає зразок до черги
    public void AddSample(string sample)
    {
        lock (_lock)
        {
            _samples.Enqueue(sample);
            Console.WriteLine($"[Надходження] Зразок '{sample}' додано до черги
(черга: {_samples.Count.ToString()})");
            Monitor.Pulse(_lock); // сповіщаємо лаборанта, що є новий зразок
        }
    }

    // Оголошуємо, що нових зразків більше не буде
    public void StopAccepting()
    {
        lock (_lock)
        {
            _accepting = false;
            Monitor.PulseAll(_lock); // розбуджуємо всіх, хто чекає — вони
перевірять умову
        }
    }

    // Лаборант бере наступний зразок для обробки
    // Повертає null, якщо черга закрита і порожня
    public string? TakeSample()
    {
        lock (_lock)
        {
            // Очікуємо, поки черга стане непорожньою або прийом не зупинять
            while (_samples.Count == 0 && _accepting)
```

```
            {
                Monitor.Wait(_lock); // звільняємо замок і чекаємо
            }

            if (_samples.Count > 0)
                return _samples.Dequeue();

            return null; // черга порожня і прийом зупинено
        }
    }
}
```

Зверніть на конструкцію while (_samples.Count == 0 && _accepting) { Monitor.Wait(_lock); }. Умова перевіряється у **циклі**, а не в if — це критично важливо. Pulse не гарантує, що після пробудження умова дійсно виконана: між сигналом і повторним захопленням замку інший потік (наприклад, ще один лаборант) міг встигнути забрати зразок, а PulseAll будить усіх, хоча зразок дістанеться лише одному. Крім того, у багатьох середовищах (і в загальній теорії умовних змінних) допускаються «хибні пробудження» (spurious wakeup); Monitor у .NET їх не документує, але цикл захищає і від них. Тому завжди перевіряємо умову заново після Wait.

Кілька споживачів з PulseAll

Monitor.Pulse сповіщає лише один потік; Monitor.PulseAll — всі. Якщо є кілька лаборантів, при додаванні партії зразків доцільно розбудити всіх одразу — хай кожен забере по одному:

```
C#
using System;
using System.Collections.Generic;
using System.Threading;

MultiWorkerQueue queue = new MultiWorkerQueue();

// Два лаборанти-споживачі
void RunWorker(string name)
{
    while (true)
    {
        string? item = queue.Consume(name);
        if (item == null) break;
        Thread.Sleep(200);
        Console.WriteLine($"[{name}] Аналіз '{item}' завершено");
    }
    Console.WriteLine($"[{name}] Зміна завершена");
}

Thread w1 = new Thread(() => RunWorker("Лаборант-1")) { Name = "Worker1" };
Thread w2 = new Thread(() => RunWorker("Лаборант-2")) { Name = "Worker2" };

```

```
w1.Start();
w2.Start();

// Виробник
Thread.Sleep(50);
string[] batch = { "Зразок-А", "Зразок-Б", "Зразок-В", "Зразок-Г", "Зразок-Д",
"Зразок-Е" };
foreach (string s in batch)
{
    queue.Produce(s);
    Thread.Sleep(80);
}
queue.Stop();

w1.Join();
w2.Join();
Console.WriteLine("Всі аналізи завершено");

class MultiWorkerQueue
{
    private readonly Queue<string> _items = new Queue<string>();
    private readonly object _lock = new object();
    private bool _running = true;

    public void Produce(string item)
    {
        lock (_lock)
        {
            _items.Enqueue(item);
            Monitor.PulseAll(_lock); // розбуджуємо всіх лаборантів
        }
    }

    public void Stop()
    {
        lock (_lock)
        {
            _running = false;
            Monitor.PulseAll(_lock);
        }
    }

    public string? Consume(string workerName)
    {
        lock (_lock)
        {
            while (_items.Count == 0 && _running)
                Monitor.Wait(_lock);

            if (_items.Count > 0)
            {
                string item = _items.Dequeue();
                Console.WriteLine($"[{workerName}] Взяв зразок: {item}");
                return item;
            }
            return null;
        }
    }
}
```

Monitor.TryEnter — спроба без блокування

Monitor.TryEnter(obj) намагається захопити замок і повертає true або false негайно, без очікування. Це корисно, коли потік має корисну альтернативну роботу, якщо замок зайнятий:

```C#
using System;
using System.Threading;

object equipmentLock = new object();

// Потік-1: тривала діагностика апарату
Thread diagnosticThread = new Thread(() =>
{
    lock (equipmentLock)
    {
        Console.WriteLine("[Діагностика] Починаю діагностику МРТ (довга операція)...");
        Thread.Sleep(400);
        Console.WriteLine("[Діагностика] Діагностику завершено");
    }
});

// Потік-2: технік, що намагається отримати доступ
Thread techThread = new Thread(() =>
{
    Thread.Sleep(50); // даємо потоку-1 захопити замок

    bool acquired = Monitor.TryEnter(equipmentLock, 100); // чекаємо не більше 100 мс
    if (acquired)
    {
        try
        {
            Console.WriteLine("[Технік] Доступ отримано — проводжу технічне обслуговування");
            Thread.Sleep(100);
        }
        finally
        {
            Monitor.Exit(equipmentLock);
        }
    }
    else
    {
        // Замок зайнятий — виконуємо альтернативну задачу
        Console.WriteLine("[Технік] МРТ зайнятий. Займусь обслуговуванням рентгену");
    }
});

diagnosticThread.Start();
techThread.Start();

diagnosticThread.Join();
techThread.Join();
```

TryEnter є важливим інструментом для уникнення дедлоків: якщо ресурс недоступний, потік не «зависає» у вічному очікуванні, а може спробувати іншу стратегію.

Monitor vs lock: коли що обирати

lock підходить для переважної більшості задач: захист критичної секції, яка виконується коротко і не потребує координації між потоками. Monitor обирають у таких ситуаціях:

— Потрібні Wait/Pulse/PulseAll для координації «виробник — споживач» або подібних патернів

— Потрібен TryEnter для спроби захоплення без блокування, зокрема з таймаутом

— Потрібен більш гранульований контроль над захопленням/звільненням замку

У всіх інших сценаріях — lock зрозуміліший, компактніший і менш схильний до помилок (наприклад, забутого Monitor.Exit()).

15.5. Клас AutoResetEvent

Monitor.Wait і Monitor.Pulse — потужні інструменти координації, але вони вимагають, щоб і виробник, і споживач використовували **один і той самий об'єкт замку**. У складніших сценаріях, де потоки координуються через незалежні сигнали, більш природним є клас AutoResetEvent.

AutoResetEvent — це примітив синхронізації, що моделює **автоматичне скидання сигнального прапорця**. Він має два стани:

— **Сигнальний** (signaled, «зелене світло»): потоки, що чекають через WaitOne(), можуть продовжити виконання

— **Несигнальний** (non-signaled, «червоне світло»): потоки блокуються на WaitOne()

Ключова особливість, яка відрізняє AutoResetEvent від ManualResetEvent: після того як **один** очікуючий потік пройшов через WaitOne(), прапорець **автоматично скидається** у несигнальний стан. Наступний потік знову буде заблокований — доки не надійде новий сигнал. Це нагадує турнікет: пропускає одного і автоматично закривається.

API класу AutoResetEvent

Метод / конструктор	Опис
new AutoResetEvent(bool initialState)	Створює об'єкт. true — початково сигнальний, false — несигнальний
Set()	Переводить у сигнальний стан. Один очікуючий потік пройде через WaitOne(), після чого стан знову стане несигнальним
Reset()	Примусово переводить у несигнальний стан
WaitOne()	Блокує поточний потік до отримання сигналу
WaitOne(int ms)	Блокує з таймаутом. Повертає true, якщо сигнал надійшов, false — якщо час вичерпано

Клінічний приклад: відправка результатів аналізу

Лаборант завершує аналіз і сигналізує лікарю, що результати готові. Лікар чекає сигналу, перш ніж відкрити результати:

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  // Початково несигнальний — лікар чекатиме
 5  AutoResetEvent resultsReady = new AutoResetEvent(false);
 6
 7  // Лікар (споживач): чекає на результати
 8  Thread doctor = new Thread(() =>
 9  {
10      Console.WriteLine("[Лікар] Чекаю на результати аналізів пацієнта Коваль
    М.В...");
11      resultsReady.WaitOne(); // блокуємось до сигналу
12      Console.WriteLine("[Лікар] Результати отримано! Призначаю лікування.");
13  });
14  doctor.Name = "Doctor";
15
16  // Лаборант (виробник): виконує аналіз і подає сигнал
17  Thread labTech = new Thread(() =>
18  {
19      Console.WriteLine("[Лаборант] Аналіз крові запущено...");
20      Thread.Sleep(400); // час аналізу
21      Console.WriteLine("[Лаборант] Аналіз завершено. Надсилаю результати.");
```

```
22      resultsReady.Set(); // AutoResetEvent: один потік пройде, потім —
    автоматичний скид
23  });
24  labTech.Name = "LabTech";
25
26  doctor.Start();
27  labTech.Start();
28
29  doctor.Join();
30  labTech.Join();
```

Після виклику resultsReady.Set() лікар отримує сигнал і прокидається. Прапорець автоматично скидається у несигнальний стан — якщо б з'явився ще один лікар, що чекає, він знову заблокувався б до наступного Set().

Координація кількох етапів: конвеєр обробки

AutoResetEvent особливо зручний для організації послідовних етапів конвеєра, де кожен наступний крок починається тільки після завершення попереднього. У клінічній системі: реєстрація → огляд → аналізи → призначення — кожен крок сигналізує готовність наступному:

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  AutoResetEvent registrationDone = new AutoResetEvent(false);
 5  AutoResetEvent examDone         = new AutoResetEvent(false);
 6  AutoResetEvent testsDone        = new AutoResetEvent(false);
 7
 8  string patientName = "Бойко О.П.";
 9
10  // Крок 1: реєстратура
11  Thread registration = new Thread(() =>
12  {
13      Console.WriteLine($"[Реєстратура] Реєструємо пацієнта {patientName}...");
14      Thread.Sleep(200);
15      Console.WriteLine($"[Реєстратура] {patientName} зареєстровано — направляємо
    до лікаря");
16      registrationDone.Set(); // дозволяємо лікарю починати
17  });
18
19  // Крок 2: огляд лікаря
20  Thread exam = new Thread(() =>
21  {
22      registrationDone.WaitOne(); // чекаємо завершення реєстрації
23      Console.WriteLine($"[Лікар] Починаю огляд: {patientName}");
24      Thread.Sleep(300);
25      Console.WriteLine($"[Лікар] Огляд завершено — призначаю аналізи");
26      examDone.Set(); // дозволяємо лабораторії
27  });
28
```

```
// Крок 3: лабораторія
Thread labWork = new Thread(() =>
{
    examDone.WaitOne(); // чекаємо завершення огляду
    Console.WriteLine($"[Лабораторія] Обробляємо зразки {patientName}...");
    Thread.Sleep(400);
    Console.WriteLine($"[Лабораторія] Аналізи готові");
    testsDone.Set(); // дозволяємо фінальний крок
});

// Крок 4: призначення лікування
Thread treatment = new Thread(() =>
{
    testsDone.WaitOne(); // чекаємо результатів аналізів
    Console.WriteLine($"[Лікар] Призначаю лікування для {patientName} на основі аналізів");
    Thread.Sleep(100);
    Console.WriteLine($"[Лікар] Призначення видано. Пацієнт може іти.");
});

Console.WriteLine($"=== Початок прийому пацієнта: {patientName} ===\n");

registration.Start();
exam.Start();
labWork.Start();
treatment.Start();

treatment.Join();

Console.WriteLine($"\n=== Прийом завершено ===");
```

Усі чотири потоки запущені одночасно, але виконуються суворо по черзі: кожен чекає сигналу від попереднього через відповідний AutoResetEvent. Це ефективніше за послідовний запуск потоків: поки реєстратура реєструє, лікар і лабораторія вже «готові» і не витрачають час на ініціалізацію.

WaitOne з таймаутом

Якщо потік не може чекати нескінченно — наприклад, після 5 секунд очікування слід надіслати сповіщення адміністратору — використовується WaitOne(ms):

```
C#
using System;
using System.Threading;

AutoResetEvent urgentSignal = new AutoResetEvent(false);

Thread responder = new Thread(() =>
{
    Console.WriteLine("[Черговий лікар] Очікую виклик (не більше 2 секунд)...");

```

```
    bool signalReceived = urgentSignal.WaitOne(2000); // максимум 2 секунди

    if (signalReceived)
    {
        Console.WriteLine("[Черговий лікар] Виклик отримано! Прямую до
пацієнта.");
    }
    else
    {
        Console.WriteLine("[Черговий лікар] Час очікування вичерпано.
Повідомляю адміністратора.");
    }
});
responder.Name = "OnCallDoctor";

// Спробуємо зі сигналом (закоментуйте наступні два рядки, щоб побачити
таймаут)
Thread caller = new Thread(() =>
{
    Thread.Sleep(800); // виклик через 0.8 секунди
    Console.WriteLine("[Диспетчер] Надсилаю виклик черговому лікарю!");
    urgentSignal.Set();
});

responder.Start();
caller.Start();

responder.Join();
caller.Join();
```

WaitOne(ms) дає потоку можливість «зупинитись і перевірити» — реалізація тайм-аутів без зайвого ускладнення коду.

AutoResetEvent vs ManualResetEvent

AutoResetEvent скидає прапорець автоматично після пробудження **одного** потоку. Якщо ж потрібно одночасно «відкрити ворота» для **всіх** очікуючих потоків і тримати їх відкритими до явного скидання, використовується ManualResetEvent:

Характеристика	**AutoResetEvent**	**ManualResetEvent**
Скидання після WaitOne	Автоматично	Вручну через Reset()
Пропускає через Set()	Один потік	Всі очікуючі
Аналогія	Турнікет	Відкриті ворота
Типовий сценарій	Передача одного завдання конкретному виконавцю	Старт усіх потоків одночасно

У медичному контексті: AutoResetEvent — це черговий реєстратор, що передає один квиток одному пацієнту; ManualResetEvent — це оголошення по радіо, що відкривається кабінет і туди одночасно йдуть усі пацієнти, яким призначено.

15.6. Клас Mutex

Mutex (від англ. *mutual exclusion* — взаємне виключення) — примітив синхронізації, що схожий на lock, але має ключову відмінність: Mutex може працювати **між процесами**, а не лише між потоками одного процесу. Це робить його незамінним інструментом у ситуаціях, коли потрібно координувати доступ до ресурсу між кількома запущеними програмами.

Крім міжпроцесного режиму, Mutex підтримує і внутрішньопроцесну синхронізацію. Однак він суттєво повільніший за lock: внутрішньопроцесний lock — це операція на рівні CLR без звернень до ядра ОС; Mutex — це об'єкт операційної системи, і кожне захоплення/звільнення вимагає системного виклику. Тому для синхронізації потоків усередині одного процесу зазвичай краще обирати lock. Mutex обирають тоді, коли міжпроцесна синхронізація є вимогою.

API класу Mutex

Метод / конструктор	Опис
new Mutex()	Локальний (внутрішньопроцесний) mutex, починає у вільному стані
new Mutex(bool initiallyOwned)	true — поточний потік одразу стає власником
new Mutex(bool initiallyOwned, string name)	Іменований (міжпроцесний) mutex. name — глобальна системна назва
WaitOne()	Захоплює mutex. Блокує, якщо зайнятий
WaitOne(int ms)	Захоплення з таймаутом. Повертає bool

Метод / конструктор	Опис
ReleaseMutex()	Звільняє mutex. Лише потік-власник може це зробити
Dispose()	Звільняє системний ресурс

Важлива особливість: mutex є **реентерабельним** (recursive). Якщо потік, що вже тримає mutex, знову викликає WaitOne(), він пройде без блокування. Але він зобов'язаний викликати ReleaseMutex() рівно стільки разів, скільки викликав WaitOne() — інакше mutex залишиться захопленим.

Базовий приклад: захист спільного ресурсу

Спочатку розглянемо Mutex у ролі звичайного внутрішньопроцесного замку — для розуміння API:

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  Mutex mutex = new Mutex();
 5  int _sharedLog = 0; // спільний лічильник записів у журналі
 6
 7  Thread[] nurses = new Thread[3];
 8  for (int i = 0; i < 3; i++)
 9  {
10      int nurseId = i + 1;
11      nurses[i] = new Thread(() =>
12      {
13          for (int j = 0; j < 5; j++)
14          {
15              mutex.WaitOne(); // захоплюємо mutex
16              try
17              {
18                  _sharedLog++;
19                  Console.WriteLine($"[Медсестра-{nurseId.ToString()}] Запис
    #{_sharedLog.ToString()} у журналі");
20                  Thread.Sleep(50);
21              }
22              finally
23              {
24                  mutex.ReleaseMutex(); // завжди звільняємо у finally
25              }
26
27              Thread.Sleep(30);
28          }
29      });
30      nurses[i].Name = $"Nurse-{nurseId}";
31  }
32
```

```
33  foreach (Thread t in nurses) t.Start();
34  foreach (Thread t in nurses) t.Join();
35
36  mutex.Dispose();
37  Console.WriteLine($"\nЗагалом записів у журналі: {_sharedLog.ToString()}");
```

Зверніть на паттерн try/finally: ReleaseMutex() розміщується у блоці finally, щоб гарантувати звільнення навіть при виникненні винятку. Якщо mutex не буде звільнений, всі інші потоки заблокуються назавжди.

Іменований Mutex — захист між процесами

Головна сила Mutex — іменований варіант, що є спільним об'єктом ОС. Два процеси, що створюють Mutex з однаковим іменем, отримують посилання на той самий системний об'єкт:

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  // Один примірник програми захоплює mutex, другий — чекатиме або отримає
    відмову
 5  const string MutexName = "Global\\ClinicalSystem_ReportGenerator";
 6
 7  bool createdNew;
 8  using Mutex mutex = new Mutex(true, MutexName, out createdNew);
 9
10  if (!createdNew)
11  {
12      Console.WriteLine("Генератор звітів вже запущено в іншому процесі.");
13      Console.WriteLine("Для безпеки даних дозволено лише один екземпляр.");
14      return;
15  }
16
17  try
18  {
19      // Лише один процес може дійти до цього місця
20      Console.WriteLine("Генератор звітів запущено. Генерую місячний звіт...");
21      Thread.Sleep(500);
22      Console.WriteLine("Місячний звіт сформовано і збережено.");
23  }
24  finally
25  {
26      mutex.ReleaseMutex(); // звільняємо при завершенні
27  }
```

Параметр out createdNew повертає true, якщо цей процес **створив** mutex (тобто є першим), або false, якщо mutex вже існував (інший процес вже запущений). Префікс Global\\ означає, що mutex видимий для всіх сесій ОС; Local\\ (або без префікса) — лише для поточної сесії. Іменовані mutex-и працюють і на Linux/macOS (.NET емулює їх через

файли в тимчасовому каталозі), тож цей патерн «лише один екземпляр програми» кросплатформений — на відміну від іменованих Semaphore і EventWaitHandle, які підтримуються лише у Windows.

Практичний сценарій: клінічна інформаційна система, де лише один запущений екземпляр може здійснювати запис до бази даних. Спроба запустити другий екземпляр завершиться повідомленням, а не пошкодженням даних.

Тайм-аут очікування

Mutex.WaitOne(ms) дозволяє не блокуватись вічно:

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  Mutex equipmentMutex = new Mutex();
 5
 6  // Перший лікар захоплює апарат УЗД надовго
 7  Thread doctor1 = new Thread(() =>
 8  {
 9      equipmentMutex.WaitOne();
10      try
11      {
12          Console.WriteLine("[Лікар-1] Використовую апарат УЗД (600 мс)...");
13          Thread.Sleep(600);
14          Console.WriteLine("[Лікар-1] Процедуру завершено, звільняю апарат");
15      }
16      finally
17      {
18          equipmentMutex.ReleaseMutex();
19      }
20  });
21
22  // Другий лікар — намагається отримати апарат, але не готовий чекати більше 200
    мс
23  Thread doctor2 = new Thread(() =>
24  {
25      Thread.Sleep(50); // дати першому захопити mutex
26      Console.WriteLine("[Лікар-2] Намагаюсь отримати апарат УЗД (чекаю до 200
    мс)...");
27
28      bool acquired = equipmentMutex.WaitOne(200);
29      if (acquired)
30      {
31          try
32          {
33              Console.WriteLine("[Лікар-2] Апарат отримано! Проводжу
    діагностику.");
34              Thread.Sleep(200);
35          }
36          finally
37          {
38              equipmentMutex.ReleaseMutex();
39          }
```

```
    }
    else
    {
        Console.WriteLine("[Лікар-2] Апарат зайнятий. Записую пацієнта на завтра.");
    }
});

doctor1.Start();
doctor2.Start();

doctor1.Join();
doctor2.Join();
equipmentMutex.Dispose();
```

AbandonedMutexException

Коли потік завершується, не звільнивши mutex, наступний потік, що чекає на нього у WaitOne(), отримає виняток AbandonedMutexException. Важлива деталь: разом із винятком цей потік **вже став власником** mutex — тому після обробки винятку він теж мусить викликати ReleaseMutex(). Це захисний механізм: наступний потік-власник отримає сигнал, що попередній «впав» без коректного завершення, і може прийняти рішення щодо стану спільного ресурсу.

C#
```
using System;
using System.Threading;

Mutex mutex = new Mutex();

// Потік, що завершується без звільнення mutex (наприклад, через помилку в логіці)
Thread crashingThread = new Thread(() =>
{
    mutex.WaitOne();
    Console.WriteLine("[Аварійний потік] Захопив mutex і завершився без звільнення!");
    // mutex.ReleaseMutex() не викликається — mutex залишиться «покинутим»
});

crashingThread.Start();
crashingThread.Join();

// Наступний потік спробує отримати mutex
Thread nextThread = new Thread(() =>
{
    try
    {
        mutex.WaitOne(); // отримаємо AbandonedMutexException
        Console.WriteLine("[Наступний потік] Отримав mutex (попередній власник впав)");
        // Тут треба перевірити стан ресурсу — він може бути пошкоджений
        mutex.ReleaseMutex();
    }
```

```
27      catch (AbandonedMutexException)
28      {
29          Console.WriteLine("[Наступний потік] Попередній власник mutex
    завершився аварійно!");
30          Console.WriteLine("[Наступний потік] Перевіряю цілісність даних...");
31          mutex.ReleaseMutex();
32      }
33  });
34
35  nextThread.Start();
36  nextThread.Join();
37  mutex.Dispose();
```

Зверніть увагу, як саме потік «падає» в цьому прикладі: він просто завершує метод, не звільнивши mutex. Імітувати падіння через throw без catch тут не можна: **необроблений виняток у будь-якому потоці .NET (не лише в головному) завершує весь процес** — до AbandonedMutexException справа просто не дійде. Тому тіло робочого потоку завжди варто обгортати в try/catch і журналювати помилки.

Mutex vs lock: підсумок

Характеристика	lock	Mutex
Область дії	Потоки одного процесу	Потоки та **різні процеси**
Продуктивність	Дуже висока (без ОС-викликів)	Нижча (об'єкт ядра ОС)
Реентерабельність	Так (повторний lock тим самим потоком проходить)	Так (рекурсивне захоплення)
Таймаут очікування	Лише через Monitor.TryEnter(obj, ms)	WaitOne(ms)
AbandonedMutexException	Ні	Так
Типовий сценарій	Захист даних усередині програми	Singleton-процес, спільний файл

Правило вибору: для синхронізації потоків усередині одного додатку — lock; якщо потрібна координація між кількома запущеними програмами або захист системного ресурсу — Mutex.

15.7. Клас Semaphore та SemaphoreSlim

Mutex і lock реалізують принцип «один доступ одночасно»: лише один потік може захопити ресурс. Але що, якщо ресурс допускає **обмежений паралелізм**? Наприклад, в лікарні є 3 кабінети УЗД — одночасно можуть прийматися 3 пацієнти, але не 4 і не 10. Або апаратура може одночасно обслуговувати не більше 5 запитів.

Для таких сценаріїв призначений **семафор** (semaphore). Семафор — це лічильник доступів: він ініціалізується максимальним числом одночасних дозволів і видає їх потокам на вимогу. Потік, що хоче отримати доступ, «забирає» один дозвіл; якщо всі дозволи вичерпані — потік блокується до тих пір, поки якийсь інший потік не «поверне» свій дозвіл. Коли Mutex — це двері з одним місцем, семафор — це стоянка з N місцями.

Semaphore і SemaphoreSlim

У .NET є два варіанти:

- **Semaphore** — об'єкт ядра ОС, як і Mutex. Може бути іменованим і доступним між процесами — але **лише у Windows**: на Linux і macOS конструктор з іменем кидає PlatformNotSupportedException (неіменований Semaphore працює всюди). Повільніший.
- **SemaphoreSlim** — легковаговий внутрішньопроцесний семафор. Набагато швидший, підтримує async/await. Є кращим вибором для переважної більшості задач.

Характеристика	Semaphore	SemaphoreSlim
Область дії	Процеси + міжпроцесна	Лише один процес
Продуктивність	Нижча (ОС-виклики)	Висока
async/await	Ні	Так (WaitAsync)
Іменований	Так	Ні
Типовий вибір	Міжпроцесне обмеження	Внутрішньопроцесний ліміт

API: SemaphoreSlim

Метод / конструктор	Опис
new SemaphoreSlim(initialCount)	initialCount доступних дозволів; максимум **не обмежено** (int.MaxValue)
new SemaphoreSlim(initialCount, maxCount)	Початкова кількість і максимальна; Release понад maxCount — SemaphoreFullException
Wait()	Забирає один дозвіл. Блокує, якщо лічильник = 0
Wait(int ms)	З таймаутом. Повертає bool
Release()	Повертає один дозвіл (лічильник++)
Release(int count)	Повертає count дозволів відразу
CurrentCount	Кількість доступних дозволів

Зверніть увагу на різницю двох конструкторів. new SemaphoreSlim(3) лише видає три дозволи на старті, але **не обмежує** їх кількість: зайвий Release() (наприклад, через помилку в логіці) мовчки збільшить лічильник до 4, і в «три кабінети» почнуть заходити четверо пацієнтів. new SemaphoreSlim(3, 3) у такій ситуації кине SemaphoreFullException і одразу покаже помилку — тому для реальних ресурсів краще задавати й максимум.

Клінічний приклад: кабінети УЗД

У медичному центрі є 3 апарати УЗД. Одночасно можуть проходити дослідження не більше трьох пацієнтів. Якщо всі три апарати зайняті — пацієнти чекають у черзі:

```
C#
 1  using System;
 2  using System.Threading;
 3
 4  // 3 кабінети УЗД — не більше 3 одночасних досліджень
 5  SemaphoreSlim usdRooms = new SemaphoreSlim(3, 3);
 6
 7  string[] patients =
 8  {
```

```
 9      "Коваль М.",  "Петренко І.", "Бойко О.",
10      "Мороз В.",   "Сидоренко Т.", "Руденко Н.",
11      "Кравченко Р.", "Гриценко Л."
12  };
13
14  Thread[] threads = new Thread[patients.Length];
15
16  for (int i = 0; i < patients.Length; i++)
17  {
18      string patient = patients[i];
19      threads[i] = new Thread(() =>
20      {
21          Console.WriteLine($"[{patient}] Чекаю на вільний кабінет (доступно:
    {usdRooms.CurrentCount.ToString()})");
22
23          usdRooms.Wait(); // займаємо один кабінет (або чекаємо)
24          try
25          {
26              Console.WriteLine($"[{patient}] Займаю кабінет. Зайнято: {(3 -
    usdRooms.CurrentCount).ToString()}/3");
27              Thread.Sleep(300); // УЗД-дослідження
28              Console.WriteLine($"[{patient}] УЗД завершено, звільняю кабінет");
29          }
30          finally
31          {
32              usdRooms.Release(); // повертаємо дозвіл (звільняємо кабінет)
33          }
34      });
35      threads[i].Name = $"Patient-{patient}";
36  }
37
38  foreach (Thread t in threads) t.Start();
39  foreach (Thread t in threads) t.Join();
40
41  Console.WriteLine($"\nВсіх пацієнтів обстежено. Вільних кабінетів:
    {usdRooms.CurrentCount.ToString()}/3");
```

Перші три пацієнти займуть кабінети і розпочнуть дослідження. Четвертий і наступні будуть чекати — Wait() заблокує їх до звільнення місця. Як тільки перший пацієнт викличе Release(), семафор підвищить лічильник і наступний очікуючий потік продовжить виконання.

Release() завжди розміщується у блоці finally — це гарантує звільнення дозволу навіть якщо у тілі потоку виник виняток.

Контроль темпу: обмеження паралельних запитів

Семафор — природний інструмент обмеження навантаження (rate limiting). Наприклад, система відправки SMS-нагадувань не може надсилати більше 5 повідомлень одночасно через обмеження SMS-шлюзу:

```
C#
 1  using System;
```

```
using System.Threading;

// SMS-шлюз дозволяє не більше 5 паралельних запитів
SemaphoreSlim smsGateway = new SemaphoreSlim(5, 5);

string[] recipients =
{
    "+380501234001", "+380501234002", "+380501234003",
    "+380501234004", "+380501234005", "+380501234006",
    "+380501234007", "+380501234008", "+380501234009",
    "+380501234010", "+380501234011", "+380501234012"
};

Thread[] senders = new Thread[recipients.Length];

for (int i = 0; i < recipients.Length; i++)
{
    string phone = recipients[i];
    int msgNum = i + 1;
    senders[i] = new Thread(() =>
    {
        smsGateway.Wait(); // чекаємо вільного слота шлюзу
        try
        {
            Console.WriteLine($"[SMS #{msgNum.ToString()}] Відправляю на
{phone}... (активних: {(5 - smsGateway.CurrentCount).ToString()}/5)");
            Thread.Sleep(200); // час відправки через шлюз
            Console.WriteLine($"[SMS #{msgNum.ToString()}] Доставлено →
{phone}");
        }
        finally
        {
            smsGateway.Release(); // звільняємо слот
        }
    });
}

Console.WriteLine($"Розсилка нагадувань: {recipients.Length.ToString()}
повідомлень через шлюз (ліміт 5)");
foreach (Thread t in senders) t.Start();
foreach (Thread t in senders) t.Join();
Console.WriteLine("Розсилку завершено");
```

Клас Semaphore — іменований, міжпроцесний

Semaphore аналогічний Mutex у контексті міжпроцесного доступу. Його іменована версія дозволяє різним програмам обмежувати спільне використання системного ресурсу:

```
C#
using System;
using System.Threading;

// Максимум 2 одночасних записи до журналу (між процесами)
const string SemName = "ClinicalAuditLog";

// У реальному сценарії: різні процеси або сервіси читають цей семафор.
```

```
// Іменований Semaphore підтримується лише у Windows — на інших ОС беремо локальний
using Semaphore sem = OperatingSystem.IsWindows()
    ? new Semaphore(2, 2, SemName)
    : new Semaphore(2, 2);

Thread[] writers = new Thread[5];
for (int i = 0; i < 5; i++)
{
    int id = i + 1;
    writers[i] = new Thread(() =>
    {
        Console.WriteLine($"[Writer-{id.ToString()}] Чекаю дозволу на запис...");
        sem.WaitOne();
        try
        {
            Console.WriteLine($"[Writer-{id.ToString()}] Пишу до журналу аудиту...");
            Thread.Sleep(200);
            Console.WriteLine($"[Writer-{id.ToString()}] Запис завершено");
        }
        finally
        {
            sem.Release();
        }
    });
}

foreach (Thread t in writers) t.Start();
foreach (Thread t in writers) t.Join();
```

Release з кількома дозволами

SemaphoreSlim.Release(int count) дозволяє повернути кілька дозволів за раз. Це корисно, коли один «виробник» генерує пакет завдань і хоче одразу дозволити кільком споживачам почати роботу:

```
C#
using System;
using System.Threading;

SemaphoreSlim batchSem = new SemaphoreSlim(0, 10); // 0 початково — ніхто не пройде

Thread[] workers = new Thread[5];
for (int i = 0; i < 5; i++)
{
    int id = i + 1;
    workers[i] = new Thread(() =>
    {
        Console.WriteLine($"[Лаборант-{id.ToString()}] Очікую партію зразків...");
        batchSem.Wait(); // чекаємо надходження зразків
        Console.WriteLine($"[Лаборант-{id.ToString()}] Отримав зразок, починаю аналіз");
        Thread.Sleep(150);
```

```
16          Console.WriteLine($"[Лаборант-{id.ToString()}] Аналіз завершено");
17      });
18      workers[i].Name = $"Lab-{id}";
19  }
20
21  foreach (Thread t in workers) t.Start();
22
23  Thread.Sleep(200); // готуємо партію
24
25  Console.WriteLine("[Прийом] Надійшла партія з 5 зразків — дозволяю всім
    лаборантам");
26  batchSem.Release(5); // одночасно розблокуємо 5 потоків
27
28  foreach (Thread t in workers) t.Join();
29  Console.WriteLine("Партію оброблено");
```

Ініціалізація SemaphoreSlim(0, 10) означає: початково 0 дозволів (всі потоки зупиняться на Wait), максимум 10. Після Release(5) лічильник стає 5, і всі 5 очікуючих потоків одночасно пробуджуються.

Загальна картина: коли який примітив обирати

Завершимо розділ зведеною таблицею всіх вивчених примітивів синхронізації:

Примітив	**Максимум потоків**	**Між процесами**	**Основний сценарій**
lock	1	Ні	Захист критичної секції (найчастіший вибір)
Lock (.NET 9)	1	Ні	Як lock, з кращою продуктивністю
Monitor	1	Ні	Wait/Pulse для координації виробника і споживача
AutoResetEvent	1 (по черзі)	Лише іменований EventWaitHandle у Windows	Сигналізація одному потоку про готовність даних

Примітив	Максимум потоків	Між процесами	Основний сценарій
Mutex	1	**Так**	Singleton-процес, захист системного файлу
Semaphore	N	**Так** (іменований — лише Windows)	Обмеження доступу до N ресурсів між процесами
SemaphoreSlim	N	Ні	Обмеження паралелізму всередині програми

Вибір примітива визначається трьома запитаннями:

4. **Скільки потоків** одночасно мають доступ до ресурсу — один або N?
5. **Потрібна міжпроцесна** синхронізація чи лише внутрішньопроцесна?
6. **Потрібна сигналізація** (один потік чекає на подію від іншого) чи лише взаємне виключення?

Для переважної більшості задач — lock і SemaphoreSlim покривають 90% потреб. Monitor — коли потрібна тонка координація через Wait/Pulse. AutoResetEvent — для явного сигналювання між потоками. Mutex і Semaphore — лише коли є реальна потреба у міжпроцесній синхронізації.

Питання для самоконтролю — Розділ 15. Багатопоточність

15.1. Введення у багатопоточність. Клас Thread

1. Що таке процес і що таке потік? Яке ключове ресурсне розмежування між ними? Чому кілька потоків одного процесу можуть ефективно обмінюватись даними, але це водночас породжує небезпеки?

2. Поясніть, чому властивість Thread.Name можна присвоїти лише один раз. Яке виключення кинеться при повторному присвоєнні, і чому це обмеження зроблено навмисно?
3. Що виведе наступний код — і чому результат може відрізнятися при кожному запуску?

```
C#
1  Thread t = new Thread(() => Console.WriteLine("Worker"));
2  t.Start();
3  Console.WriteLine("Main");
```

4. Порівняйте передній (foreground) і фоновий (background) потоки. Напишіть приклад, де неправильний вибір типу потоку може призвести до незбереження важливих даних при завершенні програми.
5. Метод Thread.Sleep(0) і Thread.Sleep(100) — у чому принципова різниця між ними? Коли доцільно використовувати Sleep(0)?
6. Який стан набуває потік після виклику Join() на ньому іншим потоком? Що станеться, якщо викликати Join() на потоці, що вже завершився?
7. Є код:

```
C#
1  Thread t = new Thread(() => { Thread.Sleep(5000); Console.WriteLine("Done"); });
2  t.IsBackground = true;
3  t.Start();
4  Console.WriteLine("Main ends");
```

Чи виведеться рядок "Done"? Обґрунтуйте відповідь.

8. Спроєктуйте систему первинного прийому пацієнта, де паралельно виконуються: вимірювання показників (передній потік), завантаження медичної картки (передній потік) і архівація старих записів (фоновий потік). Поясніть, чому архівація є фоновим потоком, а не переднім.

15.2. Створення потоків. ThreadStart та ParameterizedThreadStart

9. Що таке делегат ThreadStart? Як він відрізняється від ParameterizedThreadStart? Запишіть їх сигнатури.
10. Знайдіть і виправте помилку в наступному коді:

```
C#
1  for (int i = 0; i < 3; i++)
```

```
2  {
3      new Thread(() => Console.WriteLine(i)).Start();
4  }
```

Що виведеться без виправлення і чому? Як правильно?

11. Чому у ParameterizedThreadStart параметр має тип object? а не конкретний тип? Які два способи безпечно привести цей параметр до конкретного типу, і який з них кращий? Поясніть різницю між (T)param і param as T.
12. Порівняйте три способи передачі коду у потік: іменований метод, анонімний метод і лямбда-вираз. Для якого сценарію ви б обрали кожен? Наведіть конкретний приклад.
13. Напишіть код, що запускає 4 потоки паралельно — по одному на кожне лабораторне замовлення з масиву LabOrder[] — і чекає завершення всіх перед тим, як вивести «Всі аналізи готові».
14. Що означає рядок string patient = patients[i] перед захопленням у лямбду в циклі? Що відбудеться, якщо замість копії захопити patients[i] напряму через і?
15. Коли доцільно створювати клас-обгортку для передачі даних у потік через ParameterizedThreadStart? Наведіть приклад такого класу для клінічного сценарію.

15.3. Синхронізація потоків. Оператор lock та клас Lock

16. Що таке стан перегонів (race condition)? Чому операція count++ є небезпечною при паралельному доступі, якщо виглядає як одна операція?
17. Що виведе наступний код після завершення всіх потоків — і чи буде відповідь стабільною?

```C#
1  int n = 0;
2  Thread[] threads = new Thread[3];
3  for (int i = 0; i < 3; i++)
4      threads[i] = new Thread(() => { for (int j = 0; j < 1000; j++) n++; });
5  foreach (var t in threads) t.Start();
6  foreach (var t in threads) t.Join();
7  Console.WriteLine(n);
```

18. Чому не можна використовувати рядковий літерал або this як об'єкт-замок у lock? Яка стандартна практика для оголошення об'єкта-замка?

19. Поясніть, що відбувається «під капотом», коли компілятор обробляє конструкцію lock (obj) { ... }. Напишіть еквівалентний код через Monitor.Enter / Monitor.Exit.
20. Що таке взаємне блокування (deadlock)? Напишіть мінімальний приклад двох потоків, що гарантовано утворять дедлок, і поясніть, як його уникнути.
21. Порівняйте volatile, Interlocked і lock: що гарантує кожен засіб і яка його вартість? Для яких конкретних сценаріїв підходить кожен?
22. Клас Lock (.NET 9) надає метод EnterScope(). Чому патерн using (_lock.EnterScope()) надійніший за lock (_lock) { ... } з точки зору обробки виключень? Або вони еквівалентні — обґрунтуйте.
23. Напишіть потокобезпечний клас BedManager, що управляє кількістю вільних ліжок у лікарні (операції Admit() і Discharge()), використовуючи lock для захисту спільного стану.

15.4. Клас Monitor

24. У чому принципова різниця між Monitor і оператором lock? В яких трьох ситуаціях варто обирати Monitor замість lock?
25. Поясніть три дії, що виконує Monitor.Wait(obj) атомарно. Чому умова перевіряється у циклі while, а не в if? Що таке «хибне пробудження» (spurious wakeup)?
26. Чим відрізняється Monitor.Pulse(obj) від Monitor.PulseAll(obj)? В якому сценарії потрібен PulseAll? Наведіть клінічний приклад.
27. Де в коді можна викликати Monitor.Wait(), Monitor.Pulse() і Monitor.PulseAll()? Яке виключення виникне при порушенні цієї вимоги?
28. Знайдіть проблему в наступному коді:

```C#
object _lock = new object();
Queue<string> _queue = new Queue<string>();

void Consumer()
{
    lock (_lock)
    {
        if (_queue.Count == 0)
            Monitor.Wait(_lock);
        Console.WriteLine(_queue.Dequeue());
    }
```

```
12  }
```

Чому if тут небезпечний? Як виправити?

29. Реалізуйте патерн «виробник — споживач» для лабораторної черги: медсестра додає зразки, лаборант забирає. Використовуйте Monitor.Wait і Monitor.Pulse. Забезпечте коректне завершення при порожній закритій черзі.
30. Monitor.TryEnter(obj, 200) повернув false. Що це означає? Яку альтернативну логіку може виконати потік замість нескінченного очікування?

15.5. Клас AutoResetEvent

31. Що таке AutoResetEvent? Яка ключова особливість відрізняє його від ManualResetEvent? Поясніть аналогію з турнікетом.
32. Що відбудеться, якщо викликати Set() на AutoResetEvent, коли жоден потік не чекає на WaitOne()? Чи «накопичується» сигнал?
33. Порівняйте AutoResetEvent з Monitor.Wait/Monitor.Pulse. Яка ключова відмінність у вимогах щодо спільного об'єкта-замка?
34. Напишіть сценарій конвеєра із трьох кроків (реєстрація → огляд → аналізи), де кожен крок запускається в окремому потоці і чекає сигналу від попереднього через AutoResetEvent. Поясніть, чому всі потоки запускаються одночасно, але виконуються послідовно.
35. Що повертає WaitOne(int milliseconds) при успішному і при тайм-аутовому завершенні? Напишіть приклад використання цього варіанту для обмеження часу очікування відповіді.
36. Поясніть різницю між new AutoResetEvent(false) і new AutoResetEvent(true). В якому сценарії корисна початково сигнальна версія?
37. Клінічна система чекає підтвердження від лабораторії. Якщо підтвердження не надійде протягом 3 секунд — надіслати нагадування адміністратору. Реалізуйте цю логіку через WaitOne з тайм-аутом.

15.6. Клас Mutex

38. Яка ключова властивість Mutex відрізняє його від lock? У яких двох конкретних сценаріях слід обирати Mutex?

39. Що означає «реентерабельність» (recursive) Mutex? Що станеться, якщо потік, що тримає Mutex, викличе WaitOne() ще раз? І яке зобов'язання це накладає?
40. Поясніть призначення параметра out bool createdNew у конструкторі іменованого Mutex. Як за допомогою цього параметра реалізувати «singleton-процес» (тільки один запущений екземпляр)?
41. Що означає префікс Global\\ в імені іменованого Mutex? Чим він відрізняється від Local\\?
42. Знайдіть потенційну проблему в наступному коді:

```
C#
1  Mutex m = new Mutex();
2  m.WaitOne();
3  DoWork(); // може кинути виняток
4  m.ReleaseMutex();
```

Як виправити, щоб Mutex завжди звільнявся?

43. Що таке AbandonedMutexException? Коли він виникає і яке його практичне значення для відновлення після збою попереднього власника ресурсу?
44. Порівняйте lock і Mutex за продуктивністю, областю дії та можливістю тайм-ауту. Склавши зведену таблицю, обґрунтуйте вибір для кожного з двох сценаріїв: (а) захист спільного лічильника в одному процесі; (б) заборона запуску другого екземпляра генератора звітів.

15.7. Клас Semaphore та SemaphoreSlim

45. Що таке семафор? Поясніть аналогію зі стоянкою з N місцями. Чим він відрізняється від Mutex/lock?
46. Порівняйте Semaphore і SemaphoreSlim: за продуктивністю, підтримкою async/await, можливістю іменування. Коли слід обирати кожен?
47. Що означає new SemaphoreSlim(0, 10)? Що відбудеться з потоками, що викличуть Wait(), доки не буде викликано Release()?
48. Напишіть код для системи, де 8 пацієнтів паралельно намагаються потрапити до 3 кабінетів УЗД. Забезпечте, щоб у будь-який момент в кабінетах перебувало не більше 3 пацієнтів. Розмістіть Release() у правильному місці.

49. SemaphoreSlim.Release(5) — що означає передача числового аргументу? У якому сценарії (виробник пакетно публікує завдання) це корисно?
50. Знайдіть проблему в коді:

```C#
SemaphoreSlim sem = new SemaphoreSlim(3, 3);
sem.Wait();
ProcessData(); // може кинути виняток
sem.Release();
```

Як виправити?

51. Спроєктуйте систему обмеження навантаження (rate limiting) для SMS-розсилки: шлюз дозволяє максимум 5 паралельних запитів. Реалізуйте через SemaphoreSlim, пояснивши логіку ініціалізації і де розміщується Wait()/Release().
52. Склавши зведену таблицю всіх примітивів синхронізації розділу (lock, Monitor, AutoResetEvent, Mutex, Semaphore, SemaphoreSlim), відповідайте на три питання для кожного: скільки потоків пропускає одночасно, чи підтримує міжпроцесну синхронізацію, який основний сценарій використання.

Розділ 16. Паралельне програмування та TPL

16.1. Клас Task. Основи TPL

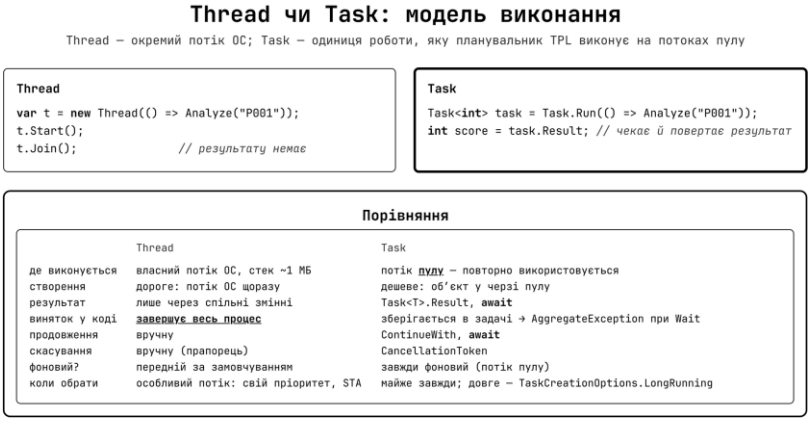
У попередньому розділі ми розглядали багатопоточність через клас Thread — прямий, але низькорівневий інструмент. Ми вручну створювали потоки, передавали делегати, викликали Join(), піклувались про пріоритети і тип потоку (передній чи фоновий). Для простих сценаріїв це цілком прийнятно, але у реальних системах з десятками паралельних операцій такий підхід стає громіздким.

TPL (Task Parallel Library) — бібліотека паралельного програмування, що входить до складу .NET, починаючи з версії 4.0. Її мета — підняти рівень абстракції: замість того щоб думати про потоки, потокові пули і синхронізацію вручну, розробник описує **завдання** (tasks) — логічні одиниці роботи — і довіряє TPL ефективно їх виконати. TPL розміщена у просторі імен System.Threading.Tasks.

Thread vs Task: у чому різниця

Ключова відмінність між Thread і Task — у тому, як вони використовують ресурси операційної системи. Кожен Thread — це окремий потік ОС із власним стеком пам'яті (~1 МБ). Створення і знищення потоку — дорога операція. Якщо програма постійно створює і знищує сотні потоків, продуктивність падає.

Task натомість використовує **пул потоків** (ThreadPool) — набір вже створених потоків, які очікують нової роботи. Коли завдання завершується, потік не знищується, а повертається у пул і чекає наступного завдання. Це суттєво знижує накладні витрати на створення і знищення потоків. Крім того, Task підтримує повернення результатів, ланцюги продовжень і скасування — все те, що з Thread довелось би реалізовувати вручну.



Thread vs Task: модель виконання

Три способи створення завдання

Спосіб 1: конструктор + Start()

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  Task task = new Task(() =>
 6  {
 7      Console.WriteLine($"[Task {Task.CurrentId}] Реєстрація пацієнта
    розпочата");
 8      Thread.Sleep(200);
 9      Console.WriteLine($"[Task {Task.CurrentId}] Реєстрацію завершено");
10  });
11
12  task.Start(); // явний запуск
13  task.Wait();  // чекаємо завершення
14
15  Console.WriteLine("Головний потік: реєстрацію підтверджено");
```

Конструктор + Start() — найбільш явний спосіб. Він дає контроль над моментом запуску: завдання можна створити заздалегідь і запустити пізніше. Task.CurrentId повертає числовий ідентифікатор поточного виконуваного завдання.

Спосіб 2: Task.Factory.StartNew()

```
C#
```

```
using System;
using System.Threading;
using System.Threading.Tasks;

Task task = Task.Factory.StartNew(() =>
{
    Console.WriteLine($"[Task {Task.CurrentId}] Лабораторний аналіз запущено
через Factory");
    Thread.Sleep(150);
    Console.WriteLine($"[Task {Task.CurrentId}] Аналіз завершено");
});

task.Wait();
Console.WriteLine("Результат аналізу отримано");
```

Task.Factory.StartNew() — старіший API, що залишається актуальним для складних сценаріїв: він приймає додаткові параметри TaskCreationOptions і TaskScheduler, недоступні в Task.Run(). Для звичайних завдань зручніший Task.Run().

Спосіб 3: Task.Run() — сучасний рекомендований підхід

```
C#
using System;
using System.Threading;
using System.Threading.Tasks;

Task task = Task.Run(() =>
{
    Console.WriteLine($"[Task {Task.CurrentId}] МРТ-сканування через
Task.Run");
    Thread.Sleep(300);
    Console.WriteLine($"[Task {Task.CurrentId}] МРТ завершено");
});

task.Wait();
Console.WriteLine("Знімки МРТ збережено в системі");
```

Task.Run() — найпростіший і найпоширеніший спосіб у сучасному коді. Він автоматично запускає завдання у пулі потоків без додаткових налаштувань. Це рекомендований підхід для переважної більшості сценаріїв.

Властивості класу Task

Кожен об'єкт Task має набір властивостей, що дозволяють отримати інформацію про стан і результат виконання:

Властивість	Тип	Опис
Id	int	Унікальний ідентифікатор завдання (статично для зовнішнього коду)
Status	TaskStatus	Поточний стан: Created, Running, RanToCompletion тощо
IsCompleted	bool	true якщо завдання завершилось (успішно, з помилкою або скасоване)
IsCompletedSuccessfully	bool	true тільки при успішному завершенні
IsCanceled	bool	true якщо завдання скасовано через CancellationToken
IsFaulted	bool	true якщо завдання завершилось із необробленим винятком
Exception	AggregateException?	Виняток (або null) — обгортка навколо всіх внутрішніх винятків

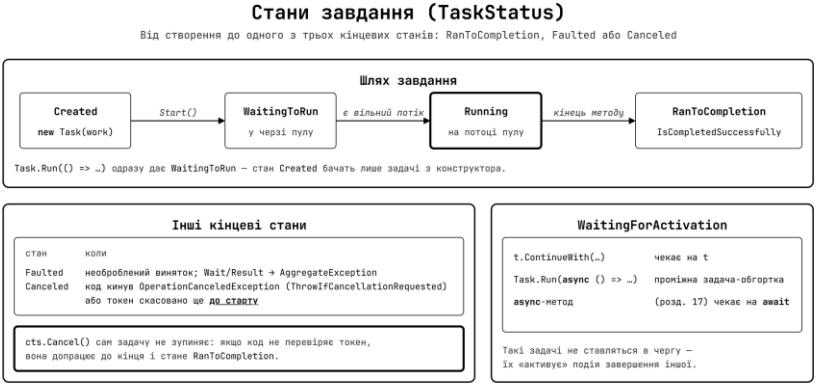
Статична властивість Task.CurrentId доступна лише всередині виконуваного завдання — повертає id поточного Task або null поза завданням.

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  Task task = Task.Run(() =>
 6  {
 7      Console.WriteLine($"Id: {Task.CurrentId.ToString()}");
 8      Console.WriteLine($"Status всередині: {Task.CompletedTask.Status}");
 9      Thread.Sleep(100);
10  });
11
12  Console.WriteLine($"Status після Run(): {task.Status}");
13  task.Wait();
```

```
14  Console.WriteLine($"Status після Wait(): {task.Status}");
15  Console.WriteLine($"IsCompleted: {task.IsCompleted.ToString()}");
16  Console.WriteLine($"IsCompletedSuccessfully:
    {task.IsCompletedSuccessfully.ToString()}");
17  Console.WriteLine($"IsFaulted: {task.IsFaulted.ToString()}");
```

Стани завдання (TaskStatus)

Завдання проходить через кілька станів протягом свого часу існування. Їх описує перерахування TaskStatus:



Стани завдання (TaskStatus)

— **Created** — завдання створено через new Task(...), але ще не запущено. Стан перед Start().

— **WaitingForActivation** — завдання не стоїть у черзі пулу, а чекає, поки його «активує» інша подія. У цьому стані перебувають продовження ContinueWith() (до завершення попереднього завдання), задачі-обгортки Task.Run(async () => ...) та задачі async-методів (розділ 17).

— **WaitingToRun** — завдання заплановано до виконання у пулі потоків, але вільний потік ще не отримано. Саме в цьому стані одразу опиняється задача з Task.Run(() => ...) (для звичайної, не async, лямбди) і задача після Start().

— **WaitingForChildrenToComplete** — код завдання вже виконано, але воно чекає на свої дочірні завдання, створені з TaskCreationOptions.AttachedToParent (розділ 16.2).

— **Running** — завдання активно виконується у потоці пулу.

— **RanToCompletion** — завдання успішно завершило виконання без винятків і скасування.

— **Canceled** — завдання завершилось через скасування за допомогою CancellationToken: або його код кинув OperationCanceledException з тим самим токеном, що був переданий у Task.Run, або токен скасували ще до старту. Сам виклик cts.Cancel() задачу не зупиняє: якщо її код токен не перевіряє, вона допрацює до кінця і стане RanToCompletion (детально — розділ 16.5).

— **Faulted** — завдання завершилось із необробленим винятком.

Очікування завершення завдань

Wait() — очікування одного завдання

task.Wait() блокує поточний (головний) потік до завершення завдання. Якщо завдання завершилось з винятком, Wait() викидає AggregateException:

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  Task ecg = Task.Run(() =>
 6  {
 7      Console.WriteLine("[ЕКГ] Запис кардіограми...");
 8      Thread.Sleep(300);
 9      Console.WriteLine("[ЕКГ] Запис завершено");
10  });
11
12  Console.WriteLine("[Main] ЕКГ запущено. Чекаю результату...");
13  ecg.Wait(); // головний потік заблокований тут
14  Console.WriteLine("[Main] Кардіограма готова — аналізую");
```

WaitAll() — очікування кількох завдань

Task.WaitAll(tasks) блокує поточний потік до завершення **всіх** переданих завдань:

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  Task blood  = Task.Run(() => { Thread.Sleep(400); Console.WriteLine("Аналіз
    крові готовий"); });
 6  Task urine  = Task.Run(() => { Thread.Sleep(250); Console.WriteLine("Аналіз
    сечі готовий"); });
```

```
 7  Task xray   = Task.Run(() => { Thread.Sleep(350); Console.WriteLine("Рентген
    готовий"); });
 8
 9  Console.WriteLine("Всі аналізи запущені паралельно...");
10  Task.WaitAll(blood, urine, xray); // чекаємо всіх трьох
11  Console.WriteLine("Всі результати отримано — формую загальний висновок");
```

WaitAny() — очікування першого завершеного

Task.WaitAny(tasks) повертає індекс першого завдання, що завершилось, і розблоковує поточний потік. Решта завдань продовжують виконуватись:

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  Task[] tests = new Task[]
 6  {
 7      Task.Run(() => { Thread.Sleep(500); Console.WriteLine("Аналіз крові готовий
    (500мс)"); }),
 8      Task.Run(() => { Thread.Sleep(200); Console.WriteLine("Аналіз сечі готовий
    (200мс)"); }),
 9      Task.Run(() => { Thread.Sleep(350); Console.WriteLine("ЕКГ готово
    (350мс)"); }),
10  };
11
12  Console.WriteLine("Чекаю на перший готовий результат...");
13  int firstIdx = Task.WaitAny(tests); // розблокується після першого
14  Console.WriteLine($"Перший результат отримано: завдання
    #{firstIdx.ToString()}");
15  Console.WriteLine("Продовжую роботу, решта завдань ще виконуються...");
16
17  Task.WaitAll(tests); // дочекаємось і решти
18  Console.WriteLine("Всі завдання завершено");
```

WaitAll підходить для сценарію «потрібні всі результати перед наступним кроком», WaitAny — для «почни обробку як тільки з'явиться перший результат» або реалізації тайм-аутів.

Синхронний запуск: RunSynchronously()

Метод RunSynchronously() виконує завдання **синхронно в поточному потоці**, без передачі до пулу потоків. Використовується рідко — наприклад, для тестування або коли потрібно явно контролювати, у якому потоці виконується завдання:

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
```

```
4
5  Task task = new Task(() =>
6  {
7      Console.WriteLine($"Завдання виконується у потоці:
   {Thread.CurrentThread.ManagedThreadId.ToString()}");
8      Console.WriteLine($"Це пул потоків?
   {Thread.CurrentThread.IsThreadPoolThread.ToString()}");
9  });
10
11 Console.WriteLine($"Головний потік:
   {Thread.CurrentThread.ManagedThreadId.ToString()}");
12 task.RunSynchronously(); // виконується тут, у поточному потоці
13 Console.WriteLine("Завдання завершено синхронно");
```

Обробка винятків у завданнях

Коли в завданні виникає виняток, він не зупиняє програму автоматично. Натомість він зберігається у властивості Task.Exception і повторно кидається при виклику Wait() чи зверненні до Result — обгорнутий у AggregateException. Читання властивості Task.Exception виняток **не кидає**: воно просто повертає цей AggregateException (або null, якщо помилки не було). А оператор await (розділ 17) кидає не обгортку, а перший внутрішній виняток:

```
C#
1  using System;
2  using System.Threading.Tasks;
3
4  Task task = Task.Run(() =>
5  {
6      Console.WriteLine("[Task] Перевіряю дані пацієнта...");
7      throw new InvalidOperationException("Медична картка не знайдена в
   системі");
8  });
9
10 try
11 {
12     task.Wait(); // тут виняток перекидається
13 }
14 catch (AggregateException ae)
15 {
16     foreach (var inner in ae.InnerExceptions)
17         Console.WriteLine($"Помилка: {inner.Message}");
18 }
19
20 Console.WriteLine($"IsFaulted: {task.IsFaulted.ToString()}");
21 Console.WriteLine($"Status: {task.Status}");
```

AggregateException є оберткою, бо одне завдання теоретично може містити кілька вкладених завдань, кожне з яких може кинути свій виняток. InnerExceptions — колекція всіх внутрішніх винятків. У простих сценаріях вона містить один елемент.

16.2. Вкладені завдання та масиви завдань. Task<T>

Розглянемо можливості TPL, що дозволяють будувати складніші структури завдань: вкладені завдання, масиви завдань з колективним очікуванням і типізовані завдання, що повертають значення.

Вкладені завдання

Завдання може запускати інше завдання зсередини свого тіла — таке завдання називається **вкладеним** (nested task). За замовчуванням вкладене завдання є **незалежним** від батьківського: батьківське завдання може завершитись раніше, ніж вкладене. Це може здивувати — інтуїтивно здається, що завдання повинно чекати на всі запущені ним задачі. Але TPL так не поводиться без явної вказівки.

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  Task outer = Task.Factory.StartNew(() =>
 6  {
 7      Console.WriteLine($"[Outer {Task.CurrentId}] Початок: відкриваю прийом
    пацієнта");
 8
 9      // вкладене завдання — незалежне за замовчуванням
10      Task inner = Task.Factory.StartNew(() =>
11      {
12          Console.WriteLine($"[Inner {Task.CurrentId}] Запит до лабораторії
    відправлено");
13          Thread.Sleep(500); // лабораторія довго обробляє
14          Console.WriteLine($"[Inner {Task.CurrentId}] Лабораторія відповіла");
15      });
16
17      Console.WriteLine($"[Outer {Task.CurrentId}] Огляд лікаря завершено (не
    чекає лабораторії)");
18      // Outer завершується — Inner ще виконується!
19  });
20
21  outer.Wait(); // чекаємо тільки outer
22  Console.WriteLine("[Main] Outer завершено. Inner може ще працювати!");
23  Thread.Sleep(600); // дамо inner доробити для демонстрації
24  Console.WriteLine("[Main] Програма завершується");
```

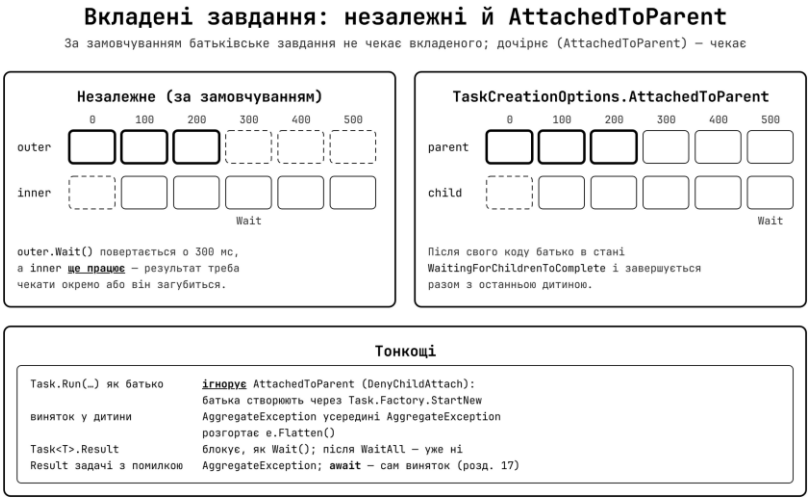
Поведінка «не чекати вкладеного» є навмисним рішенням дизайну: в більшості сценаріїв завдання запускають допоміжні підзавдання, за якими не потрібно стежити. Але якщо потрібно прив'язати вкладене завдання до батьківського, використовується TaskCreationOptions.AttachedToParent.

AttachedToParent: дочірнє завдання

TaskCreationOptions.AttachedToParent перетворює вкладене завдання на **дочірнє** (child task). Батьківське завдання не може завершитись, доки не завершаться всі його дочірні завдання: виконавши власний код, воно переходить у стан WaitingForChildrenToComplete. Якщо дочірнє завдання кине виняток, він буде перекинутий через батьківське при виклику Wait() — причому як AggregateException усередині AggregateException; розгорнути вкладені обгортки допомагає метод e.Flatten().

Важливе обмеження: батьківське завдання має бути створене через Task.Factory.StartNew або new Task(...). Task.Run навмисно створює задачі з опцією DenyChildAttach, тому прив'язати до них дочірнє завдання неможливо — AttachedToParent всередині Task.Run(...) просто ігнорується, і вкладене завдання поводиться як незалежне. Саме тому приклад нижче використовує Task.Factory.StartNew.

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  Task parent = Task.Factory.StartNew(() =>
 6  {
 7      Console.WriteLine($"[Parent {Task.CurrentId}] Відкриваю комплексне
    обстеження");
 8
 9      // дочірнє завдання — прив'язане до батька
10      Task child1 = Task.Factory.StartNew(() =>
11      {
12          Console.WriteLine($"[Child {Task.CurrentId}] Аналіз крові...");
13          Thread.Sleep(400);
14          Console.WriteLine($"[Child {Task.CurrentId}] Аналіз крові готовий");
15      }, TaskCreationOptions.AttachedToParent);
16
17      Task child2 = Task.Factory.StartNew(() =>
18      {
19          Console.WriteLine($"[Child {Task.CurrentId}] ЕКГ-запис...");
20          Thread.Sleep(300);
21          Console.WriteLine($"[Child {Task.CurrentId}] ЕКГ готово");
22      }, TaskCreationOptions.AttachedToParent);
23
24      Console.WriteLine($"[Parent {Task.CurrentId}] Дочірні завдання запущені,
    чекаю їх завершення...");
25      // Parent автоматично чекає child1 і child2
26  });
27
28  parent.Wait(); // чекає і parent, і всі його дочірні завдання
29  Console.WriteLine("[Main] Комплексне обстеження завершено — всі аналізи
    готові");
```



Вкладені завдання: незалежні vs AttachedToParent

Практичне правило: якщо вкладені завдання є логічною частиною батьківської операції і їх результат потрібен для підтвердження завершення батька — використовуйте AttachedToParent. Якщо вкладені завдання є допоміжними, їх стан не має значення для батька — залишайте їх незалежними.

Масиви завдань

Коли потрібно запустити кілька завдань і відстежити їх стан, зручно зібрати їх у масив або список. Це дозволяє застосовувати Task.WaitAll і Task.WaitAny до довільної кількості завдань без перерахування кожного вручну.

```C#
using System;
using System.Threading;
using System.Threading.Tasks;

string[] examinations = { "Загальний аналіз крові", "Аналіз сечі", "ЕКГ", "УЗД черевної порожнини" };

// Спосіб 1: створення масиву через ініціалізатор
Task[] tasks = new Task[examinations.Length];
for (int i = 0; i < examinations.Length; i++)
{
    string exam = examinations[i]; // копія для замикання
```

```
12      tasks[i] = Task.Run(() =>
13      {
14          int duration = 100 + exam.Length * 10; // різний час для різних
    обстежень
15          Console.WriteLine($"[Task {Task.CurrentId}] Початок: {exam}");
16          Thread.Sleep(duration);
17          Console.WriteLine($"[Task {Task.CurrentId}] Готово: {exam}");
18      });
19  }
20
21  Console.WriteLine("Всі обстеження запущені паралельно...");
22  Task.WaitAll(tasks); // чекаємо завершення всіх
23  Console.WriteLine("Всі обстеження завершено — готую зведений звіт лікарю");
```

WaitAll vs WaitAny: коли що обирати

Task.WaitAll — для сценарію «потрібні всі результати, щоб рухатись далі». Наприклад, перш ніж видати загальний висновок, лікар чекає на всі аналізи.

Task.WaitAny — для сценарію «реагуй на перший готовий результат». Наприклад, система моніторингу обробляє показники від першого датчика, що надіслав дані, не чекаючи решти:

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  Task[] monitors = new Task[]
 6  {
 7      Task.Run(() => { Thread.Sleep(400); Console.WriteLine("Датчик пульсу: 88
    уд/хв"); }),
 8      Task.Run(() => { Thread.Sleep(150); Console.WriteLine("Датчик SpO2: 97%");
    }),
 9      Task.Run(() => { Thread.Sleep(280); Console.WriteLine("Датчик тиску:
    122/78"); }),
10  };
11
12  Console.WriteLine("Система моніторингу: очікую перший сигнал...");
13  int first = Task.WaitAny(monitors);
14  Console.WriteLine($"Перший сигнал отримано від датчика #{first.ToString()} —
    фіксую в карті");
15  Task.WaitAll(monitors); // дочекаємо решту
16  Console.WriteLine("Всі показники зафіксовано");
```

Task<T>: завдання з результатом

До цього моменту ми розглядали завдання, що нічого не повертають (Task — аналог void). Але часто завдання обчислює щось і результат потрібен у головному потоці. Для цього існує типізована версія — Task<T>, де T — тип значення, що повертається.

Результат отримується через властивість Result. Звернення до Result **блокує поточний потік** до завершення завдання — аналогічно Wait(). Тому Result є одночасно і очікуванням, і отриманням результату.

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  // Завдання, що повертає числовий результат
 6  Task<double> bmiTask = Task.Run(() =>
 7  {
 8      double weight = 72.5; // кг
 9      double height = 1.78; // м
10      Thread.Sleep(100); // імітуємо розрахунок
11      return weight / (height * height); // формула ІМТ
12  });
13
14  Console.WriteLine("ІМТ розраховується...");
15  double bmi = bmiTask.Result; // блокується тут до отримання результату
16  Console.WriteLine($"ІМТ пацієнта: {bmi.ToString("F1")} кг/м²");
17  Console.WriteLine($"Категорія: {(bmi < 18.5 ? "Недостатня вага" : bmi < 25 ?
    "Норма" : bmi < 30 ? "Зайва вага" : "Ожиріння")}");
```

Task<T> з класом результату

У реальних клінічних задачах результат — це, як правило, не просте число, а об'єкт із кількома полями:

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  Task<LabResult> labTask = Task.Run(() =>
 6  {
 7      Console.WriteLine("[Лабораторія] Виконую аналіз крові Коваль М.В...");
 8      Thread.Sleep(400);
 9      return new LabResult("Коваль М.В.", glucose: 5.8, hemoglobin: 138.0);
10  });
11
12  LabResult result = labTask.Result; // блокуємось до завершення
13
14  Console.WriteLine($"\n=== Результат аналізу ===");
15  Console.WriteLine($"Пацієнт:    {result.PatientName}");
16  Console.WriteLine($"Глюкоза:    {result.Glucose.ToString("F1")} ммоль/л");
17  Console.WriteLine($"Гемоглобін: {result.Hemoglobin.ToString("F0")} г/л");
18  Console.WriteLine($"Висновок:   {result.Status}");
19
20  class LabResult
21  {
22      public string PatientName { get; }
23      public double Glucose     { get; }
24      public double Hemoglobin  { get; }
25      public string Status      { get; }
26
```

```
27      public LabResult(string name, double glucose, double hemoglobin)
28      {
29          PatientName = name;
30          Glucose     = glucose;
31          Hemoglobin  = hemoglobin;
32          Status      = glucose > 6.1 ? "Глюкоза підвищена" : hemoglobin < 120 ?
    "Гемоглобін знижений" : "Норма";
33      }
34  }
```

Кілька Task<T> з WaitAll

Task.WaitAll коректно працює і з масивами Task<T>. Після WaitAll до всіх результатів можна звертатись через Result без додаткового блокування — результати вже готові:

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  Task<string>[] diagnostics = new Task<string>[]
 6  {
 7      Task.Run(() => { Thread.Sleep(300); return "Кров: норма"; }),
 8      Task.Run(() => { Thread.Sleep(200); return "Сеча: норма"; }),
 9      Task.Run(() => { Thread.Sleep(400); return "Рентген: без патологій"; }),
10  };
11
12  Console.WriteLine("Паралельна діагностика запущена...");
13  Task.WaitAll(diagnostics);
14
15  Console.WriteLine("\n=== Зведений протокол обстеження ===");
16  for (int i = 0; i < diagnostics.Length; i++)
17      Console.WriteLine($"  {(i + 1).ToString()}. {diagnostics[i].Result}");
18
19  Console.WriteLine("Протокол сформовано. Направляю до лікаря.");
```

Важливий момент: звернення до Result після WaitAll не блокує — всі завдання вже завершились. Але якщо завдання завершилось з винятком, Result кине AggregateException при зверненні. Завжди обробляйте можливі винятки через try/catch, якщо завдання може завершитись з помилкою.

16.3. Продовження завдань. ContinueWith

Завдання-продовження (continuation task) — це завдання, яке автоматично запускається після завершення іншого завдання. Це ключовий механізм побудови **послідовних ланцюгів обробки** (pipelines): результат одного кроку передається наступному, а кожен крок запускається лише тоді, коли попередній завершено. На відміну

від Wait(), де поточний потік блокується в очікуванні, ContinueWith не блокує — він лише реєструє «що виконати потім» і повертає управління негайно.

Базовий синтаксис ContinueWith

```C#
using System;
using System.Threading;
using System.Threading.Tasks;

// Перше завдання — реєстрація пацієнта
Task registration = new Task(() =>
{
    Console.WriteLine($"[Task {Task.CurrentId}] Реєстрація пацієнта Бойко О.П.");
    Thread.Sleep(200);
    Console.WriteLine($"[Task {Task.CurrentId}] Реєстрацію завершено");
});

// Завдання-продовження — запускається після registration
Task examination = registration.ContinueWith(prevTask =>
{
    Console.WriteLine($"[Task {Task.CurrentId}] Огляд лікаря (після завдання {prevTask.Id.ToString()})");
    Thread.Sleep(300);
    Console.WriteLine($"[Task {Task.CurrentId}] Огляд завершено");
});

registration.Start(); // запускаємо перше завдання
examination.Wait();   // чекаємо останнього в ланцюгу

Console.WriteLine("[Main] Маршрут пацієнта пройдено");
```

Метод ContinueWith приймає делегат Action<Task> або Action<Task<T>> — параметр prevTask це посилання на попереднє завдання, що щойно завершилось. Через нього можна отримати Id, Status, Exception і, у разі Task<T>, Result попереднього кроку.

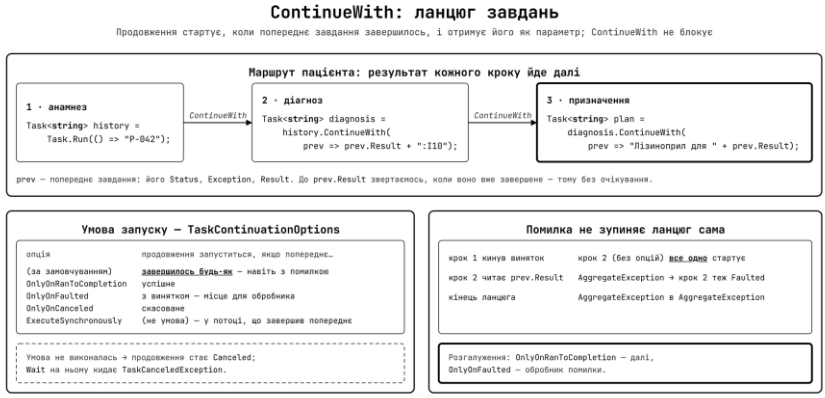
Передача результату між завданнями

Найпрактичніший сценарій ContinueWith — ланцюг обробки, де кожен крок отримує результат попереднього через Task<T>:

```C#
using System;
using System.Threading;
using System.Threading.Tasks;

// Крок 1: збір анамнезу → повертає код пацієнта
Task<string> anamnesis = Task.Run(() =>
{
```

```
 8      Console.WriteLine("[Анамнез] Збираю скарги пацієнта...");
 9      Thread.Sleep(200);
10      string patientCode = "PT-2024-0897";
11      Console.WriteLine($"[Анамнез] Карту відкрито: {patientCode}");
12      return patientCode;
13  });
14
15  // Крок 2: діагностика → отримує код і повертає діагноз
16  Task<string> diagnosis = anamnesis.ContinueWith(prev =>
17  {
18      string code = prev.Result; // результат попереднього кроку
19      Console.WriteLine($"[Діагноз] Аналізую дані картки {code}...");
20      Thread.Sleep(300);
21      string diag = "J06.9 — ГРВІ";
22      Console.WriteLine($"[Діагноз] Встановлено: {diag}");
23      return diag;
24  });
25
26  // Крок 3: призначення → отримує діагноз і повертає план лікування
27  Task<string> treatment = diagnosis.ContinueWith(prev =>
28  {
29      string diag = prev.Result;
30      Console.WriteLine($"[Лікування] Підбираю терапію для {diag}...");
31      Thread.Sleep(200);
32      string plan = "Парацетамол 500мг × 3/день, рясне пиття, постільний режим";
33      Console.WriteLine($"[Лікування] План: {plan}");
34      return plan;
35  });
36
37  treatment.Wait(); // чекаємо кінця ланцюга
38
39  Console.WriteLine($"\n=== Підсумок ===");
40  Console.WriteLine($"Діагноз:  {diagnosis.Result}");
41  Console.WriteLine($"Лікування: {treatment.Result}");
```



Ланцюг ContinueWith — клінічний маршрут пацієнта

Довгий ланцюг — pipeline

Кілька ContinueWith можна об'єднати в лінійний ланцюг, де кожен наступний крок запускається після попереднього. Такий патерн називають **pipeline** (конвеєр):

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  Task step1 = new Task(() =>
 6  {
 7      Console.WriteLine($"[Крок 1 / Task {Task.CurrentId}] Прийом та
    реєстрація");
 8      Thread.Sleep(100);
 9  });
10
11  Task step2 = step1.ContinueWith(t =>
12  {
13      Console.WriteLine($"[Крок 2 / Task {Task.CurrentId}] Попередній огляд
    медсестри (після Task {t.Id.ToString()})");
14      Thread.Sleep(150);
15  });
16
17  Task step3 = step2.ContinueWith(t =>
18  {
19      Console.WriteLine($"[Крок 3 / Task {Task.CurrentId}] Огляд лікаря та
    діагностика (після Task {t.Id.ToString()})");
20      Thread.Sleep(200);
21  });
22
23  Task step4 = step3.ContinueWith(t =>
24  {
25      Console.WriteLine($"[Крок 4 / Task {Task.CurrentId}] Призначення та видача
    рецепту (після Task {t.Id.ToString()})");
26      Thread.Sleep(100);
27  });
28
29  step1.Start();
30  step4.Wait();
31
32  Console.WriteLine("[Main] Повний маршрут пацієнта пройдено успішно");
```

Кожен крок отримує посилання на попередній Task (параметр t) і може перевірити його статус. Ланцюг виконується суворо послідовно, але **не блокує головний потік** між кроками — між кроками головний потік вільний.

TaskContinuationOptions: умовні продовження

За замовчуванням ContinueWith запускається після завершення попереднього завдання **незалежно від результату** — чи воно успішне,

чи скасоване, чи завершилось з помилкою. TaskContinuationOptions дозволяє вказати умову, при якій продовження запуститься:

Опція	Коли запускається
OnlyOnRanToCompletion	Тільки якщо попереднє завершилось успішно
OnlyOnFaulted	Тільки якщо попереднє завершилось з винятком
OnlyOnCanceled	Тільки якщо попереднє було скасовано
NotOnRanToCompletion	Якщо попереднє НЕ успішне
NotOnFaulted	Якщо попереднє НЕ з помилкою
ExecuteSynchronously	Виконати синхронно в тому ж потоці

```C#
using System;
using System.Threading;
using System.Threading.Tasks;

Task riskyAnalysis = Task.Run(() =>
{
    Console.WriteLine("[Аналіз] Запускаю аналіз зразка...");
    Thread.Sleep(200);
    throw new Exception("Зразок пошкоджено при транспортуванні");
});

// Успішне продовження — спрацює тільки якщо аналіз пройшов без помилок
Task onSuccess = riskyAnalysis.ContinueWith(
    t => Console.WriteLine($"[Успіх] Результат аналізу занесено до системи"),
    TaskContinuationOptions.OnlyOnRanToCompletion
);

// Помилкове продовження — спрацює тільки при помилці
Task onError = riskyAnalysis.ContinueWith(
    t => Console.WriteLine($"[Помилка] {t.Exception?.InnerException?.Message} — повторне взяття зразка призначено"),
    TaskContinuationOptions.OnlyOnFaulted
);

try
{
    Task.WaitAll(onSuccess, onError);
}
catch (AggregateException ae)
{
    // Завдання, що не виконались (умова не спрацювала), переходять у Canceled
    // WaitAll кине виняток для них — ігноруємо
```

```
    foreach (var ex in ae.InnerExceptions)
        if (ex is not TaskCanceledException)
            Console.WriteLine($"Неочікувана помилка: {ex.Message}");
}
```

Зверніть: якщо умова TaskContinuationOptions не виконана, продовження переходить у стан Canceled (а не просто ігнорується). Тому WaitAll на набір умовних продовжень може кинути AggregateException з TaskCanceledException всередині — це очікувана поведінка, яку треба обробляти.

Обробка помилок у ланцюгу

Помилка в ланцюгу ContinueWith сама по собі ланцюг **не зупиняє**: продовження без опцій стартує після завершення попереднього кроку незалежно від того, чи був він успішним. Виняток «підіймається» далі лише тоді, коли наступний крок звертається до prev.Result (або викликає prev.Wait()): тоді він отримує AggregateException і сам завершується з помилкою — і так до кінця ланцюга, з усе глибшими вкладеними обгортками. Правильний патерн — розмістити завдання-обробник помилок наприкінці ланцюга з OnlyOnFaulted:

```
C#
using System;
using System.Threading;
using System.Threading.Tasks;

Task<string> step1 = Task.Run(() =>
{
    Console.WriteLine("[Крок 1] Завантаження медичної картки...");
    Thread.Sleep(100);
    return "MED-2024-0042";
});

Task<string> step2 = step1.ContinueWith(t =>
{
    Console.WriteLine($"[Крок 2] Обробка картки {t.Result}...");
    Thread.Sleep(100);
    throw new InvalidOperationException("Картка заблокована адміністратором");
    return "processed"; // недосяжний код
}, TaskContinuationOptions.OnlyOnRanToCompletion);

// Продовження-обробник помилок
Task errorHandler = step2.ContinueWith(
    t => Console.WriteLine($"[Обробник] Помилка у ланцюгу:
{t.Exception?.InnerException?.Message}"),
    TaskContinuationOptions.OnlyOnFaulted
);

// Продовження при успіху
Task successHandler = step2.ContinueWith(
    t => Console.WriteLine($"[Успіх] Обробку завершено успішно"),
```

```
29      TaskContinuationOptions.OnlyOnRanToCompletion
30  );
31
32  try { Task.WaitAll(errorHandler, successHandler); }
33  catch (AggregateException) { }
34
35  Console.WriteLine("[Main] Ланцюг обробки завершено");
```

Такий підхід дозволяє будувати надійні ланцюги з розгалуженням на успішний і помилковий шляхи, не вдаючись до вкладених try/catch всередині кожного кроку.

TaskCompletionSource — Task під ручним керуванням

Task.Run() і new Task(...) створюють завдання, що виконують певний делегат у ThreadPool. Але іноді потрібен Task, що завершується **не через виконання делегата**, а в момент, коли ми самі вирішуємо. Наприклад:

— Потрібно обернути callback-based API в async-стиль

— Потрібно «сигналізувати» між частинами системи через Task

— Потрібно створити Task, що чекає зовнішньої події (таймер, WebSocket-повідомлення, результат від іншого сервісу)

Для цього існує TaskCompletionSource<T> (або TaskCompletionSource без типового параметру для Task):

```
C#
1  var tcs = new TaskCompletionSource<string>();
2
3  Task<string> task = tcs.Task; // Task, що "ще не завершений"
4
5  // Пізніше, в будь-якому місці:
6  tcs.SetResult("значення");    // Task завершується успішно
7  tcs.SetException(ex);         // Task завершується з помилкою
8  tcs.SetCanceled();            // Task скасовується
```

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  // Симуляція: очікуємо результат аналізу з лабораторії
 6  // Лабораторія "дзвонить" нам через callback — ми хочемо перетворити це на Task
 7
 8  TaskCompletionSource<string> analysisResult = new
    TaskCompletionSource<string>();
 9
10  // Симулюємо "фонову лабораторну систему", що надсилає результат через 300 мс
11  Thread labSystem = new Thread(() =>
12  {
13      Thread.Sleep(300);
```

```
    Console.WriteLine("[Лабораторія] Аналіз завершено, надсилаю результат...");

    // "Сигналізуємо" очікуючому завданню
    analysisResult.SetResult("Гемоглобін: 128 г/л — норма, Глюкоза: 5.4 ммоль/л — норма");
});
labSystem.IsBackground = true;
labSystem.Start();

// Головний потік чекає результат через Task — не через Thread.Join або callback
Console.WriteLine("[Лікар] Замовив аналіз, чекаю результату...");
string result = await analysisResult.Task; // блокує асинхронно, а не синхронно
Console.WriteLine($"[Лікар] Результат отримано: {result}");
```

C#

```
using System;
using System.Threading;
using System.Threading.Tasks;

// TaskCompletionSource з обробкою помилок
TaskCompletionSource<byte[]> imageLoad = new TaskCompletionSource<byte[]>();

Thread medicalImaging = new Thread(() =>
{
    Thread.Sleep(200);

    bool networkError = true; // симуляція помилки
    if (networkError)
    {
        imageLoad.SetException(
            new TimeoutException("Сервер DICOM-зображень не відповідає (timeout 200ms)"));
    }
    else
    {
        imageLoad.SetResult(new byte[] { 0xFF, 0xD8, 0xFF }); // JPEG header
    }
});
medicalImaging.IsBackground = true;
medicalImaging.Start();

Console.WriteLine("[PACS] Запитую рентгенівський знімок...");
try
{
    byte[] image = await imageLoad.Task;
    Console.WriteLine($"[PACS] Зображення отримано: {image.Length} байт");
}
catch (TimeoutException ex)
{
    Console.WriteLine($"[PACS] Помилка: {ex.Message}");
    Console.WriteLine("[PACS] Призначено повторний запит через 5 хв");
}
```

TaskCompletionSource є «мостом» між callback/event-моделлю і сучасним async/await. Якщо ви отримали API зі старою моделлю

(BeginXxx/EndXxx, EventHandler, callbacks) — TaskCompletionSource перетворює її у Task, який можна await-ити.

Важливо: після виклику SetResult, SetException або SetCanceled стан Task зафіксований — повторний виклик кине InvalidOperationException. Для «одноразового сигналу» є зручна альтернатива TrySetResult/TrySetException/TrySetCanceled, що повертають bool замість кидання винятку.

16.4. Клас Parallel

Клас System.Threading.Tasks.Parallel надає зручні статичні методи для паралельного виконання операцій, не вимагаючи явного створення і управління завданнями. Він автоматично розподіляє роботу між доступними ядрами процесора і чекає завершення всіх паралельних операцій перед поверненням.

На відміну від Task.Run(), де розробник явно описує кожне завдання, Parallel орієнтований на масові паралельні операції: виконання набору незалежних дій, паралельний обхід колекцій, паралельний цикл. Він ефективний, коли є багато однотипних незалежних завдань, що добре розпаралелюються.

Parallel.Invoke — паралельне виконання дій

Parallel.Invoke приймає довільний набір делегатів Action і виконує їх паралельно. Метод повертається тільки після завершення **всіх** переданих дій:

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  Console.WriteLine("Початок комплексного обстеження пацієнта Петренко І.О.");
 6
 7  Parallel.Invoke(
 8      () => {
 9          Console.WriteLine($"[Task {Task.CurrentId}] Аналіз крові...");
10          Thread.Sleep(400);
11          Console.WriteLine($"[Task {Task.CurrentId}] Аналіз крові: готово");
12      },
13      () => {
14          Console.WriteLine($"[Task {Task.CurrentId}] ЕКГ-запис...");
15          Thread.Sleep(300);
16          Console.WriteLine($"[Task {Task.CurrentId}] ЕКГ: готово");
17      },
18      () => {
```

```
        Console.WriteLine($"[Task {Task.CurrentId}] УЗД черевної порожнини...");
        Thread.Sleep(500);
        Console.WriteLine($"[Task {Task.CurrentId}] УЗД: готово");
    },
    () => {
        Console.WriteLine($"[Task {Task.CurrentId}] Рентген грудної клітини...");
        Thread.Sleep(350);
        Console.WriteLine($"[Task {Task.CurrentId}] Рентген: готово");
    }
);

Console.WriteLine("Комплексне обстеження завершено — всі результати готові");
```

Parallel.Invoke не гарантує порядку виконання і не гарантує, що кожна дія виконається в окремому потоці — якщо ядер менше, ніж дій, деякі дії будуть почергувати. До того ж частину роботи виконує сам потік, що викликав Parallel.Invoke/For/ForEach: він не просто чекає, а бере участь у циклі. Якщо в кількох діях чи ітераціях виникли винятки, Parallel дочікується завершення вже запущених і кидає один AggregateException з усіма винятками в InnerExceptions. Але результат завжди правильний: метод поверне управління тільки після завершення всіх.

Parallel.For — паралельний цикл for

Parallel.For(fromInclusive, toExclusive, body) виконує тіло циклу паралельно для всіх індексів у діапазоні [from, to). Планувальник TPL автоматично розбиває діапазон на пакети і виконує їх у потоках пулу:

C#
```
using System;
using System.Threading;
using System.Threading.Tasks;

string[] patients = { "Коваль М.", "Петренко І.", "Бойко О.", "Мороз В.", "Сидоренко Т." };

Console.WriteLine("Масовий розрахунок ризиків для пацієнтів:");

Parallel.For(0, patients.Length, i =>
{
    // i — індекс ітерації, кожна може виконуватись у різних потоках
    string patient = patients[i];
    Console.WriteLine($"[Потік {Thread.CurrentThread.ManagedThreadId.ToString()}] Розрахунок ризику: {patient}");
    Thread.Sleep(200); // імітуємо обчислення
    Console.WriteLine($"[Потік {Thread.CurrentThread.ManagedThreadId.ToString()}] Ризик {patient}: низький");
});
```

```
17
18  Console.WriteLine("Розрахунок завершено для всіх пацієнтів");
```

Зверніть: в одній ітерації використовується Thread.CurrentThread.ManagedThreadId — він показує, що різні ітерації можуть виконуватись у різних потоках пулу.

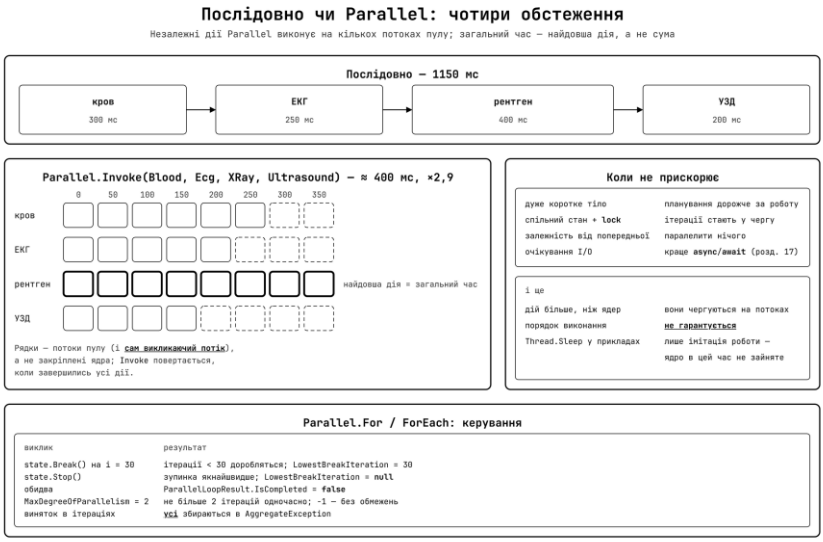
Parallel.ForEach — паралельний обхід колекції

Parallel.ForEach — аналог Parallel.For, але замість індексів він приймає колекцію IEnumerable<T> і виконує тіло для кожного елемента паралельно:

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Threading;
 4  using System.Threading.Tasks;
 5
 6  List<string> labOrders = new List<string>
 7  {
 8      "Кров-Коваль",  "Сеча-Бойко",   "Мокрота-Мороз",
 9      "Кров-Руденко", "ЕКГ-Шевченко", "Кров-Гриценко"
10  };
11
12  Console.WriteLine("Лабораторія: паралельна обробка замовлень");
13
14  ParallelLoopResult result = Parallel.ForEach(labOrders, order =>
15  {
16      Console.WriteLine($"[Потік
    {Thread.CurrentThread.ManagedThreadId.ToString()}] Обробка: {order}");
17      Thread.Sleep(200);
18      Console.WriteLine($"[Потік
    {Thread.CurrentThread.ManagedThreadId.ToString()}] Готово: {order}");
19  });
20
21  Console.WriteLine($"IsCompleted: {result.IsCompleted.ToString()}");
22  Console.WriteLine("Всі замовлення оброблено");
```

Метод повертає структуру ParallelLoopResult з двома властивостями:

- IsCompleted — true, якщо цикл пройшов до кінця без переривань
- LowestBreakIteration — індекс, на якому відбулось переривання через Break(), або null



Послідовне vs паралельне виконання

ParallelLoopState: керування циклом

Звичайний break у тілі Parallel.For викличе помилку компілятора — ви не можете використовувати break у лямбді. Натомість Parallel надає об'єкт ParallelLoopState, який передається як другий параметр:

Break() — зупинка після поточного діапазону

pls.Break() сигналізує планувальнику, що після завершення всіх ітерацій **з індексом, меншим за поточний**, подальші ітерації не потрібні. Це не негайна зупинка — ітерації з вищими індексами, що вже запущені, продовжать виконуватись:

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  Console.WriteLine("Пошук першого пацієнта з критичним показником:");
 6
 7  ParallelLoopResult result = Parallel.For(0, 20, (i, pls) =>
 8  {
 9      if (pls.ShouldExitCurrentIteration) return; // перевіряємо перед роботою
10
11      int pulse = 60 + i * 4; // симуляція: пульс зростає з індексом
```

```
12      Console.WriteLine($"  Пацієнт #{i.ToString()}: пульс {pulse.ToString()}
    уд/хв");
13      Thread.Sleep(30);
14
15      if (pulse > 110) // знайшли критичний показник
16      {
17          Console.WriteLine($"  *** Критичний пульс у пацієнта #{i.ToString()} —
    зупиняю пошук ***");
18          pls.Break();
19      }
20  });
21
22  Console.WriteLine($"\nIsCompleted: {result.IsCompleted.ToString()}");
23  Console.WriteLine($"LowestBreakIteration:
    {result.LowestBreakIteration?.ToString() ?? "null"}");
```

Stop() — негайна зупинка

pls.Stop() вимагає припинити весь цикл якнайшвидше — навіть вже запущені ітерації повинні перевіряти pls.IsStopped і завершуватись достроково. Після Stop() LowestBreakIteration завжди null, а IsCompleted — false:

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  Console.WriteLine("Перевірка показників: зупинка при виявленні небезпечного
    стану");
 6
 7  Parallel.ForEach(new[] { 72, 68, 145, 89, 55, 130 }, (pulse, pls) =>
 8  {
 9      if (pls.IsStopped) return; // якщо зупинено — виходимо негайно
10
11      Console.WriteLine($"[Потік
    {Thread.CurrentThread.ManagedThreadId.ToString()}] Пульс: {pulse.ToString()}
    уд/хв");
12
13      if (pulse > 130)
14      {
15          Console.WriteLine($"  !!! НЕБЕЗПЕКА: пульс {pulse.ToString()} — НЕГАЙНА
    ЗУПИНКА СИСТЕМИ !!!");
16          pls.Stop(); // зупиняємо весь цикл
17      }
18  });
19
20  Console.WriteLine("Перевірку завершено");
```

Break vs Stop: Break гарантує, що всі ітерації до поточної будуть виконані (корисно для пошуку з гарантованим покриттям). Stop зупиняє якнайшвидше без гарантій покриття (корисно для раннього виходу при знайденні умови).

ParallelOptions: налаштування паралелізму

ParallelOptions дозволяє налаштувати поведінку Parallel.For/ForEach/Invoke. Найважливіша властивість — MaxDegreeOfParallelism: максимальна кількість одночасно виконуваних ітерацій.

```C#
using System;
using System.Threading;
using System.Threading.Tasks;

ParallelOptions options = new ParallelOptions
{
    MaxDegreeOfParallelism = 2 // не більше 2 паралельних операцій одночасно
};

string[] patients = { "Коваль", "Бойко", "Мороз", "Петренко", "Руденко",
"Шевченко" };

Console.WriteLine("Обробка з обмеженням 2 паралельні операції:");

Parallel.ForEach(patients, options, patient =>
{
    Console.WriteLine($"[Потік
{Thread.CurrentThread.ManagedThreadId.ToString()}] Початок: {patient}");
    Thread.Sleep(300);
    Console.WriteLine($"[Потік
{Thread.CurrentThread.ManagedThreadId.ToString()}] Готово: {patient}");
});

Console.WriteLine("Завершено");
```

MaxDegreeOfParallelism = 1 фактично перетворює паралельний цикл на послідовний — корисно для налагодження. MaxDegreeOfParallelism = -1 (значення за замовчуванням) означає «без обмежень» — TPL сам вирішує.

У реальних системах обмеження паралелізму важливе: якщо кожен потік відкриває з'єднання з базою даних або відправляє мережевий запит, необмежений паралелізм може перевантажити зовнішній ресурс.

Коли Parallel не прискорює

Parallel — не срібна куля. Є сценарії, де паралелізація не допомагає або навіть шкодить:

Занадто короткі ітерації. Якщо тіло циклу займає наносекунди (наприклад, просто додає два числа), накладні витрати — виклик делегата на кожну ітерацію, розподіл діапазону між потоками, синхронізація в кінці — можуть перевищити виграш від паралелізму.

Parallel.For сам групує індекси в діапазони, тож обчислення від кількох мікросекунд на ітерацію зазвичай уже розпаралелюються вигідно; але виграш завжди варто вимірювати, а не припускати.

Операції із спільним станом без синхронізації. Якщо всі ітерації змінюють одну змінну без lock, виникне race condition. Якщо додати lock — паралелізм де-факто зникне через серіалізацію доступу.

Послідовна природа даних. Якщо кожна ітерація залежить від результату попередньої, паралелізація неможлива — задачу потрібно переосмислити.

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  // Приклад коректного паралельного накопичення через Interlocked
 6  int totalProcessed = 0;
 7
 8  Parallel.For(0, 100, i =>
 9  {
10      Thread.Sleep(5); // коротка робота
11      Interlocked.Increment(ref totalProcessed); // атомарний інкремент без lock
12  });
13
14  Console.WriteLine($"Оброблено записів: {totalProcessed.ToString()}");
```

Interlocked.Increment — атомарна операція збільшення цілого числа, яка не потребує lock. Для простих лічильників у паралельних циклах це ефективніше за lock (obj) { count++; }.

16.5. Скасування завдань. CancellationToken

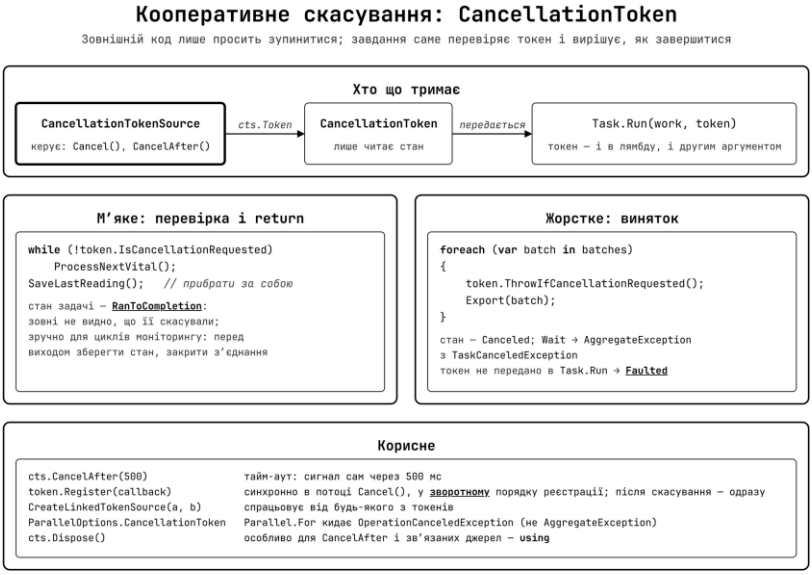
У реальних системах завдання нерідко потрібно зупинити достроково: користувач натиснув «Скасувати», сплив тайм-аут, виникла аварійна ситуація або система перевантажена. Примусове завершення потоку через Thread.Abort() давно застаріло, а в .NET Core і .NET 5+ не підтримується (метод лишився, але кидає PlatformNotSupportedException): воно небезпечне, бо залишає ресурси неприбраними. Правильний підхід — **кооперативне скасування** через CancellationToken.

Концепція кооперативного скасування полягає в тому, що завдання саме перевіряє, чи не надійшов сигнал скасування, і коректно завершується у зручний момент — після звільнення ресурсів, запису незавершеного результату, закриття з'єднань. Жодного примусового переривання — лише добровільна відповідь на запит.

Ключові типи

Тип	Роль
CancellationTokenSource	Джерело сигналу скасування. Тільки він може викликати Cancel()
CancellationToken	Токен — «квиток» з інформацією про скасування. Передається у завдання
OperationCanceledException	Виняток, що сигналізує про кооперативне скасування

Розподіл ролей чіткий: **той хто управляє** (зовнішній код) тримає CancellationTokenSource і вирішує, коли скасувати. **Завдання** отримує лише CancellationToken — воно може лише перевірити стан, але не може само себе скасувати через чужий джерело.



Механізм скасування завдань: CancellationToken

Спосіб 1: м'яке скасування через IsCancellationRequested

Найпростіший підхід — завдання регулярно перевіряє token.IsCancellationRequested і виходить через return при виявленні сигналу. Стан завдання після цього — RanToCompletion (успішне завершення), оскільки виняток не кидався:

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  CancellationTokenSource cts = new CancellationTokenSource();
 6  CancellationToken token = cts.Token;
 7
 8  Task monitoring = Task.Run(() =>
 9  {
10      Console.WriteLine("[Моніторинг] Систему моніторингу запущено");
11      int cycle = 0;
12
13      while (!token.IsCancellationRequested) // цикл до сигналу скасування
14      {
15          cycle++;
16          Console.WriteLine($"[Моніторинг] Цикл {cycle.ToString()}: пульс 72,
    тиск 120/80");
17          Thread.Sleep(200);
18      }
19
20      // Виходимо коректно — зберігаємо стан, звільняємо ресурси
21      Console.WriteLine($"[Моніторинг] Отримано сигнал зупинки. Завершено після
    {cycle.ToString()} циклів.");
22  }, token);
23
24  Thread.Sleep(700); // даємо поопрацювати
25  Console.WriteLine("[Main] Надсилаю сигнал скасування моніторингу...");
26  cts.Cancel();
27
28  monitoring.Wait();
29  Console.WriteLine($"[Main] Статус: {monitoring.Status}"); // RanToCompletion
30  cts.Dispose();
```

М'яке скасування підходить для довгих циклічних операцій, де є зручна точка перевірки між ітераціями.

Спосіб 2: жорстке скасування через ThrowIfCancellationRequested

token.ThrowIfCancellationRequested() кидає OperationCanceledException, якщо надійшов сигнал скасування. Завдання переходить у стан Canceled — але лише за умови, що той самий токен передано і в Task.Run(..., token) другим аргументом (як у прикладі нижче). Якщо токен лише захоплено лямбдою, TPL не може зв'язати виняток зі скасуванням саме

цієї задачі, і вона опиниться в стані Faulted. При виклику Wait() або Result зовнішній код отримає AggregateException з TaskCanceledException всередині:

```C#
using System;
using System.Threading;
using System.Threading.Tasks;

CancellationTokenSource cts = new CancellationTokenSource();
CancellationToken token = cts.Token;

Task dataExport = Task.Run(() =>
{
    Console.WriteLine("[Експорт] Початок експорту медичних даних...");

    for (int i = 1; i <= 20; i++)
    {
        token.ThrowIfCancellationRequested(); // кидає виняток при Cancel()

        Console.WriteLine($"[Експорт] Пакет {i.ToString()}/20 відправлено");
        Thread.Sleep(100);
    }

    Console.WriteLine("[Експорт] Експорт завершено повністю");
}, token);

Thread.Sleep(450); // скасовуємо після ~4 пакетів
Console.WriteLine("[Main] Скасовую експорт (адміністратор зупинив)...");
cts.Cancel();

try
{
    dataExport.Wait();
}
catch (AggregateException ae)
{
    foreach (var ex in ae.InnerExceptions)
    {
        if (ex is TaskCanceledException)
            Console.WriteLine("[Main] Завдання коректно скасовано");
        else
            Console.WriteLine($"[Main] Неочікувана помилка: {ex.Message}");
    }
}

Console.WriteLine($"[Main] Статус: {dataExport.Status}"); // Canceled
cts.Dispose();
```

Жорстке скасування підходить для задач, де після отримання сигналу скасування продовжувати виконання немає сенсу і завдання може бути перервано в будь-якій точці перевірки.

Перевірка стану: IsCanceled vs IsFaulted

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  CancellationTokenSource cts = new CancellationTokenSource();
 6
 7  Task task = Task.Run(() =>
 8  {
 9      Thread.Sleep(500);
10      cts.Token.ThrowIfCancellationRequested();
11  }, cts.Token);
12
13  cts.Cancel(); // скасовуємо одразу
14
15  try { task.Wait(); } catch (AggregateException) { }
16
17  Console.WriteLine($"IsCanceled:  {task.IsCanceled.ToString()}");   // true
18  Console.WriteLine($"IsFaulted:   {task.IsFaulted.ToString()}");    // false
19  Console.WriteLine($"IsCompleted: {task.IsCompleted.ToString()}");  // true (всі
    завершені стани)
20  Console.WriteLine($"Status:      {task.Status}");                    // Canceled
21  cts.Dispose();
```

Реєстрація обробника: Register()

CancellationToken.Register() дозволяє зареєструвати делегат, який буде викликаний **при скасуванні**. Це аналог події скасування — корисний для звільнення ресурсів, логування або надсилання повідомлень:

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  CancellationTokenSource cts = new CancellationTokenSource();
 6
 7  // Реєструємо обробник скасування
 8  cts.Token.Register(() =>
 9  {
10      Console.WriteLine("[Register] Сигнал скасування отримано — повідомляю
    адміністратора");
11      Console.WriteLine("[Register] Зберігаю проміжний стан до бази даних...");
12  });
13
14  Task longTask = Task.Run(() =>
15  {
16      Console.WriteLine("[Task] Тривала обробка даних розпочата...");
17      for (int i = 0; i < 10; i++)
18      {
19          cts.Token.ThrowIfCancellationRequested();
20          Console.WriteLine($"[Task] Крок {i.ToString()}...");
21          Thread.Sleep(150);
22      }
```

```
}, cts.Token);

Thread.Sleep(400);
cts.Cancel(); // обробник Register() викличеться тут

try { longTask.Wait(); } catch (AggregateException) { }
Console.WriteLine("[Main] Завдання завершено");
cts.Dispose();
```

Обробник Register() викликається синхронно в тому потоці, що викликав Cancel(). Якщо зареєстровано кілька обробників, вони викликаються у **зворотному** порядку реєстрації (останній зареєстрований — першим), а покладатися на конкретний порядок взагалі не рекомендовано. Якщо токен уже скасовано на момент Register(), обробник виконується одразу, у потоці, що реєструє.

CancelAfter — автоматичне скасування за тайм-аутом

CancellationTokenSource.CancelAfter(ms) автоматично надсилає сигнал скасування через вказану кількість мілісекунд. Це стандартний спосіб реалізації тайм-аутів:

```C#
using System;
using System.Threading;
using System.Threading.Tasks;

using CancellationTokenSource cts = new CancellationTokenSource();
cts.CancelAfter(500); // автоматичне скасування через 500мс

Task networkCall = Task.Run(() =>
{
    Console.WriteLine("[Запит] Звертаюсь до зовнішньої бази даних...");
    Thread.Sleep(800); // запит займає 800мс — більше за тайм-аут
    cts.Token.ThrowIfCancellationRequested();
    Console.WriteLine("[Запит] Дані отримано");
}, cts.Token);

try
{
    networkCall.Wait();
    Console.WriteLine("[Main] Запит успішний");
}
catch (AggregateException)
{
    Console.WriteLine("[Main] Тайм-аут: зовнішня база не відповіла вчасно");
    Console.WriteLine("[Main] Використовую кешовані дані");
}
```

Скасування у Parallel.For і Parallel.ForEach

CancellationToken передається у Parallel через ParallelOptions. При скасуванні Parallel зупиняє нові ітерації і кидає OperationCanceledException:

```
C#
using System;
using System.Threading;
using System.Threading.Tasks;

CancellationTokenSource cts = new CancellationTokenSource();

ParallelOptions options = new ParallelOptions
{
    CancellationToken    = cts.Token,
    MaxDegreeOfParallelism = 3
};

// Скасовуємо через 300мс
Task.Run(() => { Thread.Sleep(300); cts.Cancel(); });

try
{
    Parallel.For(0, 20, options, i =>
    {
        Console.WriteLine($"[Parallel] Обробка пацієнта #{i.ToString()}...");
        Thread.Sleep(200);
        Console.WriteLine($"[Parallel] Пацієнт #{i.ToString()} оброблено");
    });
}
catch (OperationCanceledException)
{
    Console.WriteLine("[Main] Паралельний цикл скасовано — отримано сигнал зупинки");
}

Console.WriteLine("[Main] Система зупинена коректно");
cts.Dispose();
```

Пов'язані токени (Linked tokens)

Іноді потрібно об'єднати кілька джерел скасування: наприклад, зовнішній тайм-аут **або** натискання «Скасувати». CancellationTokenSource.CreateLinkedTokenSource створює новий токен, що спрацьовує при скасуванні будь-якого з переданих:

```
C#
using System;
using System.Threading;
using System.Threading.Tasks;

// Два незалежних джерела скасування
using CancellationTokenSource userCancel    = new CancellationTokenSource();
```

```
using CancellationTokenSource timeoutCancel = new CancellationTokenSource();
timeoutCancel.CancelAfter(600); // тайм-аут 600мс

// Об'єднаний токен — спрацює при Cancel() на будь-якому з двох
using CancellationTokenSource linked =
    CancellationTokenSource.CreateLinkedTokenSource(userCancel.Token,
timeoutCancel.Token);

Task surgery = Task.Run(() =>
{
    Console.WriteLine("[Операція] Хірургічна операція розпочата...");
    for (int i = 1; i <= 20; i++)
    {
        linked.Token.ThrowIfCancellationRequested();
        Console.WriteLine($"[Операція] Крок {i.ToString()}/20");
        Thread.Sleep(100);
    }
    Console.WriteLine("[Операція] Операція завершена успішно");
}, linked.Token);

// Симулюємо: або користувач скасує, або тайм-аут — хто перший
// userCancel.Cancel(); // розкоментуйте, щоб перевірити ручне скасування

try
{
    surgery.Wait();
    Console.WriteLine("[Main] Операція пройшла успішно");
}
catch (AggregateException)
{
    string reason = userCancel.IsCancellationRequested ? "рішення хірурга" :
"перевищено тайм-аут";
    Console.WriteLine($"[Main] Операцію зупинено: {reason}");
}
```

Пов'язані токени — потужний інструмент для реалізації складних стратегій скасування, де кілька незалежних подій можуть ініціювати зупинку однієї операції. Після завершення їх потрібно утилізувати через Dispose() (або using), щоб звільнити ресурси.

Питання для самоконтролю — Розділ 16. Паралельне програмування та TPL

16.1. Клас Task. Основи TPL

1. Що таке TPL і яку проблему вона вирішує порівняно з прямим використанням класу Thread? Поясніть концепцію пулу потоків (ThreadPool) і чому він ефективніший за постійне створення нових потоків.

2. Порівняйте три способи створення завдання: new Task(...) + Start(), Task.Factory.StartNew() і Task.Run(). Коли кожен з них є доцільним вибором?
3. Що виведе наступний код? Чи гарантований порядок виводу?

```C#
Task t = Task.Run(() =>
{
    Console.WriteLine($"Task: {Task.CurrentId}");
});
Console.WriteLine("Main before Wait");
t.Wait();
Console.WriteLine("Main after Wait");
```

4. Поясніть різницю між Task.WaitAll і Task.WaitAny. Для яких клінічних сценаріїв підходить кожен? Що повертає WaitAny і що відбувається з рештою завдань після повернення?
5. Які стани описує перерахування TaskStatus? Намалюйте (або опишіть словами) граф переходів між станами від Created до термінального стану.
6. Що відбудеться, якщо завдання кине виняток, а виклик Wait() відбудеться? Яким типом буде виняток, що прилетить до catch? Як отримати оригінальне повідомлення про помилку?
7. Що робить Task.RunSynchronously()? В яких сценаріях він корисний і що є його обмеженнями?
8. Метод Task.CurrentId повертає null поза завданням. Напишіть невеликий приклад, що демонструє, де Task.CurrentId дорівнює null, а де — числовому значенню.

16.2. Вкладені завдання та масиви завдань. Task<T>

9. Що таке вкладене (nested) завдання і яка його поведінка за замовчуванням щодо батьківського? Чому батьківське завдання може завершитись раніше вкладеного?
10. Що означає TaskCreationOptions.AttachedToParent? Як ця опція змінює взаємозалежність між батьківським і дочірнім завданням? Як поширюються виключення дочірнього завдання?
11. Порівняйте поведінку наступних двох фрагментів:

```C#
// Фрагмент A
Task parent = Task.Factory.StartNew(() =>
{
```

```
    Task.Factory.StartNew(() => { Thread.Sleep(500);
Console.WriteLine("Inner"); });
    Console.WriteLine("Outer done");
});
parent.Wait();
Console.WriteLine("Main done");

// Фрагмент B — те саме, але з AttachedToParent
Task parent = Task.Factory.StartNew(() =>
{
    Task.Factory.StartNew(() => { Thread.Sleep(500);
Console.WriteLine("Inner"); },
        TaskCreationOptions.AttachedToParent);
    Console.WriteLine("Outer done");
});
parent.Wait();
Console.WriteLine("Main done");
```

Опишіть порядок виводу для обох фрагментів.

12. Що повертає властивість Result у Task<T>? Чому звернення до Result може заблокувати поточний потік? Коли звернення до Result після Task.WaitAll не є блокуючим?
13. Напишіть код, що паралельно запускає три Task<string>, потім через Task.WaitAll чекає завершення всіх і виводить результати у порядку їх оголошення (не обов'язково завершення).
14. Розробіть клас LabResult з полями PatientName, Glucose, Hemoglobin і метод Task<LabResult> RunLabAsync(string name). Запустіть завдання для 4 пацієнтів паралельно, дочекайтесь всіх і виведіть зведену таблицю результатів.
15. Якщо один із Task у масиві Task<T>[] завершився з виключенням, що станеться при зверненні до task.Result після WaitAll? Як безпечно отримати результати тільки успішних завдань?

16.3. Продовження завдань. ContinueWith

16. Що таке завдання-продовження (continuation task)? Яку перевагу воно дає над Wait() + новий Task.Run()?
17. Який параметр отримує делегат у ContinueWith? Що через нього можна дізнатись про попереднє завдання?
18. Що виведе наступний код і в якому порядку?

```
C#
Task<int> t = Task.Run(() => { Thread.Sleep(100); return 42; });
Task t2 = t.ContinueWith(prev => Console.WriteLine($"Результат:
{prev.Result}"));
t2.Wait();
```

```
4  Console.WriteLine("Головний потік");
```

19. Поясніть TaskContinuationOptions.OnlyOnRanToCompletion і TaskContinuationOptions.OnlyOnFaulted. У якому стані опиниться продовження, якщо умова не виконалась? Яке виключення кине WaitAll у цьому випадку?
20. Побудуйте ланцюг із чотирьох кроків через ContinueWith (реєстрація → огляд → аналізи → призначення), де кожен крок повертає результат типу string, що передається наступному. Використовуйте Task<string> і ContinueWith<string>.
21. Що таке TaskCompletionSource<T>? У яких трьох сценаріях він корисний? Поясніть різницю між SetResult і TrySetResult.
22. Реалізуйте функцію Task<string> WrapCallbackAsync(Action<string> callback), що перетворює callback-based API в Task-based за допомогою TaskCompletionSource. Покажіть, як SetResult і SetException відповідають різним сценаріям callback.
23. Знайдіть потенційну проблему в наступному коді:

```
C#
1  Task step1 = Task.Run(() => throw new Exception("fail"));
2  Task step2 = step1.ContinueWith(t => Console.WriteLine("Success"));
3  step2.Wait();
```

Чи виконається step2? Чому? Як виправити за допомогою TaskContinuationOptions?

16.4. Клас Parallel

24. Яка головна відмінність між Parallel і ручним управлінням завданнями через Task.Run? У яких сценаріях Parallel є зручнішим вибором?
25. Що гарантує Parallel.Invoke щодо завершення переданих дій? Чи гарантується порядок виконання? Чи гарантується, що кожна дія виконається в окремому потоці?
26. Порівняйте Parallel.For і Parallel.ForEach. Яка різниця в типах даних, з якими вони працюють? Що повертає ParallelLoopResult і які його властивості?
27. Що станеться, якщо написати break у тілі лямбди у Parallel.For? Як правильно перервати виконання паралельного циклу?

28. Поясніть різницю між ParallelLoopState.Break() і ParallelLoopState.Stop(). Після якого методу IsCompleted буде false? Коли LowestBreakIteration має значення?
29. Що означає MaxDegreeOfParallelism = 1 у ParallelOptions? Коли це корисно? Що означає MaxDegreeOfParallelism = -1?
30. Знайдіть проблему в наступному коді і поясніть, як її виправити:

```
C#
1  int total = 0;
2  Parallel.For(0, 1000, i =>
3  {
4      total++; // небезпечно
5  });
6  Console.WriteLine(total);
```

31. Спроєктуйте паралельне обчислення ризику для списку пацієнтів через Parallel.ForEach. Реалізуйте зупинку при виявленні першого пацієнта з критичним показником через ParallelLoopState.Stop(). Чому після зупинки результат може містити пацієнтів з індексом вище за той, на якому викликали Stop()?

16.5. Скасування завдань. CancellationToken

32. Що таке кооперативне скасування? Чому Thread.Abort() є небезпечним і чому він вилучений з .NET 5+?
33. Поясніть розподіл ролей між CancellationTokenSource і CancellationToken. Чому завдання отримує лише CancellationToken, а не CancellationTokenSource?
34. У чому різниця між м'яким скасуванням (через IsCancellationRequested) і жорстким (через ThrowIfCancellationRequested)? Який стан отримає Task після кожного варіанту скасування — RanToCompletion чи Canceled?
35. Що виведе наступний код?

```
C#
 1  CancellationTokenSource cts = new CancellationTokenSource();
 2  Task task = Task.Run(() =>
 3  {
 4      for (int i = 0; i < 10; i++)
 5      {
 6          cts.Token.ThrowIfCancellationRequested();
 7          Thread.Sleep(100);
 8          Console.WriteLine($"Step {i}");
 9      }
```

```
}, cts.Token);
Thread.Sleep(250);
cts.Cancel();
try { task.Wait(); } catch (AggregateException ae)
{
    Console.WriteLine(ae.InnerException?.GetType().Name);
}
Console.WriteLine(task.Status);
```

36. Що робить CancellationToken.Register()? У якому потоці виконується зареєстрований делегат? Для яких задач він корисний?
37. Як реалізувати тайм-аут для завдання за допомогою CancelAfter? Напишіть приклад, де завдання скасовується через 500 мс, якщо не завершилось раніше.
38. Що таке пов'язані токени (CreateLinkedTokenSource)? Наведіть приклад, де операція може бути скасована двома незалежними причинами: за тайм-аутом і за запитом користувача. Як після скасування визначити, яка саме причина спрацювала?
39. Передайте CancellationToken у Parallel.ForEach через ParallelOptions. Яке виключення кине Parallel при скасуванні? Порівняйте це з поведінкою при скасуванні звичайного Task.Run().

Розділ 17. Асинхронне програмування

17.1. async та await. Асинхронні методи

Асинхронне програмування — один з найважливіших концептів сучасної розробки програмного забезпечення. Щоб зрозуміти, навіщо воно існує, почнімо з проблеми, яку воно вирішує.

Проблема: блокуючий ввід-вивід

Більшість програм значну частину часу **чекають**: чекають відповіді від бази даних, чекають завантаження файлу з диску, чекають відповіді від зовнішнього API. У синхронному коді під час такого очікування потік просто заморожується — він не виконує жодної корисної роботи, але займає пам'ять і ресурс ОС.

Уявімо клінічну інформаційну систему. При синхронному підході: реєстратор запитує медичну картку пацієнта → програма чекає відповіді від бази даних (200 мс) → потік заблокований, інтерфейс завис. За 200 мс можна було зареєструвати ще трьох пацієнтів.

Ця проблема стає критичною у веб-застосунках: якщо сервер обслуговує тисячі одночасних запитів і кожен блокує потік на час очікування бази даних — сервер швидко вичерпає пул потоків і перестане відповідати.

Асинхронність вирішує цю проблему радикально: замість того щоб блокувати потік на час очікування, програма «відпускає» потік займатися іншою роботою. Коли очікувана операція завершується — потік (можливо, інший) повертається і продовжує виконання з того місця, де зупинився.

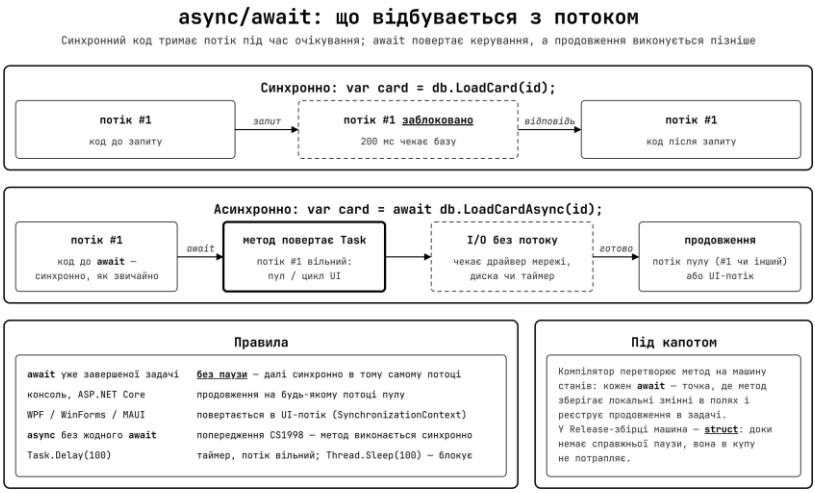
Ключові слова async та await

C# надає дві ключові синтаксичні конструкції для асинхронного програмування:

async — модифікатор, що додається до заголовку методу. Він повідомляє компілятору, що цей метод є асинхронним і може містити вирази await. Важливо: **сам по собі async не робить метод асинхронним**. Це лише дозволяє використовувати await усередині.

await — оператор, що застосовується до об'єкта Task або Task<T>. Він каже: «зупинись тут, передай управління назад викликаючому коду, і продовж виконання цього методу тоді, коли Task завершиться». await

може використовуватись **виключно всередині методу з модифікатором async**.



Модель виконання async/await: що відбувається з потоком

Ключова відмінність від блокування: при await поточний потік **не зависає в очікуванні** — метод повертає викликачу незавершену задачу, а потік іде займатися іншою роботою (повертається у пул або до циклу повідомлень UI). Саме очікування введення-виведення (мережа, диск, таймер) взагалі не займає жодного потоку — його відстежує операційна система. Коли операція завершується, продовження методу планується залежно від контексту: у консольних застосунках і ASP.NET Core — на будь-якому вільному потоці пулу (тому ж або іншому), а у WPF, WinForms і MAUI — знову в UI-потоці, завдяки SynchronizationContext (детальніше — у розділі 17.4).

Варто знати ще дві деталі. По-перше, код асинхронного методу **до першого await виконується синхронно** в потоці викликача — метод не «перескакує» в інший потік сам собою. По-друге, якщо задача, яку очікують, на момент await уже завершена, паузи не буде зовсім: виконання продовжиться далі синхронно, у тому самому потоці.

Оголошення асинхронного методу

Асинхронний метод визначається за такими ознаками:

— Модифікатор async у заголовку
— Хоча б один вираз await у тілі (без нього компілятор видасть попередження CS1998, і метод виконається повністю синхронно)
— Тип повернення: void, Task, Task<T> або ValueTask<T>
— За угодою — суфікс Async у назві

```C#
using System;
using System.Threading;
using System.Threading.Tasks;

// Виклик асинхронного методу
await LoadPatientCardAsync("PT-2024-001");
Console.WriteLine("[Main] Метод Main продовжує виконання після await");

// Асинхронний метод: async + Task + Async-суфікс
async Task LoadPatientCardAsync(string patientId)
{
    Console.WriteLine($"[LoadPatient] Запит до бази даних: {patientId}");
    await Task.Delay(500); // імітуємо мережевий запит (не блокує потік!)
    Console.WriteLine($"[LoadPatient] Картка {patientId} завантажена");
}
```

Task.Delay vs Thread.Sleep

Task.Delay — асинхронна затримка, обов'язково вживається з await. На відміну від Thread.Sleep, вона **не блокує потік** — потік звільняється під час очікування.

```C#
using System;
using System.Threading;
using System.Threading.Tasks;

Console.WriteLine("=== Thread.Sleep (блокує потік) ===");
Console.WriteLine($"До Sleep: потік
{Thread.CurrentThread.ManagedThreadId.ToString()}");
Thread.Sleep(100); // потік ЗАМОРОЖЕНО на 100мс
Console.WriteLine($"Після Sleep: потік
{Thread.CurrentThread.ManagedThreadId.ToString()}");

Console.WriteLine("\n=== Task.Delay (звільняє потік) ===");
Console.WriteLine($"До Delay: потік
{Thread.CurrentThread.ManagedThreadId.ToString()}");
await Task.Delay(100); // потік ЗВІЛЬНЕНИЙ під час очікування
Console.WriteLine($"Після Delay: потік
{Thread.CurrentThread.ManagedThreadId.ToString()}");
// ID може відрізнятися — продовжив інший потік з пулу!
```

Як async/await влаштований під капотом

Коли компілятор зустрічає метод з async, він перетворює його на **стейт-машину** (state machine) — клас, що зберігає стан виконання між точками await. Кожна точка await — це стан, де метод може «припинитись» і «продовжитись» пізніше.

Спрощено: оригінальний асинхронний метод:

```
C#
1  async Task DoWorkAsync()
2  {
3      Console.WriteLine("Крок 1");
4      await Task.Delay(200);
5      Console.WriteLine("Крок 2");
6      await Task.Delay(100);
7      Console.WriteLine("Крок 3");
8  }
```

Компілятор генерує стейт-машину (у Release-збірці — структуру, яка потрапляє в купу лише тоді, коли метод справді призупиняється) з трьома станами: State 0 (до першого await), State 1 (після першого await), State 2 (після другого await). При кожному await метод зберігає поточний стан і повертає управління; коли Task завершується, відновлення виконання планується з правильного стану. Розробнику не потрібно писати цей код вручну — async/await є синтаксичним цукром поверх цієї механіки.

Паралельний запуск асинхронних методів

Часта помилка: await кожного методу по черзі, коли методи насправді незалежні:

```
C#
 1  using System;
 2  using System.Threading.Tasks;
 3
 4  // НЕЕФЕКТИВНО: послідовний await — загальний час = сума всіх
 5  Console.WriteLine("=== Послідовно (неефективно) ===");
 6  var sw1 = System.Diagnostics.Stopwatch.StartNew();
 7  await RunTestAsync("Аналіз крові",   400);
 8  await RunTestAsync("Аналіз сечі",    250);
 9  await RunTestAsync("ЕКГ",            300);
10  sw1.Stop();
11  Console.WriteLine($"Послідовно: {sw1.ElapsedMilliseconds.ToString()} мс\n");
12
13  // ЕФЕКТИВНО: старт всіх → потім await — загальний час = максимум
14  Console.WriteLine("=== Паралельно (ефективно) ===");
15  var sw2 = System.Diagnostics.Stopwatch.StartNew();
16  var t1 = RunTestAsync("Аналіз крові",   400);
17  var t2 = RunTestAsync("Аналіз сечі",    250);
```

```
var t3 = RunTestAsync("ЕКГ",            300);
await t1; await t2; await t3;
sw2.Stop();
Console.WriteLine($"Паралельно: {sw2.ElapsedMilliseconds.ToString()} мс");

async Task RunTestAsync(string name, int ms)
{
    Console.WriteLine($"  → Початок: {name}");
    await Task.Delay(ms);
    Console.WriteLine($"  ✓ Готово:  {name} ({ms.ToString()} мс)");
}
```

Різниця суттєва: послідовно — ~950 мс, паралельно — ~400 мс. Правило: якщо задачі **незалежні** одна від одної — запускайте їх усі, потім очікуйте.

Асинхронні лямбда-вирази

Лямбди також можуть бути асинхронними — для цього перед параметрами ставиться ключове слово async. Тип такої лямбди — Func<Task>, Func<T, Task>, Func<Task<TResult>> тощо:

```
C#
using System;
using System.Threading.Tasks;

// Async-лямбда без параметрів: Func<Task>
Func<Task> sendAlert = async () =>
{
    Console.WriteLine("[Alert] Надсилаю критичне сповіщення...");
    await Task.Delay(200);
    Console.WriteLine("[Alert] Сповіщення надіслано персоналу");
};

// Async-лямбда з параметром: Func<string, Task>
Func<string, Task> notifyDoctor = async (doctorName) =>
{
    Console.WriteLine($"[Notify] Виклик лікаря: {doctorName}");
    await Task.Delay(150);
    Console.WriteLine($"[Notify] Лікар {doctorName} сповіщений");
};

// Async-лямбда що повертає результат: Func<string, Task<bool>>
Func<string, Task<bool>> checkAvailability = async (room) =>
{
    await Task.Delay(100);
    return room != "Операційна-1"; // операційна зайнята
};

await sendAlert();
await notifyDoctor("Петренко І.О.");
bool available = await checkAvailability("Палата-5");
Console.WriteLine($"Палата-5 вільна: {available.ToString()}");
```

Суфікс Async — угода іменування

Всі асинхронні методи в .NET мають суфікс Async у назві: ReadAsync, WriteAsync, GetAsync, SendAsync. Це угода, прийнята в усій екосистемі .NET, яка:

— Відрізняє асинхронну версію від синхронної (Read vs ReadAsync)

— Нагадує розробнику, що метод потребує await

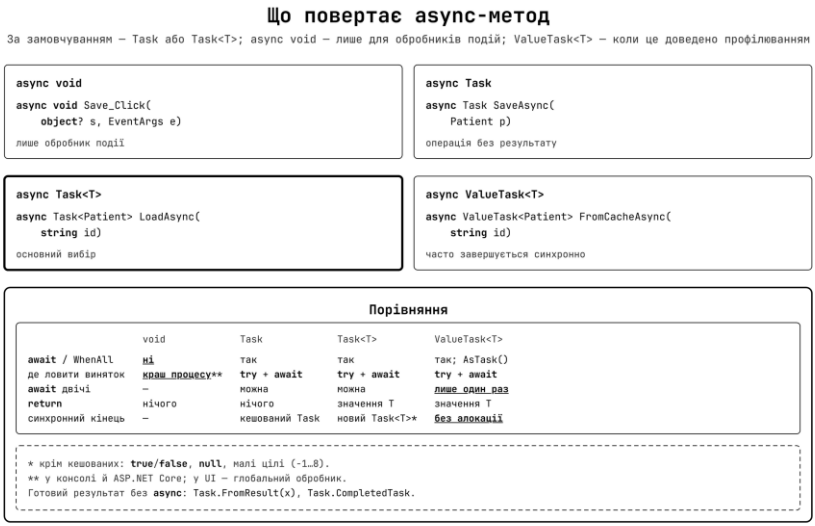
— Дозволяє надавати обидві версії в бібліотеках

```
C#
 1  using System;
 2  using System.Threading.Tasks;
 3
 4  // Синхронна версія
 5  string LoadPatientCard(string id)
 6  {
 7      System.Threading.Thread.Sleep(100); // блокує потік
 8      return $"Картка пацієнта {id}";
 9  }
10
11  // Асинхронна версія — суфікс Async
12  async Task<string> LoadPatientCardAsync(string id)
13  {
14      await Task.Delay(100); // не блокує
15      return $"Картка пацієнта {id}";
16  }
17
18  // Синхронний виклик
19  string card1 = LoadPatientCard("PT-001");
20  Console.WriteLine(card1);
21
22  // Асинхронний виклик
23  string card2 = await LoadPatientCardAsync("PT-002");
24  Console.WriteLine(card2);
```

Бібліотека .NET дотримується цієї угоди скрізь: File.ReadAllTextAsync, HttpClient.GetAsync, DbContext.SaveChangesAsync — якщо є суфікс Async, метод поверне Task і очікує await.

17.2. Типи повернення async-методів

Асинхронні методи в C# можуть повертати чотири типи: void, Task, Task<T> і ValueTask<T>. Вибір між ними — не питання смаку: кожен тип має своє призначення, обмеження та вплив на продуктивність. Правильний вибір типу повернення — ознака грамотного асинхронного коду.



Типи повернення async-методів

async void — тільки для обробників подій

async void — найнебезпечніший тип повернення. Метод нічого не повертає, тому викликаючий код не може ні отримати результат, ні дочекатися завершення, ні перехопити виняток через try/catch.

Єдиний законний сценарій використання async void — обробники подій, де підпис методу встановлений зовнішнім інтерфейсом і не може бути змінений:

```
C#
 1  using System;
 2  using System.Threading.Tasks;
 3
 4  // Симуляція обробника події — такий підпис вимагає система подій
 5  void SimulateButton_Click(object? sender, EventArgs e)
 6  {
 7      // Тут ми не можемо повернути Task, бо підпис встановлений делегатом
    EventHandler
 8      // Тому async void — єдиний варіант для обробників подій
 9      _ = HandleButtonClickAsync(); // запускаємо і не чекаємо (в реальному
    EventHandler це async void)
10  }
11
12  async Task HandleButtonClickAsync()
13  {
14      Console.WriteLine("[UI] Кнопка натиснута — початок реєстрації...");
15      await Task.Delay(300); // симуляція збереження до БД
```

```
16      Console.WriteLine("[UI] Пацієнта зареєстровано успішно");
17  }
18
19  // Демонстрація проблеми async void: його неможливо дочекатися
20  async void FireAndForget()
21  {
22      await Task.Delay(100);
23      Console.WriteLine("[async void] Збереження завершилось — але ніхто на це не
    чекав");
24  }
25
26  // Правильна версія — async Task, виняток можна перехопити
27  async Task SafeAsyncTask()
28  {
29      await Task.Delay(50);
30      throw new InvalidOperationException("Ця помилка перехоплюється через
    await");
31  }
32
33  // Демонстрація
34  Console.WriteLine("=== async void: дочекатися неможливо ===");
35  FireAndForget(); // повертає void — немає Task, нічого await-ити
36  Console.WriteLine("[Main] Іду далі, хоча збереження ще триває");
37  await Task.Delay(200); // лише «вгадуємо» час — справжнього способу дочекатися
    немає
38
39  Console.WriteLine("\n=== async Task: виняток перехоплюється ===");
40  try
41  {
42      await SafeAsyncTask(); // виняток поширюється через await
43  }
44  catch (InvalidOperationException ex)
45  {
46      Console.WriteLine($"Перехоплено: {ex.Message}");
47  }
```

З винятками все ще гірше. Виняток з async void-методу не потрапляє ні в try/catch навколо виклику (метод уже повернув керування), ні в жоден Task. Його повторно кидає той SynchronizationContext, у якому метод почав роботу: у WPF/WinForms — у циклі повідомлень UI-потоку (його можна перехопити лише глобальним обробником на кшталт Application.DispatcherUnhandledException), а в консольній програмі чи ASP.NET Core контексту немає, тож виняток кидається на потоці пулу як необроблений — і **процес аварійно завершується**. Тому приклад вище не кидає виняток з async void: інакше програма просто впала б.

```
C#
1  async void Dangerous()
2  {
3      await Task.Delay(50);
4      throw new InvalidOperationException("Збій збереження");
5  }
6
```

```
7  try { Dangerous(); }                // try/catch тут НЕ допоможе:
8  catch (Exception) { /* ніколи */ }  // у консолі весь процес завершиться
```

Ключове правило: **ніколи не використовуйте async void поза обробниками подій**. А в самих обробниках подій тіло варто повністю обгортати в try/catch. async void-метод не можна очікувати, не можна обробити його виняток ззовні, не можна перевірити його стан. Це «запустив і забув» у найгіршому сенсі.

async Task — операція без результату

async Task — стандартний тип повернення для асинхронних методів, що не повертають значення. На відміну від async void, Task дозволяє:

— очікувати завершення через await

— перехоплювати винятки через try/catch навколо await

— передавати завдання у Task.WhenAll та інші комбінатори

```
C#
 1  using System;
 2  using System.Threading.Tasks;
 3
 4  // async Task – повертає Task, але без значення всередині
 5  async Task SavePatientAsync(string patientId, string data)
 6  {
 7      Console.WriteLine($"[Save] Збереження даних пацієнта {patientId}...");
 8      await Task.Delay(200); // симуляція запиту до БД
 9      Console.WriteLine($"[Save] Дані {patientId} збережено успішно");
10      // return не потрібен – метод "повертає" Task, що сигналізує про завершення
11  }
12
13  async Task DeletePatientAsync(string patientId)
14  {
15      Console.WriteLine($"[Delete] Видалення пацієнта {patientId}...");
16      await Task.Delay(100);
17      // Симулюємо помилку – вона поширюється через Task
18      throw new UnauthorizedAccessException($"Пацієнт {patientId} захищений від
    видалення");
19  }
20
21  // Очікування завершення – викликаючий знає, коли операція завершилась
22  await SavePatientAsync("PT-001", "Анамнез: ...");
23
24  // Обробка помилок через try/catch навколо await
25  try
26  {
27      await DeletePatientAsync("PT-002");
28  }
29  catch (UnauthorizedAccessException ex)
30  {
31      Console.WriteLine($"[Main] Помилка: {ex.Message}");
32  }
33
```

```
34  Console.WriteLine("[Main] Продовжуємо роботу після обох операцій");
```

async Task завжди кращий за async void там, де підпис методу під вашим контролем. Поверніть Task — і ваш код стає тестованим, перехоплюваним і сумісним з усіма async-комбінаторами.

async Task<T> — операція з результатом

Task<T> — тип повернення для асинхронних методів, що обчислюють або отримують значення. Оператор await «розгортає» Task і повертає значення типу T. Це найпоширеніший тип для реальних асинхронних операцій:

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Threading.Tasks;
 4
 5  // async Task<string> — повертає рядок через Task
 6  async Task<string> GetPatientNameAsync(string id)
 7  {
 8      Console.WriteLine($"[DB] Запит імені для {id}...");
 9      await Task.Delay(150);
10      return $"Петренко І.О. (id={id})"; // return T, компілятор загортає у
    Task<T>
11  }
12
13  // async Task<List<string>> — повертає список
14  async Task<List<string>> GetPatientLabResultsAsync(string id)
15  {
16      Console.WriteLine($"[Lab] Завантаження результатів для {id}...");
17      await Task.Delay(200);
18      return new List<string>
19      {
20          "Гемоглобін: 135 г/л — норма",
21          "Лейкоцити: 6.2 × 10⁹/л — норма",
22          "ШОЕ: 8 мм/год — норма"
23      };
24  }
25
26  // async Task<bool> — повертає булеве значення
27  async Task<bool> CheckBedAvailabilityAsync(string ward)
28  {
29      await Task.Delay(100);
30      return ward != "BIT"; // ВІТ — переповнене
31  }
32
33  // Використання: await "розгортає" Task<T> до T
34  string name = await GetPatientNameAsync("PT-2024-007");
35  Console.WriteLine($"Пацієнт: {name}");
36
37  List<string> results = await GetPatientLabResultsAsync("PT-2024-007");
38  Console.WriteLine("\nРезультати аналізів:");
39  foreach (string r in results)
40      Console.WriteLine($"  {r}");
```

```
41
42  bool available = await CheckBedAvailabilityAsync("BIT");
43  Console.WriteLine($"\nBIT вільне: {available.ToString()}");
```

Зверніть на return у async Task<T>: ви повертаєте значення типу T, а не Task<T>. Компілятор автоматично «загортає» його у Task<T>. Це симетрично до того, як await «розгортає» Task<T> назад до T.

ValueTask<T> — для hot path без алокації

ValueTask<T> — оптимізований тип для сценаріїв, де метод **часто завершується синхронно** (без реального очікування). Різниця з Task<T>: якщо результат вже відомий і очікування не потрібне, ValueTask<T> не виділяє пам'яті у heap, оскільки є структурою (struct), а не класом.

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Threading.Tasks;
 4
 5  // Кеш результатів — більшість запитів повертатимуть синхронно
 6  Dictionary<string, string> _cache = new Dictionary<string, string>
 7  {
 8      ["PT-001"] = "Коваль М.А.",
 9      ["PT-002"] = "Бойко О.П."
10  };
11
12  // ValueTask<T>: якщо є в кеші — повертає синхронно (без алокації Task)
13  // якщо немає — виконує реальний асинхронний запит
14  async ValueTask<string> GetPatientFastAsync(string id)
15  {
16      if (_cache.TryGetValue(id, out string? cached))
17      {
18          // Синхронне повернення — ValueTask не виділяє пам'яті
19          Console.WriteLine($"[Cache] HIT: {id} → {cached}");
20          return cached;
21      }
22
23      // Асинхронний шлях — звернення до бази даних
24      Console.WriteLine($"[Cache] MISS: {id} — запит до БД...");
25      await Task.Delay(200);
26      string name = $"Невідомий пацієнт (id={id})";
27      _cache[id] = name;
28      return name;
29  }
30
31  // Використання: синтаксично ідентичне Task<T>
32  string p1 = await GetPatientFastAsync("PT-001"); // з кешу — синхронно
33  string p2 = await GetPatientFastAsync("PT-002"); // з кешу — синхронно
34  string p3 = await GetPatientFastAsync("PT-999"); // не в кеші — асинхронно
35
36  Console.WriteLine($"\nРезультати: {p1}, {p2}, {p3}");
```

ValueTask<T> слід використовувати лише коли є вимірювані докази, що алокація Task<T> є вузьким місцем (hot path, мільйони викликів). В усіх інших випадках — Task<T> є правильним вибором: він простіший, безпечніший і добре оптимізований у .NET.

Важливе обмеження: ValueTask<T> можна await-ати лише один раз. Якщо вам потрібно зберегти завдання у змінну і очікувати кілька разів — використовуйте Task<T>.

Task.FromResult і Task.CompletedTask — синхронні обгортки

Іноді потрібно повернути Task<T> або Task з методу, де фактична робота виконується синхронно. Найпоширеніший сценарій — реалізація інтерфейсу або тест-дублер:

```C#
using System;
using System.Threading.Tasks;

// Використання
IPatientRepository repo = new FakeRepository();
string name = await repo.GetNameAsync("PT-001");
Console.WriteLine($"Отримано: {name}");

await repo.SaveAsync("PT-001", "дані");
Console.WriteLine("[Main] Операції завершено");

// Інтерфейс репозиторію
interface IPatientRepository
{
    Task<string> GetNameAsync(string id);
    Task SaveAsync(string id, string data);
}

// Реальна реалізація — справді асинхронна (звертається до БД)
class DatabaseRepository : IPatientRepository
{
    public async Task<string> GetNameAsync(string id)
    {
        await Task.Delay(100); // реальний асинхронний запит
        return $"Пацієнт {id} з бази даних";
    }

    public async Task SaveAsync(string id, string data)
    {
        await Task.Delay(50);
        Console.WriteLine($"[DB] Збережено: {id}");
    }
}

// Фейкова реалізація для тестів — синхронна, але мусить відповідати інтерфейсу
class FakeRepository : IPatientRepository
{
    public Task<string> GetNameAsync(string id)
        => Task.FromResult($"Тестовий пацієнт {id}"); // синхронно, без async
```

```
40
41      public Task SaveAsync(string id, string data)
42      {
43          Console.WriteLine($"[Fake] Зберігаю (імітація): {id}");
44          return Task.CompletedTask; // Task, що вже завершений
45      }
46  }
```

Task.FromResult<T>(value) створює Task, що вже завершений зі значенням — без жодних алокацій стейт-машини. Task.CompletedTask — синглтон-Task для Task-методів без значення, що не вимагає реальної асинхронності. Обидва широко використовуються при реалізації інтерфейсів, заглушок і тест-дублерів.

Зведена таблиця типів повернення

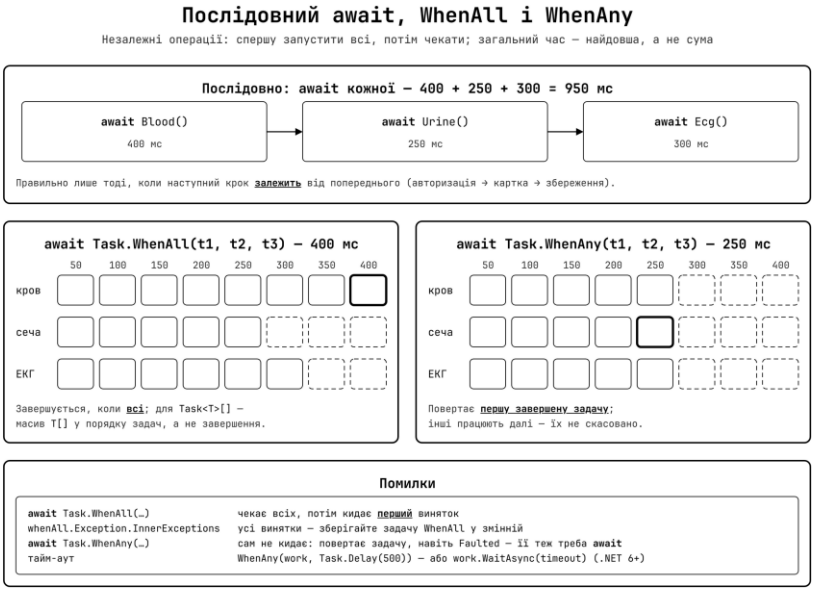
Тип	Коли використовувати	Awaitable	Перехоплення помилок	Алокація
async void	Тільки EventHandler	Ні	Ні: падіння процесу або глобальний обробник UI	Немає (Task)
async Task	Операція без результату	Так	Так	Task
async Task<T>	Операція з результатом	Так	Так	Task
async ValueTask<T>	Hot path з частим sync-шляхом	Так (1 раз)	Так	Немає (sync)

Правило вибору: за замовчуванням — Task або Task<T>. async void — лише для EventHandler. ValueTask<T> — тільки при підтвердженому профілюванням вузькому місці.

17.3. Послідовне та паралельне виконання. Task.WhenAll та Task.WhenAny

У реальних асинхронних системах рідко виникає потреба виконати лише одну операцію. Зазвичай потрібно координувати кілька: завантажити кілька ресурсів, дочекатися першого результату з набору

джерел, або збудувати складний конвеєр обробки. Для цього C# надає два потужні комбінатори: Task.WhenAll і Task.WhenAny.



Послідовне vs паралельне виконання в async

Послідовне виконання — коли порядок має значення

Якщо кожна наступна операція залежить від результату попередньої, виконання мусить бути послідовним. await кожного кроку по черзі — природній спосіб висловити цю залежність:

```
C#
 1  using System;
 2  using System.Threading.Tasks;
 3
 4  // Крок 1: авторизація → Крок 2: завантаження картки → Крок 3: збереження змін
 5  // Кожен крок залежить від попереднього — порядок обов'язковий
 6
 7  async Task<string> AuthorizeUserAsync(string login)
 8  {
 9      Console.WriteLine($"[Auth] Перевірка облікових даних: {login}");
10      await Task.Delay(100);
11      return $"token_{login}_2024";
12  }
13
14  async Task<string> LoadPatientCardAsync(string authToken, string patientId)
15  {
```

```
    Console.WriteLine($"[Load] Завантаження картки {patientId} (token: {authToken[..8]}...)");
    await Task.Delay(200);
    return $"Карта {patientId}: Петренко І.О., 45р, Діагноз: J06.9";
}

async Task SaveDiagnosisAsync(string authToken, string cardData, string diagnosis)
{
    Console.WriteLine($"[Save] Збереження діагнозу для: {cardData[..20]}...");
    await Task.Delay(150);
    Console.WriteLine($"[Save] Діагноз '{diagnosis}' збережено успішно");
}

var sw = System.Diagnostics.Stopwatch.StartNew();

// Послідовно — крок за кроком, кожен чекає попереднього
string token    = await AuthorizeUserAsync("dr.petrenko");
string card     = await LoadPatientCardAsync(token, "PT-2024-007");
await SaveDiagnosisAsync(token, card, "J06.9 — ГРВІ");

sw.Stop();
Console.WriteLine($"\nЗагальний час: {sw.ElapsedMilliseconds.ToString()} мс (послідовно)");
```

Послідовне виконання — правильний вибір, коли операції пов'язані ланцюгом залежностей. Загальний час дорівнює сумі часів усіх кроків.

Паралельне виконання — коли операції незалежні

Якщо кілька операцій не залежать одна від одної, їх можна запустити одночасно. Ключова ідея: **не await кожну операцію одразу — спочатку запусти всі, потім очікуй всіх**:

```
C#
using System;
using System.Threading.Tasks;

async Task<string> RunLabTestAsync(string testName, int durationMs)
{
    Console.WriteLine($"  → Розпочато: {testName}");
    await Task.Delay(durationMs);
    string result = $"{testName}: норма";
    Console.WriteLine($"  ✓ Завершено: {result} ({durationMs.ToString()} мс)");
    return result;
}

Console.WriteLine("=== Послідовно ===");
var sw1 = System.Diagnostics.Stopwatch.StartNew();
string r1 = await RunLabTestAsync("Аналіз крові",  400);
string r2 = await RunLabTestAsync("Аналіз сечі",   250);
string r3 = await RunLabTestAsync("ЕКГ",           300);
sw1.Stop();
```

```
Console.WriteLine($"Послідовно: {sw1.ElapsedMilliseconds.ToString()} мс\n");

Console.WriteLine("=== Паралельно (запускаємо всі, потім очікуємо) ===");
var sw2 = System.Diagnostics.Stopwatch.StartNew();
Task<string> t1 = RunLabTestAsync("Аналіз крові",  400); // запуск — не await
Task<string> t2 = RunLabTestAsync("Аналіз сечі",   250); // запуск — не await
Task<string> t3 = RunLabTestAsync("ЕКГ",           300); // запуск — не await

// Тепер очікуємо всі три
string res1 = await t1;
string res2 = await t2;
string res3 = await t3;
sw2.Stop();
Console.WriteLine($"Паралельно: {sw2.ElapsedMilliseconds.ToString()} мс");
Console.WriteLine($"Виграш: ~{(sw1.ElapsedMilliseconds -
sw2.ElapsedMilliseconds).ToString()} мс");
```

Загальний час паралельного виконання — час найдовшої операції (~400 мс замість ~950 мс). Звісно, це справедливо тільки для справді незалежних операцій.

Task.WhenAll — очікування всіх завдань

Task.WhenAll — найчистіший спосіб запустити кілька завдань паралельно і дочекатися їх усіх. Він приймає колекцію Task або масив Task<T> і повертає Task, що завершується, коли завершаться **всі** передані завдання:

```
C#
using System;
using System.Threading.Tasks;

async Task<double> MeasureGlucoseAsync()
{
    await Task.Delay(300);
    return 5.4; // ммоль/л
}

async Task<int> MeasurePulseAsync()
{
    await Task.Delay(200);
    return 72; // уд/хв
}

async Task<string> GetPatientStatusAsync()
{
    await Task.Delay(250);
    return "Стабільний";
}

var sw = System.Diagnostics.Stopwatch.StartNew();

// Task.WhenAll з Task<T> різних типів — запускаємо всі одразу
Task<double> glucoseTask = MeasureGlucoseAsync();
Task<int>    pulseTask   = MeasurePulseAsync();
```

```
Task<string> statusTask  = GetPatientStatusAsync();

await Task.WhenAll(glucoseTask, pulseTask, statusTask);

sw.Stop();

// Результати доступні через .Result (Task вже завершений — Result не блокує)
Console.WriteLine($"Глюкоза:  {glucoseTask.Result.ToString()} ммоль/л");
Console.WriteLine($"Пульс:    {pulseTask.Result.ToString()} уд/хв");
Console.WriteLine($"Статус:   {statusTask.Result}");
Console.WriteLine($"Час:      {sw.ElapsedMilliseconds.ToString()} мс
(паралельно)");
```

Task.WhenAll з масивом однотипних завдань

Якщо всі завдання мають однаковий тип Task<T>, WhenAll повертає Task<T[]> — масив результатів у порядку вихідних завдань:

```
C#
using System;
using System.Linq;
using System.Threading.Tasks;

async Task<string> GetPatientHistoryAsync(string id)
{
    await Task.Delay(100 + id.GetHashCode() % 100);
    return $"Анамнез {id}: без особливостей";
}

string[] patientIds = { "PT-001", "PT-002", "PT-003", "PT-004", "PT-005" };

// Запускаємо завдання для всіх пацієнтів одночасно
Task<string>[] tasks = patientIds
    .Select(id => GetPatientHistoryAsync(id))
    .ToArray();

// WhenAll з Task<T>[] повертає Task<T[]>
string[] histories = await Task.WhenAll(tasks);

Console.WriteLine("Анамнези всіх пацієнтів:");
foreach (string h in histories)
    Console.WriteLine($"  {h}");
```

Обробка помилок у Task.WhenAll

Task.WhenAll дочікується **всіх** завдань, навіть якщо деякі завершились з помилкою. Після завершення він кидає AggregateException, що містить всі помилки від усіх завдань, що завершились невдало:

```
C#
using System;
using System.Threading.Tasks;

async Task<string> FetchDataAsync(string source, bool willFail)
{
    await Task.Delay(100);
```

```
    if (willFail) throw new Exception($"Сервер '{source}' недоступний");
    return $"Дані з {source}";
}

Task<string> t1 = FetchDataAsync("Лабораторія",  willFail: false);
Task<string> t2 = FetchDataAsync("Рентген",       willFail: true);  // помилка
Task<string> t3 = FetchDataAsync("ЕКГ-система",  willFail: false);
Task<string> t4 = FetchDataAsync("Аптека",        willFail: true);  // помилка

try
{
    string[] results = await Task.WhenAll(t1, t2, t3, t4);
    // Цей рядок НЕ виконається — є помилки
}
catch (Exception ex)
{
    // await розгортає першу помилку з AggregateException
    Console.WriteLine($"Перша помилка: {ex.Message}");
}

// Щоб отримати ВСІ помилки — аналізуємо Task.Exception після WhenAll
Console.WriteLine("\nВсі помилки:");
foreach (Task<string> t in new[] { t1, t2, t3, t4 })
{
    if (t.IsFaulted)
        Console.WriteLine($"  ✗ {t.Exception?.InnerException?.Message}");
    else if (t.IsCompletedSuccessfully)
        Console.WriteLine($"  ✓ {t.Result}");
}
```

Важлива деталь: await Task.WhenAll(...) при помилці розгортає лише перший з винятків. Щоб отримати всі помилки, достатньо зберегти саму задачу, яку повертає WhenAll, у змінну: її властивість Exception.InnerExceptions містить винятки **всіх** завдань, що завершились невдало (або можна перевірити Exception кожного завдання окремо).

Task.WhenAny — очікування першого завдання

Task.WhenAny завершується, щойно **перше** з переданих завдань завершиться (успішно або з помилкою). Це корисно для сценаріїв: «отримати відповідь від найшвидшого сервера», «реалізувати тайм-аут», «обробляти результати по мірі готовності»:

```
C#
using System;
using System.Threading.Tasks;

async Task<string> QueryServerAsync(string server, int delay)
{
    Console.WriteLine($"  → Запит до {server}...");
    await Task.Delay(delay);
    Console.WriteLine($"  ✓ Відповідь від {server} ({delay.ToString()} мс)");
```

```
    return $"Дані від {server}";
}

// Надсилаємо запит до трьох серверів одночасно — беремо найшвидшу відповідь
Task<string> server1 = QueryServerAsync("Київ",    500);
Task<string> server2 = QueryServerAsync("Харків",  200); // найшвидший
Task<string> server3 = QueryServerAsync("Львів",   350);

Task<string> fastest = await Task.WhenAny(server1, server2, server3);
Console.WriteLine($"\nПерша відповідь: {await fastest}");
Console.WriteLine("(Інші запити продовжуються у фоні)");
```

Тайм-аут через Task.WhenAny + Task.Delay

Один з найпоширеніших патернів з WhenAny — реалізація тайм-ауту для асинхронної операції:

```
C#
using System;
using System.Threading.Tasks;

async Task<string> SlowDatabaseQueryAsync()
{
    Console.WriteLine("[DB] Виконую складний запит...");
    await Task.Delay(800); // запит займає 800мс
    return "Результат запиту";
}

int timeoutMs = 500; // тайм-аут 500мс

Task<string> queryTask   = SlowDatabaseQueryAsync();
Task         timeoutTask = Task.Delay(timeoutMs);

Task completed = await Task.WhenAny(queryTask, timeoutTask);

if (completed == timeoutTask)
{
    Console.WriteLine($"[Main] Тайм-аут! Запит перевищив {timeoutMs.ToString()}
мс");
    Console.WriteLine("[Main] Використовую кешовані дані");
}
else
{
    string result = await queryTask; // queryTask вже завершений — await не
блокує
    Console.WriteLine($"[Main] Отримано вчасно: {result}");
}
```

Зверніть: Task.WhenAny не скасовує інші завдання — SlowDatabaseQueryAsync продовжить виконуватись у фоні навіть після тайм-ауту. Щоб коректно скасувати, потрібно CancellationToken (розглядається у розділі 17.5). Починаючи з .NET 6, для тайм-ауту є і готовий метод task.WaitAsync(TimeSpan): він кидає TimeoutException, якщо задача не встигла (але саму задачу теж не скасовує).

Ще одна особливість WhenAny: сам він ніколи не кидає винятку — навіть якщо першою завершилась задача з помилкою, WhenAny просто поверне її. Щоб помилка проявилась, повернуту задачу треба ще раз очікувати: await await Task.WhenAny(...) або await winner.

Обробка результатів по мірі готовності

WhenAny дозволяє обробляти результати одразу, як вони з'являються, без очікування всіх:

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Threading.Tasks;
 4
 5  async Task<string> ProcessSampleAsync(int sampleId, int ms)
 6  {
 7      await Task.Delay(ms);
 8      return $"Зразок #{sampleId.ToString()}: аналіз завершено ({ms.ToString()}
    мс)";
 9  }
10
11  // Запускаємо аналіз 5 зразків паралельно
12  List<Task<string>> pending = new List<Task<string>>
13  {
14      ProcessSampleAsync(1, 400),
15      ProcessSampleAsync(2, 100),
16      ProcessSampleAsync(3, 250),
17      ProcessSampleAsync(4, 180),
18      ProcessSampleAsync(5, 320)
19  };
20
21  Console.WriteLine("Обробляємо результати по мірі готовності:");
22  while (pending.Count > 0)
23  {
24      Task<string> done = await Task.WhenAny(pending);
25      pending.Remove(done);
26      Console.WriteLine($"  ✓ {await done}");
27  }
28  Console.WriteLine("Всі зразки оброблено");
```

Цей патерн дозволяє одразу реагувати на перші результати, не чекаючи найповільніших — важливо для інтерактивних систем.

Порівняння: WhenAll vs WhenAny

Аспект	Task.WhenAll	Task.WhenAny
Завершується коли	Всі задачі завершились	Перша задача завершилась

Аспект	Task.WhenAll	Task.WhenAny
Результат	T[] (для Task<T>[])	Перша завершена Task<T>
Помилки	Чекає всіх; await кидає перший, усі — у Exception задачі WhenAll	Сам не кидає; помилку дає await повернутої задачі
Типові сценарії	Паралельне збирання даних	Тайм-аут, перший відповів, обробка по черзі
Скасовує інші	Ні	Ні

Обидва методи є фундаментальними інструментами координації async-операцій. Їх комбінація дозволяє будувати складні стратегії паралельного виконання без явного управління потоками.

17.4. Обробка помилок в async-методах

Асинхронний код вимагає особливої уваги до обробки помилок. Виняток, кинутий у асинхронному методі, не поширюється миттєво — він «консервується» у Task і чекає, поки хтось зробить await. Якщо await ніколи не відбудеться, виняток може бути втрачений назавжди або з'явитися в непередбаченому місці.



Поширення помилок в async-методах

Базова обробка через try/catch навколо await

Найпростіший і найправильніший підхід: обгорніть await-вираз у try/catch. Компілятор гарантує, що виняток із завдання буде «розгорнутий» і кинутий у точці await:

```
C#
 1  using System;
 2  using System.Threading.Tasks;
 3
 4  async Task<string> LoadPatientDataAsync(string id)
 5  {
 6      await Task.Delay(100);
 7      if (id == "DELETED")
 8          throw new InvalidOperationException($"Пацієнт {id} видалений з
    системи");
 9      return $"Дані пацієнта {id}: Петренко І.О., 45р";
10  }
11
12  // Правильний підхід: try/catch навколо await
13  Console.WriteLine("=== Спроба 1: неіснуючий пацієнт ===");
14  try
15  {
16      string data = await LoadPatientDataAsync("DELETED");
17      Console.WriteLine(data);
18  }
19  catch (InvalidOperationException ex)
20  {
```

```
    Console.WriteLine($"[Catch] Помилка доступу: {ex.Message}");
}

Console.WriteLine("\n=== Спроба 2: звичайний пацієнт ===");
try
{
    string data = await LoadPatientDataAsync("PT-001");
    Console.WriteLine($"[OK] {data}");
}
catch (InvalidOperationException ex)
{
    Console.WriteLine($"[Catch] {ex.Message}");
}
```

try/catch навколо await перехоплює виняток точно так само, як при синхронному виклику. Це одна з головних переваг async/await над старішими підходами (callback, ContinueWith) — обробка помилок виглядає природньо.

async void і загублені винятки

Найнебезпечніша помилка в асинхронному коді — виняток у методі async void. Оскільки async void не повертає Task, кидати й перехоплювати виняток нема де: він потрапляє безпосередньо у SynchronizationContext або у ThreadPool, що призводить до аварійного завершення застосунку:

```
C#
using System;
using System.Threading.Tasks;

// НЕБЕЗПЕЧНО: async void — якби тут був throw, try/catch ззовні не допоміг би
async void DangerousFireAndForget()
{
    await Task.Delay(50);
    Console.WriteLine("[Dangerous] У цей момент Main уже вийшов з try — ловити нема кому");
    // throw new Exception("..."); — у консолі такий виняток аварійно завершив би процес
}

// ПРАВИЛЬНО: async Task — виняток «зберігається» у Task і розгортається при await
async Task SafeOperation()
{
    await Task.Delay(50);
    throw new Exception("Ця помилка перехоплюється при await");
}

// Демонстрація: async Task — виняток перехоплюється нормально
Console.WriteLine("=== async Task: виняток перехоплюється ===");
try
{
    await SafeOperation();
```

```
}
catch (Exception ex)
{
    Console.WriteLine($"[Catch] Перехоплено: {ex.Message}");
}

// Демонстрація: async void — try/catch навколо виклику НЕ допомагає
Console.WriteLine("\n=== async void: try/catch НЕ допомагає ===");
try
{
    DangerousFireAndForget(); // повертає керування одразу — до того, як метод щось зробить
    Console.WriteLine("[Main] Вийшли з виклику, метод ще працює");
}
catch (Exception ex)
{
    Console.WriteLine($"Цей catch не побачив би винятку з async void: {ex.Message}");
}
await Task.Delay(200);

Console.WriteLine("Виняток з async void пройшов би повз catch — у консолі й ASP.NET Core це падіння процесу");
```

Чому приклад не кидає виняток насправді? Тому що в консольному застосунку немає SynchronizationContext: виняток з async void повторно кидається на потоці пулу як необроблений, і процес аварійно завершується, — до наступного рядка програма просто не дійшла б. У WPF/WinForms такий виняток потрапляє в цикл повідомлень UI і завершує застосунок, якщо його не перехопить глобальний обробник.

Правило абсолютне: **ніколи не використовуйте async void поза обробниками подій**. Якщо вам потрібна операція «запустив і забув» без очікування — збережіть Task у змінну (і переконайтесь, що виняток десь оброблюється):

```C#
using System;
using System.Threading.Tasks;

async Task SendAlertAsync(string message)
{
    await Task.Delay(100);
    Console.WriteLine($"[Alert] {message}");
    // Якщо тут кинути виняток — він залишиться у Task без обробки
}

// "Запустив і забув" — правильний спосіб
Task alertTask = SendAlertAsync("Критичне сповіщення персоналу");
// alertTask не awaited — якщо виникне виняток, він буде у Task.Exception
// Якщо потрібна обробка помилок — зберігай Task і аналізуй пізніше

await Task.Delay(200); // даємо час завершитись
Console.WriteLine($"Alert завершено: {alertTask.IsCompletedSuccessfully.ToString()}");
```

Обробка кількох помилок з Task.WhenAll

Коли Task.WhenAll чекає кількох завдань, і деякі з них завершуються з помилкою, await Task.WhenAll(...) розгортає лише **першу** помилку. Інші помилки залишаються у AggregateException, доступній через task.Exception:

```
C#
 1  using System;
 2  using System.Threading.Tasks;
 3
 4  async Task<string> FetchFromSystemAsync(string system, bool fail)
 5  {
 6      await Task.Delay(100);
 7      if (fail) throw new Exception($"Система '{system}' не відповідає");
 8      return $"Дані з {system}";
 9  }
10
11  Task<string> lab    = FetchFromSystemAsync("Лабораторія",   fail: false);
12  Task<string> xray   = FetchFromSystemAsync("Рентген",       fail: true);  //
    помилка
13  Task<string> ecg    = FetchFromSystemAsync("ЕКГ",           fail: false);
14  Task<string> pharma = FetchFromSystemAsync("Аптека",        fail: true);  //
    помилка
15
16  Task<string[]> allTasks = Task.WhenAll(lab, xray, ecg, pharma);
17
18  try
19  {
20      string[] results = await allTasks;
21  }
22  catch (Exception ex)
23  {
24      // await розгортає ТІЛЬКИ першу помилку
25      Console.WriteLine($"Перша помилка: {ex.Message}");
26  }
27
28  // Щоб отримати ВСІ помилки — перевіряємо AggregateException у Task.Exception
29  if (allTasks.Exception != null)
30  {
31      Console.WriteLine($"\nВсі помилки
    ({allTasks.Exception.InnerExceptions.Count.ToString()}):");
32      foreach (Exception inner in allTasks.Exception.InnerExceptions)
33          Console.WriteLine($"  ✗ {inner.Message}");
34  }
35
36  // Успішні результати доступні через окремі Task
37  Console.WriteLine("\nУспішні відповіді:");
38  foreach (Task<string> t in new[] { lab, xray, ecg, pharma })
39      if (t.IsCompletedSuccessfully)
40          Console.WriteLine($"  ✓ {t.Result}");
```

Паттерн: зберігати посилання на Task<T[]>, що повертає WhenAll, і перевіряти task.Exception.InnerExceptions для повного списку помилок.

finally у async-методах

finally-блок у async-методах працює так само, як у синхронних: виконується незалежно від того, успішно чи з помилкою завершився try-блок. Це важливо для гарантованого звільнення ресурсів:

```C#
using System;
using System.Threading.Tasks;

async Task ProcessPatientAsync(string patientId)
{
    Console.WriteLine($"[Process] Відкриваю з'єднання для {patientId}");
    try
    {
        Console.WriteLine($"[Process] Обробляю дані...");
        await Task.Delay(100);

        if (patientId == "ERROR")
            throw new Exception("Помилка обробки даних");

        Console.WriteLine($"[Process] Дані {patientId} оброблено");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"[Catch] Помилка: {ex.Message}");
        // Перекидаємо виняток далі
        throw;
    }
    finally
    {
        // Цей блок ЗАВЖДИ виконається — навіть при throw
        Console.WriteLine($"[Finally] Закриваю з'єднання для {patientId}");
    }
}

// Успішний випадок
try
{
    await ProcessPatientAsync("PT-001");
}
catch { }

Console.WriteLine("");

// Випадок з помилкою
try
{
    await ProcessPatientAsync("ERROR");
}
catch (Exception ex)
{
    Console.WriteLine($"[Main] Перехоплено: {ex.Message}");
}
```

await можна використовувати і в catch, і в finally-блоках (починаючи з C# 6). Це дозволяє виконувати асинхронне очищення ресурсів:

```
C#
 1  using System;
 2  using System.Threading.Tasks;
 3
 4  async Task PerformOperationAsync()
 5  {
 6      try
 7      {
 8          await Task.Delay(50);
 9          throw new Exception("Щось пішло не так");
10      }
11      catch (Exception ex)
12      {
13          // await у catch — дозволено з C# 6
14          Console.WriteLine($"[Catch] Логую помилку асинхронно: {ex.Message}");
15          await Task.Delay(30); // асинхронне логування
16          Console.WriteLine("[Catch] Помилку залоговано");
17      }
18      finally
19      {
20          // await у finally — теж дозволено
21          Console.WriteLine("[Finally] Асинхронне закриття ресурсів...");
22          await Task.Delay(20);
23          Console.WriteLine("[Finally] Ресурси звільнено");
24      }
25  }
26
27  await PerformOperationAsync();
28  Console.WriteLine("[Main] Операція завершена");
```

Глобальна обробка необроблених виключень

У продакшн-застосунках важливо передбачити глобальний обробник для необроблених async-виключень. Для консольних застосунків та ASP.NET Core це AppDomain.CurrentDomain.UnhandledException та TaskScheduler.UnobservedTaskException:

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  // Підписуємось на необроблені Task-виключення
 6  TaskScheduler.UnobservedTaskException += (sender, args) =>
 7  {
 8      Console.WriteLine($"[Global] Необроблений виняток у Task:
    {args.Exception.InnerException?.Message}");
 9      args.SetObserved(); // позначаємо як оброблений (з .NET 4.5 процес і так не
    падає)
10  };
11
```

```
// Запускаємо Task без await — виняток залишиться необробленим
Task.Run(async () =>
{
    await Task.Delay(50);
    throw new Exception("Виняток без await");
});

// Чекаємо і примусово збираємо сміття для тригеру UnobservedTaskException
await Task.Delay(200);
GC.Collect();
GC.WaitForPendingFinalizers();

Console.WriteLine("[Main] Продовжую роботу після глобального перехоплення");
```

UnobservedTaskException спрацьовує, коли Task з необробленим винятком збирається GC (тому момент спрацювання непередбачуваний і в прикладі його доводиться провокувати GC.Collect()). Починаючи з .NET Framework 4.5, «забутий» виняток задачі за замовчуванням **не** завершує процес — подія лише дає змогу його залогувати; SetObserved() позначає виняток як оброблений. Натомість AppDomain.CurrentDomain.UnhandledException дозволяє лише залогувати необроблений виняток потоку (зокрема з async void) — скасувати падіння процесу він не може. Це остання лінія захисту, але покладатись на неї як на основний механізм обробки помилок — погана практика.

Типові помилки та їх рішення

1. Забув await — виняток загублений:

```
C#
using System;
using System.Threading.Tasks;

async Task ThrowAsync()
{
    await Task.Delay(10);
    throw new Exception("Помилка!");
}

// НЕПРАВИЛЬНО: без await виняток загублений у Task
Task t = ThrowAsync(); // виняток є, але ніхто не перевірить t.Exception

await Task.Delay(100);
Console.WriteLine($"Task завершений: {t.IsCompleted.ToString()}, Faulted:
{t.IsFaulted.ToString()}");
Console.WriteLine($"Виняток: {t.Exception?.InnerException?.Message ??
"null"}");
// Виняток існує у Task, але не "вистрілив" назовні
```

2. .Result або .Wait() у async-контексті — дедлок:

Виклик .Result або .Wait() у async-контексті з SynchronizationContext (ASP.NET Classic, WinForms, WPF) призводить до дедлоку: await чекає потоку UI, а потік UI заблокований на .Result. Вирішення — завжди використовувати await:

```
C#
 1  using System;
 2  using System.Threading.Tasks;
 3
 4  async Task<string> GetDataAsync()
 5  {
 6      await Task.Delay(100);
 7      return "дані";
 8  }
 9
10  // БЕЗПЕЧНО в консольному застосунку, але ДЕДЛОК у WinForms/WPF/ASP.NET
    Classic:
11  // string data = GetDataAsync().Result; // НЕБЕЗПЕЧНО
12
13  // ПРАВИЛЬНО — завжди:
14  string data = await GetDataAsync();
15  Console.WriteLine($"Отримано: {data}");
```

Загальне правило: **async вниз по стеку до кінця**. Якщо один метод async — всі методи, що його викликають, теж мають бути async. Змішування синхронного і асинхронного коду через .Result є джерелом найважчих для діагностики дедлоків.

ConfigureAwait(false) та SynchronizationContext

Щоб зрозуміти ConfigureAwait(false), потрібно спочатку зрозуміти **SynchronizationContext**. Це механізм, що дозволяє продовженню (continuation) після await повернутися до «правильного» потоку:

- У **WinForms/WPF**: після await виконання продовжується у **потоці UI** — щоб можна було оновити елементи форми
- У **ASP.NET Classic**: після await виконання повертається до **того самого HTTP-потоку**
- У **ASP.NET Core** і **консольних** застосунках: SynchronizationContext відсутній, await продовжується у довільному ThreadPool-потоці

Саме через SynchronizationContext і виникає класичний дедлок з .Result:

```
1  1. Потік UI (або HTTP-потік) викликає GetDataAsync().Result
2  2. GetDataAsync() всередині робить await Task.Delay(100)
3  3. Task.Delay завершується — continuation чекає захоплення потоку UI
4  4. Але потік UI заблокований на .Result!
5  5. Дедлок: потік UI чекає Task, Task чекає потік UI
```

ConfigureAwait(false) каже await: «не намагайся повертатись на той самий SynchronizationContext — продовжуй у ThreadPool-потоці». Це ламає дедлок і підвищує продуктивність у бібліотечному коді:

```C#
using System;
using System.Threading;
using System.Threading.Tasks;

async Task<string> LibraryMethodAsync()
{
    // ConfigureAwait(false): після await продовжуємо в ThreadPool, не на UI-потоці
    await Task.Delay(100).ConfigureAwait(false);

    // Ця частина виконується в ThreadPool — НЕ чіпати UI-елементи тут!
    Console.WriteLine($"[Library] Потік: {Thread.CurrentThread.ManagedThreadId} (ThreadPool)");

    return "результат";
}

async Task AppMethodAsync()
{
    // У застосунку: await БЕЗ ConfigureAwait(false) — повертаємось на UI-потік
    // (у консолі SynchronizationContext == null, тому різниці немає)
    string result = await LibraryMethodAsync();

    Console.WriteLine($"[App] Потік: {Thread.CurrentThread.ManagedThreadId}");
    Console.WriteLine($"[App] Результат: {result}");
}

Console.WriteLine($"[Main] Стартовий потік: {Thread.CurrentThread.ManagedThreadId}");
await AppMethodAsync();
```

Правило використання ConfigureAwait(false):

Де	Рекомендація
Бібліотечний код (NuGet-пакети, спільні сервіси)	**Завжди** ConfigureAwait(false) — бібліотека не знає, де її використають
Код застосунку (UI, контролери)	**Без** ConfigureAwait(false) — потрібно оновлювати UI або залишатись у контексті HTTP-запиту
ASP.NET Core	Можна не турбуватись — SynchronizationContext відсутній, ConfigureAwait(false) нічого не змінює

```C#
using System;
using System.Threading.Tasks;

// Бібліотечний метод — ConfigureAwait(false) скрізь
async Task<byte[]> ReadMedicalFileAsync(string path)
{
    // Симуляція читання файлу
    await Task.Delay(50).ConfigureAwait(false);
    await Task.Delay(30).ConfigureAwait(false);

    // Повертаємо результат — caller сам вирішить, що з ним робити
    byte[] fakeData = { 0x50, 0x44, 0x46 }; // "PDF" header
    Console.WriteLine("[Lib] Файл зчитано");
    return fakeData;
}

// Код застосунку — await без ConfigureAwait(false)
async Task HandlePatientDocumentAsync(string patientId)
{
    Console.WriteLine($"[App] Читаю документ для {patientId}...");

    byte[] content = await ReadMedicalFileAsync($"/records/{patientId}.pdf");

    // Тут ми в правильному контексті — можна оновити UI або зберегти в БД
    Console.WriteLine($"[App] Документ отримано, розмір: {content.Length}
байт");
}

await HandlePatientDocumentAsync("PT-1001");

```

17.5. Скасування async-операцій. CancellationToken

Скасування — невід'ємна частина асинхронного програмування. Користувач натискає «Скасувати», сплив тайм-аут, програма завершує роботу — у всіх цих випадках потрібно коректно зупинити async-операцію, що виконується. CancellationToken у async-контексті працює за тим самим кооперативним принципом, що і у TPL (розділ 16.5), але має ряд async-специфічних особливостей.



Скасування async-операцій через CancellationToken

CancellationToken у async-методах

Async-метод приймає CancellationToken як параметр — за угодою, останній або передостанній. Токен передається у вбудовані async-методи (.NET BCL весь побудований на цій угоді), а також перевіряється вручну між операціями:

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  // Async-метод з CancellationToken: прийнятий останнім параметром
 6  async Task<string> LoadMedicalRecordAsync(string patientId, CancellationToken
    ct = default)
 7  {
 8      Console.WriteLine($"[Load] Запит картки {patientId}...");
 9
10      // Передаємо token у Task.Delay — він підтримує скасування нативно
11      await Task.Delay(300, ct); // кине OperationCanceledException при Cancel()
12
13      Console.WriteLine($"[Load] Картка {patientId} завантажена");
14      return $"Картка: Петренко І.О. (id={patientId})";
15  }
16
17  // Демонстрація успішного завершення
18  Console.WriteLine("=== Успішне завантаження ===");
19  using CancellationTokenSource cts1 = new CancellationTokenSource();
20  string record = await LoadMedicalRecordAsync("PT-001", cts1.Token);
21  Console.WriteLine($"Результат: {record}");
22
```

```
// Демонстрація скасування
Console.WriteLine("\n=== Скасування під час завантаження ===");
using CancellationTokenSource cts2 = new CancellationTokenSource();
cts2.CancelAfter(150); // скасуємо через 150мс (до завершення 300мс)

try
{
    string cancelled = await LoadMedicalRecordAsync("PT-002", cts2.Token);
    Console.WriteLine($"Результат: {cancelled}");
}
catch (OperationCanceledException)
{
    Console.WriteLine("[Main] Завантаження скасовано (тайм-аут або запит користувача)");
}
```

Task.Delay(ms, token) — async-версія затримки зі скасуванням. Коли токен скасовується, Task.Delay кидає TaskCanceledException (нащадок OperationCanceledException) одразу, не чекаючи завершення таймера. Тому в catch краще вказувати базовий OperationCanceledException — він перехопить обидва.

Перевірка токену між операціями

У довгих async-методах, що виконують кілька кроків, важливо перевіряти токен між кроками — щоб не починати нову операцію після сигналу скасування:

C#

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

async Task ProcessPatientBatchAsync(List<string> patientIds, CancellationToken ct)
{
    Console.WriteLine($"[Batch] Початок обробки {patientIds.Count.ToString()} пацієнтів");

    for (int i = 0; i < patientIds.Count; i++)
    {
        // Перевіряємо перед кожною операцією
        ct.ThrowIfCancellationRequested();

        string id = patientIds[i];
        Console.WriteLine($"[Batch] Обробка {i + 1}/{patientIds.Count.ToString()}: {id}");

        await Task.Delay(150, ct); // симуляція async-операції з токеном

        Console.WriteLine($"[Batch] {id} — оброблено");
    }

```

```
23      Console.WriteLine("[Batch] Всі пацієнти оброблені");
24  }
25
26  List<string> patients = new List<string> { "PT-001", "PT-002", "PT-003", "PT-
    004", "PT-005" };
27
28  using CancellationTokenSource cts = new CancellationTokenSource();
29  cts.CancelAfter(400); // скасуємо через 400мс (після ~2-3 пацієнтів)
30
31  try
32  {
33      await ProcessPatientBatchAsync(patients, cts.Token);
34  }
35  catch (OperationCanceledException)
36  {
37      Console.WriteLine("[Main] Пакетну обробку скасовано — оброблено часткові
    результати");
38  }
```

ct.ThrowIfCancellationRequested() — синхронна перевірка, що кидає OperationCanceledException без затримки. Розміщуйте її на початку кожного кроку або ітерації.

CancelAfter — тайм-аут для async-операцій

CancellationTokenSource.CancelAfter(ms) — стандартний спосіб реалізації тайм-ауту для async-операцій. Замість складних конструкцій з Task.WhenAny + Task.Delay, він пряно і чисто обмежує час операції:

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  async Task<byte[]> DownloadXRayAsync(string patientId, CancellationToken ct)
 6  {
 7      Console.WriteLine($"[Download] Завантаження рентгену {patientId}...");
 8      await Task.Delay(800, ct); // симуляція тривалого завантаження
 9      Console.WriteLine($"[Download] Рентген {patientId} завантажено");
10      return new byte[1024]; // симуляція даних
11  }
12
13  // Тайм-аут 500мс: якщо завантаження займає більше — скасовуємо
14  using CancellationTokenSource cts = new CancellationTokenSource(500); // або
    CancelAfter(500)
15
16  try
17  {
18      byte[] xray = await DownloadXRayAsync("PT-007", cts.Token);
19      Console.WriteLine($"[Main] Завантажено {xray.Length.ToString()} байт");
20  }
21  catch (OperationCanceledException)
22  {
23      Console.WriteLine("[Main] Тайм-аут! Рентген завантажується надто довго");
24      Console.WriteLine("[Main] Показую кешовану версію");
```

```
25  }
```

Конструктор new CancellationTokenSource(ms) — скорочення для CancelAfter(ms). Для сучасного async-коду це набагато зручніше, ніж ручна конструкція з Task.WhenAny.

Пов'язані токени у async-контексті

Реальні системи часто мають кілька незалежних причин для скасування: глобальний тайм-аут застосунку, тайм-аут конкретного запиту, та сигнал від користувача. CancellationTokenSource.CreateLinkedTokenSource об'єднує кілька токенів — новий токен спрацьовує при скасуванні будь-якого:

```
C#
 1  using System;
 2  using System.Threading;
 3  using System.Threading.Tasks;
 4
 5  async Task<string> FetchPatientSummaryAsync(string id, CancellationToken ct)
 6  {
 7      Console.WriteLine($"[Fetch] Запит зведення для {id}...");
 8
 9      // Крок 1: завантаження даних
10      ct.ThrowIfCancellationRequested();
11      await Task.Delay(200, ct);
12      Console.WriteLine($"[Fetch] Базові дані {id} завантажено");
13
14      // Крок 2: обробка
15      ct.ThrowIfCancellationRequested();
16      await Task.Delay(200, ct);
17      Console.WriteLine($"[Fetch] Дані {id} оброблено");
18
19      return $"Зведення: пацієнт {id} — стабільний стан";
20  }
21
22  // Два рівні скасування: глобальний і per-request
23  using CancellationTokenSource appShutdown = new CancellationTokenSource(); //
    глобальний
24  using CancellationTokenSource requestTimeout = new
    CancellationTokenSource(350); // тайм-аут запиту 350мс
25
26  // Об'єднаний токен — спрацьовує при будь-якому скасуванні
27  using CancellationTokenSource linked =
28      CancellationTokenSource.CreateLinkedTokenSource(appShutdown.Token,
    requestTimeout.Token);
29
30  try
31  {
32      string summary = await FetchPatientSummaryAsync("PT-010", linked.Token);
33      Console.WriteLine($"[Main] Результат: {summary}");
34  }
35  catch (OperationCanceledException)
36  {
```

```
37      string reason = appShutdown.IsCancellationRequested   ? "завершення
    застосунку"
38                    : requestTimeout.IsCancellationRequested ? "тайм-аут запиту"
39                    :                                          "невідома
    причина";
40      Console.WriteLine($"[Main] Операцію скасовано: {reason}");
41  }
```

Пов'язані токени — ключовий інструмент для ієрархічного скасування: батьківський scope (застосунок) → дочірній scope (запит) → конкретна операція.

Передача токену у бібліотечні async-методи

Більшість async-методів у .NET BCL приймають CancellationToken. Завжди передавайте токен — це гарантує коректне скасування на кожному рівні:

```
C#
 1  using System;
 2  using System.Net.Http;
 3  using System.Threading;
 4  using System.Threading.Tasks;
 5
 6  async Task<string> FetchFromApiAsync(string url, CancellationToken ct)
 7  {
 8      using HttpClient http = new HttpClient();
 9      http.Timeout = TimeSpan.FromSeconds(5);
10
11      try
12      {
13          // HttpClient.GetStringAsync приймає CancellationToken — передаємо!
14          string response = await http.GetStringAsync(url, ct);
15          return response[..Math.Min(100, response.Length)]; // перші 100
    символів
16      }
17      catch (HttpRequestException ex)
18      {
19          return $"Помилка HTTP: {ex.StatusCode}";
20      }
21  }
22
23  using CancellationTokenSource cts = new CancellationTokenSource(2000); // 2
    секунди
24
25  try
26  {
27      // Спробуємо запит до публічного API
28      string result = await FetchFromApiAsync("https://httpbin.org/delay/1",
    cts.Token);
29      Console.WriteLine($"[API] Відповідь: {result}");
30  }
31  catch (OperationCanceledException)
32  {
33      Console.WriteLine("[API] Тайм-аут або скасовано користувачем");
```

```
}
catch (Exception ex)
{
    Console.WriteLine($"[API] Помилка: {ex.Message}");
}
```

Коректна відповідь на скасування: OperationCanceledException

Коли операція скасовується, правильна поведінка — перекинути OperationCanceledException (або TaskCanceledException), а не повернути пустий результат чи ковтнути виняток. Це дозволяє коду вище в стеку знати, що операція не просто завершилась — вона була скасована:

```
C#
using System;
using System.Threading;
using System.Threading.Tasks;

async Task<string> ProcessAsync(CancellationToken ct)
{
    for (int i = 1; i <= 5; i++)
    {
        ct.ThrowIfCancellationRequested(); // кидає OperationCanceledException

        Console.WriteLine($"  Крок {i.ToString()}/5...");
        await Task.Delay(100, ct);
    }
    return "Результат обробки";
}

async Task RunWithRetryAsync(CancellationToken globalCt)
{
    for (int attempt = 1; attempt <= 3; attempt++)
    {
        // Тайм-аут для кожної спроби — але зберігаємо глобальне скасування
        using CancellationTokenSource attemptCts =
CancellationTokenSource.CreateLinkedTokenSource(globalCt);
        attemptCts.CancelAfter(250); // кожна спроба — до 250мс

        try
        {
            Console.WriteLine($"\n[Retry] Спроба {attempt.ToString()}/3:");
            string result = await ProcessAsync(attemptCts.Token);
            Console.WriteLine($"[Retry] Успіх: {result}");
            return; // виходимо при успіху
        }
        catch (OperationCanceledException) when
(!globalCt.IsCancellationRequested)
        {
            // Тайм-аут спроби — але глобальне скасування не надійшло,
повторюємо
            Console.WriteLine($"[Retry] Спроба {attempt.ToString()} перевищила
тайм-аут — повтор");
        }
        // OperationCanceledException з globalCt — пробрасуємо далі (не ловимо)
```

```
    }
    Console.WriteLine("[Retry] Всі спроби вичерпано");
}

using CancellationTokenSource globalCts = new CancellationTokenSource(1000);
try
{
    await RunWithRetryAsync(globalCts.Token);
}
catch (OperationCanceledException)
{
    Console.WriteLine("\n[Main] Глобальне скасування — зупиняємо повністю");
}
```

Конструкція catch (OperationCanceledException) when (!globalCt.IsCancellationRequested) — фільтр виключень. Вона перехоплює скасування **тільки** від тайм-ауту спроби, але пропускає через себе скасування від глобального токену. Це точний контроль над тим, яке саме скасування обробляється.

Стан Task після скасування

Тут важливо розрізняти два випадки. Задача **async-методу** переходить у стан Canceled, якщо з методу вилетів будь-який OperationCanceledException — незалежно від того, який токен його спричинив. Натомість задача, створена через **Task.Run(work, token)**, стає Canceled лише тоді, коли виняток пов'язаний з тим самим token, що переданий у Task.Run; виняток від **іншого** токена (або токен не передано другим аргументом) дасть стан Faulted. Приклад нижче показує другий, «правильний» варіант з Task.Run:

```
C#
using System;
using System.Threading;
using System.Threading.Tasks;

using CancellationTokenSource cts = new CancellationTokenSource();

// Передаємо token у Task.Run — він знає про "свій" токен
Task task = Task.Run(async () =>
{
    await Task.Delay(500, cts.Token);
}, cts.Token);

cts.Cancel(); // скасовуємо

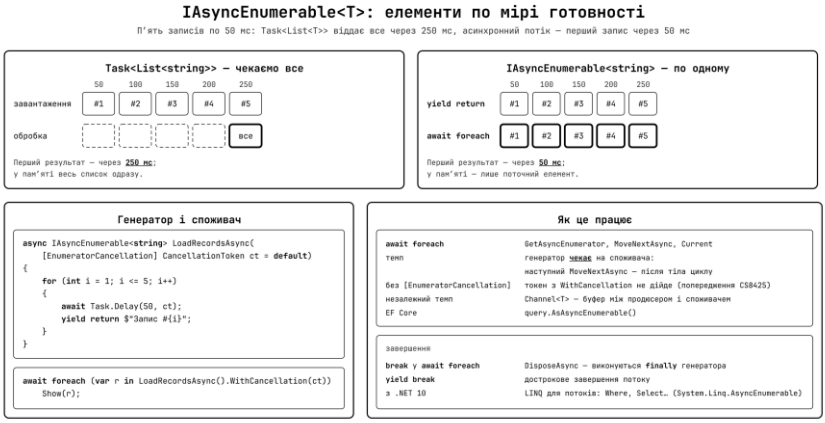
try { await task; } catch (OperationCanceledException) { }

// Task знає, що скасовано через "свій" токен → Canceled
Console.WriteLine($"Status:      {task.Status}");      // Canceled
Console.WriteLine($"IsCanceled:  {task.IsCanceled.ToString()}");   // true
Console.WriteLine($"IsFaulted:   {task.IsFaulted.ToString()}");    // false
```

Розрізнення Canceled і Faulted важливе для коду, що аналізує стан завдань після їх завершення — наприклад, при побудові retry-логіки або логування помилок.

17.6. Асинхронні потоки. IAsyncEnumerable<T>

Класичний async Task<T> відповідає на одне питання: «дай мені результат асинхронно». Але що, якщо результатів багато і вони надходять поступово — рядки з великого файлу, повідомлення з черги, дані з БД мільйонами рядків? Завантажити все у пам'ять і повернути Task<List<T>> — неефективно або неможливо. Саме для цього C# 8.0 ввів IAsyncEnumerable<T> — **асинхронні потоки даних**, де кожен елемент може бути отриманий асинхронно.



Блокуюче vs стрімінгове завантаження

Проблема: завантажити все vs обробляти по мірі надходження

Порівняємо два підходи для завантаження великого набору медичних записів:

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Threading.Tasks;
 4
 5  // Підхід 1: Task<List<T>> — чекаємо ВСЕ, потім обробляємо
 6  async Task<List<string>> LoadAllRecordsAsync(int count)
 7  {
```

```
    var result = new List<string>();
    for (int i = 1; i <= count; i++)
    {
        await Task.Delay(50); // симуляція затримки для кожного запису
        result.Add($"Запис #{i.ToString()}");
    }
    return result; // повертаємо ВСЕ одразу
}

// Підхід 2: IAsyncEnumerable<T> — отримуємо і обробляємо по одному
async IAsyncEnumerable<string> StreamRecordsAsync(int count)
{
    for (int i = 1; i <= count; i++)
    {
        await Task.Delay(50); // симуляція затримки
        yield return $"Запис #{i.ToString()}"; // повертаємо ОДРАЗУ як готовий
    }
}

// Демонстрація: Task<List<T>> — довге очікування, потім всі результати
Console.WriteLine("=== Task<List<T>>: чекаємо всього ===");
var sw1 = System.Diagnostics.Stopwatch.StartNew();
List<string> all = await LoadAllRecordsAsync(5);
sw1.Stop();
Console.WriteLine($"Отримано {all.Count.ToString()} записів через
{sw1.ElapsedMilliseconds.ToString()} мс");
foreach (string r in all)
    Console.WriteLine($"  {r}");

Console.WriteLine("\n=== IAsyncEnumerable<T>: обробляємо по мірі надходження
===");
var sw2 = System.Diagnostics.Stopwatch.StartNew();
await foreach (string record in StreamRecordsAsync(5))
    Console.WriteLine($"  ✓ {record} (через
{sw2.ElapsedMilliseconds.ToString()} мс від початку)");
sw2.Stop();
```

При IAsyncEnumerable<T> перший елемент доступний через ~50 мс, а не через 250 мс. Користувач бачить результати миттєво — це принципова різниця у сприйнятті швидкості.

Оголошення та yield return

Метод, що повертає IAsyncEnumerable<T>, використовує:

- async у заголовку
- IAsyncEnumerable<T> як тип повернення
- yield return для кожного елемента
- await між елементами для асинхронних операцій

```
C#
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
```

```
4
5  // Класичний приклад: стрімінг лабораторних результатів
6  async IAsyncEnumerable<string> GetLabResultsStreamAsync(string patientId)
7  {
8      string[] tests = { "Аналіз крові", "Аналіз сечі", "ЕКГ", "Рентген", "УЗД"
   };
9
10     Console.WriteLine($"[Lab] Починаю аналізи для {patientId}");
11
12     foreach (string test in tests)
13     {
14         // Кожен аналіз займає різний час
15         int duration = test.Length * 30; // умовна тривалість
16         await Task.Delay(duration); // асинхронне очікування
17
18         string result = $"{test}: показники в нормі";
19         Console.WriteLine($"  [Lab] {test} завершено ({duration.ToString()}
   мс)");
20
21         yield return result; // повертаємо результат одразу
22     }
23
24     Console.WriteLine($"[Lab] Всі аналізи для {patientId} завершено");
25 }
26
27 // Споживання: await foreach
28 Console.WriteLine("Реєстрація результатів у систему:");
29 await foreach (string labResult in GetLabResultsStreamAsync("PT-2024-007"))
30 {
31     // Цей блок виконується одразу при появі кожного результату
32     Console.WriteLine($"  ✓ Занесено: {labResult}");
33 }
34
35 Console.WriteLine("\nВсі результати зареєстровано");
```

yield return у async-методі — це точка, де метод «призупиняється» і повертає елемент споживачу. Споживач обробляє елемент, і лише потім метод продовжує генерувати наступний.

await foreach — споживання асинхронного потоку

await foreach — спеціальна синтаксична конструкція для обходу IAsyncEnumerable<T>. Вона асинхронно очікує кожен наступний елемент і виконує тіло циклу:

```
C#
1  using System;
2  using System.Collections.Generic;
3  using System.Threading.Tasks;
4
5  async IAsyncEnumerable<int> GeneratePatientIdsAsync()
6  {
7      int[] ids = { 101, 205, 318, 422, 537 };
8      foreach (int id in ids)
9      {
```

```
        await Task.Delay(80);
        yield return id;
    }
}

// await foreach: кожна ітерація асинхронно очікує наступний елемент
await foreach (int patientId in GeneratePatientIdsAsync())
{
    Console.WriteLine($"Обробляю пацієнта ID={patientId.ToString()}");
    await Task.Delay(20); // власна обробка теж може бути async
}

Console.WriteLine("Потік завершено");
```

Компілятор перетворює await foreach на виклики GetAsyncEnumerator(), MoveNextAsync() та Current — async-аналоги синхронного IEnumerable<T>, а наприкінці (зокрема після break чи винятку) викликає DisposeAsync(), завдяки чому виконуються блоки finally генератора. Починаючи з .NET 10, для асинхронних потоків є й вбудований LINQ — методи Where, Select, ToListAsync тощо з класу System.Linq.AsyncEnumerable (раніше для цього підключали пакет System.Linq.Async).

CancellationToken у асинхронних потоках

IAsyncEnumerable<T> підтримує скасування через атрибут [EnumeratorCancellation] у параметрах методу-генератора. Це дозволяє передавати токен через WithCancellation() при await foreach:

C#
```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using System.Threading;
using System.Threading.Tasks;

// [EnumeratorCancellation] дозволяє передавати токен через WithCancellation()
async IAsyncEnumerable<string> StreamDiagnosesAsync(
    string ward,
    [EnumeratorCancellation] CancellationToken ct = default)
{
    string[] diagnoses = {
        "J06.9 — ГРВІ",      "I10 — Гіпертонія",
        "E11.9 — Діабет 2",  "J18.9 — Пневмонія",
        "K29.5 — Гастрит",   "M54.5 — Люмбалгія"
    };

    Console.WriteLine($"[Stream] Завантаження діагнозів відділення {ward}...");

    foreach (string diagnosis in diagnoses)
    {
        ct.ThrowIfCancellationRequested(); // перевірка перед кожним елементом

```

```
        await Task.Delay(100, ct); // асинхронна затримка зі скасуванням

        Console.WriteLine($"  [Stream] Готово: {diagnosis}");
        yield return diagnosis;
    }
}

using CancellationTokenSource cts = new CancellationTokenSource();
cts.CancelAfter(350); // скасуємо через 350мс (~3 діагнози)

Console.WriteLine("Завантаження діагнозів (з обмеженням часу 350мс):");
try
{
    // WithCancellation() передає токен у генератор через
[EnumeratorCancellation]
    await foreach (string diag in
StreamDiagnosesAsync("Терапія").WithCancellation(cts.Token))
    {
        Console.WriteLine($"  ✓ Зареєстровано: {diag}");
    }
    Console.WriteLine("Всі діагнози завантажено");
}
catch (OperationCanceledException)
{
    Console.WriteLine("[Main] Стрімінг скасовано — тайм-аут");
}
```

Без [EnumeratorCancellation] токен, переданий через WithCancellation(), не потрапить у параметри методу-генератора і не спрацює.

Фільтрація та трансформація потоку

Асинхронні потоки можна обробляти у pipeline: один метод генерує, інший фільтрує або трансформує. Це async-аналог LINQ-ланцюгів для синхронних колекцій:

```
C#
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

// Генератор: потік пацієнтів з показниками
async IAsyncEnumerable<(string Name, int Pulse, double Glucose)>
GetPatientMetricsAsync()
{
    var data = new (string, int, double)[]
    {
        ("Коваль М.",    72,  5.1),
        ("Бойко О.",    145,  8.7), // критичний пульс
        ("Мороз В.",     68,  4.8),
        ("Петренко І.", 110,  12.4), // критична глюкоза
        ("Руденко С.",   78,  5.5)
    };

```

```
    foreach (var patient in data)
    {
        await Task.Delay(50); // симуляція отримання з пристрою
        yield return patient;
    }
}

// Фільтр: передає тільки критичні стани
async IAsyncEnumerable<string> GetCriticalPatientsAsync()
{
    await foreach (var (name, pulse, glucose) in GetPatientMetricsAsync())
    {
        bool critical = pulse > 120 || glucose > 10.0;
        if (critical)
        {
            string reason = pulse > 120 ? $"пульс {pulse.ToString()} уд/хв"
                                        : $"глюкоза {glucose.ToString()}
ммоль/л";
            yield return $"⚠ {name}: {reason}";
        }
    }
}

// Споживання: тільки критичні пацієнти
Console.WriteLine("Критичні стани (стрімінг з фільтрацією):");
await foreach (string alert in GetCriticalPatientsAsync())
    Console.WriteLine($"  {alert}");

Console.WriteLine("Моніторинг завершено");
```

Кожен метод у pipeline є незалежним async-генератором, що не зберігає всі дані у пам'яті — передає елемент далі одразу при отриманні.

IAsyncEnumerable vs інші підходи

Підхід	Повертає	Споживання	Зберігає в пам'яті	Підходить для
Task<List<T>>	Весь список одразу	await + foreach	Весь список	Малі набори даних
IEnumerable<T>	Елементи по одному	foreach	Один елемент	Синхронна генерація
IAsyncEnumerable<T>	Елементи по одному	await foreach	Один елемент	Async-генерація, стрімінг
Channel<T>	Черга повідомлень	ReadAllAsync()	Буфер	Продюсер-споживач

IAsyncEnumerable<T> — правильний вибір, коли:

— Джерело даних велике або нескінченне (БД-курсор, черга, сенсорний потік)

— Перший результат потрібен якнайшвидше

— Споживач обробляє дані в міру надходження (зауважте: генератор не біжить наперед — наступний елемент обчислюється лише тоді, коли споживач попросить його через MoveNextAsync; щоб продюсер і споживач працювали кожен у своєму темпі, між ними ставлять буфер — Channel<T>)

— Обробка кожного елемента асинхронна

Практичний приклад: стрімінг пацієнтів з БД

```C#
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using System.Threading;
using System.Threading.Tasks;

using CancellationTokenSource cts = new CancellationTokenSource(500);

try
{
    await ProcessWardPatientsAsync("Терапія", cts.Token);
}
catch (OperationCanceledException)
{
    Console.WriteLine("[Main] Обробку зупинено за тайм-аутом");
}

// Сервісний рівень: обробка потоку
async Task ProcessWardPatientsAsync(string ward, CancellationToken ct)
{
    var repo = new PatientRepository();
    int count = 0;

    await foreach (string patient in repo.GetPatientsStreamAsync(ward, ct))
    {
        count++;
        Console.WriteLine($"  [{count.ToString()}] Оброблено: {patient}");
    }

    Console.WriteLine($"[Service] Відділення {ward}: оброблено
{count.ToString()} пацієнтів");
}

// Симуляція репозиторію з підтримкою стрімінгу
class PatientRepository
{
```

```
    private readonly string[] _allPatients = {
        "Коваль М.А.", "Бойко О.П.", "Мороз В.І.", "Петренко І.О.",
        "Руденко С.В.", "Шевченко Т.М.", "Гриценко Д.Ю.", "Сидоренко Р.К."
    };

    // Стрімінг з пагінацією — симуляція курсору БД
    public async IAsyncEnumerable<string> GetPatientsStreamAsync(
        string ward,
        [EnumeratorCancellation] CancellationToken ct = default)
    {
        Console.WriteLine($"[Repo] Відкриваю курсор БД для відділення {ward}");

        for (int i = 0; i < _allPatients.Length; i++)
        {
            ct.ThrowIfCancellationRequested();
            await Task.Delay(60, ct); // симуляція мережевої затримки

            yield return $"{_allPatients[i]} (відділення: {ward})";
        }

        Console.WriteLine("[Repo] Курсор БД закрито");
    }
}
```

Цей патерн — «стрімінг репозиторій» — є стандартом для роботи з великими наборами даних у сучасних .NET-застосунках. IAsyncEnumerable<T> підтримується у Entity Framework Core через AsAsyncEnumerable(), у gRPC-стрімінгу, у SignalR і у всій екосистемі .NET.

Питання для самоконтролю — Розділ 17. Асинхронне програмування

17.1. async та await. Асинхронні методи

1. Яку проблему вирішує асинхронне програмування? Поясніть, що відбувається з потоком при синхронному очікуванні вводу-виводу і чому це критично для серверних застосунків із тисячами паралельних запитів.
2. Поясніть різницю між async і await. Чи робить async метод асинхронним сам по собі? Де і тільки де може вживатися await?
3. Що виведе наступний код і чому ManagedThreadId після await може відрізнятися від ManagedThreadId до?

```
C#
1  Console.WriteLine($"До: потік {Thread.CurrentThread.ManagedThreadId}");
2  await Task.Delay(100);
3  Console.WriteLine($"Після: потік {Thread.CurrentThread.ManagedThreadId}");
```

4. Порівняйте Task.Delay і Thread.Sleep. Яка принципова різниця з точки зору ресурсів потоку? Чому в async-методах слід використовувати Task.Delay?
5. Поясніть концепцію стейт-машини (state machine), у яку компілятор перетворює async-метод. Скільки станів матиме метод з двома await? Що зберігає стейт-машина між станами?
6. Знайдіть неефективність у наступному коді і перепишіть його правильно:

```C#
async Task RunTestsAsync()
{
    await RunLabAsync("Кров",   400);
    await RunLabAsync("Сеча",   250);
    await RunLabAsync("ЕКГ",    300);
}
```

Яка буде різниця у загальному часі виконання?

7. Що таке async-лямбда? Яким буде тип змінної, що зберігає async () => { await Task.Delay(100); return 42; }?
8. Поясніть угоду іменування з суфіксом Async. Чому відсутність суфікса у публічному async-методі є поганою практикою? Наведіть три приклади методів .NET BCL, що дотримуються цієї угоди.

17.2. Типи повернення async-методів

9. Перелічіть чотири допустимі типи повернення async-методів. Для кожного з них вкажіть: чи можна його await-ати, чи можна перехопити виняток ззовні, яке основне призначення.
10. Чому async void є небезпечним? Що відбувається з виключенням, кинутим у методі async void? Де єдиний законний сценарій його використання?
11. Що виведе наступний код?

```C#
async void Dangerous()
{
    await Task.Delay(50);
    throw new Exception("Oops");
}

try
{
    Dangerous();
```

```
    await Task.Delay(200);
}
catch (Exception ex)
{
    Console.WriteLine($"Caught: {ex.Message}");
}
Console.WriteLine("After try");
```

12. Поясніть різницю між Task і Task<T> як типом повернення async-методу. Як повернути значення з async Task<string>-методу? Чи потрібно явно загортати його у Task<string>?
13. Що таке ValueTask<T> і в яких сценаріях він перевершує Task<T> за продуктивністю? Яке важливе обмеження накладає ValueTask<T> на споживача?
14. Навіщо існують Task.FromResult<T>() і Task.CompletedTask? У яких двох сценаріях вони типово використовуються? Напишіть приклад синхронної реалізації async-інтерфейсу.
15. Метод async Task<int> GetCountAsync() не містить жодного await. Компілятор попередить про це. Поясніть, що відбудеться при виклику такого методу: чи буде він виконуватись синхронно чи асинхронно?
16. Розробіть інтерфейс IPatientService з методами Task<string> GetNameAsync(string id), Task SaveAsync(string id, string data) і Task<bool> ExistsAsync(string id). Реалізуйте дві версії: DatabaseService (справді async) і InMemoryService (синхронна реалізація, що повертає Task.FromResult / Task.CompletedTask).

17.3. Послідовне та паралельне виконання. Task.WhenAll та Task.WhenAny

17. Чим відрізняється послідовне виконання async-методів від паралельного? Наведіть конкретний приклад, де помилковий послідовний await замість паралельного збільшує час виконання у 3 рази.
18. Поясніть патерн «запусти всі — потім очікуй всіх» на прикладі:

```
C#
// Варіант A (послідовно)
string r1 = await FetchAsync("A");
string r2 = await FetchAsync("B");

// Варіант B (паралельно)
Task<string> t1 = FetchAsync("A");
Task<string> t2 = FetchAsync("B");
string r1 = await t1;
```

```
9  string r2 = await t2;
```

Чому t1 вже виконується під час await t2?

19. Що повертає Task.WhenAll(Task<T>[]) — яким є тип результату? Як результати впорядковані відносно вхідних завдань?
20. Task.WhenAll дочекався всіх завдань, і два з них завершились з помилкою. Що відбувається при await Task.WhenAll(...)? Скільки помилок буде у catch (Exception ex) і як отримати всі помилки?
21. Що повертає Task.WhenAny і чи скасовує воно решту завдань? Реалізуйте тайм-аут для async-операції через Task.WhenAny + Task.Delay.
22. Знайдіть і поясніть проблему у коді:

```
C#
1  async Task<string[]> GetAllDataAsync()
2  {
3      var results = new List<string>();
4      foreach (string id in ids)
5      {
6          results.Add(await FetchAsync(id));
7      }
8      return results.ToArray();
9  }
```

Як переписати через Task.WhenAll і Select?

23. Реалізуйте патерн «обробка по мірі готовності»: запустіть 5 async-завдань і обробляйте кожне одразу при завершенні — не чекаючи решти. Використовуйте Task.WhenAny у циклі while.
24. Порівняйте Task.WaitAll (з розділу 16) і Task.WhenAll (з розділу 17): в чому принципова різниця? Яке блокує потік, а яке — ні? Коли слід обирати кожен?

17.4. Обробка помилок в async-методах

25. Чому обробка помилок в async-коді вимагає окремої уваги? Що відбувається з винятком, кинутим у async-методі, якщо await ніколи не відбудеться?
26. Що виведе наступний код?

```
C#
 1  async Task<string> FailAsync()
 2  {
```

```
    await Task.Delay(50);
    throw new InvalidOperationException("fail");
    return "ok";
}

Task<string> t = FailAsync();
Console.WriteLine($"IsFaulted before Wait: {t.IsFaulted}");
await Task.Delay(200);
Console.WriteLine($"IsFaulted after delay: {t.IsFaulted}");
```

27. Напишіть правильний патерн для перехоплення виключення з async-методу через try/catch. Як виняток з Task «розгортається» при await?
28. Чим небезпечний виняток у async void-методі? Де він з'являється і чому try/catch навколо виклику не допомагає? Наведіть правильну альтернативу для сценарію «запустив і забув».
29. При await Task.WhenAll(t1, t2, t3), де t2 і t3 завершились з помилками, catch (Exception ex) отримає лише одну помилку. Як отримати всі помилки? Напишіть повний приклад з аналізом task.Exception.InnerExceptions.
30. Поясніть, чому виклик .Result або .Wait() у WinForms/WPF може призвести до дедлоку. Опишіть покрокову послідовність подій, що призводить до зависання.
31. Що таке ConfigureAwait(false)? Коли і де його слід використовувати — у бібліотечному коді чи у коді застосунку? Яку проблему він вирішує?
32. await можна використовувати у catch і finally-блоках (з C# 6). Напишіть приклад async-методу, де finally виконує асинхронне закриття ресурсу (await CloseConnectionAsync()). Чому це важливіше, ніж синхронне закриття?

17.5. Скасування async-операцій. CancellationToken

33. Чому CancellationToken є кращим підходом, ніж примусове завершення потоку? Поясніть принцип кооперативного скасування.
34. Яка угода щодо позиції параметра CancellationToken у async-методах? Що означає значення default для цього параметра?
35. Що відбудеться, якщо передати CancellationToken у Task.Delay(500, ct) і викликати Cancel() на ньому через 200 мс? Яке виключення буде кинуто і де саме?

36. Що виведе наступний код і який буде task.Status?

```C#
var cts = new CancellationTokenSource();
Task task = Task.Run(async () =>
{
    await Task.Delay(1000, cts.Token);
}, cts.Token);
cts.Cancel();
try { await task; } catch (OperationCanceledException) { }
Console.WriteLine(task.Status);
```

37. Поясніть різницю між task.IsCanceled == true і task.IsFaulted == true після скасування. За яких умов Task переходить у стан Canceled, а не Faulted, при кинутому OperationCanceledException?
38. Що означає конструкція catch (OperationCanceledException) when (!globalCt.IsCancellationRequested) — фільтр виключень при скасуванні? Коли і чому вона корисна при реалізації логіки повторних спроб (retry)?
39. Реалізуйте async-метод з трьома послідовними кроками (кожен — Task.Delay), де перевірка ct.ThrowIfCancellationRequested() виконується перед кожним кроком. Продемонструйте поведінку при скасуванні між другим і третім кроками.
40. Спроєктуйте CancellationTokenSource.CreateLinkedTokenSource для системи, де операція може бути скасована або глобальним завершенням застосунку (appShutdownCts), або індивідуальним тайм-аутом запиту. Після скасування — визначте, яка саме причина спрацювала.

17.6. Асинхронні потоки. IAsyncEnumerable<T>

41. Яку проблему вирішує IAsyncEnumerable<T> порівняно з Task<List<T>>? Наведіть сценарій, де завантаження всього списку у пам'ять є неприйнятним.
42. Що таке yield return в async-методі і як він взаємодіє з await? Що відбувається, коли споживач ще обробляє поточний елемент, — чи продовжує генератор генерувати наступний?
43. Яку синтаксичну конструкцію використовують для споживання IAsyncEnumerable<T>? Що відбувається «під капотом» при кожній ітерації await foreach?

44. Порівняйте Task<List<T>> і IAsyncEnumerable<T> за трьома параметрами: час до першого елемента, споживання пам'яті, реакція на часткові результати.
45. Напишіть async-генератор IAsyncEnumerable<string> StreamLabResultsAsync(), що імітує надходження результатів аналізів із затримкою між ними. Покажіть його споживання через await foreach.
46. Що таке атрибут [EnumeratorCancellation] і навіщо він потрібен? Що відбудеться, якщо передати токен через WithCancellation(), але не оголосити параметр з цим атрибутом?
47. Реалізуйте pipeline з двох async-генераторів: перший генерує метрики пацієнтів, другий фільтрує лише критичні стани. Покажіть, що другий генератор не зберігає всі метрики у пам'яті, а передає елементи по одному.
48. Порівняйте IAsyncEnumerable<T> з Channel<T> (назва та призначення) за колонками: «хто ініціює генерацію», «можлива буферизація», «підтримка кількох споживачів», «типовий сценарій». Для якого зі сценаріїв підходить кожен підхід?

Розділ 18. Робота з файловою системою

18.1. Класи Path, File та Directory

Файлова система — один із найважливіших механізмів збереження даних у будь-якому застосунку. Медична інформаційна система зберігає картки пацієнтів, результати аналізів, знімки, архіви журналів — усе це файли та теки на диску. C# надає потужний і зручний інструментарій для роботи з файловою системою через простір імен System.IO. Три основні статичні класи — Path, File та Directory — покривають переважну більшість сценаріїв: від складання шляхів і перевірки існування ресурсів до читання, запису та переміщення файлів.

Клас Path — маніпуляції з рядками шляхів

Path — статичний клас, що надає методи для роботи з рядковим поданням шляхів файлової системи. Важлива особливість: Path **не звертається до диску взагалі** — він виключно обробляє рядки. Це робить його операції миттєвими і безпечними щодо винятків файлової системи.

```
C#
 1  using System;
 2  using System.IO;
 3
 4  string filePath = @"C:\MedSystem\Patients\PT-2024\card_001.txt";
 5
 6  // Розбирання шляху на складові
 7  Console.WriteLine($"Тека:              {Path.GetDirectoryName(filePath)}");
 8  Console.WriteLine($"Ім'я файлу:        {Path.GetFileName(filePath)}");
 9  Console.WriteLine($"Ім'я без розширення:
    {Path.GetFileNameWithoutExtension(filePath)}");
10  Console.WriteLine($"Розширення:        {Path.GetExtension(filePath)}");
11  Console.WriteLine($"Корінь шляху:      {Path.GetPathRoot(filePath)}");
```

Метод Path.Combine дозволяє об'єднувати частини шляху незалежно від операційної системи. Використання рядкової конкатенації (dir + "\\" + file) — поширена помилка: вона ламається на Linux/macOS, де роздільник — /. Path.Combine завжди підставляє правильний роздільник:

```C#
using System;
using System.IO;

string baseDir  = @"C:\MedSystem\Patients";
string patientId = "PT-2024-007";
string fileName  = "card.json";

// Правильно: Path.Combine
string fullPath = Path.Combine(baseDir, patientId, fileName);
Console.WriteLine($"Шлях: {fullPath}");

// Платформонезалежний роздільник
Console.WriteLine($"Роздільник:      {Path.DirectorySeparatorChar}");
Console.WriteLine($"Роздільник Alt:  {Path.AltDirectorySeparatorChar}");

// Тимчасова тека системи
Console.WriteLine($"Temp:            {Path.GetTempPath()}");

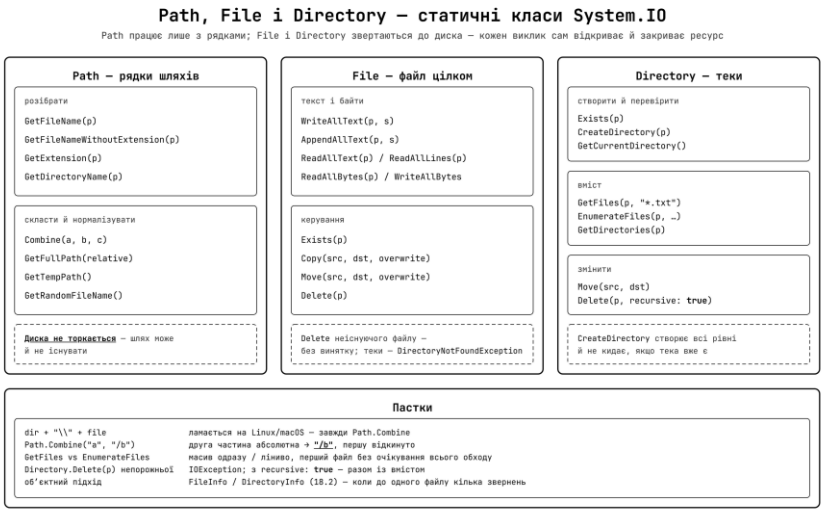
// Унікальне тимчасове ім'я файлу
Console.WriteLine($"Temp file:       {Path.GetTempFileName()}");
```

Path.GetFullPath перетворює відносний шлях на абсолютний — корисно при роботі з конфігурацією:

```C#
using System;
using System.IO;

// Відносний шлях → абсолютний (відносно поточної теки)
string relative = Path.Combine("data", "patients", "archive.json");
string absolute = Path.GetFullPath(relative);
Console.WriteLine($"Відносний: {relative}");
Console.WriteLine($"Абсолютний: {absolute}");

// Перевірка шляху на правильність
bool hasExtension = Path.HasExtension("report.pdf");
Console.WriteLine($"Має розширення: {hasExtension.ToString()}");
```



Статичні класи файлової системи та анатомія шляху

Клас Directory — робота з теками

Directory — статичний клас для операцій над директоріями файлової системи: перевірки існування, створення, видалення, перебирання вмісту.

Перевірка існування та створення

```C#
using System;
using System.IO;

string archiveDir = Path.Combine(Path.GetTempPath(), "MedArchive", "2024");

// Перевіряємо існування
if (!Directory.Exists(archiveDir))
{
    // CreateDirectory створює всі проміжні теки (аналог mkdir -p)
    Directory.CreateDirectory(archiveDir);
    Console.WriteLine($"Створено: {archiveDir}");
}
else
{
    Console.WriteLine($"Вже існує: {archiveDir}");
}

// Поточна тека
Console.WriteLine($"Поточна тека: {Directory.GetCurrentDirectory()}");
```

Directory.CreateDirectory безпечно ігнорує виклик, якщо тека вже існує — не кидає виняток при повторному створенні. Це дозволяє гарантувати існування потрібної структури теок перед записом файлів.

Перебирання вмісту теки

```
C#
 1  using System;
 2  using System.IO;
 3
 4  // Ство рюємо тестову структуру у temp
 5  string root = Path.Combine(Path.GetTempPath(), "MedTest");
 6  Directory.CreateDirectory(Path.Combine(root, "Cardiology"));
 7  Directory.CreateDirectory(Path.Combine(root, "Neurology"));
 8  File.WriteAllText(Path.Combine(root, "index.txt"), "Медичний архів 2024");
 9  File.WriteAllText(Path.Combine(root, "Cardiology", "pt001.txt"), "Петренко
    I.O.");
10  File.WriteAllText(Path.Combine(root, "Cardiology", "pt002.txt"), "Коваль
    M.A.");
11
12  // Отримати всі теки
13  string[] dirs = Directory.GetDirectories(root);
14  Console.WriteLine($"Підтеки ({dirs.Length.ToString()}):");
15  foreach (string d in dirs)
16      Console.WriteLine($"  [DIR] {Path.GetFileName(d)}");
17
18  // Отримати всі файли
19  string[] files = Directory.GetFiles(root, "*.txt");
20  Console.WriteLine($"\nФайли .txt ({files.Length.ToString()}):");
21  foreach (string f in files)
22      Console.WriteLine($"  [FILE] {Path.GetFileName(f)}");
23
24  // Рекурсивний пошук по всій ієрархії
25  string[] allFiles = Directory.GetFiles(root, "pt*.txt",
    SearchOption.AllDirectories);
26  Console.WriteLine($"\nВсі файли пацієнтів ({allFiles.Length.ToString()}):");
27  foreach (string f in allFiles)
28      Console.WriteLine($"  {f}");
```

GetFiles vs EnumerateFiles

GetFiles повертає весь масив одразу — усі шляхи зчитуються у пам'ять перед поверненням. EnumerateFiles повертає IEnumerable<string> з ліниво обчисленим результатом — елементи надходять по одному в міру обходу. Для великих теок з тисячами файлів EnumerateFiles ефективніший — перший файл доступний без очікування завершення всього обходу:

```
C#
 1  using System;
 2  using System.IO;
 3  using System.Linq;
 4
```

```
string root = Path.Combine(Path.GetTempPath(), "MedTest");
Directory.CreateDirectory(root);

// EnumerateFiles — лінивий обхід, зупиняємось на першому знайденому
string? firstPatient = Directory.EnumerateFiles(root, "pt*.txt",
SearchOption.AllDirectories)
    .FirstOrDefault();

Console.WriteLine(firstPatient != null
    ? $"Перший файл пацієнта: {firstPatient}"
    : "Файлів не знайдено");
```

Переміщення та видалення теки

```
C#
using System;
using System.IO;

string src = Path.Combine(Path.GetTempPath(), "OldArchive");
string dst = Path.Combine(Path.GetTempPath(), "NewArchive");
Directory.CreateDirectory(src);
File.WriteAllText(Path.Combine(src, "log.txt"), "тест");

// Переміщення (перейменування) теки
Directory.Move(src, dst);
Console.WriteLine($"Переміщено: {src} → {dst}");

// Видалення: другий параметр true — рекурсивне видалення разом з вмістом
Directory.Delete(dst, recursive: true);
Console.WriteLine("Видалено разом з вмістом");
```

Без параметра recursive: true виклик Delete на непорожній теці кине IOException.

Клас File — робота з файлами

File — статичний клас для «самодостатніх» операцій над файлами: кожен метод сам відкриває файл, виконує операцію і закриває його. Зауважте, що це не атомарність у строгому сенсі: якщо, наприклад, File.WriteAllText перерве збій живлення, на диску може лишитися наполовину записаний файл. Для справді атомарної заміни файлу записують дані у тимчасовий файл, а потім підміняють оригінал через File.Move(tmp, path, overwrite: true) або File.Replace. Це зручно для простих сценаріїв, де не потрібен потоковий доступ.

Запис та читання тексту

```
C#
using System;
using System.IO;
using System.Text;
```

```
4
5  string path = Path.Combine(Path.GetTempPath(), "patient_card.txt");
6
7  // Запис: створює файл або повністю перезаписує існуючий
8  string content = "Пацієнт: Петренко Іван Олексійович\n" +
9                   "Дата народження: 15.03.1978\n" +
10                  "Діагноз: J06.9 — ГРВІ\n" +
11                  "Лікар: Коваленко О.П.";
12
13 File.WriteAllText(path, content, Encoding.UTF8);
14 Console.WriteLine($"Записано: {path}");
15
16 // Читання всього вмісту
17 string loaded = File.ReadAllText(path, Encoding.UTF8);
18 Console.WriteLine($"\n--- Вміст файлу ---\n{loaded}");
19
20 // Рядки — масив рядків (кожен рядок як окремий елемент)
21 string[] lines = File.ReadAllLines(path, Encoding.UTF8);
22 Console.WriteLine($"\nРядків: {lines.Length.ToString()}");
23 Console.WriteLine($"Перший рядок: {lines[0]}");
```

Дозапис у кінець файлу

```
C#
1  using System;
2  using System.IO;
3
4  string logPath = Path.Combine(Path.GetTempPath(), "clinic_log.txt");
5
6  // AppendAllText — дописує в кінець, не стирає існуючий вміст
7  File.AppendAllText(logPath, $"[{DateTime.Now:yyyy-MM-dd HH:mm}] Прийом: Бойко
   О.П.\n");
8  File.AppendAllText(logPath, $"[{DateTime.Now:yyyy-MM-dd HH:mm}] Прийом: Мороз
   В.І.\n");
9  File.AppendAllText(logPath, $"[{DateTime.Now:yyyy-MM-dd HH:mm}] Прийом:
   Сидоренко Т.К.\n");
10
11 string log = File.ReadAllText(logPath);
12 Console.WriteLine("Журнал прийомів:");
13 Console.WriteLine(log);
```

Перевірка існування, копіювання, переміщення, видалення

```
C#
1  using System;
2  using System.IO;
3
4  string src  = Path.Combine(Path.GetTempPath(), "original.txt");
5  string copy = Path.Combine(Path.GetTempPath(), "backup.txt");
6
7  File.WriteAllText(src, "Медична картка #001");
8
9  // Перевірка існування
10 Console.WriteLine($"Існує: {File.Exists(src).ToString()}");
11
12 // Копіювання: третій параметр overwrite
13 File.Copy(src, copy, overwrite: true);
```

```
Console.WriteLine($"Скопійовано до: {copy}");

// Інформація про файл
DateTime created  = File.GetCreationTime(src);
DateTime modified = File.GetLastWriteTime(src);
Console.WriteLine($"Створено:  {created:yyyy-MM-dd HH:mm:ss}");
Console.WriteLine($"Змінено:   {modified:yyyy-MM-dd HH:mm:ss}");

// Переміщення
string moved = Path.Combine(Path.GetTempPath(), "archive_001.txt");
File.Move(copy, moved, overwrite: true);
Console.WriteLine($"Переміщено: {moved}");

// Видалення
File.Delete(src);
File.Delete(moved);
Console.WriteLine("Тимчасові файли видалено");
```

Читання/запис масиву байтів

Для бінарних даних File.ReadAllBytes і File.WriteAllBytes зчитують та записують весь файл як byte[]:

```
C#
using System;
using System.IO;

string path = Path.Combine(Path.GetTempPath(), "binary_data.bin");

// Запис байтів (наприклад, результати вимірювань)
byte[] measurements = { 0x48, 0xB4, 0x00, 0x50, 0x00, 0x46 }; // hex-кодовані дані
File.WriteAllBytes(path, measurements);

// Читання назад
byte[] loaded = File.ReadAllBytes(path);
Console.WriteLine($"Прочитано {loaded.Length.ToString()} байт");
Console.Write("Байти: ");
foreach (byte b in loaded)
    Console.Write($"{b.ToString("X2")} ");
Console.WriteLine("");

File.Delete(path);
```

Path, File та Directory разом: реальний сценарій

Об'єднаємо всі три класи у типовому сценарії медичної системи — організація архіву результатів аналізів:

```
C#
using System;
using System.IO;
using System.Linq;

// Структура архіву: MedArchive/{рік}/{місяць}/{пацієнт}.txt
```

```
string archiveRoot = Path.Combine(Path.GetTempPath(), "MedArchive");

void SaveLabResult(string patientId, string result)
{
    string year  = DateTime.Now.Year.ToString();
    string month = DateTime.Now.Month.ToString("D2");
    string dir   = Path.Combine(archiveRoot, year, month);

    // Гарантуємо існування ієрархії теок
    Directory.CreateDirectory(dir);

    string fileName = $"{patientId}_{DateTime.Now:yyyyMMdd_HHmmss}.txt";
    string filePath = Path.Combine(dir, fileName);

    File.WriteAllText(filePath, result);
    Console.WriteLine($"Збережено: {filePath}");
}

string LoadLatestResult(string patientId)
{
    string year  = DateTime.Now.Year.ToString();
    string month = DateTime.Now.Month.ToString("D2");
    string dir   = Path.Combine(archiveRoot, year, month);

    if (!Directory.Exists(dir)) return "Архів порожній";

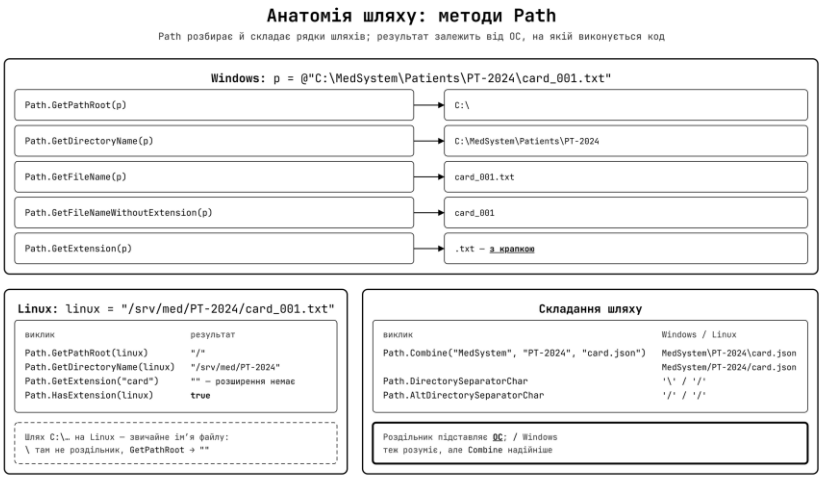
    // Знаходимо всі файли цього пацієнта і беремо найновіший
    string pattern = $"{patientId}_*.txt";
    string? latest = Directory.EnumerateFiles(dir, pattern)
        .OrderByDescending(f => f)
        .FirstOrDefault();

    return latest != null ? File.ReadAllText(latest) : "Результатів не
знайдено";
}

// Зберігаємо кілька результатів
SaveLabResult("PT001", "Гемоглобін: 135 г/л — норма\nЛейкоцити: 6.2 — норма");
SaveLabResult("PT002", "Глюкоза: 7.8 ммоль/л — вище норми\nХолестерин: 5.1 —
норма");
SaveLabResult("PT001", "Глюкоза: 4.9 ммоль/л — норма");

// Читаємо останній результат
Console.WriteLine($"\nОстанній результат PT001:\n{LoadLatestResult("PT001")}");

// Прибираємо тестові дані
if (Directory.Exists(archiveRoot))
    Directory.Delete(archiveRoot, recursive: true);
```

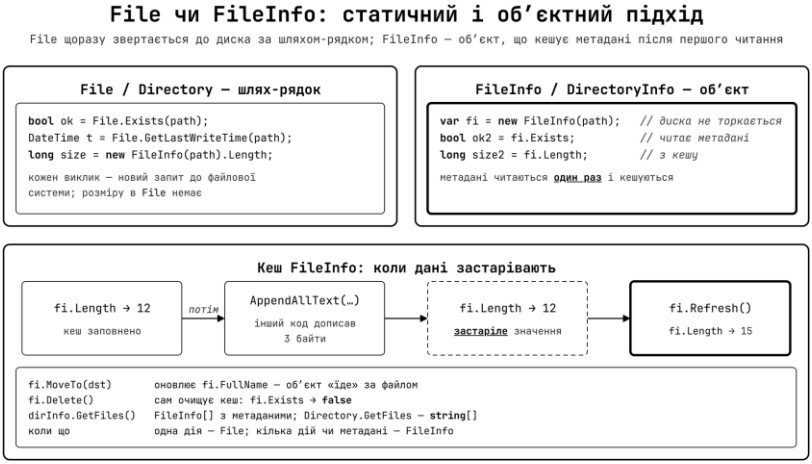


Анатомія шляху файлової системи

18.2. Класи FileInfo та DirectoryInfo

У розділі 18.1 ми розглянули статичні класи Path, File та Directory, кожен з яких виконує операцію атомарно: відкрив → зробив → закрив. Це зручно для разових дій, але неефективно, коли потрібно виконати кілька операцій над одним і тим самим ресурсом. Саме для таких сценаріїв C# надає **об'єктні аналоги** — класи FileInfo та DirectoryInfo.

На відміну від статичних класів, FileInfo і DirectoryInfo — **екземплярні класи**: ви створюєте об'єкт, що представляє конкретний файл або директорію, і далі вся робота відбувається через цей об'єкт. Ключова перевага — **кешування метаданих**: при першому зверненні до властивості (Exists, Length, LastWriteTime тощо) об'єкт один раз запитує у файлової системи всі метадані файлу і зберігає їх у собі, тож подальші звернення до властивостей диска не торкаються. (У старих джерелах можна зустріти твердження, що FileInfo ще й «один раз перевіряє права доступу» — це стосувалося перевірок безпеки коду CAS у .NET Framework; у .NET Core і .NET 5+ таких перевірок немає, а права доступу перевіряє операційна система при кожному відкритті файлу незалежно від того, чи використано File, чи FileInfo.) Зворотний бік кешування — дані можуть застаріти, якщо файл змінив інший код; про це — метод Refresh() нижче.



Статичні vs екземплярні класи файлової системи

Клас FileInfo — об'єктне представлення файлу

FileInfo надає той самий набір операцій, що і статичний File, але через інтерфейс об'єкта: метадані файлу зберігаються у властивостях, а операції викликаються як методи екземпляра.

Створення об'єкта та властивості

```
C#
using System;
using System.IO;

// Визначаємо шлях до файлу
string path = Path.Combine(Path.GetTempPath(), "patient_card_001.txt");

// Записуємо тестовий файл
File.WriteAllText(path,
    "Пацієнт: Петренко Іван Олексійович\n" +
    "Дата народження: 15.03.1978\n" +
    "Діагноз: J06.9 — ГРВІ\n" +
    "Лікар: Коваленко О.П.");

// Створюємо FileInfo — об'єкт ще не звертається до диску при конструюванні
FileInfo fi = new FileInfo(path);

// Властивості — більшість звертаються до диску при першому зверненні
Console.WriteLine($"Існує:           {fi.Exists.ToString()}");
Console.WriteLine($"Ім'я:            {fi.Name}");
Console.WriteLine($"Ім'я без розш.:  {fi.Directory?.Name}");
Console.WriteLine($"Повний шлях:     {fi.FullName}");
```

```
Console.WriteLine($"Розширення:      {fi.Extension}");
Console.WriteLine($"Розмір (байт):   {fi.Length.ToString()}");
Console.WriteLine($"Тека:            {fi.DirectoryName}");
Console.WriteLine($"Тільки читання:  {fi.IsReadOnly.ToString()}");
Console.WriteLine($"Створено:        {fi.CreationTime:yyyy-MM-dd HH:mm:ss}");
Console.WriteLine($"Змінено:         {fi.LastWriteTime:yyyy-MM-dd HH:mm:ss}");
Console.WriteLine($"Відкрито:        {fi.LastAccessTime:yyyy-MM-dd HH:mm:ss}");

File.Delete(path);
```

Методи FileInfo — операції над файлом

C#

```
using System;
using System.IO;
using System.Text;

string srcPath  = Path.Combine(Path.GetTempPath(), "med_record.txt");
string copyPath = Path.Combine(Path.GetTempPath(), "med_record_backup.txt");
string movePath = Path.Combine(Path.GetTempPath(), "archive",
"med_record_final.txt");

File.WriteAllText(srcPath, "Медична картка #007: Бойко Оксана Петрівна",
Encoding.UTF8);
Directory.CreateDirectory(Path.GetDirectoryName(movePath)!);

FileInfo fi = new FileInfo(srcPath);

// Читання через OpenText — повертає StreamReader
using (StreamReader reader = fi.OpenText())
{
    Console.WriteLine($"Вміст: {reader.ReadToEnd()}");
}

// Копіювання — повертає FileInfo копії
FileInfo copyFi = fi.CopyTo(copyPath, overwrite: true);
Console.WriteLine($"Копія: {copyFi.FullName} ({copyFi.Length.ToString()}
байт)");

// Переміщення
fi.MoveTo(movePath, overwrite: true);
Console.WriteLine($"Переміщено: {fi.FullName}"); // fi.FullName тепер вказує на
нове місце

// Перейменування — це MoveTo у тій самій теці
FileInfo movedFi = new FileInfo(movePath);
movedFi.MoveTo(Path.Combine(Path.GetDirectoryName(movePath)!,
"med_record_renamed.txt"));
Console.WriteLine($"Перейменовано: {movedFi.Name}");

// Видалення
movedFi.Delete();
copyFi.Delete();
Directory.Delete(Path.GetDirectoryName(movePath)!, recursive: true);
Console.WriteLine("Тимчасові файли прибрано");
```

fi.MoveTo(...) оновлює внутрішній стан об'єкта — після виклику fi.FullName і fi.Name відображають нове розташування файлу. Це суттєва відмінність від статичного File.Move, після якого ви маєте будувати новий шлях вручну.

Відкриття потоків через FileInfo

FileInfo надає методи для відкриття різних видів потоків — це зручно, коли потрібен потоковий доступ замість атомарного читання/запису:

```
C#
 1  using System;
 2  using System.IO;
 3  using System.Text;
 4
 5  string path = Path.Combine(Path.GetTempPath(), "stream_test.txt");
 6  File.WriteAllText(path, "Рядок 1\nРядок 2\nРядок 3", Encoding.UTF8);
 7
 8  FileInfo fi = new FileInfo(path);
 9
10  // OpenRead — відкриває FileStream тільки для читання
11  using (FileStream fs = fi.OpenRead())
12  {
13      Console.WriteLine($"FileStream: {fs.CanRead} / CanWrite:
    {fs.CanWrite.ToString()}");
14      Console.WriteLine($"Розмір потоку: {fs.Length.ToString()} байт");
15  }
16
17  // OpenText — відкриває StreamReader (текстовий)
18  using (StreamReader sr = fi.OpenText())
19  {
20      string? line;
21      int lineNum = 1;
22      while ((line = sr.ReadLine()) != null)
23          Console.WriteLine($"  [{lineNum++.ToString()}] {line}");
24  }
25
26  // AppendText — відкриває StreamWriter у режимі дозапису
27  using (StreamWriter sw = fi.AppendText())
28  {
29      sw.WriteLine($"Рядок додано: {DateTime.Now:HH:mm:ss}");
30  }
31
32  // Перевіряємо
33  Console.WriteLine($"Після дозапису: {fi.Length.ToString()} байт");
34  // Увага: fi.Length кешовано — потрібен Refresh()
35  fi.Refresh(); // оновлюємо кешовані метадані
36  Console.WriteLine($"Після Refresh(): {fi.Length.ToString()} байт");
37
38  fi.Delete();
```

fi.Refresh() — важливий метод: FileInfo кешує метадані (розмір, час змін) при першому зверненні. Якщо файл змінився після створення

об'єкта, кешовані дані застаріють. Виклик Refresh() примусово перечитує метадані з диску.

Клас DirectoryInfo — об'єктне представлення директорії

DirectoryInfo — об'єктний аналог статичного Directory. Представляє конкретну директорію і надає методи для роботи з нею та її вмістом через інтерфейс об'єкта.

Властивості та методи

```C#
using System;
using System.IO;

string dirPath = Path.Combine(Path.GetTempPath(), "MedClinic_2024");
Directory.CreateDirectory(Path.Combine(dirPath, "Cardiology"));
Directory.CreateDirectory(Path.Combine(dirPath, "Neurology"));
File.WriteAllText(Path.Combine(dirPath, "index.txt"), "Клінічний архів 2024");
File.WriteAllText(Path.Combine(dirPath, "Cardiology", "pt001.txt"), "Петренко І.О.");
File.WriteAllText(Path.Combine(dirPath, "Cardiology", "pt002.txt"), "Коваль М.А.");

DirectoryInfo di = new DirectoryInfo(dirPath);

// Властивості
Console.WriteLine($"Існує:         {di.Exists.ToString()}");
Console.WriteLine($"Ім'я:          {di.Name}");
Console.WriteLine($"Повний шлях:   {di.FullName}");
Console.WriteLine($"Батьківська:   {di.Parent?.Name}");
Console.WriteLine($"Корінь:        {di.Root.FullName}");
Console.WriteLine($"Створено:      {di.CreationTime:yyyy-MM-dd HH:mm:ss}");

// Підтеки — GetDirectories() повертає DirectoryInfo[]
DirectoryInfo[] subdirs = di.GetDirectories();
Console.WriteLine($"\nПідтеки ({subdirs.Length.ToString()}):");
foreach (DirectoryInfo sub in subdirs)
    Console.WriteLine($"  [DIR]  {sub.Name}  ({sub.CreationTime:HH:mm:ss})");

// Файли — GetFiles() повертає FileInfo[]
FileInfo[] files = di.GetFiles("*.txt");
Console.WriteLine($"\nФайли .txt ({files.Length.ToString()}):");
foreach (FileInfo f in files)
    Console.WriteLine($"  [FILE] {f.Name}  {f.Length.ToString()} байт");

// Рекурсивний пошук
FileInfo[] allFiles = di.GetFiles("*.txt", SearchOption.AllDirectories);
Console.WriteLine($"\nВсі .txt рекурсивно ({allFiles.Length.ToString()}):");
foreach (FileInfo f in allFiles)
    Console.WriteLine($"  {f.FullName}");

di.Delete(recursive: true);
```

Ключова відмінність DirectoryInfo.GetFiles() від Directory.GetFiles(): перший повертає FileInfo[] — повноцінні об'єкти з метаданими (розмір, час, атрибути). Другий повертає лише string[] — масив рядків-шляхів. Якщо вам потрібні метадані файлів після переліку, DirectoryInfo економить один додатковий виклик на кожен файл.

Створення ієрархії та навігація

```C#
using System;
using System.IO;

string root = Path.Combine(Path.GetTempPath(), "MedArchive");

// Створення ієрархії через DirectoryInfo
DirectoryInfo rootDi = new DirectoryInfo(root);
rootDi.Create(); // безпечно — не кидає при повторному викл.

DirectoryInfo year   = rootDi.CreateSubdirectory("2024");
DirectoryInfo cardio = year.CreateSubdirectory("Cardiology");
DirectoryInfo neuro  = year.CreateSubdirectory("Neurology");

// Додаємо файли через FileInfo від DirectoryInfo
string cardioFile = Path.Combine(cardio.FullName, "pt_reports.txt");
File.WriteAllText(cardioFile, "Звіти кардіологічного відділення");

// Навігація вверх по ієрархії через Parent
Console.WriteLine("Ієрархія від cardio вгору:");
DirectoryInfo? current = cardio;
while (current != null)
{
    Console.WriteLine($"  {current.FullName}");
    current = current.Parent;
}

// GetDirectories з SearchOption
Console.WriteLine($"\nВсі підтеки rootDi рекурсивно:");
foreach (DirectoryInfo d in rootDi.GetDirectories("*",
SearchOption.AllDirectories))
    Console.WriteLine($"  {d.Name}");

rootDi.Delete(recursive: true);
```

CreateSubdirectory — метод, що повертає DirectoryInfo для щойно створеної підтеки. Це дозволяє будувати ієрархії тек у fluent-стилі, не будуючи шляхи вручну. Властивість Parent забезпечує навігацію вгору, Root — одразу до кореня.

FileSystemInfo — спільний базовий клас

FileInfo і DirectoryInfo успадковують від абстрактного класу FileSystemInfo. Це дозволяє писати поліморфний код, що працює як з файлами, так і з директоріями:

```
C#
using System;
using System.IO;

string root = Path.Combine(Path.GetTempPath(), "MedScan");
Directory.CreateDirectory(Path.Combine(root, "Reports"));
File.WriteAllText(Path.Combine(root, "index.txt"), "Індекс сканування");
File.WriteAllText(Path.Combine(root, "log.txt"), "Журнал операцій");

DirectoryInfo di = new DirectoryInfo(root);

// GetFileSystemInfos() повертає FileSystemInfo[] — суміш файлів і теок
FileSystemInfo[] entries = di.GetFileSystemInfos();
Console.WriteLine($"Всього записів: {entries.Length.ToString()}");

foreach (FileSystemInfo entry in entries)
{
    // Поліморфна обробка через is-перевірку
    if (entry is FileInfo fi)
    {
        Console.WriteLine($"  [FILE] {fi.Name}  {fi.Length.ToString()} байт
змінено: {fi.LastWriteTime:HH:mm:ss}");
    }
    else if (entry is DirectoryInfo subDir)
    {
        int fileCount = subDir.GetFiles().Length;
        Console.WriteLine($"  [DIR]  {subDir.Name}  ({fileCount.ToString()}
файлів)  створено: {subDir.CreationTime:HH:mm:ss}");
    }
}

// Спільні властивості FileSystemInfo
Console.WriteLine("\nАтрибути через FileSystemInfo:");
foreach (FileSystemInfo entry in entries)
{
    Console.WriteLine($"  {entry.Name}: {entry.Attributes}");
}

di.Delete(recursive: true);
```

FileSystemInfo містить спільні властивості: Name, FullName, Exists, CreationTime, LastWriteTime, LastAccessTime, Attributes. Метод GetFileSystemInfos() повертає суміш FileInfo і DirectoryInfo об'єктів для всіх записів у теці.

FileInfo vs File — порівняння підходів

```
C#
```

```
using System;
using System.IO;
using System.Text;

string path = Path.Combine(Path.GetTempPath(), "compare_test.txt");
File.WriteAllText(path, "Тестовий вміст медичної картки пацієнта",
Encoding.UTF8);

// --- Статичний File: зручно для разових операцій ---
// Кожен виклик — окремий запит до файлової системи за шляхом-рядком
bool ex1  = File.Exists(path);
long len1 = new FileInfo(path).Length; // потрібен FileInfo для розміру
File.Copy(path, path + ".bak", overwrite: true);
File.Delete(path + ".bak");

// --- Екземплярний FileInfo: зручно для кількох операцій над тим самим файлом
---
// Метадані читаються один раз — при першому зверненні до властивості
FileInfo fi = new FileInfo(path);
bool   ex2     = fi.Exists;      // читання метаданих з диска — один раз
long   size    = fi.Length;      // вже кешовано
string ext     = fi.Extension;
DateTime mod   = fi.LastWriteTime;
FileInfo bakFi = fi.CopyTo(path + ".bak2", overwrite: true); // через об'єкт
bakFi.Delete();

Console.WriteLine($"Статичний: exists={ex1.ToString()},
size={len1.ToString()}");
Console.WriteLine($"Екземпляр: exists={ex2.ToString()}, size={size.ToString()},
ext={ext}, mod={mod:HH:mm:ss}");

fi.Delete();
```

Правило вибору:

— File / Directory — одна операція, результат не потрібен відразу, метадані не важливі

— FileInfo / DirectoryInfo — кілька операцій над тим самим ресурсом, або потрібні метадані файлу (розмір, час, атрибути)

Практичний сценарій: аналіз архіву медичних записів

Об'єднаємо можливості FileInfo та DirectoryInfo у реальному сценарії — побудова зведеного звіту по архіву медичних записів:

```
C#
using System;
using System.IO;
using System.Linq;
using System.Text;

// Створюємо тестову структуру архіву
string archive = Path.Combine(Path.GetTempPath(), "ClinicArchive");
Directory.CreateDirectory(Path.Combine(archive, "2023", "Cardiology"));
Directory.CreateDirectory(Path.Combine(archive, "2023", "Neurology"));
```

```
Directory.CreateDirectory(Path.Combine(archive, "2024", "Cardiology"));

// Генеруємо тестові файли
string[] files2023 = {
    Path.Combine(archive, "2023", "Cardiology", "pt001_report.pdf.txt"),
    Path.Combine(archive, "2023", "Cardiology", "pt002_ecg.pdf.txt"),
    Path.Combine(archive, "2023", "Neurology",  "pt003_mri.pdf.txt"),
};
string[] files2024 = {
    Path.Combine(archive, "2024", "Cardiology", "pt004_report.pdf.txt"),
    Path.Combine(archive, "2024", "Cardiology", "pt005_echo.pdf.txt"),
};

foreach (string f in files2023)
    File.WriteAllText(f, new string('X', 1024 + f.GetHashCode() % 512),
Encoding.UTF8);
foreach (string f in files2024)
    File.WriteAllText(f, new string('X', 2048 + f.GetHashCode() % 1024),
Encoding.UTF8);

// Аналіз через DirectoryInfo / FileInfo
DirectoryInfo archiveDi = new DirectoryInfo(archive);

Console.WriteLine($"=== Звіт архіву: {archiveDi.Name} ===\n");

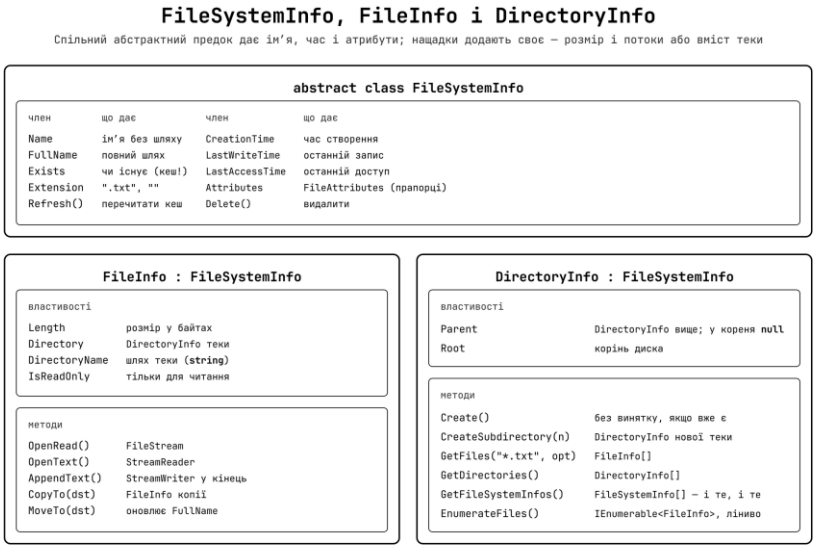
foreach (DirectoryInfo yearDir in archiveDi.GetDirectories().OrderBy(d =>
d.Name))
{
    FileInfo[] yearFiles = yearDir.GetFiles("*.txt",
SearchOption.AllDirectories);
    long totalSize = yearFiles.Sum(f => f.Length);

    Console.WriteLine($"Рік: {yearDir.Name}");
    Console.WriteLine($"  Файлів:      {yearFiles.Length.ToString()}");
    Console.WriteLine($"  Загальний розмір: {totalSize.ToString()} байт");

    // Топ-файл за розміром
    FileInfo? largest = yearFiles.OrderByDescending(f =>
f.Length).FirstOrDefault();
    if (largest != null)
        Console.WriteLine($"  Найбільший:  {largest.Name}
({largest.Length.ToString()} байт)");

    // Файли за підтеками
    foreach (DirectoryInfo deptDir in yearDir.GetDirectories())
    {
        FileInfo[] deptFiles = deptDir.GetFiles("*.txt");
        Console.WriteLine($"  [{deptDir.Name}] {deptFiles.Length.ToString()}
файлів");
    }
    Console.WriteLine("");
}

archiveDi.Delete(recursive: true);
```

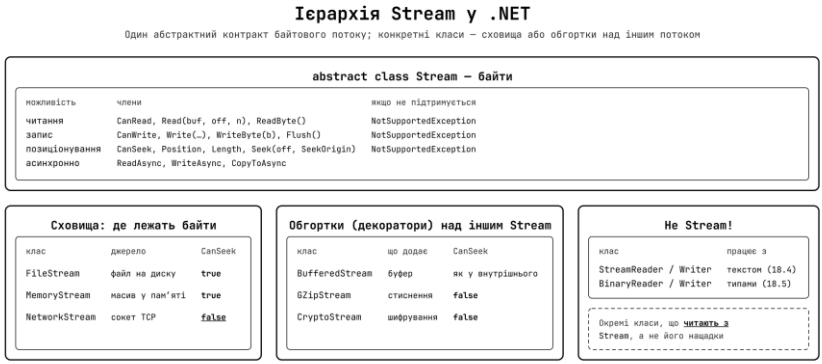


Властивості FileInfo та структура ієрархії

18.3. Stream та FileStream. Потоковий доступ до файлів

Усі класи File.WriteAllText, File.ReadAllBytes — це зручні обгортки над нижчим рівнем абстракції: **потоками (streams)**. Потік — це абстракція послідовного доступу до даних, де читання або запис відбуваються **по шматках (chunk by chunk)**, а не завантаженням усього ресурсу у пам'ять одразу. Розуміння потоків є фундаментом для роботи не лише з файлами, а й з мережевими з'єднаннями, ZIP-архівами, HTTP-відповідями, шифруванням — будь-яким I/O у .NET.

Абстрактний клас Stream визначає єдиний контракт для всіх видів потоків. FileStream — одна з конкретних реалізацій, що забезпечує потоковий доступ до файлів на диску.



Ієрархія класів Stream у .NET

Абстрактний клас Stream

Stream (простір імен System.IO) визначає мінімальний набір можливостей, якими може володіти або не володіти потік:

```
C#
 1  using System;
 2  using System.IO;
 3  using System.Text;
 4
 5  string path = Path.Combine(Path.GetTempPath(), "stream_demo.bin");
 6
 7  // Запишемо щось через FileStream, щоб мати що читати
 8  File.WriteAllText(path, "MED-DATA-2024", Encoding.ASCII);
 9
10  // Відкриваємо через абстрактний тип Stream
11  using Stream stream = File.OpenRead(path);
12
13  // Три фундаментальні властивості потоку
14  Console.WriteLine($"CanRead:   {stream.CanRead.ToString()}");    // чи можна
    читати
15  Console.WriteLine($"CanWrite:  {stream.CanWrite.ToString()}");   // чи можна
    записувати
16  Console.WriteLine($"CanSeek:   {stream.CanSeek.ToString()}");    // чи
    підтримує позиціонування
17
18  // Позиціонування — доступне лише якщо CanSeek == true
19  Console.WriteLine($"Length:    {stream.Length.ToString()} байт");
20  Console.WriteLine($"Position:  {stream.Position.ToString()} (початок)");
21
22  // Читання у буфер
23  byte[] buffer = new byte[4];
24  int read = stream.Read(buffer, 0, buffer.Length);
25  Console.WriteLine($"Прочитано: {read.ToString()} байт:
    {Encoding.ASCII.GetString(buffer, 0, read)}");
26  Console.WriteLine($"Position після Read: {stream.Position.ToString()}");
27
```

```
// Seek — переміщення позиції
stream.Seek(4, SeekOrigin.Begin);   // з початку на 4
Console.WriteLine($"Після Seek(4, Begin):
Position={stream.Position.ToString()}");

stream.Seek(-2, SeekOrigin.End);    // від кінця на -2
Console.WriteLine($"Після Seek(-2, End):
Position={stream.Position.ToString()}");

stream.Seek(2, SeekOrigin.Current); // від поточної позиції
Console.WriteLine($"Після Seek(2, Current):
Position={stream.Position.ToString()}");

File.Delete(path);
```

Stream визначає три типи операцій: **читання** (Read, ReadByte), **запис** (Write, WriteByte, Flush) і **позиціонування** (Seek, Position). Не всі потоки підтримують усі операції — мережевий потік NetworkStream не підтримує Seek (CanSeek == false), потік лише для читання не підтримує Write. Перевіряйте CanRead/CanWrite/CanSeek перед використанням.

Клас FileStream

FileStream — реалізація Stream для файлів на диску. Він підтримує всі три операції: читання, запис і позиціонування (CanSeek == true).

Відкриття файлу: FileMode та FileAccess

C#

```
using System;
using System.IO;
using System.Text;

string path = Path.Combine(Path.GetTempPath(), "filestream_test.bin");

// Варіант 1: через конструктор FileStream
// FileMode: Create, CreateNew, Open, OpenOrCreate, Append, Truncate
// FileAccess: Read, Write, ReadWrite
using (FileStream fs = new FileStream(path, FileMode.Create, FileAccess.Write))
{
    Console.WriteLine($"Відкрито для запису. CanWrite:
{fs.CanWrite.ToString()}");
    byte[] data = Encoding.UTF8.GetBytes("PATIENT-DATA-001");
    fs.Write(data, 0, data.Length);
    Console.WriteLine($"Записано {data.Length.ToString()} байт");
} // using закриває потік і звільняє ресурс

// Варіант 2: через статичний File (зручніший синтаксис)
using (FileStream fs = File.Open(path, FileMode.Open, FileAccess.Read))
{
    byte[] buf = new byte[fs.Length];
    int n = fs.Read(buf, 0, buf.Length);
    Console.WriteLine($"Прочитано {n.ToString()} байт:
{Encoding.UTF8.GetString(buf)}");
```

```
}

// Варіант 3: через FileInfo
FileInfo fi = new FileInfo(path);
using (FileStream fs = fi.Open(FileMode.Open, FileAccess.ReadWrite))
{
    fs.Seek(8, SeekOrigin.Begin); // переходимо до "001"
    byte[] patch = Encoding.ASCII.GetBytes("007");
    fs.Write(patch, 0, patch.Length);

    fs.Seek(0, SeekOrigin.Begin);
    byte[] result = new byte[fs.Length];
    fs.Read(result, 0, result.Length);
    Console.WriteLine($"Після патчу: {Encoding.UTF8.GetString(result)}");
}

File.Delete(path);
```

FileMode визначає, що відбувається при відкритті:

FileMode	Якщо файл існує	Якщо файл не існує
Create	Обрізає до нуля	Створює новий
CreateNew	Кидає IOException	Створює новий
Open	Відкриває	Кидає FileNotFoundException
OpenOrCreate	Відкриває	Створює новий
Append	Відкриває, позиція в кінці	Створює новий
Truncate	Обрізає до нуля	Кидає FileNotFoundException

Буферизований запис та Flush

```
C#
using System;
using System.IO;
using System.Text;

string path = Path.Combine(Path.GetTempPath(), "buffered_write.txt");

// Варіант 4: з явним розміром буфера
// FileStream внутрішньо буферизує запис — дані спочатку потрапляють у буфер
using (FileStream fs = new FileStream(path, FileMode.Create, FileAccess.Write,
FileShare.None, bufferSize: 4096))
{
    string header = "=== Медичний журнал 2024 ===\n";
```

```
    byte[] headerBytes = Encoding.UTF8.GetBytes(header);
    fs.Write(headerBytes, 0, headerBytes.Length);
    Console.WriteLine($"Після Write: Position={fs.Position.ToString()}");

    // Flush — примусово скидає буфер на диск
    // Без Flush дані можуть ще бути у буфері при краші
    fs.Flush();
    Console.WriteLine("Flush виконано — дані гарантовано на диску");

    string entry = $"[{DateTime.Now:HH:mm:ss}] Прийом: Петренко І.О.\n";
    byte[] entryBytes = Encoding.UTF8.GetBytes(entry);
    fs.Write(entryBytes, 0, entryBytes.Length);
}
// При виході з using: Dispose() → Flush() → Close() — автоматично

string content = File.ReadAllText(path, Encoding.UTF8);
Console.WriteLine($"Вміст файлу:\n{content}");
File.Delete(path);
```

Flush() передає вміст внутрішнього буфера FileStream операційній системі — після цього дані переживуть аварійне завершення **програми**. Але ОС теж кешує записи в пам'яті, тож при збої живлення вони ще можуть загубитися. Гарантію запису на фізичний носій дає лише перевантаження Flush(flushToDisk: true) — воно повільніше, бо чекає на диск. Для критичних даних (медичні журнали, транзакції) — викликайте Flush() після кожного запису. При звичайних ситуаціях using / Dispose автоматично виконає Flush і Close.

Паралельний доступ: FileShare

C#

```
using System;
using System.IO;
using System.Text;

string path = Path.Combine(Path.GetTempPath(), "shared_log.txt");
File.WriteAllText(path, "Початковий вміст\n", Encoding.UTF8);

// FileShare.Read — дозволяє іншим процесам читати, поки ми пишемо
using FileStream fs1 = new FileStream(path, FileMode.Append, FileAccess.Write,
FileShare.Read);
fs1.Write(Encoding.UTF8.GetBytes("Запис 1 від процесу A\n"));

// FileShare.ReadWrite — дозволяє і читання, і запис
using FileStream fs2 = new FileStream(path, FileMode.Open, FileAccess.Read,
FileShare.ReadWrite);
byte[] buf = new byte[fs2.Length];
fs2.Read(buf, 0, buf.Length);
Console.WriteLine($"Паралельне читання:\n{Encoding.UTF8.GetString(buf)}");

// Важливо: FileShare.None = виключний доступ (іншим забороняється відкривати
файл)
Console.WriteLine("FileShare.None = монопольний доступ (інші отримають
IOException)");
```

```
20
21  File.Delete(path);
```

FileShare визначає, що дозволяється іншим процесам (або потокам) робити з файлом, поки він відкритий у вас:

FileShare	Опис
None	Виключний доступ — ніхто інший не може відкрити файл
Read	Інші можуть відкривати для читання
Write	Інші можуть відкривати для запису
ReadWrite	Інші можуть відкривати для читання і запису
Delete	Інші можуть видаляти файл

Читання та запис по шматках — ефективна обробка великих файлів

Ключова перевага потокового доступу — обробка файлів будь-якого розміру при константному використанні пам'яті:

```
C#
 1  using System;
 2  using System.IO;
 3  using System.Text;
 4
 5  string srcPath  = Path.Combine(Path.GetTempPath(), "large_medical_data.bin");
 6  string dstPath  = Path.Combine(Path.GetTempPath(), "large_medical_copy.bin");
 7
 8  // Генеруємо тестовий файл ~50 KB
 9  using (FileStream gen = new FileStream(srcPath, FileMode.Create))
10  {
11      byte[] chunk = new byte[1024];
12      for (int i = 0; i < 50; i++)
13      {
14          // Заповнюємо імітаційними медичними даними
15          for (int j = 0; j < chunk.Length; j++)
16              chunk[j] = (byte)((i * 7 + j * 3) % 256);
17          gen.Write(chunk, 0, chunk.Length);
18      }
19  }
20
21  FileInfo srcFi = new FileInfo(srcPath);
22  Console.WriteLine($"Джерело: {srcFi.Length.ToString()} байт");
23
```

```
// Копіювання по шматках — використовує лише 4 КВ пам'яті незалежно від розміру файлу
const int BUFFER_SIZE = 4096;
byte[] buffer = new byte[BUFFER_SIZE];
int totalCopied = 0;

// using-блоки: файли закриються одразу після копіювання — до перевірки й повторного відкриття
using (FileStream src = File.OpenRead(srcPath))
using (FileStream dst = File.Create(dstPath))
{
    int bytesRead;
    while ((bytesRead = src.Read(buffer, 0, buffer.Length)) > 0)
    {
        dst.Write(buffer, 0, bytesRead);
        totalCopied += bytesRead;
    }
}

Console.WriteLine($"Скопійовано: {totalCopied.ToString()} байт");
Console.WriteLine($"Файли ідентичні: {(new FileInfo(dstPath).Length == srcFi.Length).ToString()}");

// Альтернатива: Stream.CopyTo — вбудований метод копіювання між потоками
using (FileStream src2 = File.OpenRead(srcPath))
using (FileStream dst2 = File.Create(dstPath))
{
    src2.CopyTo(dst2, bufferSize: 4096);
}
Console.WriteLine("CopyTo: теж по шматках, теж ефективно");

File.Delete(srcPath);
File.Delete(dstPath);
```

Зверніть увагу на using-блоки у прикладі. Оголошення using FileStream dst = ...; (без фігурних дужок) закриває потік лише в кінці всього методу. Якби тут стояли такі оголошення, спроба вдруге відкрити dstPath через File.Create завершилась би IOException («файл використовується іншим процесом»): FileStream за замовчуванням відкривається з FileShare.Read, тобто іншим дозволено лише читати. До того ж перевірка довжини незакритого файлу могла б показати не всі байти — частина ще лежала б у буфері. Тому потоки, з якими робота завершена, закривайте одразу — блоком using (...) { }.

Stream.CopyTo(Stream destination, int bufferSize) — вбудований метод для копіювання між будь-якими потоками: з файлу у файл, з файлу у мережу, з HTTP-відповіді у файл. Внутрішньо він теж читає по шматках вказаного розміру.

Seekable потік: довільний доступ до даних

```
C#
using System;
```

```
using System.IO;
using System.Text;

string path = Path.Combine(Path.GetTempPath(), "indexed_records.bin");

// Запишемо 5 «записів» фіксованого розміру (32 байти кожен)
const int RECORD_SIZE = 32;
string[] patients = { "Коваль М.А.    ", "Бойко О.П.     ", "Мороз В.І.     ",
                      "Петренко І.О.  ", "Руденко С.В.   " };

using (FileStream fs = new FileStream(path, FileMode.Create))
{
    foreach (string p in patients)
    {
        byte[] record = new byte[RECORD_SIZE];
        byte[] nameBytes = Encoding.UTF8.GetBytes(p.PadRight(RECORD_SIZE));
        Array.Copy(nameBytes, record, Math.Min(nameBytes.Length, RECORD_SIZE));
        fs.Write(record, 0, RECORD_SIZE);
    }
}

// Довільне читання: записи 4-й (індекс 3)
using (FileStream fs = new FileStream(path, FileMode.Open, FileAccess.Read))
{
    int targetIndex = 3;
    long offset = (long)targetIndex * RECORD_SIZE;

    fs.Seek(offset, SeekOrigin.Begin);
    byte[] buf = new byte[RECORD_SIZE];
    fs.Read(buf, 0, RECORD_SIZE);

    Console.WriteLine($"Запис #{targetIndex.ToString()}:
{Encoding.UTF8.GetString(buf).Trim()}");
    Console.WriteLine($"Прочитано без перебору {targetIndex.ToString()}
попередніх записів — O(1) доступ");
}

File.Delete(path);
```

Файлові потоки (CanSeek == true) підтримують **довільний доступ** — стрибок до будь-якої позиції за O(1). Це основа для баз даних фіксованих записів, індексних структур і форматів з заголовками.

MemoryStream — потік у пам'яті

MemoryStream реалізує Stream над масивом байтів у пам'яті. Не звертається до диску, але реалізує той самий інтерфейс. Корисний для тестування (замість реальних файлів), серіалізації у пам'яті та конвеєрів обробки:

```
C#
using System;
using System.IO;
using System.Text;

```

```
// MemoryStream — Stream у RAM
using MemoryStream ms = new MemoryStream();

// Записуємо у пам'ять через StreamWriter (текстовий запис у потік)
using (StreamWriter sw = new StreamWriter(ms, Encoding.UTF8, leaveOpen: true))
{
    sw.WriteLine("Пацієнт: Петренко І.О.");
    sw.WriteLine("Діагноз: J06.9");
    sw.Flush();
}

Console.WriteLine($"MemoryStream.Length = {ms.Length.ToString()} байт");

// Читаємо назад
ms.Seek(0, SeekOrigin.Begin);
using (StreamReader sr = new StreamReader(ms, Encoding.UTF8))
{
    Console.WriteLine("Вміст MemoryStream:");
    Console.WriteLine(sr.ReadToEnd());
}

// Отримати весь масив байтів
byte[] bytes = ms.ToArray();
Console.WriteLine($"ToArray(): {bytes.Length.ToString()} байт");

// Корисний паттерн: обробити у пам'яті, потім зберегти на диск
string outputPath = Path.Combine(Path.GetTempPath(), "from_memory.txt");
File.WriteAllBytes(outputPath, bytes);
Console.WriteLine($"Збережено на диск: {outputPath}");
File.Delete(outputPath);
```

leaveOpen: true у конструкторі StreamWriter — важливий параметр: без нього StreamWriter.Dispose() закриє і MemoryStream, і ми не зможемо прочитати дані після using-блоку.

Практичний сценарій: журнал медичних подій з FileStream

```
C#
using System;
using System.IO;
using System.Text;

// Бінарний журнал: кожен запис — фіксованого розміру
// Формат: [timestamp:8 байт][patientId:16 байт][eventCode:4 байт][checksum:4 байт] = 32 байти

string logPath = Path.Combine(Path.GetTempPath(), "medical_events.log");
const int ENTRY_SIZE = 32;

// Запис події
void LogEvent(FileStream log, long patientId, int eventCode)
{
    byte[] entry = new byte[ENTRY_SIZE];
    long ts = DateTimeOffset.UtcNow.ToUnixTimeSeconds();

    BitConverter.TryWriteBytes(entry.AsSpan(0,  8), ts);
```

```
    BitConverter.TryWriteBytes(entry.AsSpan(8,  8), patientId);
    BitConverter.TryWriteBytes(entry.AsSpan(16, 4), eventCode);

    int checksum = 0;
    for (int i = 0; i < 20; i++) checksum += entry[i];
    BitConverter.TryWriteBytes(entry.AsSpan(20, 4), checksum);

    log.Write(entry, 0, ENTRY_SIZE);
    log.Flush(); // Кожна подія одразу на диску
}

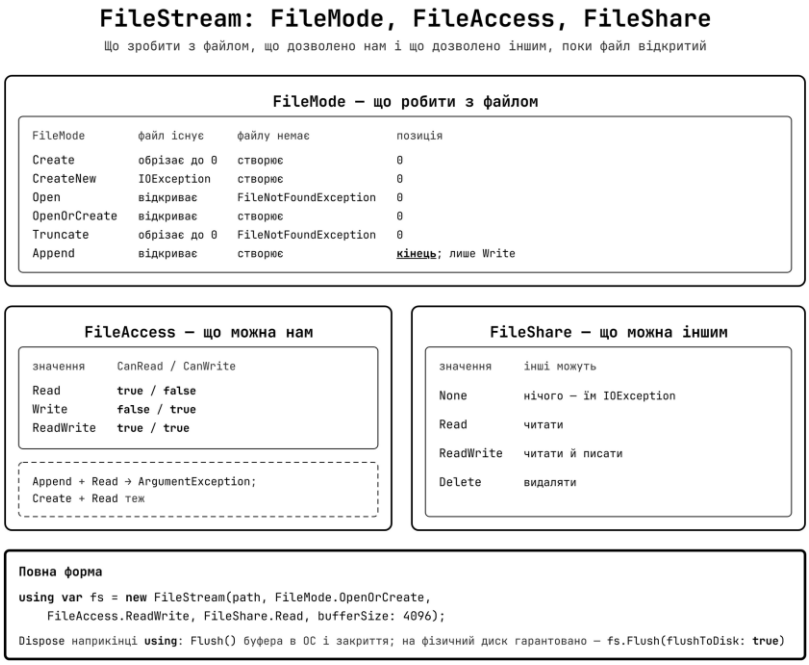
// Зберігаємо кілька подій
using (FileStream log = new FileStream(logPath, FileMode.Create,
FileAccess.Write, FileShare.Read))
{
    LogEvent(log, 1001L, 0x01); // пацієнт 1001, подія "прийом"
    LogEvent(log, 1002L, 0x01);
    LogEvent(log, 1001L, 0x02); // пацієнт 1001, подія "виписка"
    LogEvent(log, 1003L, 0x01);
}

// Читання журналу
FileInfo logFi = new FileInfo(logPath);
int entryCount = (int)(logFi.Length / ENTRY_SIZE);
Console.WriteLine($"Журнал: {entryCount.ToString()} записів,
{logFi.Length.ToString()} байт");

using (FileStream log = File.OpenRead(logPath))
{
    byte[] entry = new byte[ENTRY_SIZE];
    for (int i = 0; i < entryCount; i++)
    {
        log.Read(entry, 0, ENTRY_SIZE);
        long ts        = BitConverter.ToInt64(entry, 0);
        long patientId = BitConverter.ToInt64(entry, 8);
        int  eventCode = BitConverter.ToInt32(entry, 16);
        int  checksum  = BitConverter.ToInt32(entry, 20);

        string evtName = eventCode == 1 ? "прийом" : "виписка";
        Console.WriteLine($"  [{i.ToString()}] pid={patientId.ToString()},
подія={evtName}, ts={ts.ToString()}");
    }
}

File.Delete(logPath);
```

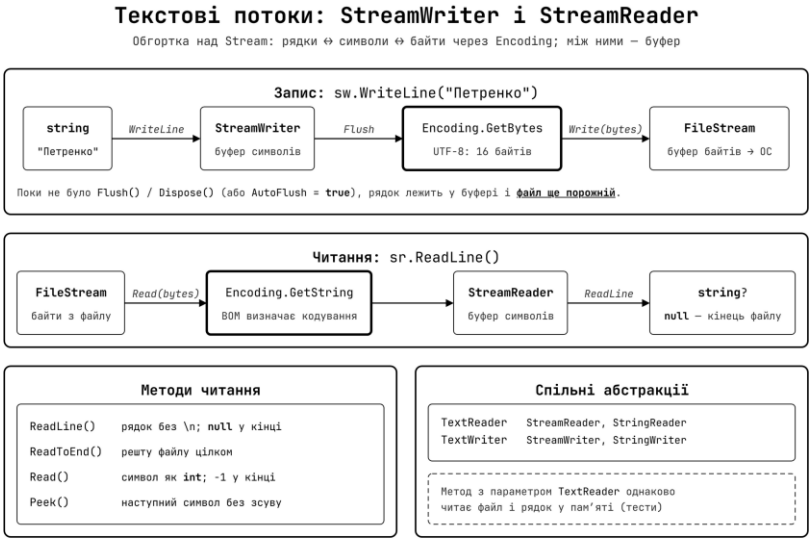


FileMode та FileAccess — параметри відкриття файлу

18.4. StreamReader та StreamWriter. Текстові потоки

FileStream із розділу 18.3 оперує **байтами** — він не знає нічого про текст, рядки, символи чи кодування. Якщо потрібно читати чи записувати текстові дані, щоразу виконувати ручне кодування/декодування байтів незручно і ненадійно. Саме для цього .NET надає StreamReader і StreamWriter — **текстові обгортки над потоком**, що прозоро обробляють кодування символів і надають зручний рядковий API.

Ключовий принцип: StreamReader / StreamWriter не є самостійними — вони **декорують** інший Stream (найчастіше FileStream). Це класичний патерн **Decorator**: обгортка додає нову поведінку (роботу з текстом), не змінюючи базову абстракцію потоку.



Потік текстових даних: StreamReader та StreamWriter

StreamWriter — запис тексту у потік

StreamWriter перетворює рядки на байти з урахуванням кодування і записує їх у базовий потік.

Базовий запис

```C#
using System;
using System.IO;
using System.Text;

string path = Path.Combine(Path.GetTempPath(), "patient_log.txt");

// Варіант 1: конструктор з рядком шляху (автоматично створює FileStream)
// Encoding.UTF8 передано явно — файл почнеться з BOM (EF BB BF). Без BOM: new UTF8Encoding(false)
using (StreamWriter sw = new StreamWriter(path, append: false, encoding: Encoding.UTF8))
{
    sw.WriteLine("=== Журнал прийомів пацієнтів ===");
    sw.WriteLine($"Дата: {DateTime.Now:yyyy-MM-dd}");
    sw.WriteLine();
    sw.Write("Пацієнт: ");       // Write без переносу рядка
    sw.WriteLine("Петренко І.О."); // WriteLine додає Environment.NewLine
    sw.WriteLine($"Час: {DateTime.Now:HH:mm}");
}
```

```
18
19  // Варіант 2: обгортка над FileStream (більший контроль над потоком)
20  using (FileStream fs = new FileStream(path, FileMode.Append, FileAccess.Write))
21  using (StreamWriter sw = new StreamWriter(fs, Encoding.UTF8))
22  {
23      sw.WriteLine();
24      sw.WriteLine("--- Другий прийом ---");
25      sw.WriteLine("Пацієнт: Бойко О.П.");
26      sw.Flush(); // примусово скинути буфер на базовий потік
27  }
28
29  string result = File.ReadAllText(path, Encoding.UTF8);
30  Console.WriteLine(result);
31  File.Delete(path);
```

Другий параметр конструктора append: false — якщо true, файл відкривається у режимі дозапису. Фактично це визначає FileMode: false → FileMode.Create, true → FileMode.Append.

AutoFlush та буферизація

```
C#
 1  using System;
 2  using System.IO;
 3  using System.Text;
 4
 5  string path = Path.Combine(Path.GetTempPath(), "clinic_events.log");
 6
 7  using StreamWriter sw = new StreamWriter(path, append: false, Encoding.UTF8);
 8
 9  // AutoFlush = true: кожен Write/WriteLine одразу передається у базовий потік
10  // Корисно для журналів, де важлива надійність запису
11  sw.AutoFlush = true;
12
13  for (int i = 1; i <= 4; i++)
14  {
15      sw.WriteLine($"[{DateTime.Now:HH:mm:ss}] Подія #{i.ToString()}:
    {GetEventDescription(i)}");
16      // З AutoFlush=true кожен рядок одразу на диску — не втратимо при краші
17  }
18
19  string GetEventDescription(int code) => code switch
20  {
21      1 => "пацієнт зареєстрований",
22      2 => "аналізи взяті",
23      3 => "лікар призначений",
24      _ => "виписка"
25  };
26
27  sw.Close(); // явний Close — аналог Dispose
28  Console.WriteLine(File.ReadAllText(path, Encoding.UTF8));
29  File.Delete(path);
```

За замовчуванням AutoFlush = false — дані накопичуються у внутрішньому буфері (зазвичай 4 KB) і записуються на диск при

заповненні буфера або при Flush()/Dispose(). Для журналів подій де критична надійність встановлюйте AutoFlush = true.

Кодування — як StreamWriter обробляє символи

```C#
using System;
using System.IO;
using System.Text;

string text = "Пацієнт: Петренко І.О. Діагноз: J06.9 — ГРВІ";

// Запишемо у різних кодуваннях
string utf8Path    = Path.Combine(Path.GetTempPath(), "enc_utf8.txt");
string utf16Path   = Path.Combine(Path.GetTempPath(), "enc_utf16.txt");
string asciiPath   = Path.Combine(Path.GetTempPath(), "enc_ascii.txt");

using (var sw = new StreamWriter(utf8Path,  false, Encoding.UTF8))
    sw.WriteLine(text);
using (var sw = new StreamWriter(utf16Path, false, Encoding.Unicode))
    sw.WriteLine(text);
using (var sw = new StreamWriter(asciiPath, false, Encoding.ASCII))
    sw.WriteLine(text);

long utf8Size  = new FileInfo(utf8Path).Length;
long utf16Size = new FileInfo(utf16Path).Length;
long asciiSize = new FileInfo(asciiPath).Length;

Console.WriteLine($"UTF-8:   {utf8Size.ToString()} байт  (1-4 байти/символ,
    кирилиця = 2 байти)");
Console.WriteLine($"UTF-16:  {utf16Size.ToString()} байт  (2 байти/символ + 2-
    байтний BOM)");
Console.WriteLine($"ASCII:   {asciiSize.ToString()} байт  (кирилиця = '?' — не
    підтримується)");

// Читання ASCII-файлу покаже знаки питання для кирилиці
string asciiRead = File.ReadAllText(asciiPath, Encoding.ASCII);
Console.WriteLine($"ASCII читання: {asciiRead}");

foreach (string p in new[] { utf8Path, utf16Path, asciiPath })
    File.Delete(p);
```

Рекомендація для нових застосунків: використовуйте UTF-8 — це стандарт де-факто для текстових файлів у сучасних системах. Тут є тонкість щодо BOM (Byte Order Mark — мітки кодування на початку файлу). Якщо кодування **не вказано** (new StreamWriter(path), File.WriteAllText(path, text)), .NET пише UTF-8 **без** BOM. Але статична властивість Encoding.UTF8 має преамбулу, тож при явній передачі Encoding.UTF8 у StreamWriter чи File.WriteAllText файл почнеться з трьох байтів EF BB BF. Більшість програм це розуміє, але деякі інструменти Linux і парсери CSV сприймають BOM як частину

першого поля. Тому для файлів без BOM передавайте new UTF8Encoding(false) або просто не вказуйте кодування.

StreamReader — читання тексту з потоку

StreamReader декодує байти з базового потоку у символи рядків. Надає методи для читання рядок за рядком, символ за символом або весь вміст одразу.

Базове читання

```
C#
 1  using System;
 2  using System.IO;
 3  using System.Text;
 4
 5  string path = Path.Combine(Path.GetTempPath(), "medical_records.txt");
 6  File.WriteAllText(path,
 7      "ID001|Петренко І.О.|J06.9|2024-03-15\n" +
 8      "ID002|Бойко О.П.|I10|2024-03-16\n" +
 9      "ID003|Коваль М.А.|E11.9|2024-03-17\n",
10      Encoding.UTF8);
11
12  // Варіант 1: з рядком шляху
13  using (StreamReader sr = new StreamReader(path, Encoding.UTF8))
14  {
15      // Метадані потоку
16      Console.WriteLine($"CurrentEncoding: {sr.CurrentEncoding.EncodingName}");
17      Console.WriteLine($"EndOfStream: {sr.EndOfStream.ToString()}");
18
19      // Читання рядок за рядком
20      string? line;
21      int lineNum = 0;
22      while ((line = sr.ReadLine()) != null)
23      {
24          lineNum++;
25          string[] parts = line.Split('|');
26          Console.WriteLine($"  [{lineNum.ToString()}] id={parts[0]},
    пацієнт={parts[1]}, діагноз={parts[2]}");
27      }
28
29      Console.WriteLine($"EndOfStream після читання:
    {sr.EndOfStream.ToString()}");
30  }
31
32  File.Delete(path);
```

Методи читання: ReadLine, Read, ReadToEnd, Peek

```
C#
 1  using System;
 2  using System.IO;
 3  using System.Text;
 4
```

```
string path = Path.Combine(Path.GetTempPath(), "read_methods_test.txt");
File.WriteAllText(path, "ABCDE\nFGHIJ\nKLMNO", Encoding.UTF8);

// Показуємо всі 4 методи читання
using (StreamReader sr = new StreamReader(path, Encoding.UTF8))
{
    // Peek: дивиться на наступний символ БЕЗ просування позиції
    int nextChar = sr.Peek();
    Console.WriteLine($"Peek(): '{(char)nextChar}' (позиція не змінилась)");

    // Read(): читає один символ і просуває позицію
    char c = (char)sr.Read();
    Console.WriteLine($"Read(): '{c.ToString()}' (позиція +1)");

    // Read(buffer, index, count): читає N символів у масив
    char[] buf = new char[3];
    int read = sr.Read(buf, 0, buf.Length);
    Console.WriteLine($"Read(buf, 0, 3): '{new string(buf, 0, read)}'
({read.ToString()} символів)");

    // ReadLine(): читає до кінця рядка (без '\n')
    string? line = sr.ReadLine(); // пропуск залишку 1-го рядка
    Console.WriteLine($"ReadLine(): '{line}'");  // "E" (залишок після "ABCD")

    string? line2 = sr.ReadLine();
    Console.WriteLine($"ReadLine(): '{line2}'"); // "FGHIJ"

    // ReadToEnd(): читає все від поточної позиції до кінця
    string rest = sr.ReadToEnd();
    Console.WriteLine($"ReadToEnd(): '{rest}'"); // "KLMNO"

    Console.WriteLine($"EndOfStream: {sr.EndOfStream.ToString()}");
}

File.Delete(path);
```

Peek() — без споживання символу. Це дозволяє перевірити, чи є ще дані, перш ніж читати: while (sr.Peek() != -1) { ... }. Повертає -1 на кінці потоку.

Обробка великих файлів рядок за рядком

```
C#
using System;
using System.IO;
using System.Text;

// Генеруємо великий тестовий файл — журнал пацієнтів
string logPath = Path.Combine(Path.GetTempPath(), "patients_log_large.txt");
using (StreamWriter gen = new StreamWriter(logPath, false, Encoding.UTF8))
{
    string[] diagnoses = { "J06.9", "I10", "E11.9", "K29.5", "M54.5" };
    for (int i = 1; i <= 1000; i++)
    {
        string date = DateTime.Now.AddMinutes(-i).ToString("yyyy-MM-dd HH:mm");
        string diag = diagnoses[i % diagnoses.Length];
```

```
14          gen.WriteLine($"{date}|PT{i.ToString("D4")}|{diag}|Лікар #{(i % 10 +
    1).ToString()}");
15      }
16  }
17
18  FileInfo logFi = new FileInfo(logPath);
19  Console.WriteLine($"Файл: {logFi.Length.ToString()} байт, ~1000 записів");
20
21  // Агрегація без завантаження всього файлу у пам'ять
22  int total = 0, j069Count = 0, i10Count = 0;
23
24  using (StreamReader sr = new StreamReader(logPath, Encoding.UTF8))
25  {
26      string? line;
27      while ((line = sr.ReadLine()) != null)
28      {
29          total++;
30          string[] parts = line.Split('|');
31          if (parts.Length > 2)
32          {
33              if (parts[2] == "J06.9") j069Count++;
34              if (parts[2] == "I10")   i10Count++;
35          }
36      }
37  }
38
39  Console.WriteLine($"Всього записів: {total.ToString()}");
40  Console.WriteLine($"J06.9 (ГРВІ): {j069Count.ToString()}");
41  Console.WriteLine($"I10 (гіпертонія): {i10Count.ToString()}");
42  Console.WriteLine("Файл оброблено рядок за рядком — пам'яті лише один рядок
    одночасно");
43
44  File.Delete(logPath);
```

ReadLine() — ідеальний вибір для обробки великих текстових файлів: тримає у пам'яті лише один рядок, обробляє гігабайтні файли без OutOfMemoryException.

StringReader та StringWriter — потоки над рядком

StringReader і StringWriter реалізують той самий текстовий інтерфейс, що StreamReader/StreamWriter, але над string або StringBuilder у пам'яті. Корисні для тестування та обробки рядкових даних без файлів:

```
C#
 1  using System;
 2  using System.IO;
 3  using System.Text;
 4
 5  // StringWriter — запис у StringBuilder
 6  var sb = new StringBuilder();
 7  using (StringWriter sw = new StringWriter(sb))
 8  {
 9      sw.WriteLine("Пацієнт: Бойко О.П.");
10      sw.WriteLine("Вік: 45");
```

```
    sw.WriteLine($"Дата: {DateTime.Now:yyyy-MM-dd}");
}
Console.WriteLine("StringWriter результат:");
Console.WriteLine(sb.ToString());

// StringReader — читання з рядка рядок за рядком
string input = "ID001|Петренко|J06.9\nID002|Бойко|I10\nID003|Коваль|E11.9";
int count = 0;
using (StringReader sr = new StringReader(input))
{
    string? line;
    while ((line = sr.ReadLine()) != null)
    {
        count++;
        string[] parts = line.Split('|');
        Console.WriteLine($"  [{count.ToString()}] {parts[1]} — {parts[2]}");
    }
}

// Корисний патерн: той самий метод обробляє і файл, і рядок
void ProcessLines(TextReader reader)
{
    string? line;
    int n = 0;
    while ((line = reader.ReadLine()) != null)
        Console.WriteLine($"    обробка рядка {++n}: {line.Split('|')[0]}");
}

Console.WriteLine("\nОдин метод — різні джерела:");
using (StringReader sr = new StringReader(input)) ProcessLines(sr);  // рядок у пам'яті
// ProcessLines(new StreamReader(path));                               // файл на диску
```

TextReader і TextWriter — абстрактні базові класи для StreamReader/StringReader та StreamWriter/StringWriter відповідно. Код, написаний проти TextReader, однаково працює з файлом, рядком або будь-яким іншим текстовим джерелом.

Практичний сценарій: CSV-парсинг медичних записів

C#
```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;

// CSV-файл з результатами аналізів
string csvPath = Path.Combine(Path.GetTempPath(), "lab_results.csv");
using (StreamWriter sw = new StreamWriter(csvPath, false, Encoding.UTF8))
{
    sw.WriteLine("patient_id,test_name,value,unit,status");
    sw.WriteLine("PT001,Гемоглобін,135.0,г/л,норма");
    sw.WriteLine("PT001,Лейкоцити,6.2,10^9/л,норма");
    sw.WriteLine("PT002,Глюкоза,7.8,ммоль/л,вище норми");
    sw.WriteLine("PT002,Холестерин,5.1,ммоль/л,норма");
```

```
    sw.WriteLine("PT003,Гемоглобін,98.0,г/л,нижче норми");
}

// Парсинг CSV
List<LabResult> ParseLabCsv(string path)
{
    var results = new List<LabResult>();
    using StreamReader sr = new StreamReader(path, Encoding.UTF8);

    string? header = sr.ReadLine(); // пропускаємо заголовок
    Console.WriteLine($"Заголовок: {header}");

    string? line;
    while ((line = sr.ReadLine()) != null)
    {
        if (string.IsNullOrWhiteSpace(line)) continue;

        string[] parts = line.Split(',');
        if (parts.Length < 5) continue;

        if (!double.TryParse(parts[2], System.Globalization.NumberStyles.Float,
            System.Globalization.CultureInfo.InvariantCulture, out double val))
            continue;

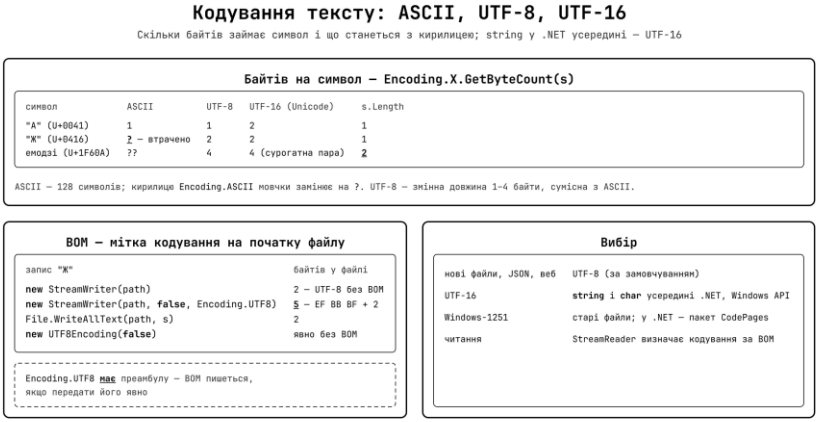
        results.Add(new LabResult(parts[0], parts[1], val, parts[3],
parts[4]));
    }
    return results;
}

List<LabResult> results = ParseLabCsv(csvPath);
Console.WriteLine($"\nЗчитано {results.Count.ToString()} записів:");
foreach (LabResult r in results)
{
    string icon = r.Status == "норма" ? "[OK]" : "[!!]";
    Console.WriteLine($"  {icon} {r.PatientId} | {r.TestName}:
{r.Value.ToString()} {r.Unit} — {r.Status}");
}

// Формуємо звіт — відхилення від норми
Console.WriteLine("\n--- Відхилення від норми ---");
foreach (LabResult r in results)
    if (r.Status != "норма")
        Console.WriteLine($"  {r.PatientId}: {r.TestName} =
{r.Value.ToString()} {r.Unit} [{r.Status}]");

File.Delete(csvPath);

// Структура запису лабораторного аналізу
record LabResult(string PatientId, string TestName, double Value, string Unit,
string Status);
```



Кодування символів у текстових потоках

18.5. BinaryReader та BinaryWriter. Бінарні потоки

StreamReader/StreamWriter чудово підходять для текстових даних. Але медичні системи часто мають справу з даними зовсім іншої природи: показники датчиків — послідовності чисел з плаваючою комою, архіви кардіограм — тисячі 16-бітних відліків, медичні зображення — мільйони байтів з пікселями. Зберігати такі дані у текстовому форматі неефективно: рядок "98.6" займає 4 байти (символи), тоді як значення float 98.6f у бінарному форматі займає рівно 4 байти **і зчитується без конвертації**. Для роботи з типізованими бінарними даними C# надає BinaryWriter та BinaryReader.

BinaryWriter і BinaryReader — теж **декоратори над Stream** (як StreamWriter/StreamReader), але вони оперують не рядками, а C#-типами безпосередньо: int, double, bool, string, byte[]. Дані записуються у внутрішньому представленні .NET (little-endian для чисел) без будь-якого текстового кодування.



Бінарний vs текстовий формат зберігання

BinaryWriter — запис типізованих даних

```C#
using System;
using System.IO;
using System.Text;

string path = Path.Combine(Path.GetTempPath(), "patient_vitals.bin");

// BinaryWriter обгортає будь-який Stream
using (BinaryWriter bw = new BinaryWriter(File.Create(path), Encoding.UTF8))
{
    // Write() перевантажений для кожного примітивного типу
    bw.Write((int)1001);           // 4 байти — id пацієнта
    bw.Write("Петренко І.О.");     // string: довжина + UTF-8 байти
    bw.Write((byte)45);            // 1 байт — вік
    bw.Write(true);                // 1 байт — чи госпіталізований
    bw.Write(98.6f);               // 4 байти — температура (float)
    bw.Write(120.0 / 80.0);        // 8 байт — індекс АТ (double)
    bw.Write((short)72);           // 2 байти — пульс (short)

    // byte[] — записується з довжиною вперед
    byte[] ecgSample = { 0x12, 0x34, 0x56, 0x78, 0x9A };
    bw.Write(ecgSample.Length); // спочатку довжина
    bw.Write(ecgSample);        // потім дані
}

FileInfo fi = new FileInfo(path);
Console.WriteLine($"Бінарний файл: {fi.Length.ToString()} байт");

// Для порівняння: той самий пацієнт у текстовому CSV
string csv = "1001,Петренко І.О.,45,true,98.6,1.5,72,12345678A";
```

```
30  Console.WriteLine($"CSV рядок:     {Encoding.UTF8.GetByteCount(csv).ToString()}
    байт");
31
32  File.Delete(path);
```

Метод Write у BinaryWriter перевантажений для всіх примітивних типів C#: bool, byte, sbyte, char, short, ushort, int, uint, long, ulong, float, double, decimal, string, а також byte[] і char[]. Кожен тип записується у точній кількості байт свого розміру.

BinaryReader — читання типізованих даних

```
C#
 1  using System;
 2  using System.IO;
 3  using System.Text;
 4
 5  string path = Path.Combine(Path.GetTempPath(), "vitals_demo.bin");
 6
 7  // Спочатку запишемо дані
 8  using (BinaryWriter bw = new BinaryWriter(File.Create(path), Encoding.UTF8))
 9  {
10      bw.Write(1001);
11      bw.Write("Петренко І.О.");
12      bw.Write((byte)45);
13      bw.Write(true);
14      bw.Write(98.6f);
15      bw.Write(120.0 / 80.0);
16      bw.Write((short)72);
17      byte[] ecg = { 0x12, 0x34, 0x56, 0x78, 0x9A };
18      bw.Write(ecg.Length);
19      bw.Write(ecg);
20  }
21
22  // Читаємо — ПОРЯДОК ЧИТАННЯ ПОВИНЕН ЗБІГАТИСЯ З ПОРЯДКОМ ЗАПИСУ
23  using (BinaryReader br = new BinaryReader(File.OpenRead(path), Encoding.UTF8))
24  {
25      int    patientId  = br.ReadInt32();
26      string name       = br.ReadString();
27      byte   age        = br.ReadByte();
28      bool   hospitalized = br.ReadBoolean();
29      float  temp       = br.ReadSingle();    // float = Single
30      double bpIndex    = br.ReadDouble();
31      short  pulse      = br.ReadInt16();     // short = Int16
32      int    ecgLength  = br.ReadInt32();
33      byte[] ecgSample  = br.ReadBytes(ecgLength);
34
35      Console.WriteLine($"ID:            {patientId.ToString()}");
36      Console.WriteLine($"Ім'я:          {name}");
37      Console.WriteLine($"Вік:           {age.ToString()}");
38      Console.WriteLine($"Госпіт.:       {hospitalized.ToString()}");
39      Console.WriteLine($"Температура:   {temp.ToString("F1")} °C");
40      Console.WriteLine($"Індекс АТ:     {bpIndex.ToString("F3")}");
41      Console.WriteLine($"Пульс:         {pulse.ToString()} уд/хв");
42      Console.Write($"ЕКГ ({ecgLength.ToString()} байт): ");
43      foreach (byte b in ecgSample) Console.Write($"{b.ToString("X2")} ");
```

```
    Console.WriteLine("");

    // Перевірка: CanRead після кінця файлу
    Console.WriteLine($"\nПрочитано до кінця: {(br.BaseStream.Position ==
br.BaseStream.Length).ToString()}");
}

File.Delete(path);
```

Критично важливо: порядок читання методами (ReadInt32, ReadString тощо) повинен **точно відповідати** порядку запису методами Write. Якщо порядок порушений — прочитаємо «сміття» або отримаємо виняток. Бінарний формат — це неявна схема даних, яку потрібно явно документувати.

BinaryReader — методи читання

```
C#
using System;
using System.IO;

string path = Path.Combine(Path.GetTempPath(), "all_types.bin");

// Запис усіх основних типів
using (BinaryWriter bw = new BinaryWriter(File.Create(path)))
{
    bw.Write(true);         // bool   — 1 байт
    bw.Write((byte)255);    // byte   — 1 байт
    bw.Write((short)1000);  // short  — 2 байти
    bw.Write(42);           // int    — 4 байти
    bw.Write(100L);         // long   — 8 байт
    bw.Write(3.14f);        // float  — 4 байти
    bw.Write(2.718281828);  // double — 8 байт
    bw.Write('A');          // char   — 1 байт (UTF-8: 'A'; кирилиця — 2)
    bw.Write("Hello");      // string — 1(len) + 5 байт
}

long fileSize = new FileInfo(path).Length;
Console.WriteLine($"Файл: {fileSize.ToString()} байт");

// Читання з відповідними методами
using (BinaryReader br = new BinaryReader(File.OpenRead(path)))
{
    Console.WriteLine($"bool:   {br.ReadBoolean().ToString()}  (1 байт)");
    Console.WriteLine($"byte:   {br.ReadByte().ToString()}  (1 байт)");
    Console.WriteLine($"short:  {br.ReadInt16().ToString()}  (2 байти)");
    Console.WriteLine($"int:    {br.ReadInt32().ToString()}  (4 байти)");
    Console.WriteLine($"long:   {br.ReadInt64().ToString()}  (8 байт)");
    Console.WriteLine($"float:  {br.ReadSingle().ToString("F5")}  (4 байти)");
    Console.WriteLine($"double: {br.ReadDouble().ToString("F9")}  (8 байт)");
    Console.WriteLine($"char:   {br.ReadChar().ToString()}  (2 байти)");
    Console.WriteLine($"string: {br.ReadString()}  (1+5 байт)");
}

File.Delete(path);
```

Тип C#	Метод BinaryReader	Розмір
bool	ReadBoolean()	1 байт
byte	ReadByte()	1 байт
sbyte	ReadSByte()	1 байт
char	ReadChar()	1–3 байти — за кодуванням BinaryWriter (UTF-8 за замовчуванням)
short	ReadInt16()	2 байти
ushort	ReadUInt16()	2 байти
int	ReadInt32()	4 байти
uint	ReadUInt32()	4 байти
long	ReadInt64()	8 байт
float	ReadSingle()	4 байти
double	ReadDouble()	8 байт
string	ReadString()	7-бітна довжина + байти
byte[]	ReadBytes(count)	count байт

Файл з фіксованими записами — індексований доступ

Бінарні файли з фіксованим розміром запису забезпечують O(1) доступ до будь-якого елемента через Seek:

```
C#
 1  using System;
 2  using System.IO;
 3  using System.Text;
 4
 5  // Структура: id(4) + name(32) + glucose(8) + pulse(2) = 46 байтів на запис
 6  const int RECORD_SIZE = 46;
 7  const int NAME_SIZE   = 32; // у БАЙТАХ: кирилиця в UTF-8 займає 2 байти на
    літеру
 8
 9  string path = Path.Combine(Path.GetTempPath(), "vitals_index.bin");
10
11  // Записуємо кілька пацієнтів
```

```
var records = new (int id, string name, double glucose, short pulse)[]
{
    (1001, "Коваль М.А.",     5.1, 72),
    (1002, "Бойко О.П.",      8.7, 145),
    (1003, "Мороз В.І.",      4.8, 68),
    (1004, "Петренко І.О.",  12.4, 110),
    (1005, "Руденко С.В.",    5.5, 78),
};

using (BinaryWriter bw = new BinaryWriter(File.Create(path), Encoding.UTF8))
{
    foreach (var (id, name, glucose, pulse) in records)
    {
        bw.Write(id);

        // Ім'я фіксованої ширини в байтах: UTF-8 байти імені, решта — нулі
        byte[] nameBytes = new byte[NAME_SIZE];
        byte[] encoded   = Encoding.UTF8.GetBytes(name);
        Array.Copy(encoded, nameBytes, Math.Min(encoded.Length, NAME_SIZE));
        bw.Write(nameBytes);

        bw.Write(glucose);
        bw.Write(pulse);
    }
}

// Читання конкретного запису по індексу — O(1)
int targetIndex = 3; // 4-й пацієнт (0-based)

using (BinaryReader br = new BinaryReader(File.OpenRead(path), Encoding.UTF8))
{
    long offset = (long)targetIndex * RECORD_SIZE;
    br.BaseStream.Seek(offset, SeekOrigin.Begin);

    int    id      = br.ReadInt32();
    string name    =
Encoding.UTF8.GetString(br.ReadBytes(NAME_SIZE)).TrimEnd('\0');
    double glucose = br.ReadDouble();
    short  pulse   = br.ReadInt16();

    Console.WriteLine($"Запис [{targetIndex.ToString()}]:");
    Console.WriteLine($"  ID:      {id.ToString()}");
    Console.WriteLine($"  Ім'я:    {name}");
    Console.WriteLine($"  Глюкоза: {glucose.ToString("F1")} ммоль/л");
    Console.WriteLine($"  Пульс:   {pulse.ToString()} уд/хв");

    // Читаємо всі записи
    Console.WriteLine("\nВсі пацієнти:");
    br.BaseStream.Seek(0, SeekOrigin.Begin);
    int count = (int)(br.BaseStream.Length / RECORD_SIZE);
    for (int i = 0; i < count; i++)
    {
        int    rid    = br.ReadInt32();
        string rname  =
Encoding.UTF8.GetString(br.ReadBytes(NAME_SIZE)).TrimEnd('\0');
        double rglu   = br.ReadDouble();
        short  rpulse = br.ReadInt16();
        string alert  = (rglu > 7.0 || rpulse > 100) ? " [!]" : "";
        Console.WriteLine($"  {rid.ToString()} | {rname,-18} |
глюк.={rglu.ToString("F1")} | пульс={rpulse.ToString()}{alert}");
```

```
    }
}

File.Delete(path);
```

Зверніть увагу на ширину поля імені. Попередня версія цього прикладу доповнювала ім'я пробілами до 20 **символів** (PadRight(20)) і копіювала перші 20 **байтів** UTF-8. Для латиниці це те саме, але кожна кирилична літера в UTF-8 займає 2 байти, тож «Петренко І.О.» (30 байтів) обрізалося до «Петренко І.», а розріз міг припасти навіть на середину літери. Правило для бінарних форматів: ширину текстового поля задають у байтах, з запасом на найдовше ім'я, а невикористані байти заповнюють нулями.

Практичний сценарій: бінарний архів ЕКГ

Реальний приклад застосування BinaryWriter/BinaryReader — зберігання даних електрокардіограми:

```C#
using System;
using System.IO;
using System.Text;

string ecgPath = Path.Combine(Path.GetTempPath(), "ecg_archive.ecg");

// Структура файлу ЕКГ:
// Заголовок: версія(4) + patientId(4) + sampleRate(4) + channelCount(4) + timestamp(8) = 24 байти
// Потім: samples_count(4) + float[] samples

void WriteEcgFile(string path, int patientId, int sampleRate, float[] samples)
{
    using BinaryWriter bw = new BinaryWriter(File.Create(path));

    // Заголовок
    bw.Write(1);                                    // версія формату
    bw.Write(patientId);                            // id пацієнта
    bw.Write(sampleRate);                           // частота дискретизації (Гц)
    bw.Write(1);                                    // кількість каналів
    bw.Write(DateTimeOffset.UtcNow.ToUnixTimeSeconds()); // timestamp

    // Дані
    bw.Write(samples.Length);
    foreach (float s in samples)
        bw.Write(s);
}

(int patientId, float[] samples, string label)[] ReadEcgFile(string path)
{
    using BinaryReader br = new BinaryReader(File.OpenRead(path));

```

```
    int version     = br.ReadInt32();
    int patientId   = br.ReadInt32();
    int sampleRate  = br.ReadInt32();
    int channels    = br.ReadInt32();
    long timestamp  = br.ReadInt64();
    int  sampleCount = br.ReadInt32();

    float[] samples = new float[sampleCount];
    for (int i = 0; i < sampleCount; i++)
        samples[i] = br.ReadSingle();

    return new[] { (patientId, samples, $"v{version.ToString()}
sr={sampleRate.ToString()}Hz ch={channels.ToString()}") };
}

// Генеруємо симульований ЕКГ-сигнал (250 Гц, 1 секунда = 250 відліків)
const int SAMPLE_RATE = 250;
float[] ecgData = new float[SAMPLE_RATE];
for (int i = 0; i < SAMPLE_RATE; i++)
{
    double t = (double)i / SAMPLE_RATE;
    ecgData[i] = (float)(Math.Sin(2 * Math.PI * 1.2 * t) + 0.3 * Math.Sin(2 *
Math.PI * 6 * t));
}

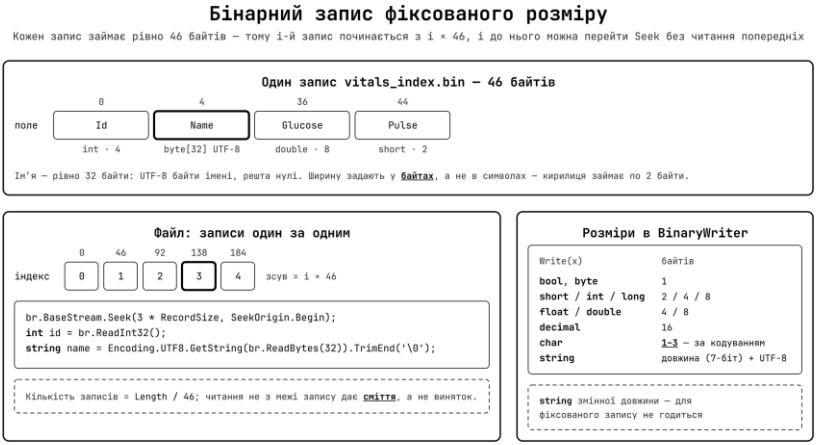
WriteEcgFile(ecgPath, 1007, SAMPLE_RATE, ecgData);

FileInfo ecgFi = new FileInfo(ecgPath);
int headerSize   = 24;
int samplesBytes = 4 * SAMPLE_RATE + 4; // 4 байт/float + 4 байти для count
Console.WriteLine($"ЕКГ файл: {ecgFi.Length.ToString()} байт");
Console.WriteLine($"  Заголовок: {headerSize.ToString()} байт");
Console.WriteLine($"  Дані:      {samplesBytes.ToString()} байт
({SAMPLE_RATE.ToString()} відліків x 4 байти)");

// Порівняємо з текстовим форматом
string csvLine = string.Join(",", Array.ConvertAll(ecgData, x =>
x.ToString("F6")));
Console.WriteLine($"\nТе саме у CSV:
{Encoding.UTF8.GetByteCount(csvLine).ToString()} байт");
Console.WriteLine($"Стиснення: {((ecgFi.Length * 100.0) /
Encoding.UTF8.GetByteCount(csvLine)).ToString("F0")}% від CSV-розміру");

var read = ReadEcgFile(ecgPath)[0];
Console.WriteLine($"\nЗчитано: patientId={read.patientId.ToString()},
{read.samples.Length.ToString()} відліків, {read.label}");
Console.WriteLine($"Перші 5 відліків: {string.Join(", ",
Array.ConvertAll(read.samples[..5], x => x.ToString("F3")))}");

File.Delete(ecgPath);
```



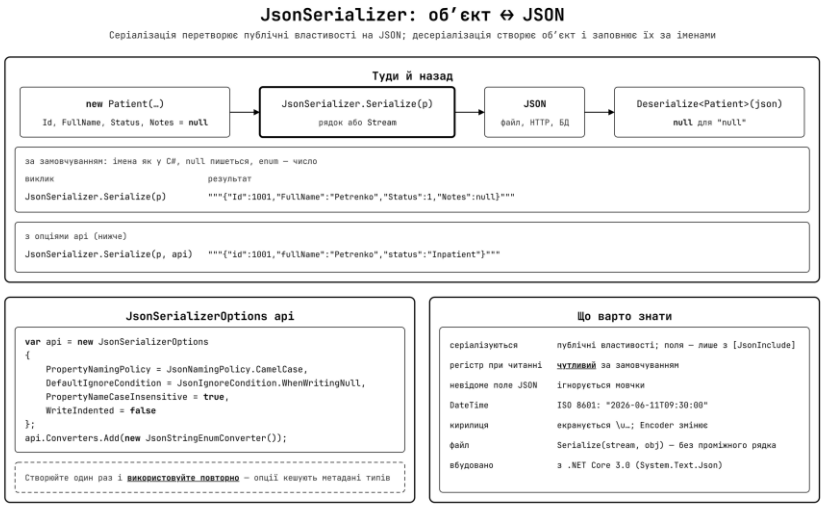
Структура бінарного запису фіксованого розміру

18.6. System.Text.Json. Серіалізація та десеріалізація

Текстові та бінарні потоки з попередніх розділів вирішують питання *як* зберегти байти. Але сучасний застосунок рідко оперує сирими байтами — він оперує **об'єктами**: пацієнтами, аналізами, рецептами, планами лікування. Зберегти об'єкт C# у файл або передати через мережу означає його **серіалізувати** — перетворити граф об'єктів у послідовність байтів. Зворотний процес — **десеріалізація**: відновити об'єкти зі збереженого представлення.

Найпоширеніший формат для серіалізації у сучасних застосунках — **JSON** (JavaScript Object Notation). Він людиночитабельний, підтримується всіма мовами і платформами, і є стандартом де-факто для REST API, конфігурацій та обміну даними між сервісами.

Починаючи з .NET Core 3.0, платформа включає System.Text.Json — **вбудовану бібліотеку серіалізації JSON**, оптимізовану для продуктивності. Вона замінила сторонній Newtonsoft.Json (Json.NET) як основний інструмент для більшості задач.



Потік JSON-серіалізації в System.Text.Json

JsonSerializer — базова серіалізація та десеріалізація

Центральний клас бібліотеки — JsonSerializer. Два основних методи: Serialize (об'єкт → рядок JSON) і Deserialize (рядок JSON → об'єкт).

```
C#
using System;
using System.Text.Json;

// Серіалізація: об'єкт → JSON рядок
var card = new PatientCard
{
    Id           = 1001,
    FullName     = "Петренко Іван Олексійович",
    Age          = 45,
    Diagnosis    = "J06.9",
    Hospitalized = false,
    Glucose      = 5.1
};

string json = JsonSerializer.Serialize(card);
Console.WriteLine("JSON (compact):");
Console.WriteLine(json);

// Відформатований JSON з відступами
var options = new JsonSerializerOptions { WriteIndented = true };
string prettyJson = JsonSerializer.Serialize(card, options);
Console.WriteLine("\nJSON (indented):");
Console.WriteLine(prettyJson);

```

```
// Десеріалізація: JSON рядок → об'єкт
PatientCard? restored = JsonSerializer.Deserialize<PatientCard>(json);
Console.WriteLine($"\nВідновлено: {restored?.FullName},
діагноз={restored?.Diagnosis}, вік={restored?.Age.ToString()}");

// Моделі даних медичної картки
class PatientCard
{
    public int    Id          { get; set; }
    public string FullName    { get; set; } = "";
    public int    Age         { get; set; }
    public string Diagnosis   { get; set; } = "";
    public bool   Hospitalized { get; set; }
    public double Glucose     { get; set; }
}
```

Зверніть увагу на поведінку за замовчуванням: імена властивостей записуються так само, як у C# (PascalCase), значення null включаються, enum записується числом, а при читанні імена властивостей **чутливі до регістру** (JSON {"id":1} не заповнить властивість Id). Кирилиця за замовчуванням екранується як \u0416 — це коректний JSON, але для людського читання задають Encoder = JavaScriptEncoder.UnsafeRelaxedJsonEscaping (або JavaScriptEncoder.Create(UnicodeRanges.All)). Усе це змінюється через JsonSerializerOptions нижче.

Ще одна особливість .NET 10: у файлових застосунках (dotnet run app.cs) за замовчуванням увімкнено Native AOT, а з ним вимкнено серіалізацію через рефлексію — виклик JsonSerializer.Serialize(obj) кине InvalidOperationException. Для таких скриптів додайте директиву #:property PublishAot=false першим рядком або використовуйте генератор вихідного коду JsonSerializerContext. У звичайних проєктах .csproj цього обмеження немає.

JsonSerializer.Serialize без опцій генерує компактний JSON без пробілів — це оптимально для мережевої передачі. WriteIndented = true додає відступи для читабельності — зручно для конфігурацій і логів.

Серіалізація до файлу та з файлу

Найчастіший сценарій — збереження об'єктів у файл і завантаження з файлу. JsonSerializer інтегрується зі Stream:

```
C#
using System;
using System.Collections.Generic;
using System.IO;
using System.Text.Json;

string filePath = Path.Combine(Path.GetTempPath(), "med_records.json");
```

```
 7
 8  var records = new List<MedRecord>
 9  {
10      new() { PatientId=1001, PatientName="Петренко І.О.", TestName="Гемоглобін",
    Value=135.0, Unit="г/л",     Status="норма",
    RecordedAt=DateTime.Now.AddHours(-2) },
11      new() { PatientId=1001, PatientName="Петренко І.О.", TestName="Глюкоза",
    Value=5.1,   Unit="ммоль/л", Status="норма",
    RecordedAt=DateTime.Now.AddHours(-2) },
12      new() { PatientId=1002, PatientName="Бойко О.П.",    TestName="Глюкоза",
    Value=8.7,   Unit="ммоль/л", Status="вище норми",
    RecordedAt=DateTime.Now.AddHours(-1) },
13      new() { PatientId=1003, PatientName="Мороз В.І.",    TestName="Гемоглобін",
    Value=98.0,  Unit="г/л",     Status="нижче норми", RecordedAt=DateTime.Now },
14  };
15
16  var options = new JsonSerializerOptions
17  {
18      WriteIndented = true,
19      // Серіалізація DateTime у ISO 8601
20  };
21
22  // Запис у файл через Stream — ефективніше ніж Serialize → string →
    WriteAllText
23  using (FileStream fs = File.Create(filePath))
24  {
25      JsonSerializer.Serialize(fs, records, options);
26  }
27
28  FileInfo fi = new FileInfo(filePath);
29  Console.WriteLine($"Записано: {fi.Length.ToString()} байт");
30
31  // Читання з файлу
32  List<MedRecord>? loaded;
33  using (FileStream fs = File.OpenRead(filePath))
34  {
35      loaded = JsonSerializer.Deserialize<List<MedRecord>>(fs);
36  }
37
38  Console.WriteLine($"Завантажено: {loaded?.Count.ToString()} записів");
39  if (loaded != null)
40  {
41      foreach (MedRecord r in loaded)
42      {
43          string icon = r.Status == "норма" ? "[OK]" : "[!!]";
44          Console.WriteLine($"  {icon} {r.PatientName} | {r.TestName}:
    {r.Value.ToString()} {r.Unit} ({r.Status})");
45      }
46  }
47
48  File.Delete(filePath);
49
50  class MedRecord
51  {
52      public int      PatientId  { get; set; }
53      public string   PatientName { get; set; } = "";
54      public string   TestName   { get; set; } = "";
55      public double   Value      { get; set; }
56      public string   Unit       { get; set; } = "";
57      public string   Status     { get; set; } = "";
```

```
    public DateTime RecordedAt { get; set; }
}
```

Передача Stream у JsonSerializer.Serialize/Deserialize ефективніша за string-варіант: бібліотека пише напряму у потік без проміжного рядкового буфера — важливо для великих об'єктів.

JsonSerializerOptions — налаштування серіалізації

JsonSerializerOptions — центральний об'єкт конфігурації, що контролює практично всі аспекти серіалізації:

```C#
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

var vitals = new PatientVitals
{
    PatientId     = 1001,
    fullName      = "Петренко І.О.",
    BloodPressure = 120.0 / 80.0,
    HeartRate     = 72,
    Notes         = null
};

// За замовчуванням: PascalCase, null включається
Console.WriteLine("=== Стандартні опції ===");
Console.WriteLine(JsonSerializer.Serialize(vitals, new JsonSerializerOptions { WriteIndented = true }));

// camelCase назви полів (стандарт JavaScript/REST)
Console.WriteLine("\n=== camelCase ===");
var camelOptions = new JsonSerializerOptions
{
    WriteIndented      = true,
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
};
Console.WriteLine(JsonSerializer.Serialize(vitals, camelOptions));

// Ігнорування null-значень
Console.WriteLine("\n=== Без null полів ===");
var noNullOptions = new JsonSerializerOptions
{
    WriteIndented      = true,
    DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull,
};
Console.WriteLine(JsonSerializer.Serialize(vitals, noNullOptions));

// Десеріалізація нечутлива до регістру
string json = """{"patientid":1002,"fullname":"Бойко О.П.","bloodpressure":1.6,"heartrate":145,"notes":null}""";
var relaxedOptions = new JsonSerializerOptions { PropertyNameCaseInsensitive = true };
PatientVitals? loaded = JsonSerializer.Deserialize<PatientVitals>(json, relaxedOptions);
```

```
Console.WriteLine($"\nНечутливо до регістру: {loaded?.fullName}, пульс={loaded?.HeartRate.ToString()}");

class PatientVitals
{
    public int    PatientId    { get; set; }
    public string fullName     { get; set; } = "";  // camelCase назва
    public double BloodPressure { get; set; }
    public int    HeartRate    { get; set; }
    public string? Notes       { get; set; }        // null-able
}
```

Типові JsonSerializerOptions для медичного застосунку

```
C#
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

// Рекомендований набір опцій для REST API / зберігання
var standardOptions = new JsonSerializerOptions
{
    WriteIndented               = false,               // компактний для API/файлів
    PropertyNamingPolicy        = JsonNamingPolicy.CamelCase, // camelCase для JS
    PropertyNameCaseInsensitive = true,                // гнучке читання
    DefaultIgnoreCondition      = JsonIgnoreCondition.WhenWritingNull, // без null
    // Enum як рядки замість чисел
    Converters = { new JsonStringEnumConverter() }
};

var apt = new Appointment { Id = 5, Doctor = "Коваленко О.П.", Status = PatientStatus.InTreatment, Room = null };
string json = JsonSerializer.Serialize(apt, standardOptions);
Console.WriteLine("Стандартний API JSON:");
Console.WriteLine(json);

// Десеріалізація
Appointment? loaded = JsonSerializer.Deserialize<Appointment>(json, standardOptions);
Console.WriteLine($"\nПовернуто: id={loaded?.Id.ToString()}, лікар={loaded?.Doctor}, статус={loaded?.Status}");

// Використовуємо enum
enum PatientStatus { Active, Discharged, InTreatment }

class Appointment
{
    public int           Id     { get; set; }
    public string        Doctor { get; set; } = "";
    public PatientStatus Status { get; set; }
    public string?       Room   { get; set; }
}
```

Атрибути керування серіалізацією

System.Text.Json надає атрибути для тонкого контролю над кожним полем і класом:

```C#
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

var record = new DiagnosisRecord
{
    PatientId     = 1001,
    DiagnosisCode = "J06.9",
    InternalNotes = "ці дані НЕ збережуться у JSON",
    PatientName   = "Петренко І.О.",
    RecordedAt    = DateTime.Now,
    Ward          = "Терапія",
    OptionalValue = null
};

string json = JsonSerializer.Serialize(record, new JsonSerializerOptions { WriteIndented = true });
Console.WriteLine("JSON з атрибутами:");
Console.WriteLine(json);

// Перевіряємо: InternalNotes відсутній у JSON
Console.WriteLine($"\nПоле 'InternalNotes' у JSON: {json.Contains("InternalNotes").ToString()} (має бути False)");
Console.WriteLine($"Поле 'patient_id' у JSON: {json.Contains("patient_id").ToString()} (має бути True)");

class DiagnosisRecord
{
    [JsonPropertyName("patient_id")]   // Перейменування поля у JSON
    public int PatientId { get; set; }

    [JsonPropertyName("diagnosis_code")]
    public string DiagnosisCode { get; set; } = "";

    [JsonIgnore]                       // Поле не включається у JSON
    public string InternalNotes { get; set; } = "внутрішні примітки";

    [JsonPropertyOrder(1)]             // Порядок полів у JSON
    public string PatientName { get; set; } = "";

    [JsonPropertyOrder(2)]
    public DateTime RecordedAt { get; set; }

    [JsonInclude]                      // Включає публічне поле (не тільки property)
    public string Ward = "";

    [JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingDefault)]
    public double? OptionalValue { get; set; } // null або 0 — не записується
}
```

Атрибут	Призначення
[JsonPropertyName("name")]	Задає ім'я поля у JSON
[JsonIgnore]	Виключає поле з серіалізації
[JsonIgnore(Condition = ...)]	Умовне виключення (null, default)
[JsonPropertyOrder(n)]	Порядок полів у виводі
[JsonInclude]	Включає публічне поле (field, не property)
[JsonRequired]	Поле обов'язкове при десеріалізації
[JsonConverter(typeof(...))]	Кастомний конвертер для типу

Робота з вкладеними структурами та колекціями

C#

```
using System;
using System.Collections.Generic;
using System.Text.Json;

var patient = new Patient
{
    Id      = 1001,
    Name    = "Петренко І.О.",
    Address = new Address { City = "Київ", Street = "вул. Хрещатик, 1" },
    Diagnoses = new List<string> { "J06.9", "I10" },
    LabResults = new Dictionary<string, double>
    {
        ["Гемоглобін"] = 135.0,
        ["Глюкоза"]    = 5.1,
        ["Холестерин"] = 4.8
    }
};

var opts = new JsonSerializerOptions { WriteIndented = true };
string json = JsonSerializer.Serialize(patient, opts);
Console.WriteLine("Вкладений JSON:");
Console.WriteLine(json);

// Десеріалізація — вкладені об'єкти відновлюються автоматично
Patient? loaded = JsonSerializer.Deserialize<Patient>(json, opts);
Console.WriteLine($"\nВідновлено:");
Console.WriteLine($"  {loaded?.Name}, м.{loaded?.Address?.City}");
Console.WriteLine($"  Діагнози: {string.Join(", ", loaded?.Diagnoses ??
new())}");
foreach (var (test, val) in loaded?.LabResults ?? new())
    Console.WriteLine($"  {test}: {val.ToString()}");

// Вкладені класи
```

```
class Address
{
    public string City   { get; set; } = "";
    public string Street { get; set; } = "";
}

class Patient
{
    public int              Id         { get; set; }
    public string           Name       { get; set; } = "";
    public Address?         Address    { get; set; }
    public List<string>     Diagnoses  { get; set; } = new();
    public Dictionary<string, double> LabResults { get; set; } = new();
}
```

Практичний сценарій: JSON-конфігурація медичного застосунку

```
C#
using System;
using System.IO;
using System.Text.Json;
using System.Text.Json.Serialization;
using System.Collections.Generic;

string configPath = Path.Combine(Path.GetTempPath(), "clinic_config.json");

// Запис конфігурації
var config = new ClinicConfig
{
    ClinicName         = "Міська клінічна лікарня №1",
    Departments        = new List<string> { "Терапія", "Кардіологія",
"Неврологія", "Хірургія" },
    MaxPatientsPerDay  = 200,
    Thresholds         = new LabThresholds { GlucoseMax=7.0, PulseMin=60,
PulseMax=100, HemoglobinMin=110.0 },
    LastUpdated        = DateTime.Now
};

var opts = new JsonSerializerOptions { WriteIndented = true };
using (FileStream fs = File.Create(configPath))
    JsonSerializer.Serialize(fs, config, opts);

Console.WriteLine($"Конфіг збережено: {new
FileInfo(configPath).Length.ToString()} байт");

// Завантаження та використання
ClinicConfig? loaded;
using (FileStream fs = File.OpenRead(configPath))
    loaded = JsonSerializer.Deserialize<ClinicConfig>(fs, opts);

Console.WriteLine($"\nКлініка: {loaded?.ClinicName}");
Console.WriteLine($"Відділення: {string.Join(", ", loaded?.Departments ??
new())}");
Console.WriteLine($"Ліміт пацієнтів/день:
{loaded?.MaxPatientsPerDay.ToString()}");
Console.WriteLine($"Порогові значення:");
```

```
Console.WriteLine($"  Глюкоза: <= {loaded?.Thresholds.GlucoseMax.ToString()} ммоль/л");
Console.WriteLine($"  Пульс: {loaded?.Thresholds.PulseMin.ToString()} - {loaded?.Thresholds.PulseMax.ToString()} уд/хв");

// Перевірка пацієнта по конфігу
double patientGlucose = 8.5;
bool criticalGlucose = loaded != null && patientGlucose > loaded.Thresholds.GlucoseMax;
Console.WriteLine($"\nГлюкоза {patientGlucose.ToString()} ммоль/л — {(criticalGlucose ? "КРИТИЧНО" : "норма")}");

File.Delete(configPath);

// Конфігурація клініки — типовий сценарій для JSON
class ClinicConfig
{
    [JsonPropertyName("clinic_name")]
    public string ClinicName { get; set; } = "";

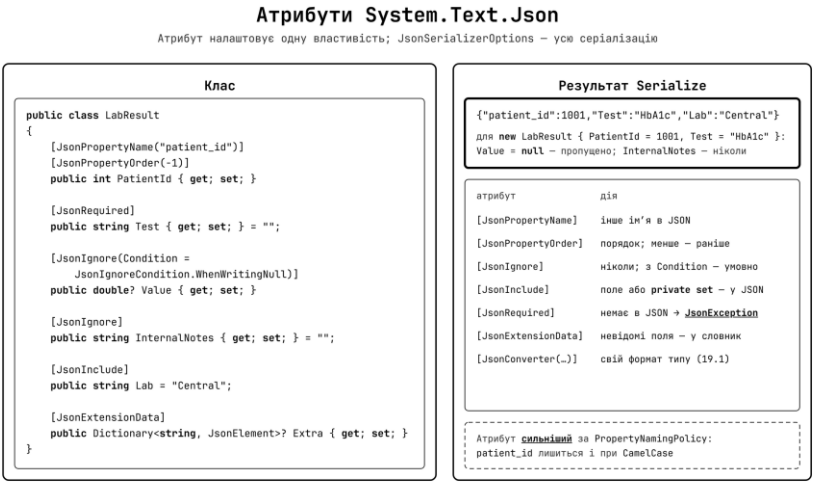
    [JsonPropertyName("departments")]
    public List<string> Departments { get; set; } = new();

    [JsonPropertyName("max_patients_per_day")]
    public int MaxPatientsPerDay { get; set; }

    [JsonPropertyName("lab_thresholds")]
    public LabThresholds Thresholds { get; set; } = new();

    [JsonPropertyName("last_updated")]
    public DateTime LastUpdated { get; set; }
}

class LabThresholds
{
    [JsonPropertyName("glucose_max")]    public double GlucoseMax    { get; set; }
    [JsonPropertyName("pulse_min")]      public int    PulseMin      { get; set; }
    [JsonPropertyName("pulse_max")]      public int    PulseMax      { get; set; }
    [JsonPropertyName("hemoglobin_min")] public double HemoglobinMin { get; set; }
}
```



Атрибути JSON та JsonSerializerOptions

Питання для самоконтролю — Розділ 18. Файлова система та потоки

18.1. Path, File, Directory

1. Чим відрізняється Path.Combine("C:\\data", "logs", "app.log") від простої конкатенації рядків через +? Чому Path.Combine є кращим вибором у кросплатформних застосунках?
2. Розгляньте код:

```
C#
1  string path = @"C:\projects\app\bin\Debug\app.exe";
2  Console.WriteLine(Path.GetDirectoryName(path));
3  Console.WriteLine(Path.GetFileNameWithoutExtension(path));
4  Console.WriteLine(Path.GetExtension(path));
```

Що виведе кожен рядок? Поясніть різницю між GetFileName і GetFileNameWithoutExtension.

3. Знайдіть помилку в наступному коді:

```
C#
1  string dir = @"C:\reports\2024\january";
2  if (!Directory.Exists(dir))
3      Directory.CreateDirectory(dir);
```

```
4  File.WriteAllText(dir + "report.txt", "data");
```

Виправте її та поясніть, яка функція Path вирішує цю проблему.

4. Порівняйте Directory.GetFiles() і Directory.EnumerateFiles(). У якому сценарії EnumerateFiles матиме суттєву перевагу в продуктивності? Наведіть приклад такого сценарію.
5. Напишіть метод CleanOldLogs(string folder, int daysOld), який видаляє з папки всі .log файли, що не змінювались більше daysOld днів. Використовуйте File.GetLastWriteTime і Directory.EnumerateFiles.
6. Що відбудеться при виклику Directory.Delete(@"C:\mydir")? Коли потрібен параметр recursive: true? Яка небезпека рекурсивного видалення?
7. Напишіть метод EnsureBackupExists(string sourceFile), який копіює файл поряд з оригіналом з суфіксом .bak, але тільки якщо .bak ще не існує або старіший за оригінал.

18.2. FileInfo та DirectoryInfo

8. Яка принципова різниця між статичним класом File і класом FileInfo? Коли виправдано використовувати FileInfo замість File.GetXxx() методів?
9. Розгляньте код:

```
C#
1  var fi = new FileInfo(@"C:\data\report.csv");
2  Console.WriteLine(fi.Length);
3  // ... зовнішній процес змінює файл ...
4  Console.WriteLine(fi.Length);
```

Чому другий Console.WriteLine може вивести хибне значення? Що треба зробити, щоб отримати актуальний розмір?

10. Поясніть концепцію кешування метаданих у FileInfo. Як це допомагає при роботі з багатьма операціями над одним файлом? Наведіть приклад коду, де FileInfo ефективніший за File.
11. Що таке FileSystemInfo? Яку роль він відіграє в ієрархії класів? Напишіть метод, що приймає FileSystemInfo і виводить ім'я та дату створення незалежно від того, файл це чи директорія.
12. Напишіть метод GetLargestFiles(string directory, int count), що повертає count найбільших файлів у директорії (включно з

вкладеними) у вигляді колекції FileInfo, відсортованих за спаданням розміру.

13. DirectoryInfo.GetFileSystemInfos() повертає FileSystemInfo[]. Чому це корисно? Напишіть код, що обходить директорію рекурсивно та виводить дерево файлів і папок з відступами відповідно до рівня вкладеності.
14. Порівняйте підходи: new FileInfo(path).Exists проти File.Exists(path). Чи є між ними різниця в поведінці? Який з них кращий у циклі з 1000 перевірок одного й того самого файлу?

18.3. Stream та FileStream

15. Що таке абстрактний клас Stream? Назвіть три найважливіших властивості Stream та поясніть їх значення. Чому Stream є абстрактним, а не інтерфейсом?
16. Розгляньте код:

```
C#
1  using var fs = new FileStream("data.bin", FileMode.Create);
2  byte[] data = Encoding.UTF8.GetBytes("Hello");
3  fs.Write(data, 0, data.Length);
4  // програма завершується
```

Що станеться з даними без явного Flush()? У яких умовах using гарантує збереження даних?

17. Поясніть різницю між FileMode.Create, FileMode.CreateNew і FileMode.OpenOrCreate. Наведіть конкретний сценарій, де помилковий вибір режиму призведе до втрати даних.
18. Навіщо потрібен параметр FileShare? Напишіть код, що відкриває файл журналу так, щоб кілька процесів могли читати його одночасно, але жоден не міг писати під час читання.
19. Реалізуйте метод CopyStream(Stream source, Stream destination, int bufferSize), що копіює дані чанками вказаного розміру. Поясніть, чому порційне копіювання краще за source.CopyTo(destination) у деяких випадках.
20. Напишіть код, що використовує Seek(offset, SeekOrigin) для читання останніх 100 байт файлу без завантаження всього файлу в пам'ять. Поясніть значення SeekOrigin.End.
21. Що таке MemoryStream? Чим він відрізняється від FileStream? Поясніть параметр leaveOpen у контексті передачі MemoryStream в інший об'єкт.

22. Розгляньте сценарій: потрібно прочитати великий файл (2 ГБ), обробити дані та записати результат. Чому не варто використовувати File.ReadAllBytes()? Як правильно організувати обробку за допомогою FileStream?

18.4. StreamReader та StreamWriter

23. Чому StreamReader і StreamWriter називають "декораторами"? Який паттерн проєктування тут реалізовано? Намалюйте (текстово) ланцюжок об'єктів при читанні UTF-8 файлу.
24. Розгляньте код:

```
C#
1  var writer = new StreamWriter("log.txt", append: true);
2  writer.WriteLine("Entry 1");
3  writer.WriteLine("Entry 2");
4  // writer.Dispose() не викликається
```

Що може статися з даними? Як AutoFlush = true змінює поведінку? Які його переваги та недоліки?

25. Напишіть метод CountLinesInFile(string path), що підраховує кількість рядків у великому текстовому файлі (сотні мегабайт) без завантаження всього файлу в пам'ять. Чому ReadLine() кращий за ReadToEnd() у цьому випадку?
26. Що таке StringReader і StringWriter? Навіщо вони потрібні, якщо є string? Напишіть приклад використання StringWriter для тестування методу, що пише до TextWriter.
27. Поясніть різницю між TextReader і StreamReader. Де в реальному коді корисно приймати TextReader замість StreamReader як параметр методу?
28. Розгляньте код:

```
C#
1  using var reader = new StreamReader("data.csv");
2  while (!reader.EndOfStream)
3  {
4      string line = reader.ReadLine();
5      ProcessLine(line);
6  }
```

Що поверне reader.Peek() на останньому рядку? Як EndOfStream пов'язаний з Peek()?

29. Напишіть метод MergeCsvFiles(string[] inputPaths, string outputPath), що об'єднує кілька CSV файлів в один, зберігаючи

заголовок тільки з першого файлу. Використовуйте StreamReader і StreamWriter.

18.5. BinaryReader та BinaryWriter

30. Чим принципово відрізняється двійковий формат від текстового при збереженні числа 3.14159265358979? Скільки байт займає double у бінарному форматі проти текстового?
31. Розгляньте код запису:

```
C#
1  using var bw = new BinaryWriter(File.Create("data.bin"));
2  bw.Write(42);
3  bw.Write("Hello");
4  bw.Write(3.14f);
5  bw.Write(true);
```

Напишіть відповідний код читання. Що станеться, якщо читати у іншому порядку (спочатку float, потім int)?

32. Поясніть концепцію записів фіксованого розміру у бінарних файлах. Напишіть структуру PatientRecord (ID: int, вік: byte, вага: float) та методи для запису й читання масиву таких записів.
33. Як використати Seek у FileStream разом з BinaryReader для читання запису за індексом без перебору всього файлу? Обчисліть зміщення для читання запису з індексом i, якщо кожен запис займає 9 байт.
34. Порівняйте розмір файлу при збереженні 1000 чисел типу double у текстовому форматі (через StreamWriter) та бінарному (через BinaryWriter). Зробіть розрахунок та поясніть різницю.
35. Напишіть метод UpdateRecord(string filePath, int index, float newWeight), що відкриває бінарний файл записів PatientRecord (ID: int, вік: byte, вага: float), знаходить запис за індексом та оновлює тільки поле ваги без переписування всього файлу.
36. Де бінарний формат є очевидним вибором, а де текстовий? Перелічіть 3 сценарії для кожного формату. Що таке ендіанність і чому вона важлива при обміні бінарними файлами між різними системами?

18.6. System.Text.Json

37. Що робить JsonSerializer.Serialize(obj, new JsonSerializerOptions { WriteIndented = true })? Коли доцільно використовувати WriteIndented = false у продуктивному середовищі?
38. Розгляньте клас:

```C#
public class Doctor
{
    public string Name { get; set; }
    public string PasswordHash { get; set; }
    public DateTime CreatedAt { get; set; }
    public List<string> Roles { get; set; }
}
```

Як серіалізувати об'єкт так, щоб: поле PasswordHash не потрапило в JSON, CreatedAt стало created_at у camelCase, а Roles стало permissions? Використайте JSON атрибути.

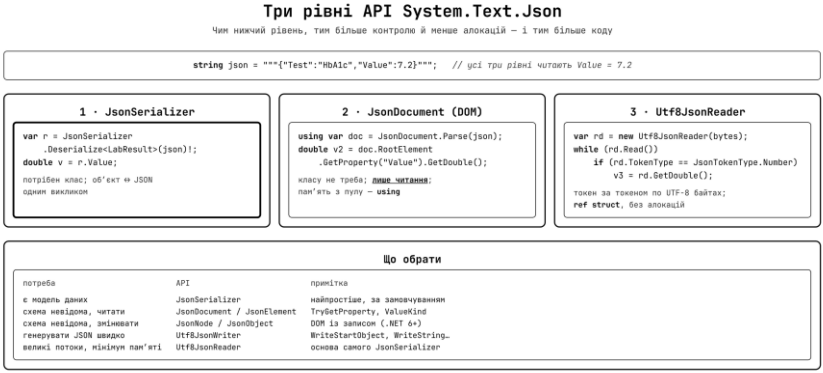
39. Поясніть різницю між JsonSerializerOptions.PropertyNamingPolicy = JsonNamingPolicy.CamelCase та атрибутом [JsonPropertyName("name")]. Який з них має пріоритет?
40. Що означає DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull? Напишіть код та покажіть вихідний JSON для об'єкта, де частина полів null.
41. Перерахуйте всі JSON атрибути, що вивчались у розділі ([JsonPropertyName], [JsonIgnore] тощо). Для кожного наведіть короткий приклад використання та поясніть ефект.
42. Напишіть метод SaveConfig<T>(T config, string path) і T LoadConfig<T>(string path), що зберігають і відновлюють конфігурацію з JSON файлу. Обробіть випадки, коли файл не існує або містить невалідний JSON.
43. Порівняйте JsonSerializer.Serialize(obj) (рядок) і JsonSerializer.Serialize(stream, obj) (в потік). Коли потокова версія є кращим вибором? Яка різниця в використанні пам'яті?
44. Що таке JsonStringEnumConverter? Без нього як серіалізується enum за замовчуванням? Напишіть приклад, де JsonStringEnumConverter зареєстрований глобально через JsonSerializerOptions.

Розділ 19. Серіалізація та десеріалізація. JSON та XML

19.1. System.Text.Json — поглиблено

У розділі 18.6 ми розглянули базові операції JsonSerializer.Serialize / Deserialize — достатньо для більшості задач. Але реальні системи стикаються з ситуаціями, де базового API не вистачає: схема JSON невідома заздалегідь, потрібна власна логіка серіалізації для специфічного типу, дані надходять як потік з HTTP і не варто буферизувати їх у рядок, або ієрархія класів потребує поліморфної серіалізації.

System.Text.Json надає **три рівні API**: від зручного JsonSerializer до низькорівневих JsonDocument/Utf8JsonReader. Чим нижчий рівень — тим більший контроль і вища продуктивність, але й більше коду.



Три рівні API System.Text.Json

JsonDocument та JsonElement — DOM без десеріалізації

JsonDocument — це легка DOM-модель для JSON, яка дозволяє читати та навігувати по JSON-документу **без прив'язки до конкретного C#-класу**. Корисно, коли схема документа невідома або змінюється — наприклад, відповідь зовнішнього API, налаштування плагінів, гетерогенні медичні записи.

```
C#
```

```
using System;
using System.Text.Json;

// Сценарій: відповідь зовнішнього API лабораторії — схема невідома заздалегідь
string labApiResponse = """
{
    "requestId": "LAB-2024-0315",
    "patient": {
        "id": 1001,
        "name": "Петренко Іван Олексійович",
        "ward": "Терапія"
    },
    "results": [
        { "test": "Гемоглобін",  "value": 135.0, "unit": "г/л",     "status": "норма" },
        { "test": "Глюкоза",     "value": 7.8,   "unit": "ммоль/л", "status": "вище норми" },
        { "test": "Лейкоцити",   "value": 6.2,   "unit": "10^9/л",  "status": "норма" }
    ],
    "processedAt": "2024-03-15T14:30:00Z"
}
""";

// JsonDocument.Parse — парсимо без прив'язки до класу
using JsonDocument doc = JsonDocument.Parse(labApiResponse);
JsonElement root = doc.RootElement;

// Читання скалярних значень
string requestId = root.GetProperty("requestId").GetString()!;
Console.WriteLine($"Запит: {requestId}");

// Навігація по вкладених об'єктах
JsonElement patient = root.GetProperty("patient");
int    patientId   = patient.GetProperty("id").GetInt32();
string patientName = patient.GetProperty("name").GetString()!;
Console.WriteLine($"Пацієнт: [{patientId.ToString()}] {patientName}");

// Перебір масиву
JsonElement results = root.GetProperty("results");
Console.WriteLine($"\nРезультати ({results.GetArrayLength().ToString()} аналізів):");
foreach (JsonElement result in results.EnumerateArray())
{
    string test   = result.GetProperty("test").GetString()!;
    double value  = result.GetProperty("value").GetDouble();
    string unit   = result.GetProperty("unit").GetString()!;
    string status = result.GetProperty("status").GetString()!;
    string icon   = status == "норма" ? "[OK]" : "[!!]";
    Console.WriteLine($"  {icon} {test}: {value.ToString()} {unit} — {status}");
}

// Перевірка існування поля
if (root.TryGetProperty("processedAt", out JsonElement ts))
    Console.WriteLine($"\nОброблено: {ts.GetDateTimeOffset():yyyy-MM-dd HH:mm}");
```

JsonDocument реалізує IDisposable — завжди використовуйте using. Внутрішньо він орендує буфер із пулу (ArrayPool<byte>), і Dispose повертає його назад. Без Dispose збирач сміття колись звільнить пам'ять, але пул її вже не отримає, тож під навантаженням виникатимуть зайві алокації. Не менш важливо: JsonElement, отримані з документа, після Dispose стають недійсними (звернення кидає ObjectDisposedException) — якщо елемент потрібен довше, викличте element.Clone(). А для DOM, який треба **змінювати**, з .NET 6 є окреме API JsonNode / JsonObject (System.Text.Json.Nodes).

TryGetProperty та перевірка типів

```
C#
 1  using System;
 2  using System.Text.Json;
 3
 4  string json = """
 5  {
 6      "patientId": 1002,
 7      "temperature": 37.2,
 8      "hospitalized": true,
 9      "notes": null,
10      "tags": ["ГРВІ", "сезонний"]
11  }
12  """;
13
14  using JsonDocument doc = JsonDocument.Parse(json);
15  JsonElement root = doc.RootElement;
16
17  // TryGetProperty — безпечне читання (не кидає при відсутності поля)
18  if (root.TryGetProperty("temperature", out JsonElement temp))
19      Console.WriteLine($"Температура: {temp.GetDouble().ToString("F1")} °C");
20
21  // Перевірка типу через ValueKind
22  foreach (JsonProperty prop in root.EnumerateObject())
23  {
24      string typeName = prop.Value.ValueKind switch
25      {
26          JsonValueKind.Number  => $"number =
    {prop.Value.GetDouble().ToString()}",
27          JsonValueKind.String  => $"string = \"{prop.Value.GetString()}\"",
28          JsonValueKind.True    => "bool = true",
29          JsonValueKind.False   => "bool = false",
30          JsonValueKind.Null    => "null",
31          JsonValueKind.Array   =>
    $"array[{prop.Value.GetArrayLength().ToString()}]",
32          JsonValueKind.Object  => "object {...}",
33          _                     => prop.Value.ValueKind.ToString()
34      };
35      Console.WriteLine($"  {prop.Name}: {typeName}");
36  }
```

JsonValueKind — перелік, що описує JSON-тип вузла: Object, Array, String, Number, True, False, Null, Undefined. Перевіряйте ValueKind перед читанням, якщо схема документа не гарантована.

Utf8JsonWriter — швидкісний запис JSON

Utf8JsonWriter — найнижчий рівень запису JSON. Він записує безпосередньо в UTF-8 байти без проміжного рядкового представлення — це найшвидший можливий спосіб генерації JSON у .NET:

```C#
using System;
using System.IO;
using System.Text;
using System.Text.Json;

using MemoryStream ms = new MemoryStream();
var writerOptions = new JsonWriterOptions { Indented = true };

using (Utf8JsonWriter writer = new Utf8JsonWriter(ms, writerOptions))
{
    writer.WriteStartObject();                          // {

    writer.WriteString("requestId", "MED-2024-0315");
    writer.WriteNumber("version", 2);

    writer.WriteStartObject("patient");                 // "patient": {
    writer.WriteNumber("id", 1001);
    writer.WriteString("name", "Петренко І.О.");
    writer.WriteBoolean("hospitalized", false);
    writer.WriteEndObject();                            // }

    writer.WriteStartArray("diagnoses");                // "diagnoses": [
    writer.WriteStringValue("J06.9");
    writer.WriteStringValue("I10");
    writer.WriteEndArray();                             // ]

    writer.WriteNull("notes");                          // "notes": null
    writer.WriteString("createdAt", DateTime.UtcNow);

    writer.WriteEndObject();                            // }
}

string json = Encoding.UTF8.GetString(ms.ToArray());
Console.WriteLine(json);
Console.WriteLine($"\nРозмір: {ms.Length.ToString()} байт");
```

Utf8JsonWriter застосовується у бібліотеках і фреймворках, де продуктивність критична — наприклад, ASP.NET Core використовує його внутрішньо для серіалізації HTTP-відповідей. Для звичайного застосунку достатньо JsonSerializer.

Кастомні конвертери: JsonConverter<T>

Стандартна серіалізація не завжди підходить для специфічних типів. JsonConverter<T> дозволяє повністю контролювати, як конкретний тип читається з JSON і записується в JSON:

```
C#
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

var test = new LabTest
{
    Name   = "Глюкоза",
    Value  = 5.4,
    Unit   = "ммоль/л",
    Normal = new NormalRange(3.9, 6.1)
};

var opts = new JsonSerializerOptions { WriteIndented = true };
string json = JsonSerializer.Serialize(test, opts);
Console.WriteLine("JSON з кастомним конвертером:");
Console.WriteLine(json);

// Десеріалізація — конвертер відпрацьовує автоматично
LabTest? loaded = JsonSerializer.Deserialize<LabTest>(json, opts);
Console.WriteLine($"\nВідновлено: {loaded?.Name} = {loaded?.Value.ToString()}
{loaded?.Unit}");
Console.WriteLine($"Норма: {loaded?.Normal}");
Console.WriteLine($"Статус: {(loaded?.Value >= loaded?.Normal.Min &&
loaded?.Value <= loaded?.Normal.Max ? "норма" : "відхилення")}");

// Кастомний тип — діапазон нормальних значень показника
struct NormalRange
{
    public double Min { get; }
    public double Max { get; }
    public NormalRange(double min, double max) { Min = min; Max = max; }
    public override string ToString() => $"{Min.ToString()}-{Max.ToString()}";
}

// За замовчуванням NormalRange серіалізується як об'єкт {"Min":...,"Max":...}
// Хочемо: рядок "4.0-6.5" для компактності

class NormalRangeConverter : JsonConverter<NormalRange>
{
    // Читання: "4.0-6.5" -> NormalRange(4.0, 6.5)
    public override NormalRange Read(ref Utf8JsonReader reader, Type
typeToConvert, JsonSerializerOptions options)
    {
        string? s = reader.GetString() ?? throw new JsonException("Expected
string for NormalRange");
        string[] parts = s.Split('-');
        if (parts.Length != 2) throw new JsonException($"Invalid NormalRange
format: {s}");
        return new NormalRange(double.Parse(parts[0]), double.Parse(parts[1]));
    }

```

```
47      // Запис: NormalRange(4.0, 6.5) -> "4.0-6.5"
48      public override void Write(Utf8JsonWriter writer, NormalRange value,
    JsonSerializerOptions options)
49      {
50          writer.WriteStringValue($"{value.Min.ToString()}-
    {value.Max.ToString()}");
51      }
52  }
53
54  class LabTest
55  {
56      public string      Name    { get; set; } = "";
57      public double      Value   { get; set; }
58      public string      Unit    { get; set; } = "";
59      [JsonConverter(typeof(NormalRangeConverter))]   // атрибут на конкретному
    полі
60      public NormalRange Normal  { get; set; }
61  }
```

Конвертер можна зареєструвати **глобально** через options.Converters.Add(new NormalRangeConverter()) замість атрибута на кожному полі — тоді він спрацює для всіх властивостей цього типу в усій програмі.

Поліморфна серіалізація: JsonDerivedType

Коли в полі може бути один з кількох підтипів, JsonSerializer за замовчуванням серіалізує лише властивості базового типу і при десеріалізації не знає, який підтип відновити. Атрибути [JsonPolymorphic] і [JsonDerivedType] вирішують це:

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Text.Json;
 4  using System.Text.Json.Serialization;
 5
 6  // Список різних підтипів
 7  var results = new List<MedicalResult>
 8  {
 9      new BloodTest   { PatientId=1001, Glucose=5.1, Hemoglobin=135.0 },
10      new EcgRecord   { PatientId=1001, HeartRate=72, Rhythm="синусовий" },
11      new ImagingData { PatientId=1002, Modality="УЗД", BodyPart="черевна
    порожнина" },
12  };
13
14  var opts = new JsonSerializerOptions { WriteIndented = true };
15  string json = JsonSerializer.Serialize(results, opts);
16  Console.WriteLine("Поліморфний JSON:");
17  Console.WriteLine(json);
18
19  // Десеріалізація відновлює правильний підтип
20  var loaded = JsonSerializer.Deserialize<List<MedicalResult>>(json, opts)!;
21  foreach (MedicalResult r in loaded)
```

```
{
    string info = r switch
    {
        BloodTest   b => $"Кров: глюкоза={b.Glucose.ToString()}",
        EcgRecord   e => $"ЕКГ: пульс={e.HeartRate.ToString()}, ритм={e.Rhythm}",
        ImagingData i => $"Знімок: {i.Modality} ({i.BodyPart})",
        _             => "невідомий тип"
    };
    Console.WriteLine($"  [{r.GetType().Name}] пацієнт {r.PatientId.ToString()}: {info}");
}

[JsonPolymorphic(TypeDiscriminatorPropertyName = "$type")]
[JsonDerivedType(typeof(BloodTest),   typeDiscriminator: "blood")]
[JsonDerivedType(typeof(EcgRecord),   typeDiscriminator: "ecg")]
[JsonDerivedType(typeof(ImagingData), typeDiscriminator: "imaging")]
abstract class MedicalResult
{
    public int    PatientId  { get; set; }
    public DateTime RecordedAt { get; set; } = DateTime.Now;
}

class BloodTest : MedicalResult
{
    public double Glucose    { get; set; }
    public double Hemoglobin { get; set; }
}

class EcgRecord : MedicalResult
{
    public int    HeartRate  { get; set; }
    public string Rhythm     { get; set; } = "";
}

class ImagingData : MedicalResult
{
    public string Modality   { get; set; } = ""; // МРТ, КТ, УЗД
    public string BodyPart   { get; set; } = "";
}
```

$type — дискримінатор типу, який JsonSerializer автоматично додає при записі і читає при десеріалізації. Ім'я поля (TypeDiscriminatorPropertyName) можна задати довільним — часто використовують "type", "kind", "$type".

Async серіалізація: SerializeAsync та DeserializeAsync

Для роботи з HTTP-потоками та великими файлами є async-версії, що не блокують потік:

C#
```
using System;
using System.Collections.Generic;
using System.IO;
```

```
using System.Text.Json;

string filePath = Path.Combine(Path.GetTempPath(), "patients_async.json");

var patients = new List<PatientRecord>
{
    new() { Id=1001, Name="Петренко І.О.", Diagnosis="J06.9",
AdmittedAt=DateTime.Now.AddDays(-3) },
    new() { Id=1002, Name="Бойко О.П.",    Diagnosis="I10",
AdmittedAt=DateTime.Now.AddDays(-1) },
    new() { Id=1003, Name="Мороз В.І.",    Diagnosis="E11.9",
AdmittedAt=DateTime.Now },
};

var opts = new JsonSerializerOptions { WriteIndented = true };

// Async запис у файл — не блокує потік під час I/O
await using (FileStream fs = File.Create(filePath))
    await JsonSerializer.SerializeAsync(fs, patients, opts);

Console.WriteLine($"Async запис: {new FileInfo(filePath).Length.ToString()}
байт");

// Async читання з файлу
await using (FileStream fs = File.OpenRead(filePath))
{
    List<PatientRecord>? loaded = await
JsonSerializer.DeserializeAsync<List<PatientRecord>>(fs, opts);
    Console.WriteLine($"Async читання: {loaded?.Count.ToString()} записів");
    foreach (var p in loaded ?? new())
        Console.WriteLine($"  [{p.Id.ToString()}] {p.Name} — {p.Diagnosis}");
}

File.Delete(filePath);

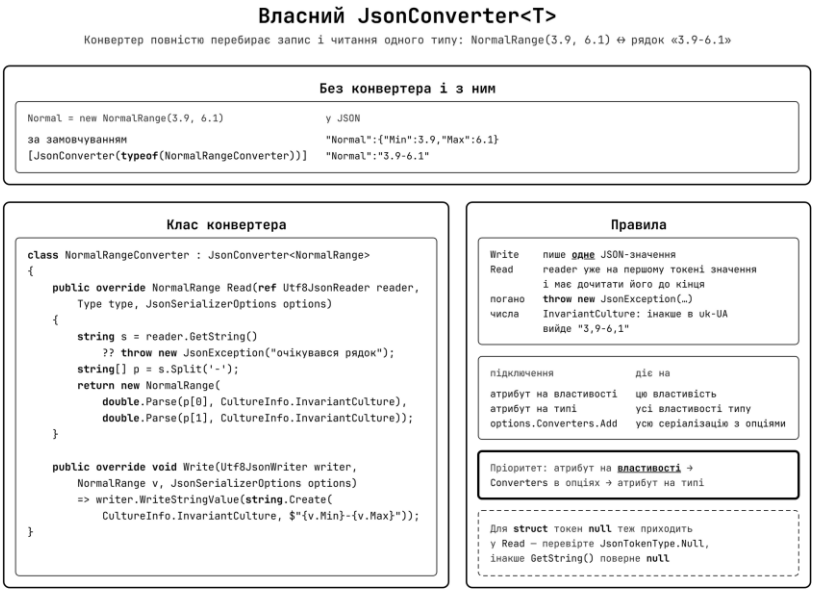
class PatientRecord
{
    public int      Id        { get; set; }
    public string   Name      { get; set; } = "";
    public string   Diagnosis { get; set; } = "";
    public DateTime AdmittedAt { get; set; }
}
```

SerializeAsync / DeserializeAsync — особливо важливі у ASP.NET Core: при відповіді на HTTP-запит серіалізація відбувається напряму у HttpResponse.Body-потік без буферизації всього JSON у пам'яті. Для консольних застосунків або Desktop різниця мінімальна, але для веб-сервісів під навантаженням — суттєва.

Практичний сценарій: агрегація гетерогенних JSON-відповідей

```
C#
using System;
using System.Collections.Generic;
using System.Text.Json;
```

```
4
5  // Симуляція: кілька API повертають різні JSON-схеми
6  // Потрібно агрегувати дані без фіксованих класів
7
8  string[] apiResponses = {
9      """{"source":"lab",     "patientId":1001, "test":"Глюкоза",    "value":5.1,
   "unit":"ммоль/л"}""",
10     """{"source":"vitals",  "patientId":1001, "pulse":72,
   "temp":36.8,  "bp":"120/80"}""",
11     """{"source":"pharmacy","patientId":1001, "drug":"Амоксицилін",
   "dose":"500mg","days":7}""",
12 };
13
14 var summary = new Dictionary<string, List<string>>();
15
16 foreach (string response in apiResponses)
17 {
18     using JsonDocument doc = JsonDocument.Parse(response);
19     JsonElement root = doc.RootElement;
20
21     string source    = root.GetProperty("source").GetString()!;
22     int    patientId = root.GetProperty("patientId").GetInt32();
23
24     if (!summary.ContainsKey(source))
25         summary[source] = new List<string>();
26
27     // Агрегуємо поля, залежно від джерела
28     switch (source)
29     {
30         case "lab":
31             string test  = root.GetProperty("test").GetString()!;
32             double value = root.GetProperty("value").GetDouble();
33             string unit  = root.GetProperty("unit").GetString()!;
34             summary[source].Add($"{test}: {value.ToString()} {unit}");
35             break;
36
37         case "vitals":
38             int    pulse = root.GetProperty("pulse").GetInt32();
39             double temp  = root.GetProperty("temp").GetDouble();
40             string bp    = root.GetProperty("bp").GetString()!;
41             summary[source].Add($"пульс={pulse.ToString()},
   t={temp.ToString("F1")}, AT={bp}");
42             break;
43
44         case "pharmacy":
45             string drug = root.GetProperty("drug").GetString()!;
46             string dose = root.GetProperty("dose").GetString()!;
47             int    days = root.GetProperty("days").GetInt32();
48             summary[source].Add($"{drug} {dose} x {days.ToString()} днів");
49             break;
50     }
51 }
52
53 Console.WriteLine("=== Зведення по пацієнту 1001 ===");
54 foreach (var (source, items) in summary)
55 {
56     Console.WriteLine($"\n[{source.ToUpper()}]");
57     foreach (string item in items)
58         Console.WriteLine($"  • {item}");
59 }
```



Кастомний JsonConverter<T> — схема роботи

JSON Source Generators — серіалізація без рефлексії

Стандартний JsonSerializer використовує **рефлексію** (reflection) під час виконання: він динамічно аналізує типи, знаходить властивості та генерує відповідний код. Це зручно, але має три недоліки:

— **Продуктивність**: рефлексія повільніша за статично сгенерований код

— **NativeAOT / trimming**: рефлексія несумісна з AOT-компіляцією та агресивним видаленням невикористаного коду

— **Розмір застосунку**: рефлексивний код не trimmed компілятором

JSON Source Generators (.NET 6+; у файлових застосунках .NET 10 з увімкненим AOT — фактично єдиний спосіб, розділ 18.6) вирішують усе це: ви оголошуєте JsonSerializerContext — і компілятор генерує весь серіалізаційний код на **етапі компіляції**, без рефлексії під час виконання.

```C#
using System;
using System.Text.Json;
using System.Text.Json.Serialization;
```

```
4
5  // Використання: замість JsonSerializer.Serialize(obj) →
6  //                        JsonSerializer.Serialize(obj, context.TypeName)
7
8  var record = new PatientRecord
9  {
10     PatientId = "PT-1001",
11     FullName  = "Петренко Іван Олексійович",
12     Age       = 67,
13     Diagnoses = new[]
14     {
15         new DiagnosisEntry { Code = "I10.9", Description = "Гіпертонія" },
16         new DiagnosisEntry { Code = "E11.9", Description = "Цукровий діабет 2
   типу" },
17     }
18 };
19
20 // Серіалізація через Source Generator (без рефлексії!)
21 string json = JsonSerializer.Serialize(record,
   MedicalJsonContext.Default.PatientRecord);
22 Console.WriteLine("=== Source Generator: Serialize ===");
23 Console.WriteLine(json);
24
25 // Десеріалізація
26 PatientRecord? parsed = JsonSerializer.Deserialize(json,
   MedicalJsonContext.Default.PatientRecord);
27 Console.WriteLine($"\n=== Parsed back ===");
28 Console.WriteLine($"ID: {parsed?.PatientId}, Вік: {parsed?.Age}");
29 Console.WriteLine($"Діагнозів: {parsed?.Diagnoses?.Length ?? 0}");
30
31 // Налаштовуємо Source Generator для наших типів (оголошення типів — після top-
   level коду)
32 [JsonSerializable(typeof(PatientRecord))]
33 [JsonSerializable(typeof(DiagnosisEntry))]
34 [JsonSourceGenerationOptions(
35     WriteIndented = true,
36     PropertyNamingPolicy = JsonKnownNamingPolicy.CamelCase)]
37 partial class MedicalJsonContext : JsonSerializerContext { }
38
39 class PatientRecord
40 {
41     public string  PatientId { get; set; } = "";
42     public string  FullName  { get; set; } = "";
43     public int     Age       { get; set; }
44     public DiagnosisEntry[]? Diagnoses { get; set; }
45 }
46
47 class DiagnosisEntry
48 {
49     public string Code        { get; set; } = "";
50     public string Description { get; set; } = "";
51 }
```

Синтаксис MedicalJsonContext.Default.PatientRecord — це сгенерований JsonTypeInfo<PatientRecord>, який передається замість опцій. Компілятор знаходить атрибут

[JsonSerializable(typeof(PatientRecord))] і генерує весь необхідний код у часткому класі MedicalJsonContext.

Режим	Продуктивність	NativeAOT	Зручність
Reflection (за замовчуванням)	Помірна	✗	★★★★★
Source Generators	**Швидший** (передусім старт: немає рефлексії при першому виклику; виграш залежить від типу)	✓	★★★★

Для нових проектів на .NET 7+ та особливо для AOT-сценаріїв (мобільні застосунки, мікросервіси з швидким стартом) рекомендується Source Generators.

Антипатерн: небезпечна десеріалізація

Не десеріалізуйте довільний JSON без валідації типу і вмісту. Декілька прикладів небезпечних паттернів:

```C#
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

// Вхідний JSON від клієнта — зловмисник додав поле isAdmin
var web = new JsonSerializerOptions(JsonSerializerDefaults.Web); // camelCase, без урахування регістру

// === НЕБЕЗПЕЧНО: десеріалізуємо запит прямо в доменну модель ===
string maliciousJson = """{"name":"Петренко","isAdmin":true}""";
var user = JsonSerializer.Deserialize<DangerousUser>(maliciousJson, web)!;
Console.WriteLine($"Ім'я: {user.Name}, IsAdmin: {user.IsAdmin}"); // True — КРИТИЧНА УРАЗЛИВІСТЬ!

// === ПРАВИЛЬНО: окремий вхідний DTO без полів безпеки ===
string safeJson = """{"fullName":"Коваль Марія","age":45,"isAdmin":true}""";
var request = JsonSerializer.Deserialize<CreatePatientRequest>(safeJson, web)!;
Console.WriteLine($"\nБезпечно: {request.FullName}, {request.Age.ToString()}р. (isAdmin просто нікуди записати)");

// Доменна модель: будь-яка публічна властивість із set заповнюється з JSON
class DangerousUser
{
    public string Name { get; set; } = "";
```

```
22      public bool IsAdmin { get; set; } = false;      // ← зловмисник може
    передати "isAdmin": true
23      public string PasswordHash { get; set; } = "";  // ← і підмінити хеш пароля
24  }
25
26  class CreatePatientRequest
27  {
28      public string FullName { get; init; } = "";
29      public int    Age      { get; init; }
30      // IsAdmin, internal flags — НЕМА у вхідному DTO
31  }
32
33  class PatientResponse
34  {
35      public string PatientId { get; init; } = "";
36      public string FullName  { get; init; } = "";
37      // PasswordHash, internal fields — НЕМА у вихідному DTO
38  }
```

Правила безпечної десеріалізації:

— **Окремі DTO** для вхідних і вихідних даних — Request не містить полів безпеки, Response не містить внутрішніх полів

— **[JsonIgnore]** для полів, що ніколи не мають надходити від клієнта

Зверніть увагу: уразливість виникає не через якийсь «небезпечний» атрибут, а сама собою — JsonSerializer заповнює **будь-яку** публічну властивість із сеттером, ім'я якої збігається з полем JSON (а з JsonSerializerDefaults.Web, як в ASP.NET Core, — ще й без урахування регістру). Атрибут [JsonInclude] на публічній властивості нічого не змінює: він потрібен лише для полів і властивостей з непублічним сеттером. Попередня версія прикладу позначала IsAdmin атрибутом [JsonInclude] і читала JSON без опцій — через чутливість до регістру "isAdmin" взагалі не потрапляв у IsAdmin, тож приклад виводив False і уразливості не демонстрував.

— **Валідація після десеріалізації**: перевіряйте діапазони значень, обов'язкові поля, допустимі enum-значення

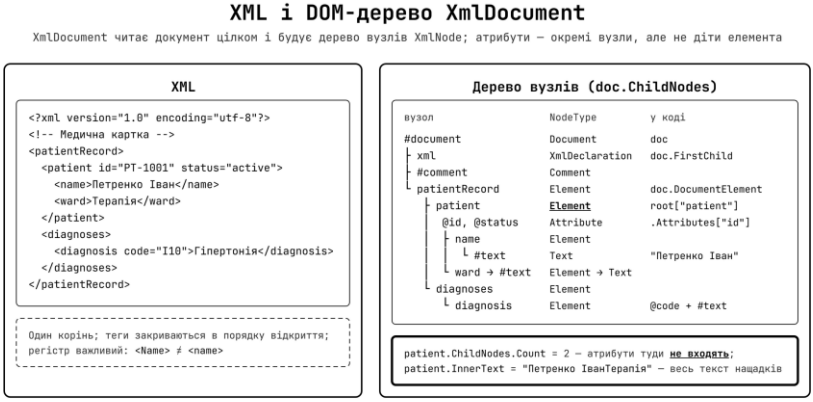
— **MaxDepth** у JsonSerializerOptions обмежує глибину вкладеності (захист від атак через переповнення стека)

19.2. XML-формат та XmlDocument. DOM-модель

XML (eXtensible Markup Language) — мова розмітки, що описує структуровані дані у вигляді ієрархії тегів. На відміну від JSON, XML підтримує атрибути як першокласний синтаксис, простори імен (xmlns) та валідацію через схеми (XSD). У медичній галузі XML — стандартний

формат: **HL7 v2/v3** і **FHIR** для обміну клінічними даними, **DICOM** заголовки, **CDA** (Clinical Document Architecture). Конфігурація .NET-застосунків (.config-файли), SOAP-сервіси, Office Open XML (.docx, .xlsx) — теж XML.

System.Xml — простір імен, що надає повний набір інструментів для роботи з XML. Клас XmlDocument реалізує **DOM** (Document Object Model) — модель, де весь XML-документ завантажується у пам'ять як дерево вузлів (XmlNode). Кожен елемент, атрибут, текстовий вузол, коментар — окремий об'єкт у цьому дереві.



Структура XML-документа та DOM-дерево

Структура XML-документа

```C#
using System;

// Повна структура XML з усіма можливими вузлами:
string xmlExample = """
<?xml version="1.0" encoding="utf-8"?>
<!-- Медична картка пацієнта: CDA-подібний формат -->
<patientRecord xmlns:med="http://clinic.ua/medical/2024">
    <patient id="PT-1001" status="active">
        <name>Петренко Іван Олексійович</name>
        <birthDate>1978-03-15</birthDate>
        <ward>Терапія</ward>
    </patient>
    <diagnoses>
        <diagnosis code="J06.9" system="ICD-10">ГРВІ</diagnosis>
        <diagnosis code="I10"   system="ICD-10">Гіпертонічна
хвороба</diagnosis>
    </diagnoses>
    <vitals recordedAt="2024-03-15T14:30:00">
```

```
        <temperature unit="C">37.2</temperature>
        <pulse unit="bpm">82</pulse>
        <bloodPressure>
            <systolic>135</systolic>
            <diastolic>85</diastolic>
        </bloodPressure>
    </vitals>
</patientRecord>
""";

// Ключові компоненти XML:
Console.WriteLine("Компоненти XML-документа:");
Console.WriteLine("  <?xml ...?>          — декларація (версія, кодування)");
Console.WriteLine("  <!-- ... -->         — коментар");
Console.WriteLine("  <patientRecord>      — кореневий елемент (root element)");
Console.WriteLine("  xmlns:med=\"...\"   — простір імен (namespace)");
Console.WriteLine("  id=\"PT-1001\"      — атрибут елемента");
Console.WriteLine("  <name>Петр...</name> — дочірній елемент з текстовим вмістом");
Console.WriteLine("  <diagnosis code=...> — елемент з атрибутами і текстом");
```

XmlDocument — завантаження та навігація

```
C#
using System;
using System.Xml;
using System.IO;

string xml = """
<?xml version="1.0" encoding="utf-8"?>
<patientRecord>
    <patient id="PT-1001" status="active">
        <name>Петренко Іван Олексійович</name>
        <birthDate>1978-03-15</birthDate>
        <ward>Терапія</ward>
    </patient>
    <diagnoses>
        <diagnosis code="J06.9">ГРВІ</diagnosis>
        <diagnosis code="I10">Гіпертонічна хвороба</diagnosis>
    </diagnoses>
    <vitals>
        <temperature unit="C">37.2</temperature>
        <pulse unit="bpm">82</pulse>
    </vitals>
</patientRecord>
""";

XmlDocument doc = new XmlDocument();

// Завантаження з рядка
doc.LoadXml(xml);

// Альтернатива — з файлу:
// doc.Load("patient.xml");
// або зі Stream:
// doc.Load(stream);

// DocumentElement — кореневий елемент
```

```
XmlElement root = doc.DocumentElement!;
Console.WriteLine($"Корінь: <{root.Name}>");
Console.WriteLine($"Дочірніх вузлів: {root.ChildNodes.Count.ToString()}");

// Доступ до дочірніх елементів
XmlNode? patientNode = root["patient"];  // перший <patient>
if (patientNode != null)
{
    // Читання атрибутів
    string id     = patientNode.Attributes?["id"]?.Value ?? "";
    string status = patientNode.Attributes?["status"]?.Value ?? "";
    Console.WriteLine($"\nПацієнт: id={id}, status={status}");

    // Читання текстового вмісту дочірніх елементів
    string name  = patientNode["name"]?.InnerText  ?? "";
    string bdate = patientNode["birthDate"]?.InnerText ?? "";
    string ward  = patientNode["ward"]?.InnerText  ?? "";
    Console.WriteLine($"  Ім'я: {name}");
    Console.WriteLine($"  Дата народження: {bdate}");
    Console.WriteLine($"  Відділення: {ward}");
}

// Перебір дочірніх вузлів колекції
XmlNode? diagsNode = root["diagnoses"];
if (diagsNode != null)
{
    Console.WriteLine($"\nДіагнози
({diagsNode.ChildNodes.Count.ToString()}):");
    foreach (XmlNode diag in diagsNode.ChildNodes)
    {
        string code = diag.Attributes?["code"]?.Value ?? "";
        string text = diag.InnerText;
        Console.WriteLine($"  [{code}] {text}");
    }
}
```

XmlNode — базовий клас для всіх вузлів DOM. XmlElement — конкретний тип для елементів (тегів). Властивість InnerText повертає весь текстовий вміст вузла (включно з вкладеними тегами), InnerXml — весь XML-вміст як рядок.

SelectSingleNode та SelectNodes — XPath у XmlDocument

C#
```
using System;
using System.Xml;

string xml = """
<?xml version="1.0" encoding="utf-8"?>
<clinic>
    <patient id="PT-1001" ward="Терапія">
        <name>Петренко Іван Олексійович</name>
        <diagnosis code="J06.9">ГРВІ</diagnosis>
        <diagnosis code="I10">Гіпертонія</diagnosis>
    </patient>
    <patient id="PT-1002" ward="Кардіологія">
```

```
        <name>Бойко Оксана Петрівна</name>
        <diagnosis code="I21.0">Інфаркт міокарда</diagnosis>
    </patient>
    <patient id="PT-1003" ward="Терапія">
        <name>Мороз Василь Іванович</name>
        <diagnosis code="E11.9">Цукровий діабет 2 типу</diagnosis>
    </patient>
</clinic>
""";

XmlDocument doc = new XmlDocument();
doc.LoadXml(xml);

// SelectSingleNode — перший вузол за XPath
XmlNode? first = doc.SelectSingleNode("/clinic/patient[1]/name");
Console.WriteLine($"Перший пацієнт: {first?.InnerText}");

// Пошук за значенням атрибуту
XmlNode? byId = doc.SelectSingleNode("/clinic/patient[@id='PT-1002']");
Console.WriteLine($"Пацієнт PT-1002: {byId?["name"]?.InnerText}");

// SelectNodes — всі вузли за XPath
XmlNodeList? therapyPatients =
doc.SelectNodes("/clinic/patient[@ward='Терапія']");
Console.WriteLine($"\nПацієнти у Терапії
({therapyPatients?.Count.ToString()}):");
if (therapyPatients != null)
    foreach (XmlNode p in therapyPatients)
        Console.WriteLine($"  {p.Attributes?["id"]?.Value}:
{p["name"]?.InnerText}");

// Пошук по всьому документу: // = будь-який рівень вкладеності
XmlNodeList? allDiags = doc.SelectNodes("//diagnosis[@code='I10' or
@code='I21.0']");
Console.WriteLine($"\nКардіологічні діагнози ({allDiags?.Count.ToString()}):");
if (allDiags != null)
    foreach (XmlNode diag in allDiags)
        Console.WriteLine($"  [{diag.Attributes?["code"]?.Value}]
{diag.InnerText}");
```

XPath-вирази у SelectSingleNode/SelectNodes — потужний засіб вибірки: /root/child (точний шлях), //element (будь-яка глибина), [@attr='val'] (фільтр за атрибутом), [position()] (за позицією).

Створення XML-документа програмно

C#

```
using System;
using System.IO;
using System.Xml;

XmlDocument doc = new XmlDocument();

// XML-декларація
XmlDeclaration decl = doc.CreateXmlDeclaration("1.0", "utf-8", null);
doc.AppendChild(decl);
```

```

// Кореневий елемент
XmlElement root = doc.CreateElement("medicalReport");
root.SetAttribute("version", "1.0");
root.SetAttribute("generatedAt", DateTime.Now.ToString("yyyy-MM-ddTHH:mm:ss"));
doc.AppendChild(root);

// Коментар
XmlComment comment = doc.CreateComment("Автоматично згенеровано медичною системою");
root.AppendChild(comment);

// Елемент пацієнта з атрибутами
XmlElement patient = doc.CreateElement("patient");
patient.SetAttribute("id", "PT-1001");
patient.SetAttribute("status", "active");
root.AppendChild(patient);

// Дочірні елементи з текстовим вмістом
void AddElement(XmlElement parent, string tagName, string text)
{
    XmlElement el = doc.CreateElement(tagName);
    el.InnerText = text;
    parent.AppendChild(el);
}

AddElement(patient, "name",      "Петренко Іван Олексійович");
AddElement(patient, "birthDate", "1978-03-15");
AddElement(patient, "ward",      "Терапія");

// Вкладена структура: diagnoses -> diagnosis[]
XmlElement diagnoses = doc.CreateElement("diagnoses");
root.AppendChild(diagnoses);

var diagData = new[] {
    ("J06.9", "ICD-10", "ГРВІ"),
    ("I10",   "ICD-10", "Гіпертонічна хвороба"),
};
foreach (var (code, system, text) in diagData)
{
    XmlElement diag = doc.CreateElement("diagnosis");
    diag.SetAttribute("code",   code);
    diag.SetAttribute("system", system);
    diag.InnerText = text;
    diagnoses.AppendChild(diag);
}

// Збереження у рядок
using StringWriter sw = new StringWriter();
using XmlTextWriter xw = new XmlTextWriter(sw) { Formatting = Formatting.Indented, Indentation = 2 };
doc.WriteTo(xw);
Console.WriteLine(sw.ToString());

// Збереження у файл
string path = Path.Combine(Path.GetTempPath(), "patient_report.xml");
doc.Save(path);
Console.WriteLine($"\nЗбережено: {path}");
File.Delete(path);
```

Зміна існуючого XML

```C#
using System;
using System.Xml;

string xml = """
<?xml version="1.0" encoding="utf-8"?>
<patients>
    <patient id="PT-1001">
        <name>Петренко І.О.</name>
        <ward>Терапія</ward>
    </patient>
</patients>
""";

XmlDocument doc = new XmlDocument();
doc.LoadXml(xml);

XmlElement root = doc.DocumentElement!;

// Зміна атрибуту існуючого вузла
XmlNode? p1 = doc.SelectSingleNode("/patients/patient[@id='PT-1001']");
p1?.Attributes?["id"]?.OwnerElement?.SetAttribute("status", "discharged");

// Зміна текстового вмісту
if (p1?["ward"] != null)
    p1["ward"]!.InnerText = "Виписаний";

// Додавання нового пацієнта
XmlElement newPatient = doc.CreateElement("patient");
newPatient.SetAttribute("id", "PT-1002");
newPatient.SetAttribute("status", "active");

XmlElement nameEl = doc.CreateElement("name");
nameEl.InnerText = "Бойко О.П.";
newPatient.AppendChild(nameEl);

XmlElement wardEl = doc.CreateElement("ward");
wardEl.InnerText = "Кардіологія";
newPatient.AppendChild(wardEl);

root.AppendChild(newPatient);

// Видалення вузла
XmlNode? toRemove = doc.SelectSingleNode("/patients/patient[@id='PT-
1001']/ward");
toRemove?.ParentNode?.RemoveChild(toRemove);

// Результат
using System.IO.StringWriter sw = new System.IO.StringWriter();
using XmlTextWriter xw = new XmlTextWriter(sw) { Formatting =
Formatting.Indented };
doc.WriteTo(xw);
Console.WriteLine(sw.ToString());
```

Практичний сценарій: збереження результатів обстеження у XML

```C#
using System;
using System.Collections.Generic;
using System.IO;
using System.Xml;

string BuildExamXml(PatientExam exam)
{
    XmlDocument doc = new XmlDocument();
    doc.AppendChild(doc.CreateXmlDeclaration("1.0", "utf-8", null));

    XmlElement root = doc.CreateElement("examination");
    root.SetAttribute("version",  "2.0");
    root.SetAttribute("exportedAt", DateTime.UtcNow.ToString("yyyy-MM-ddTHH:mm:ssZ"));
    doc.AppendChild(root);

    // Пацієнт
    XmlElement pt = doc.CreateElement("patient");
    pt.SetAttribute("id", exam.PatientId.ToString());
    pt.InnerText = exam.PatientName;
    root.AppendChild(pt);

    // Дата обстеження
    XmlElement dateEl = doc.CreateElement("examDate");
    dateEl.InnerText = exam.ExamDate.ToString("yyyy-MM-dd");
    root.AppendChild(dateEl);

    // Результати
    XmlElement resultsEl = doc.CreateElement("labResults");
    resultsEl.SetAttribute("count", exam.Results.Count.ToString());
    root.AppendChild(resultsEl);

    foreach (ExamResult r in exam.Results)
    {
        XmlElement res = doc.CreateElement("result");
        res.SetAttribute("status", r.Status);
        res.SetAttribute("ref",    r.ReferenceRange);

        XmlElement testEl = doc.CreateElement("test");  testEl.InnerText = r.TestName;
        XmlElement valEl  = doc.CreateElement("value"); valEl.InnerText = r.Value.ToString("F2");
        valEl.SetAttribute("unit", r.Unit);

        res.AppendChild(testEl);
        res.AppendChild(valEl);
        resultsEl.AppendChild(res);
    }

    using StringWriter sw = new StringWriter();
    using XmlTextWriter xw = new XmlTextWriter(sw) { Formatting = Formatting.Indented, Indentation = 4 };
    doc.WriteTo(xw);
    return sw.ToString();
```

```
}

// Тест
var exam = new PatientExam(1001, "Петренко Іван Олексійович", DateTime.Now, new List<ExamResult>
{
    new("Гемоглобін",   135.0, "г/л",     "норма",       "120-160"),
    new("Глюкоза",      7.8,   "ммоль/л", "вище норми",  "3.9-6.1"),
    new("Лейкоцити",    6.2,   "10^9/л",  "норма",       "4.0-9.0"),
    new("Холестерин",   5.1,   "ммоль/л", "норма",       "<5.2"),
});

string resultXml = BuildExamXml(exam);
Console.WriteLine(resultXml);

// Збереження
string path = Path.Combine(Path.GetTempPath(), "exam_result.xml");
File.WriteAllText(path, resultXml, System.Text.Encoding.UTF8);
Console.WriteLine($"\nФайл: {new FileInfo(path).Length.ToString()} байт");
File.Delete(path);

record ExamResult(string TestName, double Value, string Unit, string Status, string ReferenceRange);
record PatientExam(int PatientId, string PatientName, DateTime ExamDate, List<ExamResult> Results);
```



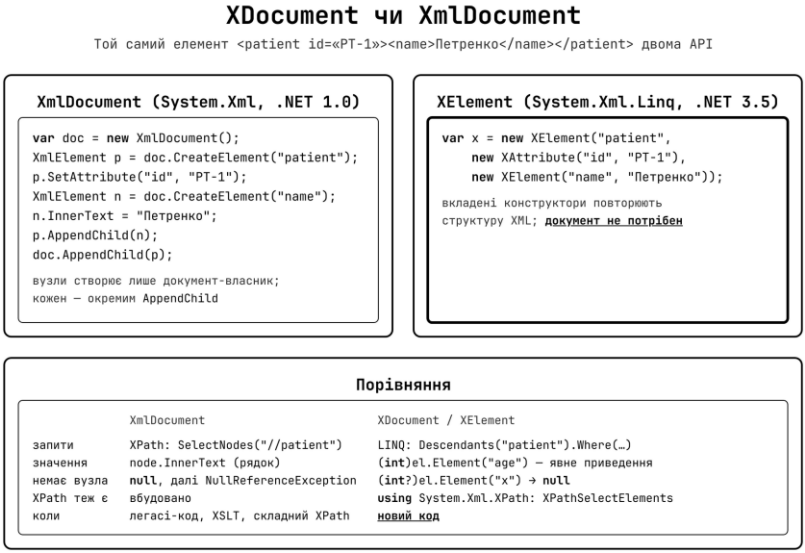
Методи XmlDocument по категоріях

19.3. XDocument та LINQ to XML

XmlDocument з'явився у .NET 1.0 і повністю реалізує стандарт DOM рівня 1. Але його API проектувався за зразком Java DOM і є досить багатослівним: щоб додати один елемент, потрібно викликати

CreateElement, SetAttribute, AppendChild — три окремих виклики. Навігація потребує явного приведення типів. А LINQ-запити до XmlNodeList — неможливі без ручного перебору.

.NET 3.5 разом із LINQ представив принципово інший підхід: System.Xml.Linq. Класи XDocument, XElement, XAttribute проектувалися **з нуля** під функціональний стиль і LINQ. Результат — код у 2–4 рази коротший, XML будується декларативно (через вкладені конструктори), а LINQ-запити працюють нативно на будь-якій колекції вузлів.



XDocument vs XmlDocument — порівняння API

Декларативне створення XML через XElement

Ключова перевага XDocument/XElement — **вкладений конструктор**. XML будується в точності так, як він виглядає: вкладені теги = вкладені об'єкти:

```
C#
 1  using System;
 2  using System.Xml.Linq;
 3
 4  // Один виклик — весь документ
 5  XDocument doc = new XDocument(
```

```
    new XDeclaration("1.0", "utf-8", null),
    new XComment("Медична картка пацієнта"),
    new XElement("patientRecord",
        new XAttribute("version", "2.0"),
        new XAttribute("generatedAt", DateTime.Now.ToString("yyyy-MM-
ddTHH:mm:ss")),

        new XElement("patient",
            new XAttribute("id", "PT-1001"),
            new XAttribute("status", "active"),
            new XElement("name",      "Петренко Іван Олексійович"),
            new XElement("birthDate", "1978-03-15"),
            new XElement("ward",      "Терапія")
        ),

        new XElement("diagnoses",
            new XElement("diagnosis",
                new XAttribute("code",   "J06.9"),
                new XAttribute("system", "ICD-10"),
                "ГРВІ"),
            new XElement("diagnosis",
                new XAttribute("code",   "I10"),
                new XAttribute("system", "ICD-10"),
                "Гіпертонічна хвороба")
        ),

        new XElement("vitals",
            new XAttribute("recordedAt", "2024-03-15T14:30:00"),
            new XElement("temperature", new XAttribute("unit", "C"),   37.2),
            new XElement("pulse",       new XAttribute("unit", "bpm"), 82),
            new XElement("bloodPressure",
                new XElement("systolic",  135),
                new XElement("diastolic",  85))
        )
    )
);

// Виведення з відступами
Console.WriteLine(doc.ToString());
```

Порівняйте: у XmlDocument той самий документ потребував би близько 40 рядків з явними CreateElement/AppendChild. Тут вся структура видна одним поглядом.

Збереження та завантаження

C#

```
using System;
using System.IO;
using System.Linq;
using System.Xml.Linq;

string path = Path.Combine(Path.GetTempPath(), "clinic_xdoc.xml");

XDocument doc = new XDocument(
    new XElement("clinic",
```

```
        new XElement("patient", new XAttribute("id","PT-1001"),
            new XElement("name","Петренко І.О."),
            new XElement("ward","Терапія")),
        new XElement("patient", new XAttribute("id","PT-1002"),
            new XElement("name","Бойко О.П."),
            new XElement("ward","Кардіологія"))
    )
);

// Save — з автоматичними відступами
doc.Save(path);
Console.WriteLine($"Збережено: {new FileInfo(path).Length.ToString()} байт");
Console.WriteLine(File.ReadAllText(path));

// Load — завантаження з файлу
XDocument loaded = XDocument.Load(path);
Console.WriteLine($"Кореневий елемент: {loaded.Root?.Name}");
Console.WriteLine($"Пацієнтів:
{loaded.Root?.Elements("patient").Count().ToString()}");

// Parse — завантаження з рядка
string xmlStr = "<root><item>A</item><item>B</item></root>";
XDocument parsed = XDocument.Parse(xmlStr);
Console.WriteLine($"Parse: {parsed.Root?.Elements().Count().ToString()}
елементів");

File.Delete(path);
```

Навігація по документу

```
C#
using System;
using System.Linq;
using System.Xml.Linq;

XDocument doc = XDocument.Parse("""
<clinic>
    <patient id="PT-1001" ward="Терапія">
        <name>Петренко Іван Олексійович</name>
        <diagnosis code="J06.9">ГРВІ</diagnosis>
        <diagnosis code="I10">Гіпертонія</diagnosis>
    </patient>
    <patient id="PT-1002" ward="Кардіологія">
        <name>Бойко Оксана Петрівна</name>
        <diagnosis code="I21.0">Інфаркт міокарда</diagnosis>
    </patient>
    <patient id="PT-1003" ward="Терапія">
        <name>Мороз Василь Іванович</name>
        <diagnosis code="E11.9">Діабет 2 типу</diagnosis>
    </patient>
</clinic>
""");

XElement root = doc.Root!;

// Element(name) — перший прямий дочірній елемент з такою назвою
XElement? first = root.Element("patient");
Console.WriteLine($"Перший: {first?.Attribute("id")?.Value}");
```

```
28
29  // Elements(name) — всі прямі дочірні з такою назвою
30  Console.WriteLine($"\nВсі пацієнти
    ({root.Elements("patient").Count().ToString()}):");
31  foreach (XElement p in root.Elements("patient"))
32  {
33      string id   = p.Attribute("id")?.Value ?? "";
34      string ward = p.Attribute("ward")?.Value ?? "";
35      string name = p.Element("name")?.Value ?? "";
36      Console.WriteLine($"  [{id}] {name} — {ward}");
37  }
38
39  // Descendants(name) — всі нащадки з назвою на будь-якій глибині
40  Console.WriteLine($"\nВсі діагнози
    ({root.Descendants("diagnosis").Count().ToString()}):");
41  foreach (XElement diag in root.Descendants("diagnosis"))
42  {
43      string code = diag.Attribute("code")?.Value ?? "";
44      Console.WriteLine($"  [{code}] {diag.Value}");
45  }
46
47  // Ancestors — навігація вгору
48  XElement? oneDiag = root.Descendants("diagnosis").FirstOrDefault();
49  Console.WriteLine($"\nБатьківський пацієнт діагнозу:
    {oneDiag?.Parent?.Element("name")?.Value}");
```

Метод	Що повертає
Element("name")	Перший прямий дочірній XElement з такою назвою
Elements("name")	Всі прямі дочірні XElement з такою назвою
Elements()	Всі прямі дочірні XElement
Descendants("name")	Всі нащадки з назвою (будь-яка глибина)
Descendants()	Всі нащадки
Ancestors("name")	Всі батьківські вузли з назвою (вгору)
Attribute("name")	XAttribute або null
Attributes()	Всі атрибути

LINQ-запити до XML

Найсильніша сторона XDocument — нативна інтеграція з LINQ. Будь-який метод, що повертає IEnumerable<XElement>, можна фільтрувати, трансформувати, групувати через Where, Select, GroupBy, OrderBy:

```
C#
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

XDocument doc = XDocument.Parse("""
<clinic date="2024-03-15">
    <patient id="PT-1001" ward="Терапія">
        <name>Петренко І.О.</name>
        <age>45</age>
        <diagnosis code="J06.9" severity="mild">ГРВІ</diagnosis>
    </patient>
    <patient id="PT-1002" ward="Кардіологія">
        <name>Бойко О.П.</name>
        <age>62</age>
        <diagnosis code="I21.0" severity="critical">Інфаркт міокарда</diagnosis>
    </patient>
    <patient id="PT-1003" ward="Терапія">
        <name>Мороз В.І.</name>
        <age>38</age>
        <diagnosis code="E11.9" severity="moderate">Діабет 2 типу</diagnosis>
    </patient>
    <patient id="PT-1004" ward="Кардіологія">
        <name>Руденко С.В.</name>
        <age>55</age>
        <diagnosis code="I10" severity="moderate">Гіпертонія</diagnosis>
    </patient>
</clinic>
""");

// 1. Фільтрація: пацієнти з відділення Кардіологія
var cardio = doc.Root!
    .Elements("patient")
    .Where(p => p.Attribute("ward")?.Value == "Кардіологія")
    .Select(p => p.Element("name")?.Value)
    .ToList();
Console.WriteLine($"Кардіологія ({cardio.Count.ToString()}): {string.Join(", ", cardio)}");

// 2. Сортування за віком
var byAge = doc.Root
    .Elements("patient")
    .OrderBy(p => int.Parse(p.Element("age")?.Value ?? "0"))
    .Select(p => $"{p.Element("name")?.Value} ({p.Element("age")?.Value} р.)");
Console.WriteLine($"\nЗа віком: {string.Join(" | ", byAge)}");

// 3. Критичні пацієнти
var critical = doc.Root
    .Descendants("diagnosis")
    .Where(d => d.Attribute("severity")?.Value == "critical")
```

```
    .Select(d => new {
        Patient = d.Parent?.Element("name")?.Value,
        Diag    = d.Value,
        Code    = d.Attribute("code")?.Value
    });
Console.WriteLine("\nКритичні стани:");
foreach (var c in critical)
    Console.WriteLine($"  [{c.Code}] {c.Patient}: {c.Diag}");

// 4. Групування за відділенням
var byWard = doc.Root
    .Elements("patient")
    .GroupBy(p => p.Attribute("ward")?.Value)
    .Select(g => new { Ward = g.Key, Count = g.Count(), AvgAge = g.Average(p =>
int.Parse(p.Element("age")?.Value ?? "0")) });
Console.WriteLine("\nПо відділеннях:");
foreach (var w in byWard)
    Console.WriteLine($"  {w.Ward}: {w.Count.ToString()} пацієнтів, сер. вік
{w.AvgAge.ToString("F0")} р.");
```

Зміна документа

```
C#
using System;
using System.Linq;
using System.Xml.Linq;

XDocument doc = XDocument.Parse("""
<patients>
    <patient id="PT-1001" status="active">
        <name>Петренко І.О.</name>
        <ward>Терапія</ward>
    </patient>
</patients>
""");

XElement root = doc.Root!;

// 1. Зміна значення атрибута
XElement? p1 = root.Elements("patient").FirstOrDefault(p =>
p.Attribute("id")?.Value == "PT-1001");
p1?.SetAttributeValue("status", "discharged");

// 2. Зміна тексту елемента
p1?.Element("ward")?.SetValue("Виписаний");

// 3. Додавання нового атрибута
p1?.SetAttributeValue("dischargedAt", DateTime.Now.ToString("yyyy-MM-dd"));

// 4. Додавання нового дочірнього елемента
p1?.Add(new XElement("note", "Виписаний за покращення стану"));

// 5. Видалення елемента
p1?.Element("ward")?.Remove();

// 6. Додавання нового пацієнта
root.Add(new XElement("patient",
    new XAttribute("id", "PT-1002"),
```

```
    new XAttribute("status", "active"),
    new XElement("name", "Бойко О.П."),
    new XElement("ward", "Кардіологія")
));

Console.WriteLine("Після змін:");
Console.WriteLine(doc.ToString());
```

SetAttributeValue — зручний метод: якщо атрибут існує — оновлює значення, не існує — додає. При значенні null — видаляє атрибут. SetValue — аналогічно для текстового вмісту елемента.

Генерація XML з колекцій C#

Особливо зручно використовувати конструктор XElement з IEnumerable<XElement> — можна передавати LINQ-вираз як аргумент:

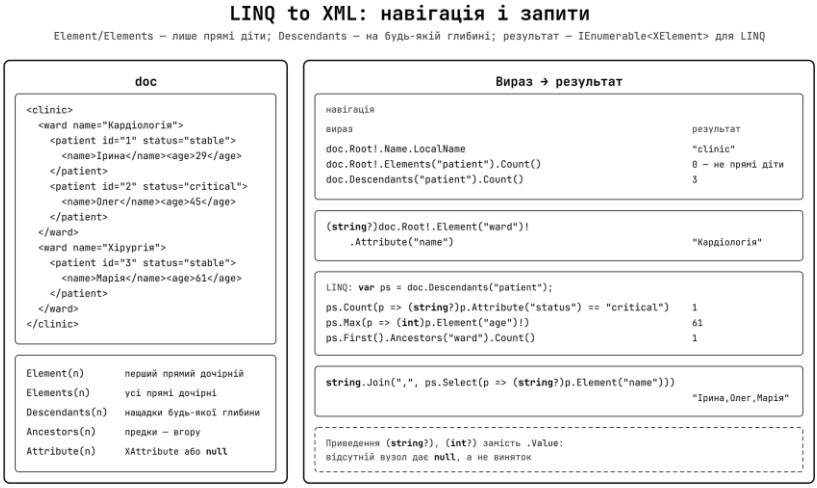
```
C#
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

var results = new List<LabResult>
{
    new(1001, "Петренко І.О.", "Гемоглобін",  135.0, "г/л",     "норма"),
    new(1001, "Петренко І.О.", "Глюкоза",     5.1,   "ммоль/л", "норма"),
    new(1002, "Бойко О.П.",    "Глюкоза",     8.7,   "ммоль/л", "вище норми"),
    new(1002, "Бойко О.П.",    "Холестерин",  5.8,   "ммоль/л", "вище норми"),
    new(1003, "Мороз В.І.",    "Гемоглобін",  98.0,  "г/л",     "нижче норми"),
};

// Генерація XML з групуванням по пацієнту
XDocument report = new XDocument(
    new XDeclaration("1.0", "utf-8", null),
    new XElement("labReport",
        new XAttribute("generatedAt", DateTime.Now.ToString("yyyy-MM-
ddTHH:mm:ss")),
        new XAttribute("totalResults", results.Count.ToString()),

        // LINQ прямо у конструкторі XElement
        from g in results.GroupBy(r => r.PatientId)
        select new XElement("patient",
            new XAttribute("id",   g.Key.ToString()),
            new XAttribute("name", g.First().Name),
            new XAttribute("resultsCount", g.Count().ToString()),
            from r in g
            select new XElement("result",
                new XAttribute("status", r.Status),
                new XElement("test",  r.Test),
                new XElement("value", new XAttribute("unit", r.Unit),
r.Value.ToString("F2"))
            )
        )
    )
);
```

```
37
38  Console.WriteLine(report.ToString());
39
40  // Підрахунок відхилень
41  int abnormal = report.Descendants("result")
42      .Count(r => r.Attribute("status")?.Value != "норма");
43  Console.WriteLine($"\nВідхилень від норми: {abnormal.ToString()} з
    {results.Count.ToString()}");
44
45  record LabResult(int PatientId, string Name, string Test, double Value, string
    Unit, string Status);
```

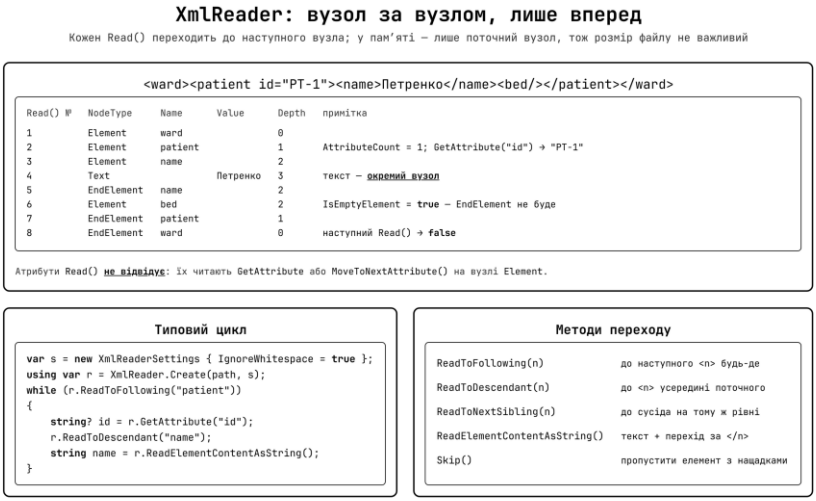


LINQ to XML — схема запитів через Descendants та Elements

19.4. XmlReader та XmlWriter. Потоковий XML

XmlDocument і XDocument зручні, але мають спільний недолік: вони завантажують **весь XML-документ у пам'ять** як дерево об'єктів. Для файлу в 10 МВ це прийнятно. Для XML-експорту з 500 000 лабораторних результатів, XML-архіву DICOM-даних або потоку HL7-повідомлень — DOM-підхід призведе до OutOfMemoryException або неприйнятного часу завантаження.

XmlReader і XmlWriter реалізують **потоковий (streaming)** підхід: читання і запис відбуваються **послідовно, вузол за вузлом**, без побудови повного дерева в пам'яті. Використання пам'яті константне незалежно від розміру файлу — тільки один поточний вузол.



XmlReader — forward-only потокове читання XML

XmlReader — forward-only читання

XmlReader рухається по XML **в одному напрямку** — вперед. Не можна повернутися до попереднього вузла (на відміну від DOM). Кожен виклик Read() просуває позицію на наступний вузол і встановлює NodeType.

Базове читання

```C#
using System;
using System.IO;
using System.Xml;

string xml = """
<?xml version="1.0" encoding="utf-8"?>
<clinic>
    <patient id="PT-1001" ward="Терапія">
        <name>Петренко Іван Олексійович</name>
        <age>45</age>
        <diagnosis code="J06.9">ГРВІ</diagnosis>
    </patient>
    <patient id="PT-1002" ward="Кардіологія">
        <name>Бойко Оксана Петрівна</name>
        <age>62</age>
        <diagnosis code="I21.0">Інфаркт міокарда</diagnosis>
    </patient>
</clinic>
```

```
"""";

var settings = new XmlReaderSettings
{
    IgnoreWhitespace = true,   // пропускати текстові вузли з пробілами
    IgnoreComments   = true,   // пропускати коментарі
};

using XmlReader reader = XmlReader.Create(new StringReader(xml), settings);

Console.WriteLine("Всі вузли (тип | назва | значення):");
while (reader.Read())
{
    string indent = new string(' ', reader.Depth * 2);
    switch (reader.NodeType)
    {
        case XmlNodeType.XmlDeclaration:
            Console.WriteLine($"{indent}[Decl] <?xml {reader.Value}?>");
            break;
        case XmlNodeType.Element:
            Console.Write($"{indent}[Elem] <{reader.Name}");
            if (reader.HasAttributes)
            {
                // Зберігаємо атрибути поточного елемента
                for (int i = 0; i < reader.AttributeCount; i++)
                {
                    reader.MoveToAttribute(i);
                    Console.Write($" {reader.Name}=\"{reader.Value}\"");
                }
                reader.MoveToElement(); // повернутись до елемента після
атрибутів
            }
            Console.WriteLine(reader.IsEmptyElement ? "/>" : ">");
            break;
        case XmlNodeType.Text:
            Console.WriteLine($"{indent}[Text] \"{reader.Value}\"");
            break;
        case XmlNodeType.EndElement:
            Console.WriteLine($"{indent}[End]  </{reader.Name}>");
            break;
    }
}
```

reader.Depth показує рівень вкладеності поточного вузла — зручно для відступів. reader.IsEmptyElement — true для <tag/> без вмісту.

ReadToFollowing, ReadElementContentAsString та GetAttribute

```
C#
using System;
using System.IO;
using System.Xml;
using System.Collections.Generic;

string xml = """
<?xml version="1.0"?>
<clinic>
```

```
    <patient id="PT-1001" ward="Терапія">
        <name>Петренко Іван Олексійович</name>
        <age>45</age>
        <diagnosis code="J06.9">ГРВІ</diagnosis>
        <diagnosis code="I10">Гіпертонія</diagnosis>
    </patient>
    <patient id="PT-1002" ward="Кардіологія">
        <name>Бойко Оксана Петрівна</name>
        <age>62</age>
        <diagnosis code="I21.0">Інфаркт міокарда</diagnosis>
    </patient>
</clinic>
""";

var settings = new XmlReaderSettings { IgnoreWhitespace = true };
using XmlReader reader = XmlReader.Create(new StringReader(xml), settings);

int patientCount = 0;
var allDiagnoses = new List<string>();

// ReadToFollowing — переходить до наступного елемента з такою назвою
while (reader.ReadToFollowing("patient"))
{
    patientCount++;
    // GetAttribute — читання атрибуту поточного елемента
    string id   = reader.GetAttribute("id") ?? "";
    string ward = reader.GetAttribute("ward") ?? "";

    // ReadToDescendant — переходить до першого дочірнього з такою назвою
    reader.ReadToDescendant("name");
    // ReadElementContentAsString — читає текст і просуває позицію за
EndElement
    string name = reader.ReadElementContentAsString();

    // reader вже стоїть на <age>: ReadElementContentAsString перейшов за
</name>
    // до наступного вузла. ReadToNextSibling("age") тут пропустив би поточний
<age>!
    int age = reader.ReadElementContentAsInt();

    Console.WriteLine($"  [{id}] {name}, {age.ToString()} р. ({ward})");

    // Читаємо всі diagnosis для цього пацієнта: reader уже на першому
<diagnosis>
    while (reader.NodeType == XmlNodeType.Element && reader.Name ==
"diagnosis")
    {
        string code = reader.GetAttribute("code") ?? "";
        string text = reader.ReadElementContentAsString();
        allDiagnoses.Add($"[{code}] {text}");
    }
}

Console.WriteLine($"\nПацієнтів: {patientCount.ToString()}");
Console.WriteLine($"Всі діагнози ({allDiagnoses.Count.ToString()}):");
foreach (string diag in allDiagnoses)
    Console.WriteLine($"  {diag}");
```

Метод	Опис
Read()	Наступний вузол; false = кінець
ReadToFollowing("name")	Перемотати до наступного елемента з назвою
ReadToDescendant("name")	Перемотати до першого дочірнього
ReadToNextSibling("name")	Перемотати до **наступного** сусіда з назвою (поточний вузол не перевіряється!)
ReadElementContentAsString()	Прочитати текст і перейти за </tag> — на наступний вузол
ReadElementContentAsInt()	Те саме для int
ReadElementContentAsDouble()	Те саме для double
GetAttribute("name")	Атрибут поточного Element-вузла
Skip()	Пропустити поточний елемент з усіма дочірніми

Обробка великого XML-файлу потоково

```
C#
 1  using System;
 2  using System.IO;
 3  using System.Xml;
 4  using System.Collections.Generic;
 5
 6  // Генеруємо великий тестовий XML
 7  string xmlPath = Path.Combine(Path.GetTempPath(), "large_clinic.xml");
 8  using (XmlWriter gen = XmlWriter.Create(xmlPath, new XmlWriterSettings { Indent
    = true }))
 9  {
10      gen.WriteStartDocument();
11      gen.WriteStartElement("clinicExport");
12      gen.WriteAttributeString("date", DateTime.Now.ToString("yyyy-MM-dd"));
13
14      string[] diagnoses = { "J06.9", "I10", "E11.9", "K29.5", "M54.5" };
15      string[] wards     = { "Терапія", "Кардіологія", "Неврологія" };
16
17      for (int i = 1; i <= 500; i++)
18      {
19          gen.WriteStartElement("patient");
20          gen.WriteAttributeString("id",   $"PT-{i.ToString("D4")}");
21          gen.WriteAttributeString("ward", wards[i % wards.Length]);
22          gen.WriteElementString("name",   $"Пацієнт #{i.ToString()}");
23          gen.WriteElementString("age",    (20 + i % 60).ToString());
```

```
        gen.WriteElementString("diagnosis", diagnoses[i % diagnoses.Length]);
        gen.WriteEndElement();
    }

    gen.WriteEndElement();
    gen.WriteEndDocument();
}

var fileInfo = new FileInfo(xmlPath);
Console.WriteLine($"XML: {fileInfo.Length.ToString()} байт, ~500 пацієнтів");

// Потокова обробка — пам'яті потрібно лише для одного пацієнта
int total = 0, cardioCount = 0;
var settings = new XmlReaderSettings { IgnoreWhitespace = true };

using (XmlReader reader = XmlReader.Create(xmlPath, settings))
{
    while (reader.ReadToFollowing("patient"))
    {
        total++;
        string ward = reader.GetAttribute("ward") ?? "";
        if (ward == "Кардіологія") cardioCount++;

        // Skip() — пропускаємо вміст пацієнта без читання
        reader.Skip();
    }
}

Console.WriteLine($"Оброблено: {total.ToString()} пацієнтів");
Console.WriteLine($"Кардіологія: {cardioCount.ToString()}");
Console.WriteLine("Витрачено пам'яті: ~константна (лише один вузол
одночасно)");

File.Delete(xmlPath);
```

XmlWriter — потоковий запис XML

XmlWriter записує XML **послідовно** — елемент за елементом. Він гарантує синтаксичну коректність: відкритий тег без WriteEndElement призведе до помилки при Dispose.

Базовий запис

```
C#
using System;
using System.IO;
using System.Text;
using System.Xml;

var settings = new XmlWriterSettings
{
    Indent            = true,          // відступи
    IndentChars       = "    ",        // 4 пробіли
    Encoding          = Encoding.UTF8,
    OmitXmlDeclaration = false,        // включати <?xml ...?>
    NewLineChars      = "\n",          // Unix-стиль переносів
```

```
};

using MemoryStream ms = new MemoryStream();
using (XmlWriter writer = XmlWriter.Create(ms, settings))
{
    writer.WriteStartDocument();

    // Коментар
    writer.WriteComment("Медична звітність 2024");

    writer.WriteStartElement("medReport");
    writer.WriteAttributeString("version",     "1.0");
    writer.WriteAttributeString("generatedAt", DateTime.Now.ToString("yyyy-MM-ddTHH:mm:ss"));

    // Пацієнт 1
    writer.WriteStartElement("patient");
    writer.WriteAttributeString("id",     "PT-1001");
    writer.WriteAttributeString("status", "active");

    writer.WriteElementString("name",      "Петренко Іван Олексійович");
    writer.WriteElementString("birthDate", "1978-03-15");
    writer.WriteElementString("ward",      "Терапія");

    // Вкладений елемент з атрибутом і текстом
    writer.WriteStartElement("diagnosis");
    writer.WriteAttributeString("code",   "J06.9");
    writer.WriteAttributeString("system", "ICD-10");
    writer.WriteString("ГРВІ");          // текстовий вміст
    writer.WriteEndElement();            // </diagnosis>

    writer.WriteEndElement(); // </patient>

    // WriteRaw — записує XML-рядок без escaping
    writer.WriteRaw("\n    <!-- кінець звіту -->");

    writer.WriteEndElement(); // </medReport>
    writer.WriteEndDocument();
}

string result = Encoding.UTF8.GetString(ms.ToArray());
Console.WriteLine(result);
```

Серіалізація колекції у XML через XmlWriter

```
C#
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using System.Xml;

var patients = new List<Patient>
{
    new(1001, "Петренко І.О.", "Терапія",     new[]{"J06.9","I10"},    5.1, 72),
    new(1002, "Бойко О.П.",   "Кардіологія", new[]{"I21.0"},          8.7, 145),
```

```
    new(1003, "Мороз В.І.",   "Неврологія",  new[]{"G43.9","G40.9"},  4.8, 68),
};

string path = Path.Combine(Path.GetTempPath(), "patients_export.xml");

var settings = new XmlWriterSettings { Indent = true, Encoding = Encoding.UTF8 };
using (XmlWriter w = XmlWriter.Create(path, settings))
{
    w.WriteStartDocument();
    w.WriteStartElement("clinicExport");
    w.WriteAttributeString("exportedAt", DateTime.Now.ToString("O"));
    w.WriteAttributeString("count",      patients.Count.ToString());

    foreach (Patient p in patients)
    {
        w.WriteStartElement("patient");
        w.WriteAttributeString("id",   p.Id.ToString());
        w.WriteAttributeString("ward", p.Ward);

        w.WriteElementString("name", p.Name);

        // Числові значення
        w.WriteStartElement("vitals");
        w.WriteElementString("glucose", p.Glucose.ToString("F1"));
        w.WriteElementString("pulse",   p.Pulse.ToString());
        w.WriteEndElement(); // </vitals>

        // Масив діагнозів
        w.WriteStartElement("diagnoses");
        foreach (string code in p.Diagnoses)
            w.WriteElementString("code", code);
        w.WriteEndElement(); // </diagnoses>

        // Умовний елемент — тільки якщо критичний стан
        bool critical = p.Glucose > 7.0 || p.Pulse > 120;
        if (critical)
        {
            w.WriteStartElement("alert");
            w.WriteAttributeString("level", "high");
            w.WriteString(p.Glucose > 7.0 ? $"Глюкоза: {p.Glucose.ToString("F1")}" : $"Пульс: {p.Pulse.ToString()}");
            w.WriteEndElement();
        }

        w.WriteEndElement(); // </patient>
    }

    w.WriteEndElement(); // </clinicExport>
    w.WriteEndDocument();
}

Console.WriteLine(File.ReadAllText(path, Encoding.UTF8));
Console.WriteLine($"Файл: {new FileInfo(path).Length.ToString()} байт");
File.Delete(path);

record Patient(int Id, string Name, string Ward, string[] Diagnoses, double Glucose, int Pulse);
```

XmlReader + XmlWriter: трансформація великих файлів

Найпотужніший патерн — читати через XmlReader і одночасно писати через XmlWriter. Це дозволяє трансформувати XML будь-якого розміру при константному використанні пам'яті:

```C#
using System;
using System.IO;
using System.Xml;

// Генеруємо вхідний файл
string inputPath  = Path.Combine(Path.GetTempPath(), "input_clinic.xml");
string outputPath = Path.Combine(Path.GetTempPath(), "output_critical.xml");

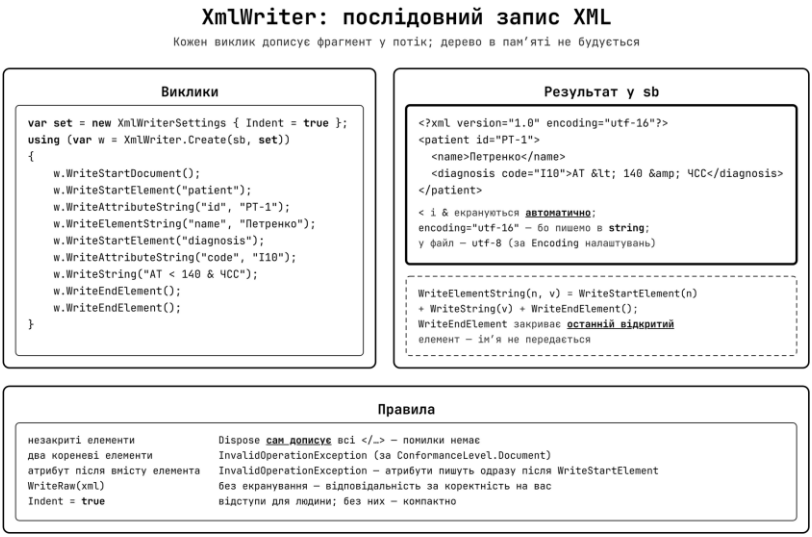
// Вхідний XML з пацієнтами
using (XmlWriter gen = XmlWriter.Create(inputPath, new XmlWriterSettings { Indent = true }))
{
    gen.WriteStartDocument();
    gen.WriteStartElement("patients");
    var data = new[] {
        ("PT-1001","Петренко І.О.", 5.1,  72,  "Терапія"),
        ("PT-1002","Бойко О.П.",    8.7,  145, "Кардіологія"),
        ("PT-1003","Мороз В.І.",    4.8,  68,  "Терапія"),
        ("PT-1004","Руденко С.В.", 12.4,  110, "Кардіологія"),
    };
    foreach (var (id,name,gluc,pulse,ward) in data)
    {
        gen.WriteStartElement("patient");
        gen.WriteAttributeString("id", id);
        gen.WriteAttributeString("ward", ward);
        gen.WriteElementString("name", name);
        gen.WriteElementString("glucose", gluc.ToString("F1"));
        gen.WriteElementString("pulse", pulse.ToString());
        gen.WriteEndElement();
    }
    gen.WriteEndElement();
    gen.WriteEndDocument();
}

// Трансформація: копіюємо лише критичних пацієнтів
int processed = 0, written = 0;
var rSettings = new XmlReaderSettings { IgnoreWhitespace = true };
var wSettings = new XmlWriterSettings { Indent = true };

using (XmlReader reader = XmlReader.Create(inputPath, rSettings))
using (XmlWriter writer = XmlWriter.Create(outputPath, wSettings))
{
    writer.WriteStartDocument();
    writer.WriteStartElement("criticalPatients");
    writer.WriteAttributeString("filter", "glucose>7 OR pulse>120");

    while (reader.ReadToFollowing("patient"))
    {
        processed++;
        string id   = reader.GetAttribute("id") ?? "";
        string ward = reader.GetAttribute("ward") ?? "";
```

```
51
52          // Читаємо дані пацієнта у змінні
53          reader.ReadToDescendant("name");
54          string name   = reader.ReadElementContentAsString();
55          double gluc   = reader.ReadElementContentAsDouble();
56          int    pulse  = reader.ReadElementContentAsInt();
57
58          bool critical = gluc > 7.0 || pulse > 120;
59          if (!critical) continue;
60
61          // Пишемо у вихідний файл
62          written++;
63          writer.WriteStartElement("patient");
64          writer.WriteAttributeString("id",   id);
65          writer.WriteAttributeString("ward", ward);
66          writer.WriteElementString("name",   name);
67          writer.WriteElementString("glucose", gluc.ToString("F1"));
68          writer.WriteElementString("pulse",   pulse.ToString());
69
70          string reason = gluc > 7.0
71              ? $"глюкоза {gluc.ToString("F1")} ммоль/л"
72              : $"пульс {pulse.ToString()} уд/хв";
73          writer.WriteElementString("alertReason", reason);
74
75          writer.WriteEndElement();
76      }
77
78      writer.WriteEndElement();
79      writer.WriteEndDocument();
80  }
81
82  Console.WriteLine($"Оброблено: {processed.ToString()} пацієнтів");
83  Console.WriteLine($"Критичних: {written.ToString()}");
84  Console.WriteLine($"\nРезультат:");
85  Console.WriteLine(File.ReadAllText(outputPath));
86
87  File.Delete(inputPath);
88  File.Delete(outputPath);
```

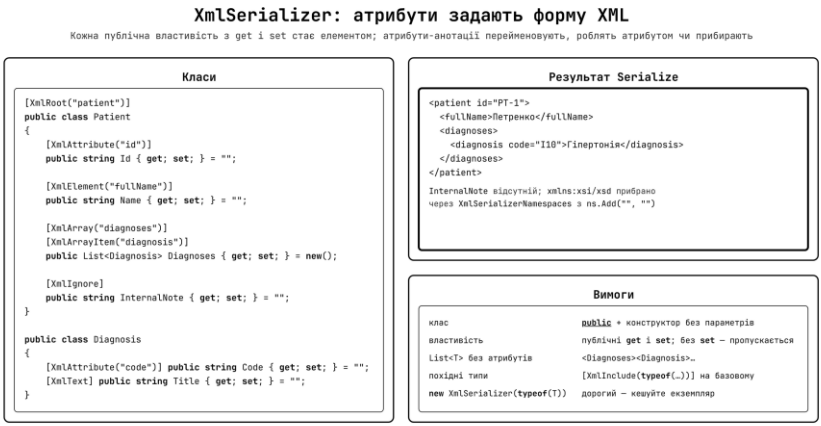


XmlWriter — послідовний запис елементів і атрибутів

19.5. XmlSerializer — атрибутна серіалізація

XmlSerializer із простору імен System.Xml.Serialization перетворює .NET-об'єкти у XML і назад за допомогою **атрибутів-анотацій** прямо на класах та властивостях. Це класичний підхід «contract-first»: ви описуєте структуру XML декораторами, а серіалізатор будує і читає документ автоматично.

На відміну від JsonSerializer, XmlSerializer потребує **публічного конструктора без параметрів** і **публічних властивостей або полів** — це жорстка вимога, пов'язана із генерацією коду серіалізатора під час створення екземпляра XmlSerializer. Ця генерація дорога (рефлексія й компіляція збірки в пам'яті), тому конструктор new XmlSerializer(typeof(T)) кешує результат лише для простих перевантажень; у гарячому коді екземпляр серіалізатора варто створити один раз і зберегти в static readonly-полі. Сам клас також має бути **публічним** — інакше конструктор кине InvalidOperationException, а властивості без сеттера (лише get) серіалізатор мовчки пропускає.



XmlSerializer — атрибути та схема потоку

Базова серіалізація та десеріалізація

```C#
using System;
using System.IO;
using System.Xml.Serialization;

// Серіалізація
Patient patient = new Patient { Id = 1001, Name = "Петренко Іван", Ward = "Терапія", Age = 45 };

XmlSerializer serializer = new XmlSerializer(typeof(Patient));

// У рядок через StringWriter
using StringWriter sw = new StringWriter();
serializer.Serialize(sw, patient);
string xml = sw.ToString();
Console.WriteLine(xml);

// Десеріалізація
using StringReader sr = new StringReader(xml);
Patient? loaded = (Patient?)serializer.Deserialize(sr);
Console.WriteLine($"\nЗавантажено: {loaded?.Name}, відділення: {loaded?.Ward}");

public class Patient
{
    public int    Id       { get; set; }
    public string Name     { get; set; } = "";
    public string Ward     { get; set; } = "";
    public int    Age      { get; set; }
}
```

За замовчуванням ім'я кореневого елемента збігається з ім'ям класу, а кожна властивість стає дочірнім елементом. Атрибути-анотації дозволяють повністю змінити цю поведінку.

Атрибути XmlSerializer

[XmlRoot] — кореневий елемент

```
C#
 1  using System;
 2  using System.IO;
 3  using System.Xml.Serialization;
 4
 5  PatientRecord rec = new PatientRecord
 6  {
 7      PatientId    = "PT-1001",
 8      Name         = "Петренко Іван Олексійович",
 9      Ward         = "Терапія",
10      Age          = 45,
11      LastAccessed = DateTime.Now   // буде проігноровано
12  };
13
14  XmlSerializer xs = new XmlSerializer(typeof(PatientRecord));
15  using StringWriter sw = new StringWriter();
16  xs.Serialize(sw, rec);
17  Console.WriteLine(sw.ToString());
18
19  [XmlRoot("patient", Namespace = "http://clinic.ua/2024")]
20  public class PatientRecord
21  {
22      // Властивість → XML-атрибут (не елемент)
23      [XmlAttribute("id")]
24      public string PatientId { get; set; } = "";
25
26      // Властивість → елемент з іншою назвою
27      [XmlElement("fullName")]
28      public string Name { get; set; } = "";
29
30      // Властивість → елемент у просторі імен
31      [XmlElement("ward", Namespace = "http://clinic.ua/2024")]
32      public string Ward { get; set; } = "";
33
34      // Властивість повністю ігнорується
35      [XmlIgnore]
36      public DateTime LastAccessed { get; set; }
37
38      // Числова властивість без зміни імені
39      public int Age { get; set; }
40  }
```

Атрибут	Що робить
[XmlRoot("tag")]	Перейменовує кореневий елемент

Атрибут	Що робить
[XmlElement("tag")]	Серіалізує властивість як XML-елемент з даним ім'ям
[XmlAttribute("attr")]	Серіалізує властивість як XML-атрибут
[XmlIgnore]	Виключає властивість з серіалізації
[XmlText]	Серіалізує властивість як текстовий вміст елемента
[XmlArray("list")]	Перейменовує обгортковий елемент колекції
[XmlArrayItem("item")]	Перейменовує елементи всередині колекції

[XmlArray] та [XmlArrayItem] — серіалізація колекцій

```C#
using System;
using System.Collections.Generic;
using System.IO;
using System.Xml.Serialization;

FullPatientRecord record = new FullPatientRecord
{
    Id   = "PT-1001",
    Name = "Петренко Іван Олексійович",
    Diagnoses = new List<DiagnosisEntry>
    {
        new() { Code = "J06.9", Description = "ГРВІ" },
        new() { Code = "I10",   Description = "Гіпертонія" }
    },
    Vitals = new List<VitalSign>
    {
        new() { Type = "temperature", Unit = "C",   Value = "37.2" },
        new() { Type = "pulse",       Unit = "bpm", Value = "82"   }
    }
};

XmlSerializer xs = new XmlSerializer(typeof(FullPatientRecord));
using StringWriter sw = new StringWriter();
xs.Serialize(sw, record);
Console.WriteLine(sw.ToString());

public class DiagnosisEntry
{
    [XmlAttribute("code")]
    public string Code { get; set; } = "";

```

```
    [XmlAttribute("system")]
    public string System { get; set; } = "ICD-10";

    [XmlText]
    public string Description { get; set; } = "";
}

public class VitalSign
{
    [XmlAttribute("type")]
    public string Type  { get; set; } = "";

    [XmlAttribute("unit")]
    public string Unit  { get; set; } = "";

    [XmlText]
    public string Value { get; set; } = "";
}

[XmlRoot("patientRecord")]
public class FullPatientRecord
{
    [XmlAttribute("id")]
    public string Id { get; set; } = "";

    [XmlElement("name")]
    public string Name { get; set; } = "";

    // List → <diagnoses><diagnosis ...>
    [XmlArray("diagnoses")]
    [XmlArrayItem("diagnosis")]
    public List<DiagnosisEntry> Diagnoses { get; set; } = new();

    // List → <vitals><sign ...>
    [XmlArray("vitals")]
    [XmlArrayItem("sign")]
    public List<VitalSign> Vitals { get; set; } = new();
}
```

Серіалізація з простором імен

```
C#
using System;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

ClinicDocument doc = new ClinicDocument
{
    ClinicName = "Міська клінічна лікарня №5",
    RegDate    = "2024-01-01"
};

XmlSerializer xs   = new XmlSerializer(typeof(ClinicDocument));
XmlSerializerNamespaces ns = new XmlSerializerNamespaces();
ns.Add("clinic", "http://clinic.ua/schema/2024");
ns.Add("",       "");   // прибирає зайвий xsi/xsd

```

```
using StringWriter sw = new StringWriter();
xs.Serialize(sw, doc, ns);
Console.WriteLine(sw.ToString());

[XmlRoot("clinic", Namespace = "http://clinic.ua/schema/2024")]
public class ClinicDocument
{
    [XmlElement("name")]
    public string ClinicName { get; set; } = "";

    [XmlElement("registrationDate")]
    public string RegDate { get; set; } = "";
}
```

XmlSerializerNamespaces дозволяє додавати або прибирати простори імен з вихідного XML. Якщо передати ns.Add("", "") — серіалізатор не додає зайвих xmlns:xsi та xmlns:xsd.

Десеріалізація з файлу та валідація

C#

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Xml.Serialization;

string xmlData = """
<?xml version="1.0" encoding="utf-16"?>
<examination patientId="PT-1002">
  <examDate>2024-03-15</examDate>
  <conclusion>Стан задовільний</conclusion>
  <tests>
    <test name="Гемоглобін" status="норма">135</test>
    <test name="Глюкоза" status="вище норми">8.7</test>
    <test name="Холестерин" status="норма">4.9</test>
  </tests>
</examination>
""";

XmlSerializer xs = new XmlSerializer(typeof(ExamRecord));

// Десеріалізація
using StringReader sr = new StringReader(xmlData);
ExamRecord? exam = (ExamRecord?)xs.Deserialize(sr);

if (exam != null)
{
    Console.WriteLine($"Пацієнт: {exam.PatientId}");
    Console.WriteLine($"Дата: {exam.ExamDate}");
    Console.WriteLine($"Висновок: {exam.Conclusion}");
    Console.WriteLine($"Тестів: {exam.Tests.Count.ToString()}");
    foreach (TestResult t in exam.Tests)
        Console.WriteLine($"  {t.Name}: {t.Value} [{t.Status}]");
}

[XmlRoot("examination")]
```

```
public class ExamRecord
{
    [XmlAttribute("patientId")]
    public string PatientId { get; set; } = "";

    [XmlElement("examDate")]
    public string ExamDate { get; set; } = "";

    [XmlElement("conclusion")]
    public string Conclusion { get; set; } = "";

    [XmlArray("tests")]
    [XmlArrayItem("test")]
    public List<TestResult> Tests { get; set; } = new();
}

public class TestResult
{
    [XmlAttribute("name")]
    public string Name { get; set; } = "";

    [XmlAttribute("status")]
    public string Status { get; set; } = "";

    [XmlText]
    public string Value { get; set; } = "";
}
```

Наслідування та [XmlInclude]

C#

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Xml.Serialization;

LabContainer lab = new LabContainer
{
    Tests = new List<MedicalTest>
    {
        new BloodTest { TestId = "BT-001", Date = "2024-03-15", Hemoglobin = 135.0, Glucose = 5.1 },
        new EcgRecord { TestId = "EC-001", Date = "2024-03-15", HeartRate = 72, Rhythm = "Синусовий" }
    }
};

XmlSerializer xs = new XmlSerializer(typeof(LabContainer));
using StringWriter sw = new StringWriter();
xs.Serialize(sw, lab);
string xml = sw.ToString();
Console.WriteLine(xml);

// Roundtrip — читаємо назад
using StringReader sr = new StringReader(xml);
LabContainer? loaded = (LabContainer?)xs.Deserialize(sr);
Console.WriteLine($"\nЗавантажено {loaded?.Tests.Count.ToString()} тестів:");
foreach (MedicalTest t in loaded?.Tests ?? new())
```

```
    Console.WriteLine($"  [{t.GetType().Name}] {t.TestId} — {t.Date}");

// Базовий клас — треба оголосити всі похідні типи
[XmlInclude(typeof(BloodTest))]
[XmlInclude(typeof(EcgRecord))]
public abstract class MedicalTest
{
    [XmlAttribute("id")]
    public string TestId { get; set; } = "";

    [XmlElement("date")]
    public string Date { get; set; } = "";
}

public class BloodTest : MedicalTest
{
    [XmlElement("hemoglobin")]
    public double Hemoglobin { get; set; }

    [XmlElement("glucose")]
    public double Glucose { get; set; }
}

public class EcgRecord : MedicalTest
{
    [XmlElement("heartRate")]
    public int HeartRate { get; set; }

    [XmlElement("rhythm")]
    public string Rhythm { get; set; } = "";
}

[XmlRoot("lab")]
public class LabContainer
{
    // XmlSerializer серіалізує через [XmlInclude] типи з xsi:type
    [XmlElement("test")]
    public List<MedicalTest> Tests { get; set; } = new();
}
```

[XmlInclude] обов'язковий, якщо ви серіалізуєте колекцію базового типу і очікуєте отримати правильні похідні типи після десеріалізації. Серіалізатор додає xsi:type="BloodTest" у XML, і при зворотному читанні відновлює правильний тип.

Практичний сценарій: збереження протоколу обходу

```
C#
using System;
using System.Collections.Generic;
using System.IO;
using System.Xml.Serialization;

WardRoundProtocol protocol = new WardRoundProtocol
{
    Date      = DateTime.Now.ToString("yyyy-MM-dd"),
```

```
    Physician = "Коваленко О.В.",
    Patients  = new List<WardPatient>
    {
        new("РТ-1001", "Петренко І.О.", "Терапія",     "Стан покращився, t=36.8", "задовільний"),
        new("РТ-1002", "Бойко О.П.",    "Кардіологія", "ЕКГ в нормі, АТ 125/80",  "стабільний"),
        new("РТ-1003", "Мороз В.І.",    "Терапія",     "Скарги на слабкість", "спостереження"),
    }
};

XmlSerializer xs = new XmlSerializer(typeof(WardRoundProtocol));
string path = Path.Combine(Path.GetTempPath(), "ward_round.xml");

// Серіалізація у файл
using (FileStream fs = File.Create(path))
    xs.Serialize(fs, protocol);

Console.WriteLine($"Збережено ({new FileInfo(path).Length.ToString()} байт):");
Console.WriteLine(File.ReadAllText(path));

// Десеріалізація з файлу
using FileStream fsr = File.OpenRead(path);
WardRoundProtocol? loaded = (WardRoundProtocol?)xs.Deserialize(fsr);

Console.WriteLine($"\nПротокол від {loaded?.Date}, лікар: {loaded?.Physician}");
Console.WriteLine($"Пацієнтів: {loaded?.Patients.Count.ToString()}");
foreach (WardPatient p in loaded?.Patients ?? new())
    Console.WriteLine($"  [{p.Id}] {p.Name} — {p.Status}");

File.Delete(path);

[XmlRoot("wardRound")]
public class WardRoundProtocol
{
    [XmlAttribute("date")]
    public string Date { get; set; } = "";

    [XmlAttribute("physician")]
    public string Physician { get; set; } = "";

    [XmlArray("patients")]
    [XmlArrayItem("patient")]
    public List<WardPatient> Patients { get; set; } = new();
}

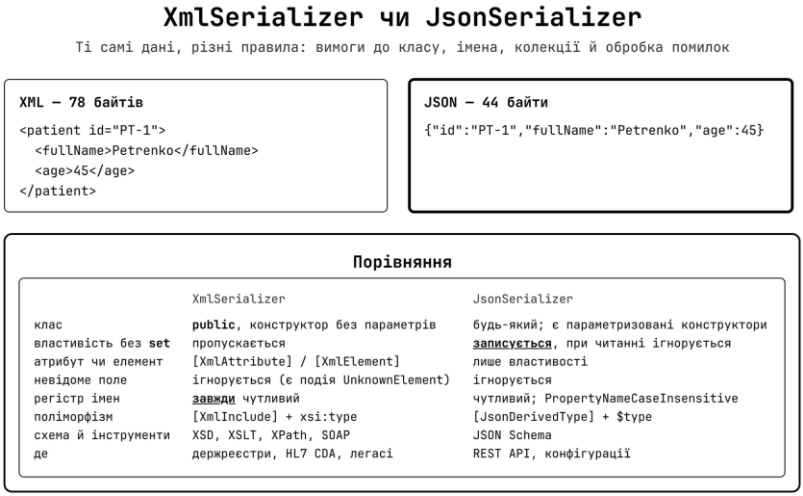
public class WardPatient
{
    [XmlAttribute("id")]
    public string Id { get; set; } = "";

    [XmlElement("name")]
    public string Name { get; set; } = "";

    [XmlElement("ward")]
    public string Ward { get; set; } = "";

    [XmlElement("notes")]
```

```
65      public string Notes { get; set; } = "";
66
67      [XmlElement("status")]
68      public string Status { get; set; } = "";
69
70      // Порожній конструктор обов'язковий для XmlSerializer
71      public WardPatient() {}
72
73      public WardPatient(string id, string name, string ward, string notes,
    string status)
74      {
75          Id = id; Name = name; Ward = ward; Notes = notes; Status = status;
76      }
77  }
```



XmlSerializer — порівняння з JsonSerializer за підходом

19.6. XPath та вибір між JSON і XML

XPath (XML Path Language) — мова запитів до XML-документа. Вона дозволяє точно адресувати будь-який вузол або атрибут через вирази-шляхи: /root/child, //descendant, @attribute, [predicat]. XPath вбудований у XmlDocument через SelectSingleNode/SelectNodes, а для XDocument доступний через розширення System.Xml.XPath.

Після опанування всього інструментарію розділу — природне запитання: **коли обирати JSON, а коли XML?** Відповідь залежить від

домену, вимог до схеми, набору інструментів і команди. Розглянемо обидва аспекти.



XPath — синтаксис виразів та осі

Синтаксис XPath

XPath-вираз описує шлях по дереву вузлів. Основні конструкції:

Вираз	**Значення**
/clinic	Кореневий елемент <clinic>
/clinic/patient	Прямі нащадки <patient>
//diagnosis	Всі <diagnosis> на будь-якій глибині
@id	Атрибут id поточного вузла
patient[1]	Перший елемент <patient>
patient[last()]	Останній <patient>
patient[@ward='Терапія']	<patient> з атрибутом ward="Терапія"
*	Будь-який елемент
node()	Будь-який вузол (включно з текстом)
..	Батьківський вузол

Вираз	Значення
.	Поточний вузол

XPath-oci

Осі дозволяють переміщатися по дереву у будь-якому напрямку:

Вісь	Вирізка
child::	Прямі дочірні (за замовчуванням)
descendant::	Всі нащадки (рекурсивно)
parent::	Батьківський вузол
ancestor::	Всі батьки вгору до кореня
following-sibling::	Сусіди після поточного вузла
preceding-sibling::	Сусіди перед поточним вузлом
attribute::	Атрибути (скорочення: @)
self::	Сам вузол

XPath-функції

```
count(nodes)          — кількість вузлів у наборі
normalize-space(str)  — прибрати зайві пробіли
contains(str, sub)    — перевірка входження підрядка
starts-with(str, pre) — перевірка початку рядка
string-length(str)    — довжина рядка
not(expr)             — логічне заперечення
position()            — позиція вузла в батьківському наборі
last()                — кількість вузлів в поточному наборі
```

XPath у XmlDocument

C#

```
using System;
using System.Xml;

string xml = """
<?xml version="1.0" encoding="utf-8"?>
<clinic date="2024-03-15">
    <patient id="PT-1001" ward="Терапія" status="active">
        <name>Петренко Іван Олексійович</name>
        <age>45</age>
```

```
        <diagnosis code="J06.9" severity="mild">ГРВІ</diagnosis>
        <diagnosis code="I10"   severity="moderate">Гіпертонія</diagnosis>
    </patient>
    <patient id="PT-1002" ward="Кардіологія" status="active">
        <name>Бойко Оксана Петрівна</name>
        <age>62</age>
        <diagnosis code="I21.0" severity="critical">Інфаркт міокарда</diagnosis>
    </patient>
    <patient id="PT-1003" ward="Терапія" status="discharged">
        <name>Мороз Василь Іванович</name>
        <age>38</age>
        <diagnosis code="E11.9" severity="moderate">Діабет 2 типу</diagnosis>
    </patient>
</clinic>
""";

XmlDocument doc = new XmlDocument();
doc.LoadXml(xml);

// 1. Точний шлях — перший пацієнт
XmlNode? first = doc.SelectSingleNode("/clinic/patient[1]/name");
Console.WriteLine($"Перший: {first?.InnerText}");

// 2. За значенням атрибуту
XmlNode? byId = doc.SelectSingleNode("/clinic/patient[@id='PT-1002']/name");
Console.WriteLine($"PT-1002: {byId?.InnerText}");

// 3. Фільтр за відділенням
XmlNodeList? therapy = doc.SelectNodes("/clinic/patient[@ward='Терапія']");
Console.WriteLine($"\nТерапія ({therapy?.Count.ToString()}):");
foreach (XmlNode p in therapy!)
    Console.WriteLine($"  {p["name"]?.InnerText} [{p.Attributes?["status"]?.Value}]");

// 4. contains() — часткове співпадіння
XmlNodeList? ivan = doc.SelectNodes("//patient[contains(name, 'Іван')]");
Console.WriteLine($"\nПацієнти з 'Іван' ({ivan?.Count.ToString()}):");
foreach (XmlNode p in ivan!)
    Console.WriteLine($"  {p["name"]?.InnerText}");

// 5. Комбінований OR у предикаті
XmlNodeList? cardioOrCrit = doc.SelectNodes(
    "//diagnosis[@severity='critical' or @severity='moderate']");
Console.WriteLine($"\nCritical/moderate діагнози ({cardioOrCrit?.Count.ToString()}):");
foreach (XmlNode d in cardioOrCrit!)
    Console.WriteLine($"  [{d.Attributes?["code"]?.Value}] {d.InnerText} — {d.Attributes?["severity"]?.Value}");

// 6. not() — активні пацієнти
XmlNodeList? active = doc.SelectNodes("/clinic/patient[not(@status='discharged')]");
Console.WriteLine($"\nАктивні пацієнти: {active?.Count.ToString()}");

// 7. count() і last()
XmlNode? lastPatient = doc.SelectSingleNode("/clinic/patient[last()]");
Console.WriteLine($"Останній пацієнт: {lastPatient?["name"]?.InnerText}");

// Evaluate повертає object: для числових виразів XPath 1.0 це завжди double
```

```
double total =
    (double)doc.CreateNavigator()!.Evaluate("count(/clinic/patient)");
Console.WriteLine($"Всього пацієнтів: {total.ToString()}");
```

XPath у XDocument (System.Xml.XPath)

```
C#
using System;
using System.Linq;
using System.Xml.Linq;
using System.Xml.XPath;

XDocument doc = XDocument.Parse("""
<clinic date="2024-03-15">
    <patient id="PT-1001" ward="Терапія">
        <name>Петренко Іван Олексійович</name>
        <age>45</age>
        <diagnosis code="J06.9">ГРВІ</diagnosis>
    </patient>
    <patient id="PT-1002" ward="Кардіологія">
        <name>Бойко Оксана Петрівна</name>
        <age>62</age>
        <diagnosis code="I21.0">Інфаркт міокарда</diagnosis>
    </patient>
</clinic>
""");

// XPathSelectElement — перший вузол
XElement? pt = doc.XPathSelectElement("/clinic/patient[@ward='Кардіологія']");
Console.WriteLine($"Кардіологія: {pt?.Element("name")?.Value}");

// XPathSelectElements — колекція, повертає IEnumerable<XElement>
var all = doc.XPathSelectElements("//patient");
Console.WriteLine($"Всього пацієнтів: {all.Count().ToString()}");

// XPathEvaluate — арифметичні функції XPath
double count = (double)doc.XPathEvaluate("count(//patient)");
Console.WriteLine($"count(): {count.ToString()}");

// Пошук з contains
var withIvan = doc.XPathSelectElements("//patient[contains(name, 'Іван')]");
foreach (XElement p in withIvan)
    Console.WriteLine($"  {p.Element("name")?.Value}");

// XPathNavigator — позиційна навігація
System.Xml.XPath.XPathNavigator? nav = doc.CreateNavigator();
nav?.MoveToRoot();
Console.WriteLine($"\nNavigator root: {nav?.Name}");
```

XPathNavigator для складної навігації

```
C#
using System;
using System.Xml;
using System.Xml.XPath;

```

```
string xml = """
<clinic>
    <department name="Терапія">
        <patient id="PT-1001"><name>Петренко</name><age>45</age></patient>
        <patient id="PT-1002"><name>Мороз</name><age>38</age></patient>
    </department>
    <department name="Кардіологія">
        <patient id="PT-1003"><name>Бойко</name><age>62</age></patient>
    </department>
</clinic>
""";

XmlDocument doc = new XmlDocument();
doc.LoadXml(xml);

XPathNavigator nav = doc.CreateNavigator()!;

// Скомпілювати вираз для повторного використання
XPathExpression expr = nav.Compile("//patient[age > 40]");

XPathNodeIterator it = nav.Select(expr);
Console.WriteLine($"Пацієнти старші 40 ({it.Count.ToString()}):");
while (it.MoveNext())
{
    XPathNavigator cur = it.Current!;
    string id   = cur.GetAttribute("id", "");
    string name = cur.SelectSingleNode("name")?.Value ?? "";
    string age  = cur.SelectSingleNode("age")?.Value  ?? "";
    Console.WriteLine($"  [{id}] {name}, {age} р.");
}

// Навігація через батьківський вузол
XPathNavigator? firstPatient = nav.SelectSingleNode("//patient[@id='PT-
1001']");
firstPatient?.MoveToParent();
Console.WriteLine($"\nВідділення PT-1001: {firstPatient?.GetAttribute("name",
"")}");

// Агрегація через XPath 1.0
double avgAge = (double)nav.Evaluate("sum(//age) div count(//age)");
Console.WriteLine($"Середній вік: {avgAge.ToString("F1")} р.");
```

XPathExpression дозволяє скомпілювати XPath один раз і виконувати багаторазово без повторного парсингу — важливо при роботі з великими документами або частими запитами.

JSON vs XML — порівняльний аналіз

```
C#
using System;

// Один і той же об'єкт — в JSON та XML:

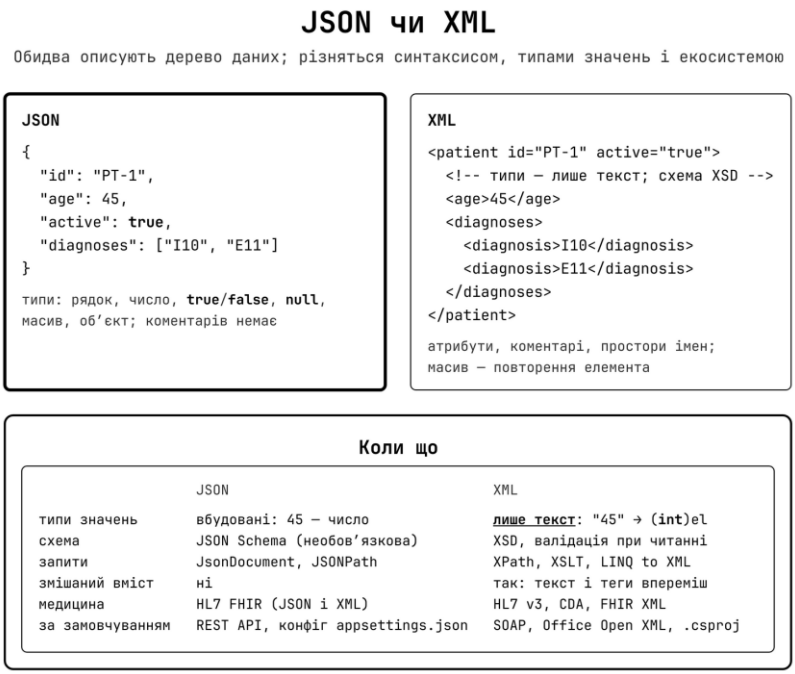
string json = """
{
  "id": "PT-1001",
```

```
  "name": "Петренко І.О.",
  "age": 45,
  "ward": "Терапія",
  "diagnoses": [
    { "code": "J06.9", "text": "ГРВІ" },
    { "code": "I10",   "text": "Гіпертонія" }
  ]
}
""";

string xml = """
<?xml version="1.0"?>
<patient id="PT-1001">
  <name>Петренко І.О.</name>
  <age>45</age>
  <ward>Терапія</ward>
  <diagnoses>
    <diagnosis code="J06.9">ГРВІ</diagnosis>
    <diagnosis code="I10">Гіпертонія</diagnosis>
  </diagnoses>
</patient>
""";

Console.WriteLine($"JSON: {json.Length.ToString()} символів");
Console.WriteLine($"XML:  {xml.Length.ToString()} символів");
Console.WriteLine($"XML більший на: {((xml.Length - json.Length) * 100.0 /
json.Length).ToString("F0")}%");
Console.WriteLine("");

// Порівняння за критеріями
Console.WriteLine("Критерій              JSON         XML");
Console.WriteLine(new string('-', 55));
Console.WriteLine($"{"Розмір",-22}{"менший",-13}{"більший"}");
Console.WriteLine($"{"Читаність",-22}{"висока",-13}{"висока"}");
Console.WriteLine($"{"Атрибути",-22}{"немає",-13}{"є"}");
Console.WriteLine($"{"Коментарі",-22}{"немає",-13}{"є"}");
Console.WriteLine($"{"Схема",-22}{"JSON Schema",-13}{"XSD/DTD"}");
Console.WriteLine($"{"Простори імен",-22}{"немає",-13}{"є (xmlns)"}");
Console.WriteLine($"{"LINQ у .NET",-22}{"JsonDocument",-
13}{"XDocument+LINQ"}");
Console.WriteLine($"{"Серіалізатор .NET",-22}{"JsonSerializer",-
13}{"XmlSerializer"}");
Console.WriteLine($"{"Потокове читання",-22}{"Utf8JsonReader",-
13}{"XmlReader"}");
Console.WriteLine($"{"REST API",-22}{"стандарт",-13}{"застарілий"}");
Console.WriteLine($"{"HL7 / FHIR",-22}{"FHIR R4 JSON",-13}{"HL7 v2/v3 XML"}");
Console.WriteLine($"{"SOAP веб-сервіси",-22}{"не підходить",-13}{"стандарт"}");
Console.WriteLine($"{"Office (.docx)",-22}{"ні",-13}{"так"}");
Console.WriteLine($"{"Config .NET",-22}{"appsettings.json",-
13}{"app.config"}");
```



JSON vs XML — коли що обирати

Правила вибору формату

Обирайте JSON, якщо:

— Ви будуєте REST API або SPA-клієнт — JSON є індустріальним стандартом HTTP

— Дані — масиви, вкладені об'єкти без метаданих-атрибутів

— Потрібна максимальна стислість і простота парсингу у браузері

— Команда використовує JavaScript/TypeScript або мобільних клієнтів

Обирайте XML, якщо:

— Обмін з медичними системами: HL7 v2, HL7 v3, CDA, FHIR R2/R3 — там XML

— Потрібні атрибути як першокласний синтаксис (не поле-обгортка)

— SOAP-сервіси або WCF (корпоративний legacy)
— Документи з коментарями та просторами імен (XML схема обов'язкова)
— Office Open XML (.docx, .xlsx, .pptx) — це ZIP з XML всередині
— Конфіг .NET Framework: Web.config, App.config

Обидва прийнятні, коли:

— FHIR R4 підтримує і JSON, і XML — обирайте за можливостями клієнта
— Для зберігання у БД: JSON підтримується у PostgreSQL, MySQL, SQL Server нативно; XML — через XML-тип або nvarchar

Підсумок розділу 19

```C#
using System;

Console.WriteLine("=== Розділ 19: Серіалізація та десеріалізація ===\n");

Console.WriteLine("19.1 System.Text.Json advanced:");
Console.WriteLine("  JsonDocument / JsonElement     — DOM без прив'язки до класу");
Console.WriteLine("  Utf8JsonWriter                 — найшвидший запис JSON");
Console.WriteLine("  JsonConverter<T>               — кастомна логіка серіалізації");
Console.WriteLine("  [JsonPolymorphic]              — поліморфний JSON");
Console.WriteLine("  SerializeAsync / DeserializeAsync — асинхронний I/O\n");

Console.WriteLine("19.2 XML + XmlDocument:");
Console.WriteLine("  XmlDocument.LoadXml / Load     — DOM у пам'яті");
Console.WriteLine("  SelectSingleNode / SelectNodes  — XPath-запити");
Console.WriteLine("  CreateElement / AppendChild     — побудова XML програмно\n");

Console.WriteLine("19.3 XDocument + LINQ to XML:");
Console.WriteLine("  new XElement(new XElement(...))  — декларативна побудова");
Console.WriteLine("  Elements() / Descendants()       — IEnumerable<XElement>");
Console.WriteLine("  Where / Select / GroupBy / OrderBy — нативний LINQ\n");

Console.WriteLine("19.4 XmlReader + XmlWriter:");
Console.WriteLine("  XmlReader.Create(stream)         — потокове читання O(1) RAM");
Console.WriteLine("  NodeType: Element/Text/EndElement — автомат станів");
Console.WriteLine("  XmlWriter.Create(stream, settings)— послідовний запис\n");

Console.WriteLine("19.5 XmlSerializer:");
Console.WriteLine("  [XmlRoot] [XmlElement] [XmlAttribute] [XmlIgnore]");
Console.WriteLine("  [XmlArray] [XmlArrayItem]        — колекції");
Console.WriteLine("  [XmlInclude]                     — поліморфізм\n");
```

```
31
32  Console.WriteLine("19.6 XPath + JSON vs XML:");
33  Console.WriteLine("  /path, //any, @attr, [predicate] — XPath-адресація");
34  Console.WriteLine("  XPathNavigator, XPathExpression  — скомпільовані запити");
35  Console.WriteLine("  JSON: REST API, браузер, стислість");
36  Console.WriteLine("  XML: HL7, SOAP, Office, схема, простори імен");
```

Питання для самоконтролю — Розділ 19. Серіалізація JSON та XML

19.1. System.Text.Json — розширені можливості

1. Навіщо JsonDocument використовує ArrayPool і чому обов'язковий блок using? Що відбудеться, якщо не викликати Dispose() на JsonDocument?
2. Розгляньте код:

```
C#
1  using var doc = JsonDocument.Parse(jsonString);
2  JsonElement root = doc.RootElement;
3  if (root.TryGetProperty("patient", out JsonElement patient))
4  {
5      Console.WriteLine(patient.GetProperty("name").GetString());
6      Console.WriteLine(patient.GetProperty("age").GetInt32());
7  }
```

Чим TryGetProperty відрізняється від GetProperty? Коли і яке з них використовувати?

3. Перелічіть значення JsonValueKind. Напишіть метод PrintJsonStructure(JsonElement element, int depth), що рекурсивно обходить JSON та виводить структуру з відступами, розрізняючи об'єкти, масиви та примітивні значення.
4. Реалізуйте за допомогою Utf8JsonWriter JSON-документ такого вигляду:

```
JSON
1  { "report": { "date": "2024-01-15", "items": [1, 2, 3], "valid": true } }
```

Назвіть методи, що ви використаєте для кожного елементу.

5. Для чого потрібен JsonConverter<T>? Реалізуйте конвертер DateOnlyJsonConverter, що серіалізує DateOnly у рядок формату "dd.MM.yyyy" і десеріалізує назад. Як зареєструвати його через атрибут і через глобальні налаштування?

6. Поясніть [JsonPolymorphic] і [JsonDerivedType]. Яку проблему вони вирішують? Напишіть ієрархію Animal (базовий) → Dog, Cat з дискримінатором та продемонструйте, як JSON виглядає після серіалізації списку.
7. Що таке JSON Source Generators? Яку проблему з рефлексією вони вирішують і в яких середовищах особливо важливі (AOT, Blazor WASM)? Напишіть мінімальний приклад з [JsonSerializable] та JsonSerializerContext.
8. Назвіть щонайменше 3 проблеми безпеки при десеріалізації JSON від зовнішніх джерел. Чому рекомендують мати окремі DTO для вводу та виводу? Що таке MaxDepth і чому важливо його обмежувати?

19.2. XML та XmlDocument

9. Поясніть структуру XML-документа: що таке декларація, елемент, атрибут, namespace? Напишіть XML-фрагмент для опису лікаря з атрибутом id та вкладеними елементами.
10. Розгляньте код:

```C#
var doc = new XmlDocument();
doc.LoadXml(xml);
XmlNode root = doc.DocumentElement;
foreach (XmlNode child in root.ChildNodes)
{
    Console.WriteLine(child.InnerText);
}
```

Чим InnerText відрізняється від InnerXml? Наведіть XML, де ці два значення різні.

11. Напишіть XPath-вираз і відповідний C# код для XmlDocument.SelectNodes(), що знаходить всі елементи <appointment> де атрибут status дорівняє "confirmed" і дочірній елемент <doctor> має значення "Smith".
12. Порівняйте SelectSingleNode і SelectNodes. Що повертає SelectSingleNode, якщо нічого не знайдено? Що станеться, якщо передати хибний XPath? Напишіть безпечний виклик з перевіркою результату.
13. Програмно побудуйте за допомогою XmlDocument наступну структуру:

```xml
<patients>
  <patient id="1">
    <name>John Doe</name>
    <age>35</age>
  </patient>
</patients>
```

Назвіть методи CreateElement, CreateAttribute, SetAttribute, AppendChild у правильному порядку.

14. Як зберегти XmlDocument у файл? Як зберегти у рядок? Напишіть обидва варіанти та поясніть, коли кожен з них доцільний.
15. Що таке XmlTextWriter і коли він використовується разом з XmlDocument.WriteTo()? Покажіть, як налаштувати форматований вивід з відступами.

19.3. XDocument та LINQ to XML

16. Порівняйте стиль створення XML-документа в XmlDocument проти XDocument. У чому перевага декларативного синтаксису XDocument? Відтворіть той самий XML-фрагмент обома способами.
17. Розгляньте код:

```csharp
var doc = XDocument.Parse(xml);
var names = doc.Root
    .Elements("patient")
    .Where(p => (int)p.Element("age") > 30)
    .Select(p => (string)p.Element("name"))
    .OrderBy(n => n);
```

Поясніть кожен крок LINQ-ланцюжка. Що станеться, якщо елемент <age> відсутній?

18. Чим Element("name") відрізняється від Elements("name") та Descendants("name")? Наведіть XML, де ці три методи дадуть різні результати.
19. Напишіть метод, що завантажує XML-файл пацієнтів, знаходить пацієнта за ID, оновлює його вік та зберігає файл назад. Використайте SetValue або SetAttributeValue.
20. Реалізуйте LINQ запит по XDocument, що групує лікарів за спеціальністю і для кожної групи виводить кількість та середній стаж. Використайте GroupBy і Select.

21. Напишіть метод List<Patient> ParsePatients(XDocument doc), що витягує всі записи пацієнтів з XML і повертає строго типізований список C# об'єктів. Обробіть відсутні необов'язкові елементи.
22. Як XDocument порівнюється з XmlDocument за продуктивністю? Коли краще використовувати XmlDocument, а коли XDocument? Що таке LINQ to XML і чим воно відрізняється від XPath у XmlDocument?

19.4. XmlReader та XmlWriter

23. Що означає "forward-only" читання? У яких сценаріях XmlReader є єдиним правильним вибором порівняно з XmlDocument або XDocument?
24. Розгляньте код:

```
C#
 1  using var reader = XmlReader.Create("data.xml");
 2  while (reader.Read())
 3  {
 4      if (reader.NodeType == XmlNodeType.Element && reader.Name == "patient")
 5      {
 6          string id = reader.GetAttribute("id");
 7          reader.ReadToDescendant("name");
 8          string name = reader.ReadElementContentAsString();
 9      }
10  }
```

Поясніть кожен крок. Що відбудеться, якщо елемент <name> відсутній всередині <patient>?

25. Назвіть 5 значень XmlNodeType, які зустрічаються при читанні реального XML. Напишіть умову, що ігнорує пробіли та коментарі за допомогою XmlReaderSettings.
26. Чим ReadToFollowing, ReadToDescendant і ReadToNextSibling відрізняються один від одного? Намалюйте (текстово) XML-дерево та покажіть, до якого вузла переміститься кожен метод від поточної позиції.
27. Реалізуйте метод XmlToJsonStream(Stream xmlInput, Stream jsonOutput), що зчитує XML пацієнтів за допомогою XmlReader і записує їх у JSON за допомогою Utf8JsonWriter, не завантажуючи весь документ у пам'ять.
28. Напишіть за допомогою XmlWriter документ з правильною декларацією, коментарем, кількома елементами та атрибутами.

Налаштуйте XmlWriterSettings для форматованого виводу з Indent = true та кодуванням UTF-8.

29. Опишіть паттерн "XmlReader + XmlWriter трансформація". Напишіть метод FilterLargeXml(string inputPath, string outputPath, Predicate<string> keepPatient), що читає великий XML пацієнтів і записує тільки тих, що задовольняють умову.

19.5. XmlSerializer

30. Які вимоги до класу для роботи з XmlSerializer? Що станеться, якщо клас не має публічного конструктора без параметрів? Чому поля та властивості мають бути публічними?
31. Розгляньте код:

```
C#
 1  [XmlRoot("medicalRecord")]
 2  public class Record
 3  {
 4      [XmlAttribute("id")]
 5      public int Id { get; set; }
 6      [XmlElement("patientName")]
 7      public string Name { get; set; }
 8      [XmlIgnore]
 9      public string InternalNotes { get; set; }
10  }
```

Як виглядатиме XML після серіалізації? Як виглядатиме XML без жодних атрибутів?

32. Поясніть різницю між [XmlElement] і [XmlAttribute]. Яка між ними семантична різниця з погляду XML-моделювання? Коли краще використовувати атрибут, а коли елемент?
33. Як серіалізувати список за допомогою [XmlArray] і [XmlArrayItem]? Напишіть клас Hospital зі списком Doctors та покажіть очікуваний XML. Як виглядатиме XML без цих атрибутів?
34. Що таке [XmlInclude]? Напишіть ієрархію Payment → CashPayment, CardPayment з можливістю серіалізації через XmlSerializer. Чим це відрізняється від [JsonPolymorphic]?
35. Як позбутися namespace-ів xmlns:xsi та xmlns:xsd у серіалізованому XML? Напишіть код з XmlSerializerNamespaces, що генерує чистий XML без зайвих просторів імен.

36. Порівняйте XmlSerializer з ручним підходом через XmlDocument. Коли варто обрати XmlSerializer, а коли ручне керування? Які обмеження є у XmlSerializer?

19.6. XPath та порівняння JSON і XML

37. Поясніть XPath-вирази: /root/patient, //patient, patient[@status='active'], patient[1], patient[last()]. Чим / відрізняється від //?
38. Напишіть XPath для таких задач:

- Знайти всі <appointment> елементи на будь-якому рівні вкладеності - Знайти <patient>, у якого <age> більше 30 - Знайти перший <doctor> з атрибутом specialty, що містить слово "cardio" - Підрахувати кількість <appointment> зі статусом "pending"

39. Які вісі XPath (axes) ви знаєте? Поясніть різницю між child::, descendant::, following-sibling:: і ancestor::. Наведіть приклад XPath з явною назвою вісі.
40. Порівняйте використання XPath у XmlDocument.SelectNodes() і в XDocument.XPathSelectElements(). Що треба підключити для використання XPath у XDocument? Напишіть однаковий запит обома способами.
41. Що таке XPathNavigator і XPathExpression? Яку перевагу дає компіляція XPath-виразу через XPathExpression.Compile()? Коли це суттєво впливає на продуктивність?
42. Заповніть таблицю порівняння JSON і XML за ознаками: розмір, читабельність, підтримка атрибутів, namespace-и, стандартна схема, типовий API (REST/SOAP), інструменти запитів. Для яких галузей/стандартів XML залишається обов'язковим?
43. Дано задачу: потрібно зберігати конфігурацію для веб-API. Аргументуйте вибір JSON або XML, враховуючи: читабельність, розмір, підтримку в .NET, тип клієнтів (браузер/мобільний/корпоративний).
44. Напишіть метод, що приймає XML-рядок медичного запису і за допомогою XPathNavigator та скомпільованого XPathExpression виконує багаторазовий пошук різних полів. Порівняйте продуктивність з некомпільованим підходом.

Розділ 20. SOLID Принципи

20.1. Вступ до SOLID. Проблеми без архітектурних принципів

SOLID — це акронім із п'яти принципів об'єктно-орієнтованого проектування, зібраних і сформульованих Робертом Мартіном (Robert C. Martin, відомий у спільноті як «Uncle Bob») у статтях другої половини 1990-х. Не всі вони придумані Мартіном: принцип відкритості/закритості сформулював Бертран Мейєр (1988), принцип підстановки — Барбара Лісков (1987); заслуга Мартіна в тому, що він переосмислив їх у термінах залежностей між модулями, додав SRP, ISP і DIP та звів у єдину систему, і систематизованих у книзі «Agile Software Development: Principles, Patterns, and Practices» (2002). Сам акронім склав Майкл Фезерс (Michael Feathers) у 2004 році. За понад двадцять років ці принципи стали індустріальним стандартом — вони описують, якою має бути архітектура коду, що призначений жити довго і змінюватися без катастрофічних наслідків.

Важливо розуміти з самого початку: SOLID — це **не суворі правила**, яким треба сліпо слідувати, і не алгоритм із єдиною правильною відповіддю. Це набір **евристик** — практичних настанов, що виникли з колосального досвіду супроводу великих систем. Кожен принцип відповідає на конкретне питання: «Яка властивість поганого коду робить його болючим для зміни, і як від неї позбутися?»

Що таке «хороший» і «поганий» код

Студенти-початківці часто оцінюють код за критерієм «працює / не працює». Але в реальних проектах набагато важливіший інший критерій: **як легко цей код змінити**. Програмне забезпечення за означенням є «м'яким» (software — soft + ware), тобто таким, що має змінюватися. Вимоги змінюються, бізнес змінюється, закони змінюються, технології змінюються. Код, який неможливо змінити без болю, — це код, який не виконує свого призначення.

Роберт Мартін описував симптоми «поганого» дизайну чотирма поняттями:

Жорсткість (Rigidity) — код важко змінити, бо будь-яка зміна тягне за собою каскад інших змін. Виправив один клас — зламалось

п'ять. Щоб додати нову функцію, доводиться переписувати існуючий код.

Крихкість (Fragility) — зміна в одному місці несподівано ламає зовсім не пов'язані частини системи. Відремонтував звіти — перестали надсилатися email. Ніхто не може пояснити, чому. З часом команда боїться чіпати код взагалі.

Нерухомість (Immobility) — неможливо перевикористати компонент в іншому проекті, бо він занадто щільно сплетений з рештою системи. Хочеш взяти «тільки логіку бронювання» — разом з нею тягнеться база даних, email-відправник і PDF-генератор.

В'язкість (Viscosity) — правильне рішення важче реалізувати, ніж неправильне. Щоб зробити «як треба», потрібно переписати багато; щоб зробити «як-небудь» — достатньо одного рядка. Розробники обирають швидкий шлях, і система деградує.

Ці чотири симптоми — не окремі проблеми, а прояви одного й того ж явища: **надмірної зв'язаності** (tight coupling) і **недостатньої зв'язності** (low cohesion). SOLID — це систематизований спосіб досягти протилежного: слабкої зв'язаності і сильної зв'язності.

Технічний борг: ціна «швидких» рішень

Кожне рішення, що спрощує код зараз, але ускладнює його зміну в майбутньому, — це **технічний борг**. Як фінансовий борг, він накопичує «відсотки»: чим довше він існує, тим більше зусиль потрібно для кожної наступної зміни. Системи, де технічний борг накопичувався роками, перетворюються на «вавилонські вежі» — конструкції, в яких ніхто не розуміє, що відбувається, і всі бояться щось чіпати.

Характерний приклад: розробник поспішав, додав логіку відправки email прямо у клас збереження пацієнта — «тимчасово». Через рік клас збереження пацієнта не можна протестувати без SMTP-сервера. Через два роки ніхто вже не пам'ятає, чому email там взагалі є.

God Class: коли один клас знає все

Найпоширеніший прояв жорсткості й крихкості — **«God Class»** (клас-Бог): клас, що бере на себе надто багато відповідальностей. Він «знає» все про всіх підсистемах і «робить» все сам. Поява такого класу — майже завжди наслідок того, що функціонал «по-швидкому» додавали в одне місце замість того, щоб проектувати архітектуру.

Ось як виглядає типова еволюція клінічної системи без принципів:

C#
```
using System;
using System.Collections.Generic;

// ─── Використання — виглядає зручно, але... ──────────────────────
var clinic = new ClinicManager();
clinic.AddDoctor("Петренко І.О.");
clinic.AddPatient("Бойко Олена");
clinic.BookAppointment("Бойко Олена", "Петренко І.О.",
DateTime.Now.AddHours(2));
Console.WriteLine(clinic.GenerateReport());
Console.WriteLine($"Рахунок: {clinic.CalculateBilling("Бойко
Олена").ToString()} грн");
clinic.SaveToFile("clinic.dat");

// ─── God Class — еволюція без принципів ──────────────────────────
// Спочатку клас здавався невеличким. З кожним тижнем до нього
// «дочіплювали» нову відповідальність, бо «тут вже є доступ до даних».

class ClinicManager  // 800+ рядків у реальному проекті
{
    // --- Пацієнти ---
    private List<string> _patients = new();
    public void AddPatient(string name)    { _patients.Add(name);
Console.WriteLine($"Додано: {name}"); }
    public void RemovePatient(string name) { _patients.Remove(name);
Console.WriteLine($"Видалено: {name}"); }
    public List<string> GetPatients()      { return _patients; }

    // --- Лікарі ---
    private List<string> _doctors = new();
    public void AddDoctor(string name)    { _doctors.Add(name); }
    public List<string> GetDoctors()      { return _doctors; }

    // --- Бронювання ---
    private List<string> _appointments = new();
    public void BookAppointment(string patient, string doctor, DateTime time)
    {
        string entry = $"{patient} → {doctor} @ {time:HH:mm}";
        _appointments.Add(entry);
        Console.WriteLine($"[Booking] Заброньовано: {entry}");
        // А ще відразу надсилаємо email — «для зручності»
        SendEmailNotification(patient, $"Ваш запис: {entry}");
    }

    // --- Email — відповідальність #3 у класі пацієнтів?! ---
    public void SendEmailNotification(string to, string message)
    {
        // У реальності — SMTP-клієнт прямо тут
        Console.WriteLine($"[Email] → {to}: {message}");
    }

    // --- Збереження у файл --- відповідальність #4 ---
    public void SaveToFile(string path)
    {
        Console.WriteLine($"[File] Зберігаю {_patients.Count} пацієнтів у
{path}");
    }

    // --- Генерація звіту --- відповідальність #5 ---
```

```
55      public string GenerateReport()
56      {
57          return $"Звіт: пацієнтів={_patients.Count}, лікарів={_doctors.Count},
    записів={_appointments.Count}";
58      }
59
60      // --- Рахунки --- відповідальність #6 ---
61      public decimal CalculateBilling(string patient)
62      {
63          return _appointments.FindAll(a => a.StartsWith(patient)).Count * 500m;
64      }
65  }
```

Цей клас **технічно працює**. Але подивимося, що відбувається, коли вимоги змінюються:

— Замовник хоче надсилати SMS замість email → треба змінити ClinicManager

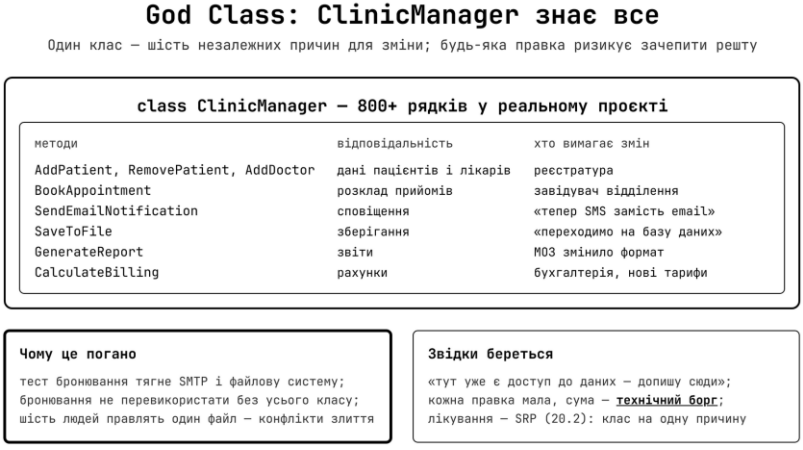
— Потрібно зберігати у базу даних замість файлу → треба змінити ClinicManager

— Нова формула рахунків → треба змінити ClinicManager

— Хочемо написати тест на бронювання → неможливо без SMTP-сервера і файлової системи

— Хочемо перевикористати логіку бронювання в іншому модулі → разом з нею тягнеться весь ClinicManager

Клас має **шість різних причин для зміни** — і будь-яка зміна несе ризик зламати всі інші шість частин.



God Class — ClinicManager із шістьма відповідальностями

Каскад змін: як один рядок ламає систему

Розглянемо конкретну ситуацію: потрібно змінити формат рядка у звіті. Здавалося б, незначна правка.

```C#
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;

var sys = new TightClinicSystem();
sys.Book("Мороз Василь", "Петренко І.О.");
sys.Book("Бойко Олена", "Коваленко М.А.");
sys.ExportReport();
Console.WriteLine($"Лікар Петренко: {sys.CountByDoctor("Петренко І.О.").ToString()} записів");
Console.WriteLine($"Мороз активний: {sys.HasActiveAppointment("Мороз Василь").ToString()}");

// Тепер змінимо формат рядка на "DD.MM.YYYY;PATIENT;DOCTOR;STATUS"
// → треба виправити Book, CountByDoctor, ExportReport, HasActiveAppointment
// → чотири місця, і це лише в одному класі!
Console.WriteLine("\n[Проблема] Зміна формату = 4 місця для редагування у цьому класі");
Console.WriteLine("           У реальному проекті таких місць можуть бути десятки");

// Спрощена модель щільно зв'язаної системи
// де кожен компонент знає про внутрішній формат інших

class TightClinicSystem
{
    private List<string> _log = new(); // формат: "YYYY-MM-DD|PATIENT|DOCTOR|STATUS"

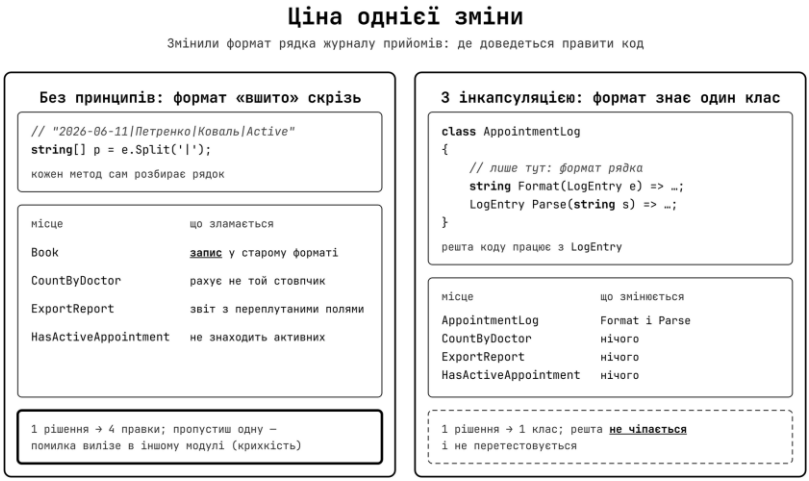
    public void Book(string patient, string doctor)
    {
        // Ключове рішення: формат рядка "зашитий" усюди
        string entry = $"{DateTime.Today:yyyy-MM-dd}|{patient}|{doctor}|ACTIVE";
        _log.Add(entry);
        Console.WriteLine($"[Book] {entry}");
    }

    // Цей метод знає про формат "yyyy-MM-dd|..." — зв'язаність!
    public int CountByDoctor(string doctor)
    {
        int count = 0;
        foreach (var e in _log)
        {
            string[] parts = e.Split('|'); // очікує саме цей формат
            if (parts.Length >= 3 && parts[2] == doctor) count++;
        }
        return count;
    }

    // Цей метод теж знає про формат — ще одна зв'язаність
    public void ExportReport()
```

```
47      {
48          Console.WriteLine("[Report] Активні записи:");
49          foreach (var e in _log)
50          {
51              string[] parts = e.Split('|'); // знову той самий формат
52              if (parts.Length >= 4 && parts[3] == "ACTIVE")
53                  Console.WriteLine($"  {parts[0]}: {parts[1]} → {parts[2]}");
54          }
55      }
56
57      // Ще один компонент із залежністю від формату
58      public bool HasActiveAppointment(string patient)
59      {
60          foreach (var e in _log)
61          {
62              string[] parts = e.Split('|'); // четверта залежність від формату
63              if (parts.Length >= 4 && parts[1] == patient && parts[3] ==
    "ACTIVE")
64                  return true;
65          }
66          return false;
67      }
68  }
```

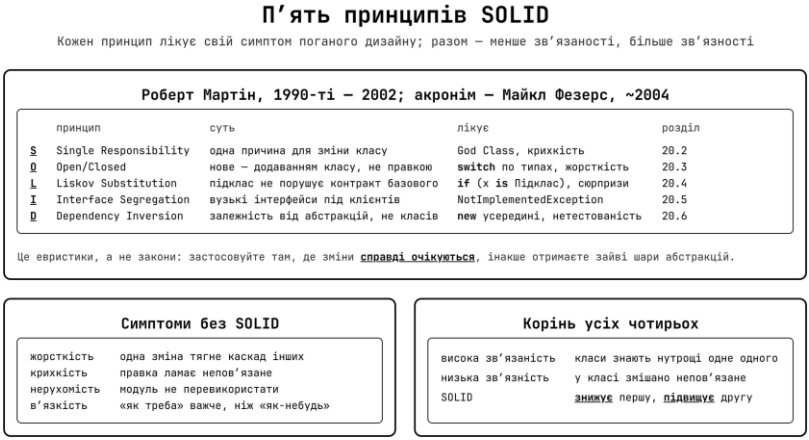
Це і є **жорсткість**: зміна одного рішення (формату рядка) потребує пошуку і виправлення всіх місць, де це рішення «вшите». У великих кодових базах розробники часто не знають про всі такі місця — звідси й **крихкість**: щось ламається там, де не очікували.



Ціна зміни — без SOLID каскад, з SOLID ізоляція

П'ять принципів: короткий огляд

SOLID — це п'ять принципів, кожен із яких «лікує» один конкретний симптом поганого дизайну:



SOLID — огляд п'яти принципів

S — Single Responsibility Principle (SRP) *Принцип єдиної відповідальності*: клас має мати лише одну причину для зміни. «Причина для зміни» — це конкретна роль у системі. Клас, що одночасно керує даними, надсилає повідомлення і генерує звіти, має три причини для зміни. SRP вимагає розбити його на три окремі класи.

O — Open/Closed Principle (OCP) *Принцип відкритості/закритості*: програмні сутності мають бути відкриті для розширення, але закриті для модифікації. Додавання нової поведінки не повинно вимагати зміни існуючого, вже перевіреного коду. Нова поведінка реалізується через розширення (нові класи, нові реалізації інтерфейсів), а не через правку старого.

L — Liskov Substitution Principle (LSP) *Принцип підстановки Ліскова* (Barbara Liskov, 1987): об'єкт підкласу має бути взаємозамінним з об'єктом базового класу без порушення коректності програми. Якщо замінити базовий клас підкласом — жодна частина системи не повинна «здивуватися». Це означає, що ієрархія успадкування має відображати справжнє відношення «є» (is-a), а не зручність повторного використання коду.

I — Interface Segregation Principle (ISP) *Принцип розподілу інтерфейсів*: клієнти не повинні залежати від методів, яких вони не використовують. Великий інтерфейс, що об'єднує непов'язані операції, змушує кожного реалізатора брати на себе зайві зобов'язання. ISP вимагає ділити великі інтерфейси на вузькоспеціалізовані.

D — Dependency Inversion Principle (DIP) *Принцип інверсії залежностей*: модулі верхнього рівня не повинні залежати від модулів нижнього рівня — обидва мають залежати від абстракцій. Абстракції не повинні залежати від деталей — деталі мають залежати від абстракцій. На практиці це означає: бізнес-логіка не знає, що конкретно зберігає дані (файл? БД? пам'ять?) — вона знає лише інтерфейс сховища.

Чому SOLID вивчають саме після ООП

Принципи SOLID мають сенс лише тоді, коли вже відомі: класи, об'єкти, успадкування, поліморфізм, інтерфейси, узагальнення. Саме тому вони стоять наприкінці курсу — це не введення у нові синтаксичні конструкції, а **синтез** усього вивченого в єдиний спосіб мислення про архітектуру.

Кожен принцип вирішує проблему, яку студент вже міг зустріти під час виконання попередніх лаб:

- SRP вирішує проблему «God Class», який ріс з кожною лабою
- OCP вирішує проблему «треба додати новий тип прийому, але не хочеться чіпати AppointmentManager»
- LSP вирішує проблему «підклас поводиться не як базовий клас»
- ISP вирішує проблему «мусив реалізувати методи, що не потрібні»
- DIP вирішує проблему «неможливо протестувати PatientManager без бази даних»

Наступні п'ять секцій розбирають кожен принцип детально — з прикладами порушення, поясненням суті і рефакторингом на конкретному коді клінічної системи.

20.2. SRP — Принцип єдиної відповідальності

Single Responsibility Principle (SRP) — перший і найфундаментальніший принцип SOLID. Його формулювання звучить коротко: «Клас повинен мати лише одну причину для зміни». Ця одна

фраза є одночасно і простою для запам'ятовування, і досить тонкою для правильного розуміння.

Ключове слово тут — «причина». Не «одна функція», не «одна задача» у звичайному розумінні. Роберт Мартін пояснював «причину для зміни» як **актора** — конкретну роль або підрозділ організації, чиї вимоги диктують зміни в коді. Клас порушує SRP, якщо його може «змусити змінитися» більш ніж один актор.

У контексті медичної системи: якщо клас AppointmentManager змінюється і коли відділ бронювання переглядає правила запису, і коли IT-відділ переходить на нову базу даних, і коли маркетинг хоче додати SMS-сповіщення — він обслуговує трьох акторів, а значить має три причини для зміни. Це порушення SRP.

Чому «одна відповідальність» — це не «одна функція»

Поширена помилка — інтерпретувати SRP буквально: «один клас = один метод». Це хибне прочитання. Клас може мати десятки методів і при цьому повністю дотримуватися SRP, якщо всі ці методи служать одній меті, одному актору, одній «осі змін».

Прикладом правильного і повного класу з однією відповідальністю є клас Patient з усіма своїми властивостями та методами перевірки: вони всі разом відповідають за коректне представлення і поведінку пацієнта в системі. Ніхто окрім «доменної моделі» не є причиною їх зміни.

Навпаки, клас із двома методами — SaveToDatabase() і SendEmail() — порушує SRP, навіть якщо у ньому лише ці два методи: зміна бази даних і зміна поштового шлюзу — це дві різні причини.

Приклад порушення SRP: клас із трьома акторами

Розглянемо клас AppointmentManager, що традиційно «виростає» в системах без архітектурних принципів:

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3
 4  var manager = new AppointmentManager_Bad();
 5  manager.Book("Мороз Василь", "Петренко І.О.", DateTime.Today.AddHours(10));
 6  manager.Book("Бойко Олена",  "Петренко І.О.", DateTime.Today.AddHours(10)); //
    зайнято
 7  manager.Cancel("Мороз Василь", "Петренко І.О.");
 8
 9  // ── ПОРУШЕННЯ SRP ──────────────────────────────────────────────
10  // Три відповідальності: бізнес-логіка + збереження + сповіщення
11
12  class AppointmentManager_Bad
```

```
{
    private List<(string Patient, string Doctor, DateTime Time, string Status)> _data = new();

    // ── Відповідальність 1: Бізнес-логіка бронювання ──────────────
    // Актор: відділ прийому — змінює правила запису

    public bool Book(string patient, string doctor, DateTime time)
    {
        // Перевірка: лікар не зайнятий у цей час
        foreach (var a in _data)
            if (a.Doctor == doctor && a.Time == time && a.Status == "Active")
            {
                Console.WriteLine($"[Book] ВІДМОВА: {doctor} зайнятий о {time:HH:mm}");
                return false;
            }

        _data.Add((patient, doctor, time, "Active"));
        Console.WriteLine($"[Book] Заброньовано: {patient} → {doctor} о {time:HH:mm}");

        // Тут же — зберігаємо і сповіщаємо. Все в одному методі.
        SaveToDatabase(patient, doctor, time);
        SendEmailConfirmation(patient, doctor, time);
        return true;
    }

    public void Cancel(string patient, string doctor)
    {
        for (int i = 0; i < _data.Count; i++)
        {
            var a = _data[i];
            if (a.Patient == patient && a.Doctor == doctor && a.Status == "Active")
            {
                _data[i] = (a.Patient, a.Doctor, a.Time, "Cancelled");
                Console.WriteLine($"[Cancel] Скасовано: {patient} у {doctor}");
                SendCancellationAlert(patient);
                return;
            }
        }
    }

    // ── Відповідальність 2: Збереження даних ──────────────────────
    // Актор: IT-відділ — змінює коли переходять на нову БД

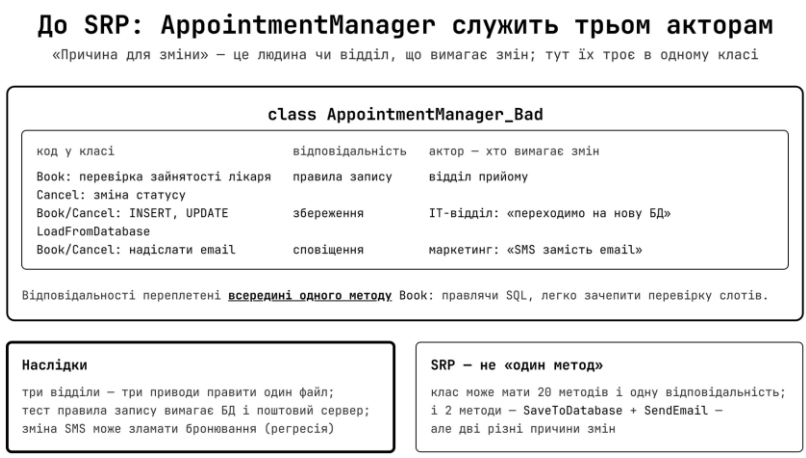
    private void SaveToDatabase(string patient, string doctor, DateTime time)
    {
        // Реально: SQL INSERT або ORM-виклик
        Console.WriteLine($"[DB] INSERT: {patient}, {doctor}, {time:yyyy-MM-dd HH:mm}");
    }

    public void LoadFromDatabase()
    {
        // Реально: SELECT з таблиці appointments
        Console.WriteLine("[DB] SELECT * FROM appointments");
    }

```

```
68      // ── Відповідальність 3: Сповіщення ──────────────────────────
69      // Актор: відділ маркетингу — змінює коли хочуть SMS замість email
70
71      private void SendEmailConfirmation(string patient, string doctor, DateTime
    time)
72      {
73          Console.WriteLine($"[Email] → {patient}: Запис до {doctor} о
    {time:HH:mm} підтверджено");
74      }
75
76      private void SendCancellationAlert(string patient)
77      {
78          Console.WriteLine($"[Email] → {patient}: Ваш запис скасовано");
79      }
80  }
```

Що не так із цим класом? Щоразу, коли **будь-який** із трьох акторів змінює вимоги — змінюється весь клас. Це ризик регресії: правлячи логіку збереження, можна ненавмисно зачепити логіку сповіщень. Тестуючи правила бронювання, доводиться піднімати базу даних і SMTP-сервер.



До SRP — три відповідальності в одному класі

Рефакторинг: розбиваємо на три класи

Відповідно до SRP, кожна «вісь змін» стає окремим класом. Бізнес-логіка, збереження і сповіщення — три різні відповідальності, три різні класи.

C#

```
using System;
using System.Collections.Generic;

// ── Використання ─────────────────────────────────────────────
var coordinator = new ClinicCoordinator(
    new AppointmentService(),
    new AppointmentRepository(),
    new AppointmentNotifier()
);

coordinator.Book("Мороз Василь", "Петренко І.О.",
DateTime.Today.AddHours(10));
coordinator.Book("Бойко Олена",  "Петренко І.О.",
DateTime.Today.AddHours(10)); // зайнято
coordinator.Cancel("Мороз Василь", "Петренко І.О.");

// ─── ПРАВИЛЬНО: кожен клас — одна відповідальність ──────────

// ── 1. Тільки бізнес-логіка ──────────────────────────────────
class AppointmentService
{
    private List<(string Patient, string Doctor, DateTime Time, string
Status)> _appointments = new();

    // Повертає true/false — і нічого більше. Не зберігає, не надсилає.
    public bool Book(string patient, string doctor, DateTime time)
    {
        foreach (var a in _appointments)
            if (a.Doctor == doctor && a.Time == time && a.Status == "Active")
                return false; // слот зайнятий

        _appointments.Add((patient, doctor, time, "Active"));
        return true;
    }

    public bool Cancel(string patient, string doctor)
    {
        for (int i = 0; i < _appointments.Count; i++)
        {
            var a = _appointments[i];
            if (a.Patient == patient && a.Doctor == doctor && a.Status ==
"Active")
            {
                _appointments[i] = (a.Patient, a.Doctor, a.Time, "Cancelled");
                return true;
            }
        }
        return false;
    }

    public int CountActive(string doctor) =>
        _appointments.FindAll(a => a.Doctor == doctor && a.Status ==
"Active").Count;
}

// ── 2. Тільки збереження ─────────────────────────────────────
class AppointmentRepository
{
    // Лише відповідальність перед IT-відділом: зберегти і отримати.
```

```
    // Ніякої бізнес-логіки, ніяких email.

    public void Save(string patient, string doctor, DateTime time)
    {
        Console.WriteLine($"[DB] Збережено: {patient} → {doctor} @ {time:HH:mm}");
    }

    public void UpdateStatus(string patient, string doctor, string status)
    {
        Console.WriteLine($"[DB] Статус змінено: {patient}/{doctor} → {status}");
    }
}

// ── 3. Тільки сповіщення ──────────────────────────────────────
class AppointmentNotifier
{
    // Лише відповідальність перед маркетингом: надіслати повідомлення.
    // Хочуть SMS — змінюємо тільки цей клас. Решта не торкається.

    public void SendConfirmation(string patient, string doctor, DateTime time)
    {
        Console.WriteLine($"[Email] → {patient}: Запис до {doctor} о {time:HH:mm} підтверджено");
    }

    public void SendCancellation(string patient)
    {
        Console.WriteLine($"[Email] → {patient}: Ваш запис скасовано");
    }
}

// ── Оркестратор (координує три класи) ─────────────────────────
class ClinicCoordinator
{
    private readonly AppointmentService    _service;
    private readonly AppointmentRepository _repo;
    private readonly AppointmentNotifier   _notifier;

    public ClinicCoordinator(AppointmentService s, AppointmentRepository r, AppointmentNotifier n)
    {
        _service  = s;
        _repo     = r;
        _notifier = n;
    }

    public void Book(string patient, string doctor, DateTime time)
    {
        if (!_service.Book(patient, doctor, time))
        {
            Console.WriteLine($"[Clinic] {doctor} зайнятий о {time:HH:mm} — запис неможливий");
            return;
        }
        _repo.Save(patient, doctor, time);
        _notifier.SendConfirmation(patient, doctor, time);
        Console.WriteLine($"[Clinic] Запис успішний: {patient} → {doctor}");
    }
```

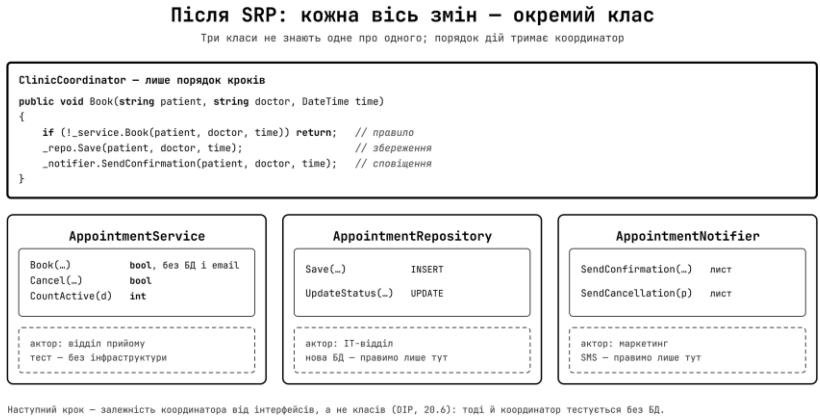
```
110
111     public void Cancel(string patient, string doctor)
112     {
113         if (!_service.Cancel(patient, doctor))
114         {
115             Console.WriteLine($"[Clinic] Активний запис не знайдено");
116             return;
117         }
118         _repo.UpdateStatus(patient, doctor, "Cancelled");
119         _notifier.SendCancellation(patient);
120         Console.WriteLine($"[Clinic] Запис скасовано: {patient}");
121     }
122 }
```

Тепер кожен клас має рівно одну причину для зміни:

— Змінюються правила бронювання → редагуємо лише AppointmentService

— Переходимо з SQL на MongoDB → редагуємо лише AppointmentRepository

— Додаємо SMS замість Email → редагуємо лише AppointmentNotifier



Після SRP — кожен клас незалежний і має одну відповідальність

Переваги SRP для тестування

Найвідчутніша практична перевага SRP — **тестованість**. Коли відповідальності розділені, кожен клас можна тестувати ізольовано:

```
C#
 1  using System;
```

```
using System.Collections.Generic;

// ── Симуляція юніт-тестів ──────────────────────────────────
void Assert(bool condition, string testName)
{
    Console.WriteLine(condition ? $"✓ PASS: {testName}" : $"✗ FAIL:
{testName}");
}

var svc = new AppointmentService_Testable();
var time1 = new DateTime(2024, 3, 15, 10, 0, 0);
var time2 = new DateTime(2024, 3, 15, 11, 0, 0);

// Тест 1: успішне бронювання
bool result1 = svc.Book("Мороз В.", "Петренко", time1);
Assert(result1 == true, "Book — успішне бронювання");

// Тест 2: подвійне бронювання того самого слоту
bool result2 = svc.Book("Бойко О.", "Петренко", time1);
Assert(result2 == false, "Book — слот уже зайнятий");

// Тест 3: різний час — обидва успішні
bool result3 = svc.Book("Бойко О.", "Петренко", time2);
Assert(result3 == true, "Book — інший час, успішно");

// Тест 4: лічильник активних записів
int count = svc.CountActiveForDoctor("Петренко");
Assert(count == 2, $"CountActive — очікуємо 2, маємо {count.ToString()}");

// Тест AppointmentService — без бази даних і без email
class AppointmentService_Testable
{
    private List<(string Patient, string Doctor, DateTime Time, string Status)>
_appointments = new();

    public bool Book(string patient, string doctor, DateTime time)
    {
        foreach (var a in _appointments)
            if (a.Doctor == doctor && a.Time == time && a.Status == "Active")
                return false;
        _appointments.Add((patient, doctor, time, "Active"));
        return true;
    }

    public int CountActiveForDoctor(string doctor) =>
        _appointments.FindAll(a => a.Doctor == doctor && a.Status ==
"Active").Count;
}
```

Зверніть увагу: жодного SMTP-сервера, жодної бази даних. AppointmentService тестується в ізоляції, бо він нічого не знає про інфраструктуру.

Реальний SRP: клас Patient як зразок

Правильно побудований клас Patient є хорошим прикладом дотримання SRP: він повністю відповідає за **представлення і валідацію пацієнта як доменної сутності** і не робить нічого іншого.

```C#
using System;

// Що НЕ має бути в Patient:
// ✗ SaveToDatabase()     — це відповідальність PatientRepository
// ✗ SendWelcomeEmail()   — це відповідальність PatientNotifier
// ✗ GenerateReport()     — це відповідальність ReportService

var patient = new Patient(1, "Іван", "Петренко", new DateTime(1985, 6, 20), "A(II)+");
Console.WriteLine(patient.ToString());
Console.WriteLine($"Вік: {patient.Age.ToString()} р.");
Console.WriteLine($"Неповнолітній: {patient.IsMinor.ToString()}");

// Клас Patient — відповідальність: доменна модель пацієнта
// Причина для зміни: ТІЛЬКИ зміна в бізнес-правилах щодо пацієнта
class Patient
{
    public int    Id          { get; }
    public string FirstName   { get; private set; }
    public string LastName    { get; private set; }
    public DateTime DateOfBirth { get; }
    public string BloodType   { get; }

    public Patient(int id, string firstName, string lastName,
                   DateTime dateOfBirth, string bloodType)
    {
        if (string.IsNullOrWhiteSpace(firstName))
            throw new ArgumentException("Ім'я не може бути порожнім", nameof(firstName));
        if (string.IsNullOrWhiteSpace(lastName))
            throw new ArgumentException("Прізвище не може бути порожнім", nameof(lastName));
        if (dateOfBirth > DateTime.Today)
            throw new ArgumentException("Дата народження не може бути в майбутньому", nameof(dateOfBirth));

        Id          = id;
        FirstName   = firstName;
        LastName    = lastName;
        DateOfBirth = dateOfBirth;
        BloodType   = bloodType;
    }

    // Доменна поведінка: вік — це бізнес-знання про пацієнта
    public int Age => (DateTime.Today - DateOfBirth).Days / 365;

    // Доменна поведінка: чи є пацієнт неповнолітнім
    public bool IsMinor => Age < 18;

    // Форматування — частина відповідальності «представлення пацієнта»
```

```
    public string FullName => $"{LastName} {FirstName}";

    public override string ToString() =>
        $"{FullName}, {Age.ToString()} р., група крові: {BloodType}";
}
```

Ознаки порушення SRP

На практиці порушення SRP легко впізнати за кількома симптомами:

Клас має методи, що змінюються з різних причин. Якщо ви описуєте клас і вживаєте слово «і» — «цей клас зберігає дані **і** надсилає повідомлення» — скоріш за все, є порушення.

Клас ImportManager, ExportManager, ProcessManager. Суфікси -Manager і -Helper часто є сигналом «сміттєвого» класу, куди скинули все, що не вклалося в чітку відповідальність.

Методи класу природно групуються у непересічні підмножини. Якщо методи A, B, C ніколи не використовують спільного стану класу, і методи X, Y, Z — теж ніколи не перетинаються з першою групою — це два кандидати на розбиття.

Тест одного методу вимагає ініціалізації непов'язаних залежностей. Якщо для тесту методу BookAppointment треба налаштовувати SMTP — SRP порушено.

SRP і розмір класу

SRP не означає, що класи мають бути крихітними. Добре спроектований PatientService може мати 20–30 методів і при цьому повністю дотримуватися принципу — якщо всі ці методи обслуговують одну бізнес-відповідальність «управління пацієнтами». SRP — це про **осі змін**, а не про кількість рядків.

Водночас SRP не рекомендує нескінченно дробити класи. «Мікроклас» на два рядки, що нічого не приховує і нічого не інкапсулює — це не дотримання принципу, а його карикатура. Баланс досягається питанням: «Чи є тут більше однієї причини для зміни?» — і не більше.

```C#
using System;
using System.Collections.Generic;

var registry = new PatientRegistry();
int id1 = registry.Register("Мороз Василь Іванович");
int id2 = registry.Register("Бойко Олена Петрівна");
int id3 = registry.Register("Коваль Михайло Андрійович");
registry.PrintAll();

```

```
Console.WriteLine($"\nПошук #{id1.ToString()}: {registry.FindById(id1)}");
registry.Deregister(id2);
Console.WriteLine($"Пацієнтів після видалення: {registry.Count.ToString()}");

// ── Демонстрація: великий, але правильний клас ──────────────────────
// PatientRegistry відповідає за одне: реєстр пацієнтів у межах клініки.
// Багато методів — одна вісь змін: бізнес-правила щодо реєстру.

class PatientRegistry
{
    private Dictionary<int, string> _patients = new(); // id → ПІБ
    private int _nextId = 1;

    public int Register(string fullName)
    {
        int id = _nextId++;
        _patients[id] = fullName;
        Console.WriteLine($"[Registry] Зареєстровано #{id.ToString()}:
{fullName}");
        return id;
    }

    public bool Deregister(int id)
    {
        bool removed = _patients.Remove(id);
        if (removed) Console.WriteLine($"[Registry] Видалено
#{id.ToString()}");
        return removed;
    }

    public string? FindById(int id) =>
        _patients.TryGetValue(id, out string? name) ? name : null;

    public bool Exists(int id) => _patients.ContainsKey(id);

    public int Count => _patients.Count;

    public void PrintAll()
    {
        Console.WriteLine($"[Registry] Пацієнтів:
{_patients.Count.ToString()}");
        foreach (var (id, name) in _patients)
            Console.WriteLine($"  #{id.ToString()}: {name}");
    }
}
```

Шість методів, одна відповідальність — «бути реєстром пацієнтів». Це правильне застосування SRP.

20.3. OCP — Принцип відкритості/закритості

Open/Closed Principle (OCP) сформулював Бертран Мейєр (Bertrand Meyer) у 1988 році, а пізніше переосмислив і популяризував Роберт Мартін. Принцип звучить так: «Програмні сутності (класи, модулі,

функції) мають бути **відкриті для розширення**, але **закриті для модифікації»**.

На перший погляд ця фраза виглядає парадоксально: як можна розширити поведінку класу, не змінюючи його? Відповідь полягає у правильному використанні абстракцій — інтерфейсів і базових класів. Новий варіант поведінки додається через нову реалізацію абстракції (новий клас), а не через правку існуючого коду.

Чому це важливо? Існуючий код — це код, який вже **протестований**, **задокументований** і **розгорнутий у виробництво**. Кожна його зміна несе ризик: навіть невелика правка може зламати щось, що раніше працювало коректно. OCP мінімізує цей ризик: нова функціональність з'являється у новому коді, а старий залишається незайманим.

Порушення OCP: switch і if-ланцюги над типом

Найпоширеніше порушення OCP — це код, що перевіряє тип об'єкта через switch або if-ланцюги і виконує різну логіку залежно від цього типу. Такий патерн означає, що при появі кожного нового типу доведеться відкривати і змінювати вже існуючий клас.

```C#
using System;

var calc = new CostCalculator_Bad();

var appts = new[]
{
    new Appointment_Bad { PatientName = "Мороз В.",   Type = AppointmentType.Regular,    DurationMin = 45 },
    new Appointment_Bad { PatientName = "Бойко О.",   Type = AppointmentType.Urgent,     DurationMin = 20 },
    new Appointment_Bad { PatientName = "Коваль М.",  Type = AppointmentType.Specialist, DurationMin = 60 },
};

foreach (var a in appts)
{
    decimal cost = calc.CalculateCost(a);
    string name  = calc.GetTypeName(a);
    Console.WriteLine($"{a.PatientName}: {name} — {cost.ToString()} грн");
}

// ── ПОРУШЕННЯ OCP ──────────────────────────────────────────────
// Щоразу, як з'являється новий тип прийому, треба відкривати
// CostCalculator і дописувати новий case. Це пряме порушення:
// клас не закритий для модифікації.

enum AppointmentType { Regular, Urgent, Specialist, VIP }

```

```
class Appointment_Bad
{
    public string       PatientName { get; set; } = "";
    public AppointmentType Type     { get; set; }
    public int          DurationMin { get; set; }
}

class CostCalculator_Bad
{
    // Цей метод треба відкривати при кожному новому типі прийому
    public decimal CalculateCost(Appointment_Bad appt)
    {
        switch (appt.Type)
        {
            case AppointmentType.Regular:
                // 500 грн базова + 5 грн за хвилину понад 30
                decimal regular = 500m;
                if (appt.DurationMin > 30)
                    regular += (appt.DurationMin - 30) * 5m;
                return regular;

            case AppointmentType.Urgent:
                // Базова × 1.6 (коефіцієнт терміновості)
                return 500m * 1.6m;

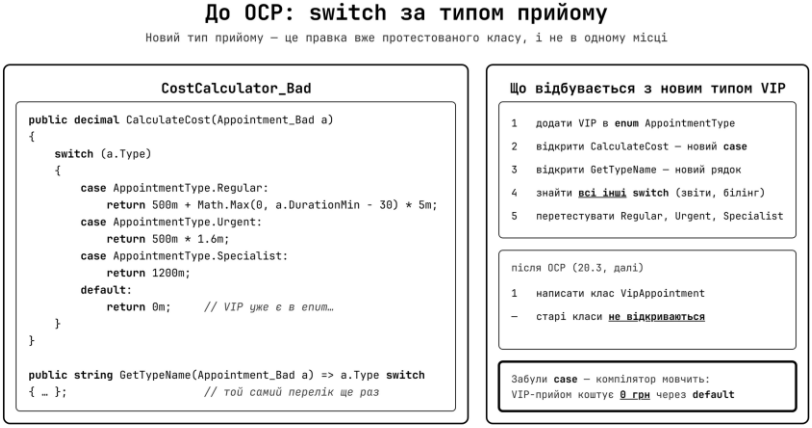
            case AppointmentType.Specialist:
                // Фіксована вартість без урахування тривалості
                return 1200m;

            // ← Новий тип VIP: треба ВІДКРИТИ цей файл і дописати case
            // ← Ризик: можна випадково зачепити Regular або Urgent
            // ← Цей switch дублюється в GenerateReport(), in
BillingService()...

            default:
                return 0m;
        }
    }

    // Другий метод з тим самим switch — OCP-порушення множиться
    public string GetTypeName(Appointment_Bad appt)
    {
        return appt.Type switch
        {
            AppointmentType.Regular    => "Первинний прийом",
            AppointmentType.Urgent     => "Терміновий прийом",
            AppointmentType.Specialist => "Консультація спеціаліста",
            _                          => "Невідомий тип",
        };
    }
}
```

Проблема не в тому, що switch-вирази погані самі по собі. Проблема в тому, що тут AppointmentType виступає точкою варіативності — місцем, де система повинна підтримувати розширення. І ця точка варіативності **не захищена абстракцією**: вона відкрита для прямого перебирання типів скрізь у коді.



До OCP — switch-ланцюги вимагають зміни існуючого коду

Рефакторинг: абстракція як точка розширення

OCP вирішується введенням **абстракції для точки варіативності**. Замість того, щоб клас-калькулятор знав про всі типи прийомів — кожен тип прийому знає свою вартість. Новий тип = новий клас, що реалізує інтерфейс. Калькулятор не змінюється.

```C#
using System;
using System.Collections.Generic;

// ─── Використання ──────────────────────────────────────────
var calculator = new CostCalculator();

// Список прийомів: різні типи, але один і той самий CostCalculator
var appointments = new List<(string Patient, IAppointmentType Type, int
Duration)>
{
    ("Мороз Василь",   new RegularAppointment(),                 45),
    ("Бойко Олена",    new UrgentAppointment(),                  20),
    ("Коваль Михайло", new SpecialistAppointment("Кардіологія"), 60),
    ("Сидоренко Тетяна",new VipAppointment(),                   90),
};

foreach (var (patient, type, duration) in appointments)
    calculator.PrintInvoice(type, patient, duration);

// ─── ПРАВИЛЬНО: OCP через абстракцію ──────────────────────

// Абстракція: контракт для будь-якого типу прийому
interface IAppointmentType
{
```

```
    string TypeName    { get; }
    decimal CalculateCost(int durationMinutes);
    string GetDescription();
}

// ─── Конкретні реалізації — окремі класи, не switch ──────────────

class RegularAppointment : IAppointmentType
{
    public string TypeName => "Первинний прийом";

    public decimal CalculateCost(int durationMinutes)
    {
        // 500 базова + 5 грн за кожну хвилину понад 30
        decimal cost = 500m;
        if (durationMinutes > 30)
            cost += (durationMinutes - 30) * 5m;
        return cost;
    }

    public string GetDescription() => "Базовий прийом у загальноклінічного
лікаря";
}

class UrgentAppointment : IAppointmentType
{
    public string TypeName => "Терміновий прийом";

    public decimal CalculateCost(int durationMinutes)
    {
        // Коефіцієнт терміновості ×1.6 до базової
        return 500m * 1.6m;
    }

    public string GetDescription() => "Позачерговий прийом при загостренні
стану";
}

class SpecialistAppointment : IAppointmentType
{
    private readonly string _speciality;

    public SpecialistAppointment(string speciality) => _speciality =
speciality;

    public string TypeName => $"Консультація: {_speciality}";

    public decimal CalculateCost(int durationMinutes)
    {
        // Фіксована вартість спеціалістів — 1200 грн
        return 1200m;
    }

    public string GetDescription() => $"Консультація спеціаліста:
{_speciality}";
}

// ─── Новий тип: VIP — НЕ ТОРКАЄМОСЬ існуючих класів ──────────────
class VipAppointment : IAppointmentType
{
```

```
    public string TypeName => "VIP-обслуговування";

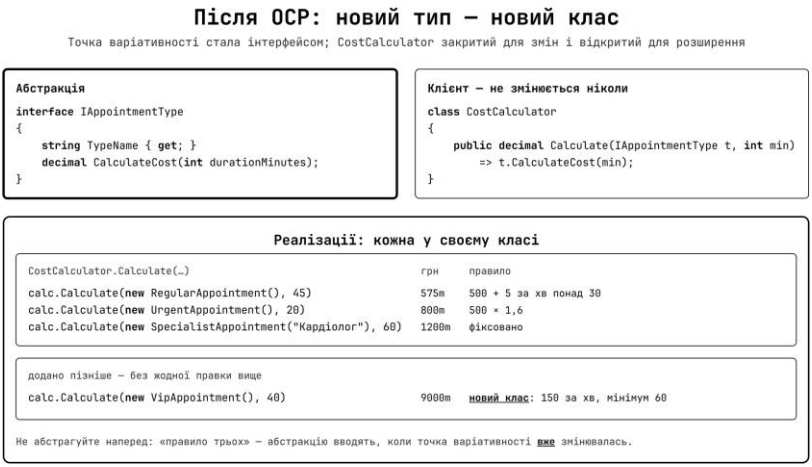
    public decimal CalculateCost(int durationMinutes)
    {
        // Погодинна ставка: 150 грн/хв, мінімум 60 хв
        int billableMinutes = Math.Max(durationMinutes, 60);
        return billableMinutes * 150m;
    }

    public string GetDescription() => "Індивідуальне обслуговування у VIP-кабінеті";
}

// ─── Калькулятор: ЗАКРИТИЙ для модифікації, але відкритий для розширення ──
class CostCalculator
{
    // Метод не знає про конкретні типи — лише про абстракцію
    public decimal Calculate(IAppointmentType type, int durationMinutes)
    {
        return type.CalculateCost(durationMinutes);
    }

    public void PrintInvoice(IAppointmentType type, string patient, int durationMinutes)
    {
        decimal cost = type.CalculateCost(durationMinutes);
        Console.WriteLine($"Пацієнт:    {patient}");
        Console.WriteLine($"Тип:        {type.TypeName}");
        Console.WriteLine($"Опис:       {type.GetDescription()}");
        Console.WriteLine($"Тривалість: {durationMinutes.ToString()} хв");
        Console.WriteLine($"Вартість:   {cost.ToString()} грн");
        Console.WriteLine(new string('-', 40));
    }
}
```

Зверніть увагу: CostCalculator не змінився при додаванні VipAppointment. Ми лише написали новий клас, що реалізує IAppointmentType. Старий код — протестований, стабільний — залишився незайманим.



Після OCP — новий тип = новий клас, без зміни калькулятора

OCP і поліморфізм: зв'язок із попередніми темами

OCP безпосередньо спирається на **поліморфізм** — механізм, що дозволяє CostCalculator.Calculate приймати будь-яку реалізацію IAppointmentType і викликати правильну версію CalculateCost. Це і є «відкритість для розширення»: нові реалізації інтерфейсу автоматично підтримуються існуючим кодом.

Важлива деталь: OCP не вимагає, щоб клас був закритий від самого початку. Мартін описував це як «прийняти першу кулю» (*take the first bullet*): спочатку пишемо найпростіший код, а коли надходить перша зміна певного виду — вводимо абстракцію, що захистить клас від подальших змін цього ж виду. Близьке за духом «правило трьох» (*three strikes and you refactor*) належить Дону Робертсу й популяризоване Мартіном Фаулером у книзі «Refactoring» (1999): коли схожий код з'являється втретє — час виділити спільну абстракцію. Обидва правила кажуть одне: точку варіативності треба побачити в реальних змінах, а не вгадувати наперед. Передчасна абстракція — теж помилка: вона ускладнює код заради змін, які можуть ніколи не настати.

Стратегія (Strategy Pattern) — класичне застосування OCP

OCP часто реалізується через патерн **Стратегія (Strategy)**: алгоритм виноситься в інтерфейс, а конкретні реалізації передаються ззовні. Це дозволяє замінювати поведінку без зміни класу, що її використовує.

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3
 4  var billing = new BillingService();
 5  billing.GenerateBill("Мороз Василь",    1200m, new NoDiscount());
 6  billing.GenerateBill("Пенсіонер Іван",  1200m, new PercentageDiscount(30));
 7  billing.GenerateBill("Абонемент Олена", 1200m, new FlatDiscount(300m));
 8  billing.GenerateBill("Бойко Тетяна",   1200m, new SeasonalDiscount("Літо",
    15));
 9
10  // ── Strategy Pattern: алгоритм знижки ──────────────────────────────
11
12  interface IDiscountStrategy
13  {
14      decimal Apply(decimal originalCost);
15      string Description { get; }
16  }
17
18  class NoDiscount : IDiscountStrategy
19  {
20      public string Description => "Без знижки";
21      public decimal Apply(decimal cost) => cost;
22  }
23
24  class PercentageDiscount : IDiscountStrategy
25  {
26      private readonly int _percent;
27      public PercentageDiscount(int percent) => _percent = percent;
28      public string Description => $"Знижка {_percent.ToString()}%";
29      public decimal Apply(decimal cost) => cost * (1 - _percent / 100m);
30  }
31
32  class FlatDiscount : IDiscountStrategy
33  {
34      private readonly decimal _amount;
35      public FlatDiscount(decimal amount) => _amount = amount;
36      public string Description => $"Знижка {_amount.ToString()} грн";
37      public decimal Apply(decimal cost) => Math.Max(0, cost - _amount);
38  }
39
40  // Новий тип знижки — без торкання до Billing ────────────────────────
41  class SeasonalDiscount : IDiscountStrategy
42  {
43      private readonly int _percent;
44      private readonly string _seasonName;
45      public SeasonalDiscount(string season, int pct) { _seasonName = season;
    _percent = pct; }
46      public string Description => $"Сезонна знижка «{_seasonName}» —
    {_percent.ToString()}%";
47      public decimal Apply(decimal cost) => cost * (1 - _percent / 100m);
48  }
```

```
49
50  // BillingService: закритий для модифікації, відкритий для нових знижок
51  class BillingService
52  {
53      public void GenerateBill(string patient, decimal baseCost,
    IDiscountStrategy discount)
54      {
55          decimal final = discount.Apply(baseCost);
56          Console.WriteLine($"Пацієнт: {patient}");
57          Console.WriteLine($"  Базова вартість:  {baseCost.ToString()} грн");
58          Console.WriteLine($"  {discount.Description}");
59          Console.WriteLine($"  До сплати:        {final.ToString("F2")} грн");
60          Console.WriteLine();
61      }
62  }
```

BillingService.GenerateBill не змінювався при додаванні SeasonalDiscount. Це OCP у дії.

OCP через успадкування: Template Method

Ще один класичний спосіб реалізувати OCP — **шаблонний метод (Template Method)**: базовий клас визначає алгоритм, а підкласи перевизначають конкретні кроки.

```
C#
 1  using System;
 2
 3  new CardiologyExam().Conduct("Мороз Василь");
 4  new NeurologyExam().Conduct("Бойко Олена");
 5  new LabExam().Conduct("Коваль Михайло");
 6
 7  // ── Template Method: базовий алгоритм закрито, кроки — відкрито ──
 8
 9  abstract class MedicalExamination
10  {
11      // Шаблонний метод: визначає порядок дій — закритий
12      public void Conduct(string patient)
13      {
14          Console.WriteLine($"\n=== Обстеження: {patient} ===");
15          RegisterPatient(patient);
16          PerformExamination(patient);   // абстрактний — відкритий для підкласів
17          RecordResults(patient);
18          ScheduleFollowUp(patient);     // може бути перевизначений
19          Console.WriteLine($"=== Завершено ===");
20      }
21
22      private void RegisterPatient(string patient)
23          => Console.WriteLine($"[Reception] Реєстрація: {patient}");
24
25      // Відкритий для розширення — підклас визначає конкретне обстеження
26      protected abstract void PerformExamination(string patient);
27
28      private void RecordResults(string patient)
29          => Console.WriteLine($"[Records] Результати внесено до картки
    {patient}");
```

```
30
31      // Virtual: підклас може перевизначити, але не зобов'язаний
32      protected virtual void ScheduleFollowUp(string patient)
33          => Console.WriteLine($"[Schedule] Повторний огляд через 4 тижні");
34  }
35
36  // Кожен підклас розширює алгоритм без зміни базового класу
37  class CardiologyExam : MedicalExamination
38  {
39      protected override void PerformExamination(string patient)
40      {
41          Console.WriteLine($"[Cardio] ЕКГ, ехокардіографія, вимірювання тиску:
    {patient}");
42      }
43
44      protected override void ScheduleFollowUp(string patient)
45          => Console.WriteLine($"[Schedule] Кардіологічний контроль через 2
    тижні");
46  }
47
48  class NeurologyExam : MedicalExamination
49  {
50      protected override void PerformExamination(string patient)
51      {
52          Console.WriteLine($"[Neuro] МРТ, рефлекси, когнітивні тести:
    {patient}");
53      }
54      // ScheduleFollowUp не перевизначено — використовується базова реалізація
55  }
56
57  // Новий тип обстеження — без зміни MedicalExamination ──────────────
58  class LabExam : MedicalExamination
59  {
60      protected override void PerformExamination(string patient)
61      {
62          Console.WriteLine($"[Lab] Аналіз крові, сечі, біохімія: {patient}");
63      }
64
65      protected override void ScheduleFollowUp(string patient)
66          => Console.WriteLine($"[Schedule] Результати готові через 2 дні");
67  }
```

Коли OCP особливо важливий

OCP найважливіший у трьох ситуаціях:

Бібліотеки та фреймворки: код, що використовується іншими командами або проектами, не може бути змінений клієнтами — тому він має бути розширюваним через абстракції.

Місця з частими змінами: якщо ви вже тричі змінювали один і той самий клас через схожі причини (новий тип знижки, новий формат звіту, новий канал сповіщень) — час закрити його від цих змін.

Плагін-архітектури: системи, де поведінка налаштовується зовні (наприклад, медичні модулі у клінічній системі), проектуються через OCP від самого початку.

Межі OCP: не все треба абстрагувати

OCP — не заклик до надмірної абстракції. Абстракція коштує: вона ускладнює код, збільшує кількість файлів і вимагає більше для розуміння. Абстракція виправдана лише там, де очікується реальна варіативність.

Практичне правило: не закривайте клас від змін, поки не побачите реальну потребу в розширенні. Замість цього пишіть простий код, а коли з'являється другий або третій різновид — рефакторингом виявляйте абстракцію. Це узгоджується зі принципом «YAGNI» (You Aren't Gonna Need It — вам це не знадобиться) і зі стилем «Agile», в якому OCP і з'явився.

```
C#
using System;
using System.Collections.Generic;

var data = new List<string>
{
    "Петренко І.О. — 12 прийомів", "Коваленко М.А. — 9 прийомів", "Бойко О.Р. — 7 прийомів"
};

var printer = new ReportPrinter();
printer.Print(new PlainTextFormatter(), "Рейтинг лікарів", data);
printer.Print(new CsvFormatter(),       "Рейтинг лікарів", data);
printer.Print(new MarkdownFormatter(),  "Рейтинг лікарів", data);

// ─── Демонстрація: поліморфна обробка списку через OCP ──────────

interface IReportFormatter
{
    string Format(string title, List<string> rows);
}

class PlainTextFormatter : IReportFormatter
{
    public string Format(string title, List<string> rows)
    {
        var sb = new System.Text.StringBuilder();
        sb.AppendLine($"=== {title} ===");
        foreach (var row in rows) sb.AppendLine($"  {row}");
        return sb.ToString();
    }
}

class CsvFormatter : IReportFormatter
{
    public string Format(string title, List<string> rows)
    {
        var sb = new System.Text.StringBuilder();
        sb.AppendLine($"\"Звіт\",\"{title}\"");
        foreach (var row in rows) sb.AppendLine($"\"{row}\"");
        return sb.ToString();
    }
```

```
}

// Новий формат — без зміни ReportPrinter
class MarkdownFormatter : IReportFormatter
{
    public string Format(string title, List<string> rows)
    {
        var sb = new System.Text.StringBuilder();
        sb.AppendLine($"## {title}");
        foreach (var row in rows) sb.AppendLine($"- {row}");
        return sb.ToString();
    }
}

// ReportPrinter: закритий — ніколи не зміниться при нових форматах
class ReportPrinter
{
    public void Print(IReportFormatter formatter, string title, List<string>
rows)
    {
        Console.WriteLine(formatter.Format(title, rows));
    }
}
```

20.4. LSP — Принцип підстановки Ліскова

Liskov Substitution Principle (LSP) сформулювала Барбара Ліскова (Barbara Liskov) у 1987 році у доповіді «Data abstraction and hierarchy» на конференції OOPSLA. У доповіді принцип звучав так: «якщо для кожного об'єкта o1 типу S існує об'єкт o2 типу T такий, що для всіх програм P, визначених у термінах T, поведінка P не змінюється при підстановці o1 замість o2, то S є підтипом T». Строге формулювання, яке найчастіше цитують сьогодні, Лісков і Жаннетт Вінг (Jeannette Wing) дали пізніше — у статті «A behavioral notion of subtyping» (1994), і воно доволі математичне: «Якщо q(x) — деяка властивість, що є істинною для об'єктів x типу T, то q(y) має бути істинною для об'єктів y типу S, де S є підтипом T». На практиці це перефразовують простіше: **об'єкт підкласу має бути взаємозамінним з об'єктом базового класу без порушення коректності програми**.

Іншими словами: якщо десь у програмі очікується MedicalRecord, то передача Diagnosis (підкласу MedicalRecord) не повинна спричинити жодних несподіванок. Клієнтський код не повинен знати, який саме підклас він отримав — він знає лише контракт базового класу, і цього достатньо.

LSP — це, по суті, **перевірка правильності ієрархії успадкування**. Порушення LSP означає, що ієрархія побудована неправильно: або підклас не є справжнім «різновидом» базового класу,

або базовий клас взяв на себе більше обіцянок, ніж може виконати кожен підклас.

Контракт методу: передумови, постумови, інваріанти

Щоб зрозуміти LSP глибоко, треба знати поняття **контракту** методу або класу. Контракт складається з трьох частин:

Передумови (preconditions) — умови, що мають виконуватися перед викликом методу. Наприклад: patientId не може бути від'ємним, date не може бути в минулому. Підклас **не може посилювати** передумови: якщо базовий клас приймає будь-яке ціле число, підклас не може раптом вимагати лише числа більше нуля — клієнт про це не знає і не перевіряє.

Постумови (postconditions) — гарантії щодо стану після виклику методу. Наприклад: метод Book гарантує, що після виклику запис існує. Підклас **не може послаблювати** постумови: якщо базовий клас гарантує додавання запису — підклас не може «вирішити» цього не робити.

Інваріанти (invariants) — умови, що завжди мають виконуватися для об'єкта протягом усього його існування. Наприклад: Patient.Id завжди додатне, Appointment.Duration завжди більше нуля. Підклас зобов'язаний зберігати інваріанти базового класу.

LSP у термінах контракту: **підклас може послаблювати передумови і посилювати постумови, але не навпаки**.

Класичний приклад порушення LSP: прямокутник і квадрат

Найвідоміший приклад порушення LSP — «проблема прямокутника і квадрата». Квадрат математично є прямокутником (кожен квадрат — прямокутник), тому здається логічним успадкувати Square від Rectangle. Але це ламає LSP:

```C#
using System;

// Код, що працює з Rectangle — нічого не знає про Square
void SetAndPrint(Rectangle r)
{
    r.Width  = 5;
    r.Height = 3;
    // Очікуємо: 5×3 = 15
    Console.WriteLine($"{r}: площа = {r.Area().ToString()}");
}

var rect = new Rectangle();
SetAndPrint(rect);   // Rectangle(5×3): площа = 15 ✓
```

```
14
15  var sq = new Square();
16  SetAndPrint(sq);     // Square(3×3): площа = 9  ✗ (очікувалось 15!)
17  // LSP порушено: підстановка Square замість Rectangle дає неочікуваний
    результат
18
19  // ─── Класичний приклад порушення LSP ──────────────────────────────
20
21  class Rectangle
22  {
23      public virtual int Width  { get; set; }
24      public virtual int Height { get; set; }
25
26      public int Area() => Width * Height;
27
28      public override string ToString() =>
    $"Rectangle({Width.ToString()}×{Height.ToString()})";
29  }
30
31  class Square : Rectangle
32  {
33      // Квадрат примушує Width == Height — але ламає контракт Rectangle!
34      public override int Width
35      {
36          set { base.Width = value; base.Height = value; } // змінює обидва
37          get => base.Width;
38      }
39      public override int Height
40      {
41          set { base.Width = value; base.Height = value; } // змінює обидва
42          get => base.Height;
43      }
44  }
```

Код SetAndPrint нічого не знає про Square. Він встановлює Width=5, Height=3 і очікує площу 15. Але Square «тихо» змінює обидва значення на останнє присвоєне. Поведінка змінилась — контракт порушено. Клієнт отримує несподіваний результат без жодного повідомлення про помилку.

Рішення: прибрати хибну ієрархію. Rectangle і Square — це дві незалежні форми, що не перебувають у відношенні «є» в контексті цього коду. Обидва можуть реалізовувати інтерфейс IShape, але не успадковувати одне від одного.

Порушення LSP у клінічній системі

Розглянемо реалістичний приклад порушення LSP у нашій системі. Є ієрархія MedicalRecord з підкласами Diagnosis, LabResult, Prescription. Новий розробник вирішує додати ReadOnlyRecord — «медичний запис тільки для читання» (наприклад, для архіву):

```
C#
```

```
using System;
using System.Collections.Generic;

// ─── Клієнтський код — не знає про ReadOnlyRecord ──────────────────
void ProcessRecord(MedicalRecord record, string diagCode)
{
    Console.WriteLine($"Обробка: {record.GetSummary()}");
    record.AddDiagnosis(diagCode); // очікує: OK для будь-якого MedicalRecord
}

var active = new ActiveDiagnosis(1, 101, "J06.9");
ProcessRecord(active, "I10"); // ✓ Працює

var archived = new ReadOnlyRecord(2, 102);
try
{
    ProcessRecord(archived, "E11.9"); // ✗ Вибух у runtime!
}
catch (InvalidOperationException ex)
{
    Console.WriteLine($"[Помилка] {ex.Message}");
    Console.WriteLine("LSP порушено: підстановка ReadOnlyRecord зламала клієнтський код");
}

// ─── ПОРУШЕННЯ LSP ─────────────────────────────────────────────────

abstract class MedicalRecord
{
    public int     Id        { get; protected set; }
    public int     PatientId { get; protected set; }
    public DateTime Date     { get; protected set; }
    protected List<string> _diagnoses = new();

    protected MedicalRecord(int id, int patientId)
    {
        Id = id; PatientId = patientId; Date = DateTime.Today;
    }

    // Контракт: метод доступний і виконує додавання
    public virtual void AddDiagnosis(string code)
    {
        _diagnoses.Add(code);
        Console.WriteLine($"[Record #{Id.ToString()}] Діагноз додано: {code}");
    }

    public virtual string GetSummary() =>
        $"Record #{Id.ToString()}, пацієнт #{PatientId.ToString()}, " +
        $"діагнозів: {_diagnoses.Count.ToString()}";
}

// Звичайний підклас — дотримується контракту ──────────────────────
class ActiveDiagnosis : MedicalRecord
{
    public string DiagnosisCode { get; private set; }

    public ActiveDiagnosis(int id, int patientId, string code) : base(id, patientId)
    {
        DiagnosisCode = code;
```

```
        _diagnoses.Add(code); // власний діагноз
    }

    // Розширює базовий клас, не порушуючи контракту
    public override string GetSummary() =>
        base.GetSummary() + $" | Основний: {DiagnosisCode}";
}

// ⚠ Підклас, що ПОРУШУЄ LSP ──────────────────────────────────────
class ReadOnlyRecord : MedicalRecord
{
    public ReadOnlyRecord(int id, int patientId) : base(id, patientId) { }

    // Порушення: посилює передумову до «ніколи не виклич мене»
    public override void AddDiagnosis(string code)
    {
        // Клієнт очікує: OK. Отримує: виняток!
        throw new InvalidOperationException("Цей запис у режимі тільки для
читання");
    }
}
```

Порушення LSP: підклас, що відмовляється від контракту

ProcessRecord чесно працює з MedicalRecord — і падає, коли йому підставили ReadOnlyRecord

Базовий клас — контракт: «AddDiagnosis додає діагноз»

```
abstract class MedicalRecord
{
    public virtual void AddDiagnosis(string code) => _diagnoses.Add(code);
    public virtual string GetSummary() => …;
}
```

ActiveDiagnosis — дотримується

```
class ActiveDiagnosis : MedicalRecord
{
    // AddDiagnosis успадковано без змін
    public override string GetSummary() =>
        base.GetSummary() + " | Основний: …";
}
```

розширює поведінку, не звужуючи її

ReadOnlyRecord — порушує

```
class ReadOnlyRecord : MedicalRecord
{
    public override void AddDiagnosis(string code)
        => throw new InvalidOperationException(
            "…тільки для читання");
}
```

посилює передумову до «не викликай мене»

Клієнт: void ProcessRecord(MedicalRecord record, string diagCode)

виклик	результат
`ProcessRecord(new ActiveDiagnosis(1, 101, "J06.9"), "I10")`	діагноз додано, тепер їх 2
`ProcessRecord(new ReadOnlyRecord(2, 102), "E11.9")`	`InvalidOperationException` — у runtime

Компілятор мовчить: за типом ReadOnlyRecord є MedicalRecord. Помилку побачить лише користувач.

Порушення LSP — ReadOnlyRecord кидає виняток там, де очікується нормальна робота

Виправлення: правильна ієрархія

Якщо ReadOnlyRecord не може виконати контракт MedicalRecord — він не повинен бути підкласом MedicalRecord. Рішення: або введення окремої ієрархії, або інтерфейсів, що розділяють читання і запис.

```
C#
using System;
using System.Collections.Generic;

// ── Клієнтський код: чіткі контракти ──────────────────────────────
void DisplayRecord(IMedicalRecord record)
{
    // Може лише читати — LSP гарантований
    Console.WriteLine(record.GetSummary());
}

void EditRecord(IEditableMedicalRecord record, string code)
{
    // Може редагувати — LSP гарантований: тільки EditableRecord тут буде
    record.AddDiagnosis(code);
}

var active   = new ActiveRecord(1, 101);
var archived = new ArchivedRecord(2, 102, new DateTime(2023, 5, 10), "Хронічна
гіпертонія");

DisplayRecord(active);    // ✓
DisplayRecord(archived);  // ✓
EditRecord(active, "J06.9"); // ✓

// ArchivedRecord компілятор просто не дозволить передати в EditRecord
// EditRecord(archived, "I10"); // ← помилка компіляції, не runtime!
Console.WriteLine("LSP дотримано: помилка виявляється на етапі компіляції");

// ─── ПРАВИЛЬНО: ієрархія відповідає LSP ──────────────────────────

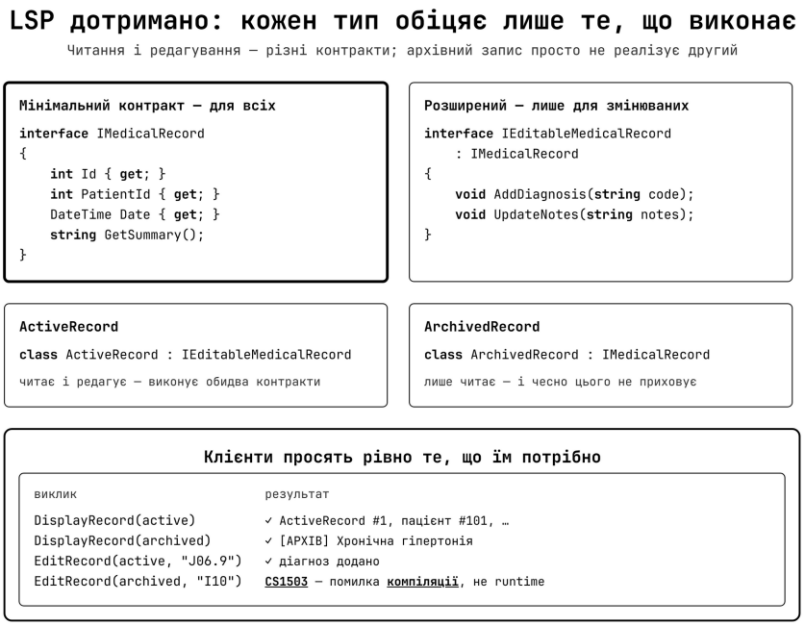
// Мінімальний контракт: тільки те, що виконає КОЖЕН нащадок
interface IMedicalRecord
{
    int     Id        { get; }
    int     PatientId { get; }
    DateTime Date     { get; }
    string  GetSummary();
}

// Розширений контракт: тільки для записів, що підтримують модифікацію
interface IEditableMedicalRecord : IMedicalRecord
{
    void AddDiagnosis(string code);
    void UpdateNotes(string notes);
}

// ── Активний запис: реалізує обидва інтерфейси ────────────────────
class ActiveRecord : IEditableMedicalRecord
{
    private List<string> _diagnoses = new();
```

```
50      private string _notes = "";
51
52      public int      Id        { get; }
53      public int      PatientId { get; }
54      public DateTime Date      { get; } = DateTime.Today;
55
56      public ActiveRecord(int id, int patientId) { Id = id; PatientId =
    patientId; }
57
58      public void AddDiagnosis(string code)
59      {
60          _diagnoses.Add(code);
61          Console.WriteLine($"[Record #{Id.ToString()}] Діагноз: {code}");
62      }
63
64      public void UpdateNotes(string notes)
65      {
66          _notes = notes;
67          Console.WriteLine($"[Record #{Id.ToString()}] Нотатки оновлено");
68      }
69
70      public string GetSummary() =>
71          $"ActiveRecord #{Id.ToString()}, пацієнт #{PatientId.ToString()}, " +
72          $"діагнозів: {_diagnoses.Count.ToString()}";
73  }
74
75  // ── Архівний запис: реалізує лише IMedicalRecord — і це чесно ──────
76  class ArchivedRecord : IMedicalRecord
77  {
78      private readonly string _summary;
79
80      public int      Id        { get; }
81      public int      PatientId { get; }
82      public DateTime Date      { get; }
83
84      public ArchivedRecord(int id, int patientId, DateTime date, string summary)
85      {
86          Id = id; PatientId = patientId; Date = date; _summary = summary;
87      }
88
89      public string GetSummary() => $"[АРХІВ] {_summary}";
90      // Немає AddDiagnosis — і клієнт про це знає через тип
91  }
```

Тепер замість runtime-вибуху — помилка компіляції: спробувати передати ArchivedRecord туди, де очікується IEditableMedicalRecord, компілятор не дозволить. Це принципова перевага: помилка виявлена раніше, і жодного несподіваного стану у виробництві.



Правильна ієрархія — підкласи розширюють, не звужують контракт

Ознаки порушення LSP

Практично LSP порушується в кількох характерних ситуаціях:

Виняток у перевизначеному методі: підклас перевизначає метод і кидає NotSupportedException, InvalidOperationException або NotImplementedException. Це класичне «пустий метод — порушення LSP».

Перевірка типу в клієнтському коді: якщо код пише if (record is ReadOnlyRecord) — це ознака того, що ієрархія неправильна. Клієнт не повинен знати про підкласи.

Ослаблення постумов: базовий метод гарантує, що після виклику Save(entity) дані збережені. Підклас перевизначає Save і нічого не робить — «тихе» порушення без жодного виключення.

Посилення передумов: базовий клас приймає будь-який рядок як ім'я. Підклас раптом вимагає, щоб ім'я не перевищувало 50 символів. Клієнти базового класу про це не знають.

```
C#
```

```
using System;

// ─── Поліморфний код — правильний LSP ──────────────────────────────
void PrintRecord(MedicalRecordBase record)
{
    // Для БУДЬ-ЯКОГО підкласу контракт виконається: GetSummary() не null,
    // GetStatus() — один з очікуваних рядків
    Console.WriteLine($"[{record.GetStatus()}] {record.GetSummary()}");
    Console.WriteLine($"  Дата: {record.Date:dd.MM.yyyy}, пацієнт
#{record.PatientId.ToString()}");
}

var records = new MedicalRecordBase[]
{
    new DiagnosisRecord(1, 101, 5, "J06.9", false, "ГРВІ, 7 днів лікування"),
    new DiagnosisRecord(2, 101, 5, "I10",   true,  "Гіпертонія, постійне
спостереження"),
    new PrescriptionRecord(3, 101, 5, "Ібупрофен 400мг", 5, "3 рази на день
після їжі"),
    new PrescriptionRecord(4, 102, 7, "Метформін 500мг", 30, "Вранці
натщесерце"),
};

foreach (var r in records) PrintRecord(r);

// ─── Демонстрація правильно побудованої ієрархії MedicalRecord ─────

abstract class MedicalRecordBase
{
    public int      Id        { get; }
    public int      PatientId { get; }
    public int      DoctorId  { get; }
    public DateTime Date      { get; }
    public string   Notes     { get; protected set; }

    protected MedicalRecordBase(int id, int patientId, int doctorId, string
notes)
    {
        if (id <= 0)        throw new ArgumentException("Id має бути
додатнім");
        if (patientId <= 0) throw new ArgumentException("PatientId має бути
додатнім");
        Id = id; PatientId = patientId; DoctorId = doctorId;
        Date = DateTime.Today; Notes = notes;
    }

    // Контракт: завжди повертає рядок з описом (ніколи null)
    public abstract string GetSummary();

    // Контракт: завжди повертає рядок — "Активний" або "Неактивний"
    public virtual string GetStatus() => "Активний";
}

class DiagnosisRecord : MedicalRecordBase
{
    public string DiagnosisCode { get; }
    public bool   IsChronic     { get; }

    public DiagnosisRecord(int id, int patientId, int doctorId,
                           string code, bool isChronic, string notes)
```

```
54          : base(id, patientId, doctorId, notes)
55      {
56          DiagnosisCode = code; IsChronic = isChronic;
57      }
58
59      // Виконує контракт: повертає не-null рядок
60      public override string GetSummary() =>
61          $"Діагноз [{DiagnosisCode}]{(IsChronic ? " (хронічний)" : "")} —
    {Notes}";
62  }
63
64  class PrescriptionRecord : MedicalRecordBase
65  {
66      public string   MedicationName { get; }
67      public int      DurationDays   { get; }
68      public DateTime ExpiresAt      { get; }
69
70      public PrescriptionRecord(int id, int patientId, int doctorId,
71                                string medication, int duration, string notes)
72          : base(id, patientId, doctorId, notes)
73      {
74          MedicationName = medication; DurationDays = duration;
75          ExpiresAt = DateTime.Today.AddDays(duration);
76      }
77
78      public override string GetSummary() =>
79          $"Рецепт: {MedicationName}, {DurationDays.ToString()} днів — {Notes}";
80
81      // Розширює постумову: уточнює статус (не звужує, не ламає)
82      public override string GetStatus() =>
83          ExpiresAt >= DateTime.Today ? "Активний" : "Прострочений";
84  }
```

Жодного switch, жодного is-перевірки. Кожен підклас виконує контракт базового — GetSummary() завжди повертає не-null рядок, GetStatus() завжди повертає статус. PrescriptionRecord лише **розширює** постумову GetStatus(), уточнюючи «Активний» або «Прострочений» — але не звужує і не ламає.

LSP і тестування: правило «замінюваності»

Практичний тест на дотримання LSP: якщо у вас є тест, що перевіряє поведінку базового класу, він **повинен проходити і для кожного підкласу** без змін. Це «правило замінюваності».

```
C#
 1  using System;
 2
 3  // Тест-перевірка, що має проходити для БУДЬ-якого Doctor ——————————
 4  void AssertDoctorContract(Doctor d)
 5  {
 6      decimal fee = d.GetConsultationFee();
 7      string spec = d.GetSpeciality();
 8
```

```
    bool feeOk  = fee > 0;
    bool specOk = spec != null && spec.Length <= 100;

    Console.WriteLine($"{d.Name} ({d.GetType().Name}):");
    Console.WriteLine($"  Fee={fee.ToString()} → {(feeOk ? "✓" : "✗ порушення!")}");
    Console.WriteLine($"  Speciality='{spec}' → {(specOk ? "✓" : "✗ порушення!")}");
}

var doctors = new Doctor[]
{
    new GeneralPractitioner("Петренко І.О."),
    new Specialist("Коваленко М.А.", "Кардіологія", 1200m),
    new Specialist("Бойко О.Р.",     "Неврологія",  1000m),
};

foreach (var d in doctors) AssertDoctorContract(d);

// Базовий клас
abstract class Doctor
{
    public string Name { get; }
    public Doctor(string name) { Name = name; }

    // Контракт: завжди повертає додатнє число
    public abstract decimal GetConsultationFee();

    // Контракт: ніколи не повертає рядок довший за 100 символів
    public abstract string GetSpeciality();
}

class GeneralPractitioner : Doctor
{
    public GeneralPractitioner(string name) : base(name) { }
    public override decimal GetConsultationFee() => 500m;     // ✓ > 0
    public override string  GetSpeciality()      => "Загальна практика"; // ✓ < 100 символів
}

class Specialist : Doctor
{
    private readonly string _field;
    private readonly decimal _fee;
    public Specialist(string name, string field, decimal fee) : base(name)
    { _field = field; _fee = fee; }
    public override decimal GetConsultationFee() => _fee;    // ✓ > 0 (якщо fee > 0)
    public override string  GetSpeciality()      => _field;  // ✓ < 100 символів
}
```

Якщо всі підкласи проходять однаковий тест контракту — LSP дотримано. Якщо хоч один підклас порушує — є проблема в ієрархії.

20.5. ISP — Принцип розподілу інтерфейсів

Interface Segregation Principle (ISP) сформулював Роберт Мартін у 1996 році у статті для журналу C++ Report. Принцип звучить так: «Клієнти не повинні залежати від методів, яких вони не використовують». Або у іншому формулюванні: «Краще багато вузькоспеціалізованих інтерфейсів, ніж один великий».

ISP — це принцип про **проектування інтерфейсів з точки зору клієнта**. Інтерфейс «поганий» не тому, що він великий сам по собі, а тому, що він змушує реалізатора брати на себе зобов'язання, які йому не потрібні. Якщо клас ReceptionDesk реалізує IClinicService і мусить написати GeneratePdfReport() та ExportToCsv() — хоча йому вони ніколи не знадобляться — це і є порушення ISP.

Наслідки такого «жирного» інтерфейсу (fat interface):

— Реалізатор пише «заглушки» — методи, що кидають NotImplementedException або просто нічого не роблять

— Зміна в інтерфейсі (навіть у методах, що клас не використовує) вимагає перекомпіляції і перевірки всіх реалізаторів

— Тест на ReceptionDesk раптом залежить від PDF-бібліотеки, хоча ReceptionDesk з PDF нічого спільного не має

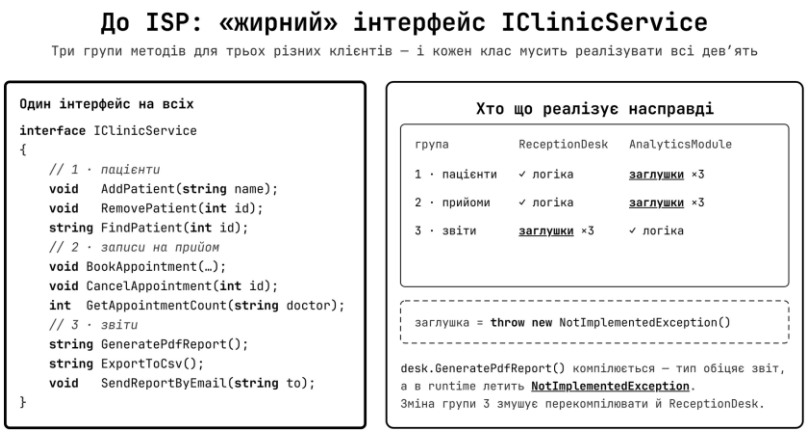
«Жирний» інтерфейс: класична помилка

Коли систему проектують поспіхом або «зверху вниз» без аналізу клієнтів, природно з'являється один великий інтерфейс, що об'єднує всі можливості системи:

```
C#
 1  using System;
 2
 3  // ── Проблема стає очевидною ──────────────────────────────────────
 4  var desk = new ReceptionDesk();
 5  desk.AddPatient("Мороз Василь");
 6  desk.BookAppointment("Мороз Василь", "Петренко І.О.",
    DateTime.Today.AddHours(10));
 7  try { desk.GeneratePdfReport(); }    // ✗ Вибух у runtime
 8  catch (NotImplementedException) { Console.WriteLine("[Помилка] ReceptionDesk не
    вміє PDF — але змушений реалізовувати метод"); }
 9
10  // ─── ПОРУШЕННЯ ISP — «жирний» інтерфейс ──────────────────────────
11
12  interface IClinicService
13  {
14      // Група 1: Управління пацієнтами
15      void   AddPatient(string name);
16      void   RemovePatient(int id);
17      string FindPatient(int id);
```

```
18
19      // Група 2: Записи на прийом
20      void BookAppointment(string patient, string doctor, DateTime time);
21      void CancelAppointment(int id);
22      int  GetAppointmentCount(string doctor);
23
24      // Група 3: Звіти та аналітика
25      string GeneratePdfReport();
26      string ExportToCsv();
27      void   SendReportByEmail(string to);
28  }
29
30  // ── ReceptionDesk потребує лише груп 1 і 2 ──────────────────────────
31  // Але мусить реалізувати ВСЕ — включно з PDF і Email, яких не знає
32
33  class ReceptionDesk : IClinicService
34  {
35      public void   AddPatient(string name)          =>
    Console.WriteLine($"Реєстрація: {name}");
36      public void   RemovePatient(int id)            =>
    Console.WriteLine($"Виписка: #{id.ToString()}");
37      public string FindPatient(int id)              => $"Пацієнт
    #{id.ToString()}";
38      public void   BookAppointment(string p, string d, DateTime t) =>
39
    Console.WriteLine($"Запис: {p} → {d}");
40      public void   CancelAppointment(int id)        =>
    Console.WriteLine($"Скасовано: #{id.ToString()}");
41      public int    GetAppointmentCount(string doc)  => 0;
42
43      // ↓ Вимушені заглушки — ReceptionDesk не знає про PDF і Email!
44      public string GeneratePdfReport() => throw new NotImplementedException("Не
    наша відповідальність");
45      public string ExportToCsv()       => throw new NotImplementedException("Не
    наша відповідальність");
46      public void   SendReportByEmail(string to) => throw new
    NotImplementedException("Не наша відповідальність");
47  }
48
49  // ── AnalyticsModule потребує лише групи 3 ──────────────────────────
50  class AnalyticsModule : IClinicService
51  {
52      // ↓ Вимушені заглушки — AnalyticsModule нічого не знає про пацієнтів!
53      public void   AddPatient(string name)          => throw new
    NotImplementedException();
54      public void   RemovePatient(int id)            => throw new
    NotImplementedException();
55      public string FindPatient(int id)              => throw new
    NotImplementedException();
56      public void   BookAppointment(string p, string d, DateTime t) => throw new
    NotImplementedException();
57      public void   CancelAppointment(int id)        => throw new
    NotImplementedException();
58      public int    GetAppointmentCount(string doc)  => throw new
    NotImplementedException();
59
60      // Реальна логіка — тільки тут
61      public string GeneratePdfReport() { Console.WriteLine("PDF згенеровано");
    return "report.pdf"; }
```

```
62      public string ExportToCsv()       { Console.WriteLine("CSV експортовано");
    return "data.csv";   }
63      public void   SendReportByEmail(string to)     =>
    Console.WriteLine($"Надіслано на {to}");
64  }
```



До ISP — «жирний» інтерфейс примушує реалізовувати непотрібне

Рефакторинг: розбиття на вузькі інтерфейси

Відповідно до ISP, кожна «група» методів, що обслуговує окремого клієнта, стає окремим інтерфейсом. Клас реалізує стільки інтерфейсів, скільки «ролей» він виконує:

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3
 4  // ── Клієнтський код — залежить від мінімального контракту ──────────
 5  void RegisterPatient(IPatientService svc, string name) =>
    svc.AddPatient(name);
 6  void MakeReport(IReportService svc, string email)
 7  {
 8      string pdf = svc.GeneratePdfReport();
 9      svc.SendReportByEmail(email);
10      Console.WriteLine($"Звіт: {pdf}");
11  }
12
13  var desk     = new ReceptionDesk();
14  var analytics= new AnalyticsModule();
15  var full     = new FullClinicService();
16
17  RegisterPatient(desk, "Мороз Василь");
18  RegisterPatient(desk, "Бойко Олена");
```

```
Console.WriteLine($"Записів лікаря:
{desk.GetAppointmentCount("Петренко").ToString()}");

MakeReport(analytics, "admin@clinic.ua");
MakeReport(full,      "boss@clinic.ua");

// Перевірка: desk не передається в MakeReport — компілятор не дозволить
// MakeReport(desk, "test@test.com"); // ← помилка компіляції
Console.WriteLine("\n✓ ISP дотримано: кожен клас реалізує лише свої методи");

// ─── ПРАВИЛЬНО: вузькі інтерфейси за роллю ──────────────────────────

// Кожен інтерфейс — мінімально необхідний контракт для конкретного клієнта

interface IPatientService
{
    void   AddPatient(string name);
    void   RemovePatient(int id);
    string FindPatient(int id);
}

interface IAppointmentService
{
    void BookAppointment(string patient, string doctor, DateTime time);
    void CancelAppointment(int id);
    int  GetAppointmentCount(string doctor);
}

interface IReportService
{
    string GeneratePdfReport();
    string ExportToCsv();
    void   SendReportByEmail(string to);
}

// ── ReceptionDesk: реалізує лише те, що потрібно ────────────────────
class ReceptionDesk : IPatientService, IAppointmentService
{
    private Dictionary<int, string> _patients     = new();
    private List<string>            _appointments = new();
    private int _nextId = 1;

    public void   AddPatient(string name)  { _patients[_nextId++] = name;
Console.WriteLine($"+ {name}"); }
    public void   RemovePatient(int id)    { _patients.Remove(id);
Console.WriteLine($"- #{id.ToString()}"); }
    public string FindPatient(int id)      => _patients.TryGetValue(id, out
var n) ? n : "Не знайдено";

    public void BookAppointment(string patient, string doctor, DateTime time)
    {
        _appointments.Add($"{patient}|{doctor}|{time:HH:mm}");
        Console.WriteLine($"Запис: {patient} → {doctor} о {time:HH:mm}");
    }
    public void CancelAppointment(int id)        =>
Console.WriteLine($"Скасовано: #{id.ToString()}");
    public int  GetAppointmentCount(string doc)  =>
        _appointments.FindAll(a => a.Contains($"|{doc}|")).Count;

    // ✓ Нема жодної заглушки для PDF — ReceptionDesk чесно не знає про звіти
```

```
}

// ── AnalyticsModule: реалізує лише звіти ──────────────────────
class AnalyticsModule : IReportService
{
    public string GeneratePdfReport()
    {
        Console.WriteLine("[Analytics] Генерація PDF-звіту...");
        return "clinic_report_2024.pdf";
    }
    public string ExportToCsv()
    {
        Console.WriteLine("[Analytics] Експорт у CSV...");
        return "clinic_data_2024.csv";
    }
    public void SendReportByEmail(string to)
        => Console.WriteLine($"[Analytics] Звіт надіслано на {to}");
}

// ── FullClinicService: реалізує всі ролі ──────────────────────
class FullClinicService : IPatientService, IAppointmentService, IReportService
{
    public void   AddPatient(string name)  => Console.WriteLine($"[Full] Пацієнт: {name}");
    public void   RemovePatient(int id)    => Console.WriteLine($"[Full] Виписано: #{id.ToString()}");
    public string FindPatient(int id)      => $"[Full] Пацієнт #{id.ToString()}";
    public void   BookAppointment(string p, string d, DateTime t) =>
                                              Console.WriteLine($"[Full] Запис: {p}→{d}");
    public void   CancelAppointment(int id)       => Console.WriteLine($"[Full] Скасовано");
    public int    GetAppointmentCount(string doc) => 5;
    public string GeneratePdfReport() => "[Full] report.pdf";
    public string ExportToCsv()       => "[Full] data.csv";
    public void   SendReportByEmail(string to)    => Console.WriteLine($"[Full] Email: {to}");
}
```



Після ISP — маленькі інтерфейси, кожен клієнт бере лише потрібне

ISP і принцип найменшого знання

ISP тісно пов'язаний із «принципом найменшого знання» (Principle of Least Knowledge або Law of Demeter): компонент повинен знати якомога менше про решту системи. Якщо ReceptionDesk реалізує IReportService, він «знає» про PDF і Email — навіть якщо сам їх не використовує. Вузькі інтерфейси природно мінімізують це «зайве знання».

ISP у реальному коді: інтерфейси як «ролі»

Хороший спосіб думати про ISP: кожен інтерфейс — це **роль**, яку клас може виконувати. Один клас може грати кілька ролей. Роль описана мінімально: лише те, що потрібно клієнту цієї ролі.

```C#
using System;
using System.Collections.Generic;

// ── Утиліти — кожна залежить від мінімального контракту ──────────
void SaveAll(IEnumerable<ISaveable> items)
{
    foreach (var item in items) item.Save();
}

void ArchiveOld(IEnumerable<IArchivable> items, string reason)
{
    foreach (var item in items)
        if (!item.IsArchived) item.Archive(reason);
}
```

```
15
16  void PrintAudit(IAuditable item)
17  {
18      Console.WriteLine($"  Створено: {item.CreatedAt:dd.MM.yyyy HH:mm}
    ({item.CreatedBy})");
19      Console.WriteLine($"  Змінено:  {(item.UpdatedAt.HasValue ?
    item.UpdatedAt.Value.ToString("dd.MM.yyyy HH:mm") : "ніколи")}");
20  }
21
22  var report = new MedicalReport("Кардіологічний огляд", "Норма", "Петренко
    І.О.");
23  var record1 = new PatientRecord(101, "Гіпертонія I ст.", "Коваленко М.А.");
24  var record2 = new PatientRecord(102, "ГРВІ", "Петренко І.О.");
25
26  // ISaveAble — save report and records
27  SaveAll(new ISaveable[] { report, record1, record2 });
28
29  // ISendable — тільки report
30  report.Send("patient@example.com");
31
32  // IArchivable — тільки records
33  ArchiveOld(new IArchivable[] { record1, record2 }, "Виписка");
34
35  // IAuditable — audit для record1
36  Console.WriteLine($"\nАудит запису пацієнта
    #{record1.PatientId.ToString()}:");
37  PrintAudit(record1);
38
39  // ─── ISP: інтерфейси як ролі для медичної системи ──────────────────
40
41  // Роль: об'єкт, що можна зберегти
42  interface ISaveable
43  {
44      void Save();
45  }
46
47  // Роль: об'єкт, що можна заархівувати
48  interface IArchivable
49  {
50      void Archive(string reason);
51      bool IsArchived { get; }
52  }
53
54  // Роль: об'єкт, що можна надіслати
55  interface ISendable
56  {
57      void Send(string recipient);
58  }
59
60  // Роль: об'єкт із аудит-слідом
61  interface IAuditable
62  {
63      DateTime CreatedAt  { get; }
64      DateTime? UpdatedAt { get; }
65      string    CreatedBy { get; }
66  }
67
68  // ── MedicalReport: зберігається і надсилається, але не архівується──
69  class MedicalReport : ISaveable, ISendable, IAuditable
70  {
```

```
    public string Title     { get; }
    public string Content   { get; }
    public DateTime CreatedAt  { get; } = DateTime.Now;
    public DateTime? UpdatedAt { get; private set; }
    public string    CreatedBy { get; }

    public MedicalReport(string title, string content, string doctor)
    {
        Title = title; Content = content; CreatedBy = doctor;
    }

    public void Save()
    {
        Console.WriteLine($"[Report] Збережено: '{Title}'");
    }

    public void Send(string recipient)
    {
        Console.WriteLine($"[Report] '{Title}' → {recipient}");
    }
}

// ── PatientRecord: зберігається, архівується, має аудит ──────────
class PatientRecord : ISaveable, IArchivable, IAuditable
{
    public int    PatientId { get; }
    public string Content   { get; }
    public DateTime CreatedAt  { get; } = DateTime.Now;
    public DateTime? UpdatedAt { get; private set; }
    public string    CreatedBy { get; }
    public bool      IsArchived{ get; private set; }

    public PatientRecord(int patientId, string content, string doctor)
    {
        PatientId = patientId; Content = content; CreatedBy = doctor;
    }

    public void Save()
    {
        UpdatedAt = DateTime.Now;
        Console.WriteLine($"[Record] Пацієнт #{PatientId.ToString()}
збережено");
    }

    public void Archive(string reason)
    {
        IsArchived = true;
        UpdatedAt  = DateTime.Now;
        Console.WriteLine($"[Record] Пацієнт #{PatientId.ToString()}
архівовано: {reason}");
    }
}
```

Коли ISP не потрібен

Варто розуміти: ISP — не заклик дробити кожен інтерфейс до одного методу. Інтерфейс IComparable<T> з одним методом CompareTo —

правильний приклад вузького інтерфейсу. Але IEnumerable<T> з одним GetEnumerator — теж правильний. Розмір інтерфейсу не є самоціллю.

Інтерфейс потрібно розбивати тоді, коли **різні клієнти використовують різні підмножини методів**. Якщо всі клієнти використовують всі методи — об'єднаний інтерфейс абсолютно правильний. Критерій: «Чи є хоч один клієнт, що змушений залежати від методу, який йому не потрібен?» — якщо так, ISP застосовується.

```C#
using System;
using System.Collections.Generic;

var dashboard = new PatientDashboard(new LabSystemAdapter());
dashboard.ShowLabResults(101, new[] { "CBC", "BMP", "HbA1c" });

// ─── Правильне застосування ISP: адаптер для зовнішньої системи ────

// Зовнішня система лабораторних аналізів має великий API
// Ми використовуємо лише маленьку частину — виразимо це через вузький інтерфейс

interface ILabResultProvider
{
    // Тільки те, що потрібно клінічній системі
    string GetResult(string testCode, int patientId);
    bool   IsResultReady(string testCode, int patientId);
    DateTime GetResultDate(string testCode, int patientId);
}

// Наш адаптер реалізує вузький інтерфейс і «перекладає» до зовнішньої системи
class LabSystemAdapter : ILabResultProvider
{
    // Уявна зовнішня система з великим API — ми беремо лише 3 методи
    public string   GetResult(string code, int patientId)
    {
        // Реально: HTTP-запит до лабораторної системи
        return $"[Лаб. система] {code} для #{patientId.ToString()}: в нормі";
    }
    public bool     IsResultReady(string code, int patientId)   => true;
    public DateTime GetResultDate(string code, int patientId)   => DateTime.Today;
}

// Клінічна система залежить від мінімального ILabResultProvider
class PatientDashboard
{
    private readonly ILabResultProvider _lab;

    public PatientDashboard(ILabResultProvider lab) => _lab = lab;

    public void ShowLabResults(int patientId, string[] testCodes)
    {
        Console.WriteLine($"\nЛабораторні результати пацієнта #{patientId.ToString()}:");
        foreach (var code in testCodes)
        {
```

```
45              if (_lab.IsResultReady(code, patientId))
46              {
47                  string result = _lab.GetResult(code, patientId);
48                  DateTime date = _lab.GetResultDate(code, patientId);
49                  Console.WriteLine($"  [{code}] {date:dd.MM.yyyy}: {result}");
50              }
51              else
52              {
53                  Console.WriteLine($"  [{code}] Ще не готово");
54              }
55          }
56      }
57  }
```

Клас PatientDashboard залежить від трьох методів ILabResultProvider, а не від усього великого API лабораторної системи. Якщо лабораторна система повністю замінюється — пишеться новий адаптер, PatientDashboard не змінюється.

20.6. DIP — Принцип інверсії залежностей

Dependency Inversion Principle (DIP) — п'ятий і в певному сенсі найвагоміший принцип SOLID, бо він підсумовує і об'єднує всі попередні. Сформулював його Роберт Мартін у 1994–1996 роках. Принцип складається з двох частин:

1. **Модулі верхнього рівня не повинні залежати від модулів нижнього рівня. Обидва мають залежати від абстракцій.**
2. **Абстракції не повинні залежати від деталей. Деталі мають залежати від абстракцій.**

Щоб зрозуміти ці слова конкретно: «модуль верхнього рівня» — це бізнес-логіка (AppointmentService, PatientService), а «модуль нижнього рівня» — це інфраструктура (SqlRepository, SmtpNotifier, FileLogger). «Деталь» — це конкретна реалізація (SQL, SMTP, CSV). «Абстракція» — це інтерфейс або абстрактний клас, що описує лише що треба зробити, але не як.

Без DIP бізнес-логіка знає про конкретну базу даних, конкретний поштовий шлюз, конкретний файловий формат. Зміна будь-якого з них вимагає змін у бізнес-логіці. Тест бізнес-логіки потребує реального SQL-сервера. Це й є «неправильний напрямок залежностей» — бізнес залежить від деталей.

Інверсія — не означає «перевернути все з ніг на голову». Вона означає: хто диктує умови? Без DIP — деталь (SQL) диктує умови бізнес-логіці. З DIP — бізнес-логіка диктує умови (через інтерфейс), а деталь їх виконує.

Пряма залежність: проблема

```
C#
using System;
using System.Collections.Generic;

var svc = new AppointmentService_Bad();
svc.Book("Мороз Василь",  "Петренко І.О.", DateTime.Today.AddHours(9));
svc.Book("Бойко Олена",   "Петренко І.О.", DateTime.Today.AddHours(10));

// Проблеми:
// ✗ Написати тест без реального SQL-сервера неможливо
// ✗ Перейти на MongoDB = переписати AppointmentService_Bad
// ✗ Використати InMemory-репозиторій для розробки — неможливо
Console.WriteLine("\n⚠ AppointmentService залежить від
SqlAppointmentRepository безпосередньо");

// ─── ПОРУШЕННЯ DIP: AppointmentService знає про SqlRepository ──────

// Конкретна реалізація — нижній рівень
class SqlAppointmentRepository
{
    private List<string> _db = new(); // імітація таблиці в БД

    public void Save(string patient, string doctor, DateTime time)
    {
        string record = $"{patient}|{doctor}|{time:yyyy-MM-dd HH:mm}";
        _db.Add(record);
        Console.WriteLine($"[SQL] INSERT: {record}");
    }

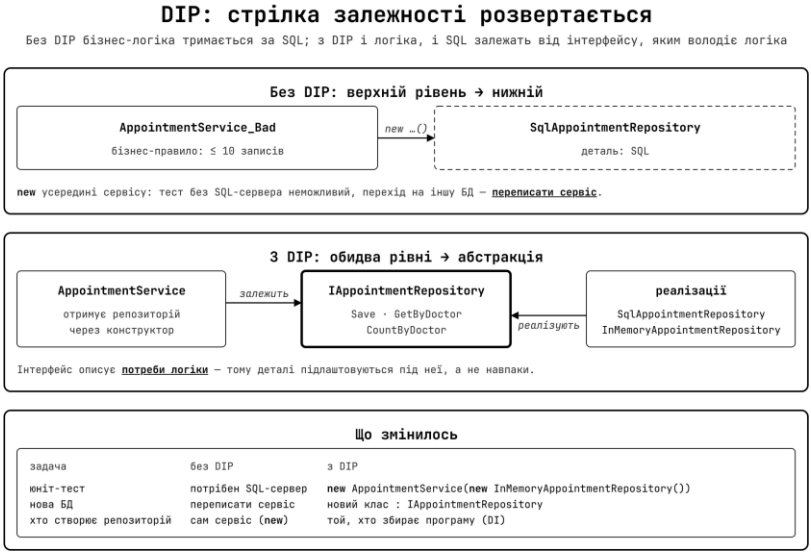
    public List<string> GetByDoctor(string doctor)
    {
        var result = _db.FindAll(r => r.Contains($"|{doctor}|"));
        Console.WriteLine($"[SQL] SELECT WHERE doctor='{doctor}':
{result.Count.ToString()} rows");
        return result;
    }
}

// Бізнес-логіка — верхній рівень
// ⚠ Залежить безпосередньо від SqlAppointmentRepository
class AppointmentService_Bad
{
    // Жорстка залежність: тут «зашита» конкретна реалізація
    private readonly SqlAppointmentRepository _repository = new
SqlAppointmentRepository();

    public void Book(string patient, string doctor, DateTime time)
    {
        // Бізнес-правило: лікар не може приймати більш ніж 10 пацієнтів на
день
        var existingToday = _repository.GetByDoctor(doctor);
        if (existingToday.Count >= 10)
        {
            Console.WriteLine($"[Service] {doctor}: максимум записів на
сьогодні");
            return;
```

```
        }

        _repository.Save(patient, doctor, time);
        Console.WriteLine($"[Service] Заброньовано: {patient} → {doctor}");
    }
}
```



DIP — інверсія напряму залежностей через абстракцію

Рефакторинг: вводимо абстракцію

Рішення — виразити потреби бізнес-логіки через **інтерфейс**. Бізнес-логіка тепер залежить від абстракції IAppointmentRepository, а не від конкретного SqlAppointmentRepository. Конкретна реалізація «вводиться» ззовні — через конструктор (Dependency Injection).

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3
 4  // ── Виробничий варіант: використовуємо SQL ──────────────────────
 5  Console.WriteLine("=== Виробничий варіант (SQL) ===");
 6  var sqlRepo      = new SqlAppointmentRepository();
 7  var prodService  = new AppointmentService(sqlRepo); // SQL "вводиться" ззовні
 8
 9  prodService.Book("Мороз Василь",  "Петренко І.О.",
    DateTime.Today.AddHours(9));
```

```
prodService.Book("Бойко Олена",   "Петренко І.О.",
DateTime.Today.AddHours(11));
prodService.PrintDoctorSchedule("Петренко І.О.");

// ── Тест / розробка: InMemory — ТОЙ САМИЙ AppointmentService ────────
Console.WriteLine("\n=== Тестовий варіант (InMemory) ===");
var memRepo     = new InMemoryAppointmentRepository();
var testService = new AppointmentService(memRepo); // InMemory "вводиться"
ззовні

testService.Book("Тест Пацієнт",  "Тест Лікар", DateTime.Today.AddHours(14));
testService.Book("Тест Пацієнт2", "Тест Лікар", DateTime.Today.AddHours(15));
testService.PrintDoctorSchedule("Тест Лікар");

// AppointmentService НЕ змінювався між варіантами — лише репозиторій змінився
Console.WriteLine("\n✓ AppointmentService не знає, SQL він чи InMemory");

// ─── ПРАВИЛЬНО: DIP через інтерфейс і Dependency Injection ──────────

// Абстракція — диктує умови, що мають виконати конкретні реалізації
interface IAppointmentRepository
{
    void         Save(string patient, string doctor, DateTime time);
    List<string> GetByDoctor(string doctor);
    int          CountByDoctor(string doctor);
}

// ── Конкретна реалізація 1: SQL ──────────────────────────────────────
class SqlAppointmentRepository : IAppointmentRepository
{
    private List<string> _db = new();

    public void Save(string patient, string doctor, DateTime time)
    {
        _db.Add($"{patient}|{doctor}|{time:HH:mm}");
        Console.WriteLine($"[SQL] Збережено: {patient} → {doctor}");
    }

    public List<string> GetByDoctor(string doctor) =>
        _db.FindAll(r => r.Contains($"|{doctor}|"));

    public int CountByDoctor(string doctor) => GetByDoctor(doctor).Count;
}

// ── Конкретна реалізація 2: In-Memory (для тестів і розробки) ────────
class InMemoryAppointmentRepository : IAppointmentRepository
{
    private List<(string Patient, string Doctor, DateTime Time)> _store =
new();

    public void Save(string patient, string doctor, DateTime time)
    {
        _store.Add((patient, doctor, time));
        Console.WriteLine($"[Memory] Збережено: {patient} → {doctor}");
    }

    public List<string> GetByDoctor(string doctor) =>
        _store.FindAll(a => a.Doctor == doctor)
              .ConvertAll(a => $"{a.Patient}|{a.Doctor}|{a.Time:HH:mm}");

```

```
    public int CountByDoctor(string doctor) =>
        _store.FindAll(a => a.Doctor == doctor).Count;
}

// ── Бізнес-логіка: залежить ТІЛЬКИ від абстракції ──────────────────
class AppointmentService
{
    private readonly IAppointmentRepository _repo;

    // Dependency Injection через конструктор
    public AppointmentService(IAppointmentRepository repo)
    {
        _repo = repo;
    }

    public bool Book(string patient, string doctor, DateTime time)
    {
        // Бізнес-правило: не більше 10 записів на день
        if (_repo.CountByDoctor(doctor) >= 10)
        {
            Console.WriteLine($"[Service] {doctor}: максимум записів
вичерпано");
            return false;
        }

        _repo.Save(patient, doctor, time);
        Console.WriteLine($"[Service] ✓ Заброньовано: {patient} → {doctor} о
{time:HH:mm}");
        return true;
    }

    public void PrintDoctorSchedule(string doctor)
    {
        var appointments = _repo.GetByDoctor(doctor);
        Console.WriteLine($"\nРозклад {doctor}
({appointments.Count.ToString()} записів):");
        foreach (var a in appointments)
        {
            string[] parts = a.Split('|');
            Console.WriteLine($"  {(parts.Length > 2 ? parts[2] : "?")} —
{(parts.Length > 0 ? parts[0] : "?")}");
        }
    }
}
```

Dependency Injection: три способи введення залежностей

DIP визначає принцип («залежте від абстракцій»), а **Dependency Injection (DI)** — механізм його реалізації. DI — це патерн, що описує, як саме залежність «вводиться» (inject) у клас ззовні. Є три основних способи:

```
C#
using System;
using System.Collections.Generic;

```

```
// ── Демонстрація трьох способів ──────────────────────────────
Console.WriteLine("=== Constructor Injection ===");
var svc1 = new PatientServiceV1(new ConsoleLogger());
svc1.Register("Мороз Василь");

Console.WriteLine("\n=== Property Injection ===");
var svc2 = new PatientServiceV2();
svc2.Logger = new FileLogger("clinic.log");
svc2.Register("Бойко Олена");

var svc2b = new PatientServiceV2(); // Logger = null — все одно працює
svc2b.Register("Коваль Михайло");

Console.WriteLine("\n=== Method Injection ===");
var svc3 = new PatientServiceV3();
svc3.Register("Сидоренко Тетяна", new ConsoleLogger());
svc3.Register("Петрів Іван",      new FileLogger("audit.log"));

interface IPatientLogger
{
    void Log(string message);
}

class ConsoleLogger : IPatientLogger
{
    public void Log(string msg) => Console.WriteLine($"[LOG] {msg}");
}

class FileLogger : IPatientLogger
{
    private string _path;
    public FileLogger(string path) => _path = path;
    public void Log(string msg) => Console.WriteLine($"[FILE:{_path}] {msg}");
}

// ── Спосіб 1: Constructor Injection ──────────────────────────
// Рекомендований. Залежність обов'язкова, очевидна, незмінна після створення.
class PatientServiceV1
{
    private readonly IPatientLogger _logger;

    public PatientServiceV1(IPatientLogger logger) // ← залежність через
конструктор
    {
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    public void Register(string name)
    {
        _logger.Log($"Зареєстровано: {name}");
        Console.WriteLine($"[Service] Пацієнт додано: {name}");
    }
}

// ── Спосіб 2: Property Injection ─────────────────────────────
// Для необов'язкових залежностей. Клас може працювати і без них.
class PatientServiceV2
{
    public IPatientLogger? Logger { get; set; } // ← залежність через
властивість
```

```
62
63      public void Register(string name)
64      {
65          Logger?.Log($"Зареєстровано: {name}"); // null-safe: якщо немає —
    просто не логуємо
66          Console.WriteLine($"[Service] Пацієнт: {name}");
67      }
68  }
69
70  // ── Спосіб 3: Method Injection ──────────────────────────────────
71  // Якщо залежність потрібна лише в одному методі, не для всього класу.
72  class PatientServiceV3
73  {
74      public void Register(string name, IPatientLogger logger) // ← залежність
    через метод
75      {
76          logger.Log($"Зареєстровано: {name}");
77          Console.WriteLine($"[Service] Пацієнт: {name}");
78      }
79  }
```

Constructor Injection є загальновизнаним стандартом для обов'язкових залежностей: клас не може існувати без своїх залежностей, тому вони передаються в конструкторі. Property Injection і Method Injection вживаються значно рідше — лише тоді, коли залежність справді необов'язкова або специфічна для одного виклику.

DI-контейнер: автоматичне керування залежностями

У великих системах ланцюжки залежностей стають довгими: AppointmentService → IAppointmentRepository → IDbConnection → ... Вручну конструювати всю ієрархію об'єктів у Program.cs нереально. Для цього існують **DI-контейнери** — бібліотеки, що автоматично вирішують, яку реалізацію підставити для кожного інтерфейсу.

У .NET стандартний DI-контейнер — Microsoft.Extensions.DependencyInjection. Він реєструє «прив'язки» (interface → implementation) і сам конструює об'єкти з потрібними залежностями:

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3
 4  // ── Налаштування контейнера ──────────────────────────────────
 5  var container = new SimpleContainer();
 6  container.Register<IDoctorRepository,  SqlDoctorRepository>();
 7  container.Register<IPatientRepository, SqlPatientRepository>();
 8
 9  // Resolve: отримуємо залежності через контейнер
10  var doctorRepo  = container.Resolve<IDoctorRepository>();
11  var patientRepo = container.Resolve<IPatientRepository>();
```

```
12
13  // Конструюємо сервіс, передаючи залежності
14  var bookingService = new AppointmentBookingService(doctorRepo, patientRepo);
15  bookingService.Book(101, 5, DateTime.Today.AddHours(14));
16
17  // ─── Хочемо протестувати? Замінюємо реалізації ──────────────────────
18  Console.WriteLine("\n=== Тестовий варіант: InMemory ===");
19
20  // Той самий AppointmentBookingService — інші реалізації
21  var testService = new AppointmentBookingService(
22      new TestDoctorRepository(),
23      new TestPatientRepository()
24  );
25  testService.Book(999, 1, DateTime.Today.AddHours(10));
26  Console.WriteLine("✓ AppointmentBookingService не змінювався");
27
28  // ─── Симуляція DI-контейнера (спрощена версія) ──────────────────────
29  // У реальних проектах: Microsoft.Extensions.DependencyInjection
30
31  class SimpleContainer
32  {
33      private Dictionary<Type, Func<object>> _registrations = new();
34
35      public void Register<TInterface, TImplementation>()
36          where TImplementation : TInterface, new()
37      {
38          _registrations[typeof(TInterface)] = () => new TImplementation();
39      }
40
41      public void RegisterFactory<TInterface>(Func<object> factory)
42      {
43          _registrations[typeof(TInterface)] = factory;
44      }
45
46      public T Resolve<T>() =>
47          (T)(_registrations.TryGetValue(typeof(T), out var factory)
48              ? factory()
49              : throw new InvalidOperationException($"Не зареєстровано:
    {typeof(T).Name}"));
50  }
51
52  // ─── Інтерфейси і реалізації ────────────────────────────────────────
53  interface IDoctorRepository
54  {
55      string GetById(int id);
56  }
57
58  interface IPatientRepository
59  {
60      string GetById(int id);
61  }
62
63  class SqlDoctorRepository : IDoctorRepository
64  {
65      public string GetById(int id) => $"[SQL] Лікар #{id.ToString()}: Петренко
    І.О.";
66  }
67
68  class SqlPatientRepository : IPatientRepository
69  {
```

```
70      public string GetById(int id) => $"[SQL] Пацієнт #{id.ToString()}: Мороз
    Василь";
71  }
72
73  class AppointmentBookingService
74  {
75      private readonly IDoctorRepository  _doctors;
76      private readonly IPatientRepository _patients;
77
78      public AppointmentBookingService(IDoctorRepository doctors,
    IPatientRepository patients)
79      {
80          _doctors  = doctors;
81          _patients = patients;
82      }
83
84      public void Book(int patientId, int doctorId, DateTime time)
85      {
86          string patient = _patients.GetById(patientId);
87          string doctor  = _doctors.GetById(doctorId);
88          Console.WriteLine($"Запис: {patient} → {doctor} о {time:HH:mm}");
89      }
90  }
91
92  class TestDoctorRepository : IDoctorRepository
93  {
94      public string GetById(int id) => $"[Test] Тест-Лікар #{id.ToString()}";
95  }
96
97  class TestPatientRepository : IPatientRepository
98  {
99      public string GetById(int id) => $"[Test] Тест-Пацієнт #{id.ToString()}";
100 }
```

Варто чесно уточнити, що саме показує цей приклад. SimpleContainer — навчальна модель: він лише зіставляє інтерфейс із фабрикою (() => new TImplementation()), тому реалізації мусять мати конструктор без параметрів (обмеження new()), а AppointmentBookingService ми все одно конструюємо вручну. Справжній контейнер Microsoft.Extensions.DependencyInjection робить головне, чого тут немає, — **автоматичне впровадження через конструктор**: при запиті сервісу він дивиться на параметри його конструктора, рекурсивно створює кожну залежність і передає її. Окрім того, він керує **часом життя** об'єктів (AddTransient — новий екземпляр щоразу, AddScoped — один на область, наприклад HTTP-запит, AddSingleton — один на весь застосунок) і викликає Dispose для створених ним об'єктів. У реальному проєкті той самий приклад виглядав би так:

```
C#
1  var services = new ServiceCollection();
2  services.AddScoped<IDoctorRepository, SqlDoctorRepository>();
```

```
services.AddScoped<IPatientRepository, SqlPatientRepository>();
services.AddScoped<AppointmentBookingService>();   // залежності знайде сам

using var provider = services.BuildServiceProvider();
var booking = provider.GetRequiredService<AppointmentBookingService>();
```

Детально контейнер, реєстрація сервісів і часи життя розглядаються в розділі 21 «Generic Host».

DIP і тестованість: підроблені залежності

Найпрактичніша перевага DIP — **можливість підміняти залежності у тестах**. Якщо клас залежить від IAppointmentRepository, а не від SqlAppointmentRepository, тест може передати «підроблений» репозиторій, що повертає заздалегідь задані дані:

```C#
using System;
using System.Collections.Generic;

// ─── Симуляція тестів — без БД, без файлів, без мережі ──────────
void Test(string name, string expected, string actual)
{
    bool pass = expected == actual;
    Console.WriteLine($"{(pass ? "✓" : "✗")} {name}");
    if (!pass) Console.WriteLine($"    Очікувалось: {expected}\n    Отримано: {actual}");
}

// Тест 1: успішний запис
var store1 = new FakeAppointmentStore();
var svc1   = new SchedulingService(store1);
Test("Успішний запис",
     "OK: Мороз Василь → Петренко І.О.",
     svc1.Schedule("Мороз Василь", "Петренко І.О."));

// Тест 2: дублікат
var store2 = new FakeAppointmentStore();
store2.Preload("Бойко Олена", "Коваленко М.А."); // налаштовуємо початковий стан
var svc2   = new SchedulingService(store2);
Test("Дублікат запису",
     "DUPLICATE: Бойко Олена вже записаний до Коваленко М.А.",
     svc2.Schedule("Бойко Олена", "Коваленко М.А."));

// Тест 3: лікар переповнений
var store3 = new FakeAppointmentStore();
store3.Preload("P1", "Бойко О.Р.");
store3.Preload("P2", "Бойко О.Р.");
store3.Preload("P3", "Бойко О.Р."); // 3 записи — максимум
var svc3   = new SchedulingService(store3);
Test("Переповнений лікар",
     "FULL: Бойко О.Р. має максимальну кількість записів (3)",
     svc3.Schedule("P4", "Бойко О.Р."));

```

```
// ── DIP + тестові дублі ──────────────────────────────────────────

interface IAppointmentStore
{
    void   Add(string patient, string doctor);
    int    Count(string doctor);
    bool   Exists(string patient, string doctor);
}

// Бізнес-логіка: перевіряє обмеження і зберігає
class SchedulingService
{
    private readonly IAppointmentStore _store;
    private const int MaxPerDoctor = 3; // для демонстрації — маленький ліміт

    public SchedulingService(IAppointmentStore store) => _store = store;

    public string Schedule(string patient, string doctor)
    {
        if (_store.Exists(patient, doctor))
            return $"DUPLICATE: {patient} вже записаний до {doctor}";

        if (_store.Count(doctor) >= MaxPerDoctor)
            return $"FULL: {doctor} має максимальну кількість записів
({MaxPerDoctor.ToString()})";

        _store.Add(patient, doctor);
        return $"OK: {patient} → {doctor}";
    }
}

// ── Тестовий дубль (Stub/Fake) ──────────────────────────────────
class FakeAppointmentStore : IAppointmentStore
{
    private List<(string Patient, string Doctor)> _data = new();

    // Можна налаштовувати для різних тестових сценаріїв
    public void Preload(string patient, string doctor) => _data.Add((patient,
doctor));

    public void   Add(string patient, string doctor) => _data.Add((patient,
doctor));
    public int    Count(string doctor) => _data.FindAll(x => x.Doctor ==
doctor).Count;
    public bool   Exists(string patient, string doctor) =>
        _data.Exists(x => x.Patient == patient && x.Doctor == doctor);
}
```

Три тести — жодного SQL, жодного файлу, жодної мережі. Кожен тест налаштовує FakeAppointmentStore у потрібний стан і перевіряє лише бізнес-логіку SchedulingService. Це і є головна практична цінність DIP.

Підсумок розділу 20: SOLID як єдина система

Хоча кожен принцип можна вивчати окремо, вони утворюють єдину архітектурну систему:

```
C#
using System;

Console.WriteLine("=== Розділ 20: SOLID Принципи ===\n");

Console.WriteLine("20.1 Вступ до SOLID:");
Console.WriteLine("  God Class       — клас із кількома відповідальностями");
Console.WriteLine("  Жорсткість      — будь-яка зміна тягне каскад");
Console.WriteLine("  Крихкість       — зміна в одному місці ламає інше");
Console.WriteLine("  Технічний борг  — ціна «швидких» рішень накопичується");
Console.WriteLine();

Console.WriteLine("20.2 SRP — Single Responsibility Principle:");
Console.WriteLine("  Один клас      → одна причина для зміни → один актор");
Console.WriteLine("  Рефакторинг    → AppointmentService + Repository + Notifier");
Console.WriteLine("  Перевага       → кожен клас тестується ізольовано");
Console.WriteLine();

Console.WriteLine("20.3 OCP — Open/Closed Principle:");
Console.WriteLine("  Відкрито для розширення  → новий клас, що реалізує інтерфейс");
Console.WriteLine("  Закрито для модифікації  → існуючий код не торкається");
Console.WriteLine("  Інструменти    → interface, abstract class, Strategy, Template Method");
Console.WriteLine();

Console.WriteLine("20.4 LSP — Liskov Substitution Principle:");
Console.WriteLine("  Підклас замінює базовий  → без порушення поведінки");
Console.WriteLine("  Контракт       → передумови, постумови, інваріанти");
Console.WriteLine("  Ознаки порушення → throw у override, if (x is SubType)");
Console.WriteLine();

Console.WriteLine("20.5 ISP — Interface Segregation Principle:");
Console.WriteLine("  Вузькі інтерфейси        → клієнт бачить лише те, що йому потрібно");
Console.WriteLine("  Жирний інтерфейс          → вимушені заглушки у реалізаторів");
Console.WriteLine("  Інтерфейс = роль          → ISaveable, IArchivable, ISendable");
Console.WriteLine();

Console.WriteLine("20.6 DIP — Dependency Inversion Principle:");
Console.WriteLine("  Верхній рівень → залежить від абстракції (interface)");
Console.WriteLine("  Нижній рівень  → реалізує абстракцію (конкретний клас)");
Console.WriteLine("  DI             → залежність вводиться ззовні (конструктор)");
Console.WriteLine("  Перевага       → тести без БД, заміна реалізації без змін у логіці");
Console.WriteLine();

Console.WriteLine("Зв'язок принципів:");
Console.WriteLine("  SRP дає маленькі класи → їх легше ізолювати через DIP");
```

```
45  Console.WriteLine("  OCP використовує абстракції → що описані через ISP");
46  Console.WriteLine("  LSP гарантує → що підстановки у OCP не зламають систему");
47  Console.WriteLine("  DIP зв'язує все → через інтерфейси (ISP) між рівнями
    (SRP)");
```

Питання для самоконтролю — Розділ 20. Принципи SOLID

20.1. Вступ до SOLID

1. Назвіть 4 симптоми поганого дизайну (за Робертом Мартіном) та поясніть кожен. Чому "жорсткість" (Rigidity) та "крихкість" (Fragility) часто виникають разом?
2. Розгляньте клас:

```
C#
1  public class ClinicManager
2  {
3      public void AddPatient(Patient p) { ... }
4      public void ScheduleAppointment(int patientId, DateTime dt) { ... }
5      public void SendReminderEmail(int appointmentId) { ... }
6      public void GeneratePdfReport(List<Appointment> list) { ... }
7      public void SaveToDatabase(Patient p) { ... }
8      public void ValidateInsurance(Patient p) { ... }
9  }
```

Скільки "відповідальностей" у цього класу? Перелічіть їх. Як називається цей антипаттерн?

3. Що таке технічний борг? Наведіть аналогію з реального фінансового боргу. Як накопичення технічного боргу впливає на здатність команди додавати нові функції?
4. Поясніть, як пов'язані між собою принципи SOLID. Чому порушення одного принципу часто означає порушення іншого? Наведіть приклад ланцюгової реакції.
5. Що таке "Immobility" (нерухомість) коду? Чому її важко помітити під час розробки? Наведіть конкретний приклад, де код у одному модулі неможливо перевикористати в іншому через погані залежності.
6. Чому існування класів з суфіксами Manager, Helper, Processor, Utils часто є сигналом проблеми дизайну? В яких випадках такі назви все ж є прийнятними?

20.2. Single Responsibility Principle

7. SRP стверджує: "клас має мати одну причину для змін". Що означає "причина для змін" і чому вона пов'язана з поняттям "актор"? Чим це відрізняється від "клас має мати лише одну функцію"?
8. Розгляньте клас:

```C#
public class AppointmentManager
{
    public void Book(Appointment a) { /* записати в БД */ }
    public void Cancel(int id) { /* видалити з БД */ }
    public void SendConfirmation(Appointment a) { /* відправити email */ }
    public List<Appointment> GetReport(DateTime from, DateTime to) { /* LINQ */
}
}
```

Визначте акторів, що зацікавлені в цьому класі. Запропонуйте рефакторинг з розподілом по відповідних класах.

9. Студент каже: "Я розбив клас на 5 маленьких — тепер у кожного по одному методу. Це SRP?" Оцініть цю думку. Де межа між правильним розбиттям і надмірною фрагментацією?
10. Клас Patient містить:

```C#
public string Name { get; set; }
public DateTime BirthDate { get; set; }
public int CalculateAge() { ... }
public bool IsAdult() { ... }
```

Чи порушує цей клас SRP? Обґрунтуйте відповідь.

11. Як SRP полегшує написання юніт-тестів? Наведіть конкретний приклад: клас, що порушує SRP, і як це ускладнює тестування; після рефакторингу — як тести спрощуються.
12. Назвіть 4 ознаки порушення SRP у коді. Для кожної наведіть приклад в контексті медичної інформаційної системи.
13. Реалізуйте рефакторинг класу ReportManager, що зараз містить: завантаження даних з БД, форматування звіту, відправку на email і збереження PDF. Розбийте на окремі класи з чіткими відповідальностями та покажіть, як вони взаємодіють через оркестратор.

20.3. Open/Closed Principle

14. Сформулюйте OCP. Що означає "відкритий для розширення, закритий для модифікації"? Чому обидві частини важливі одночасно?
15. Розгляньте код:

```C#
public decimal CalculateFee(Appointment a)
{
    return a.Type switch
    {
        "General" => 100m,
        "Specialist" => 250m,
        "Emergency" => 500m,
        _ => throw new ArgumentException("Unknown type")
    };
}
```

Яка проблема виникне при додаванні нового типу прийому "Telemedicine"? Як переписати цей код, щоб він відповідав OCP?

16. Поясніть, як паттерн Стратегія (Strategy) реалізує OCP. Намалюйте (текстово) UML-подібну структуру для системи знижок: різні стратегії знижок без зміни класу BillingService.
17. Чим паттерн Template Method відрізняється від Strategy у контексті OCP? Наведіть медичний приклад, де Template Method є більш природним вибором.
18. Студент каже: "OCP вимагає робити все через інтерфейс. Я завжди пишу IService перед реалізацією." Оцініть це твердження. Що таке "правило трьох" і як воно обмежує надмірне використання OCP?
19. Де OCP найбільш важливий: на початку проєкту чи коли вже є стабільна кодова база? Поясніть зв'язок між OCP і принципом YAGNI (You Aren't Gonna Need It).
20. Реалізуйте систему нотифікацій, що відповідає OCP: базовий клас/інтерфейс INotificationChannel та реалізації EmailNotification, SmsNotification, PushNotification. Покажіть, як NotificationService відправляє через всі канали без if/switch по типу.

20.4. Liskov Substitution Principle

21. Сформулюйте LSP своїми словами. Чим "підстановна сумісність" відрізняється від звичайної "типової сумісності" в

ООП? Наведіть приклад типу, що успадковується, але порушує LSP.

22. Розгляньте класичний приклад:

```C#
public class Rectangle
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }
    public int Area() => Width * Height;
}
public class Square : Rectangle
{
    public override int Width { set { base.Width = value; base.Height = value; } }
    public override int Height { set { base.Width = value; base.Height = value; } }
}
```

Напишіть тест, що проходить для Rectangle, але падає для Square. Чому це порушення LSP?

23. Що таке "контракт" класу в контексті LSP? Поясніть три складові контракту: передумови (preconditions), постумови (postconditions) та інваріанти. Як підклас може (і не може) їх змінювати?

24. Розгляньте код:

```C#
public class ReadOnlyRecord : MedicalRecord
{
    public override void Update(string data)
        => throw new NotSupportedException("Record is read-only");
}
```

Поясніть, чому це порушення LSP, навіть якщо компілятор не скаржиться. Як правильно вирішити цю проблему через розбиття інтерфейсів?

25. Як перевірити, чи дотримується LSP у системі? Опишіть правило "тест базового класу повинен проходити для підкласів". Напишіть клас AppointmentTests з тестом, що може перевірити будь-яку реалізацію IScheduler.

26. Назвіть 4 типові ознаки порушення LSP у коді. Що сигналізує про проблему: is, as, NotSupportedException, перевірка типу в клієнтському коді?

27. Студент пропонує: "Нехай ArchiveRecord успадковується від MedicalRecord, але перевизначає Delete() як порожній метод —

нічого не робить, але й не падає." Чи це виправляє порушення LSP? Обґрунтуйте відповідь.

28. Дизайн задача: є ієрархія Payment → CashPayment, CardPayment, InsurancePayment. Клас InsurancePayment вимагає попередньої авторизації перед Process(). Як спроєктувати ієрархію, щоб не порушувати LSP?

20.5. Interface Segregation Principle

29. Сформулюйте ISP. Чому "жирний" інтерфейс є проблемою? Поясніть термін "залежність від непотрібних методів" та її наслідки.
30. Розгляньте інтерфейс:

```C#
public interface IClinicService
{
    void RegisterPatient(Patient p);
    Appointment BookAppointment(int patientId);
    void CancelAppointment(int id);
    List<Patient> GetAllPatients();
    Report GenerateMonthlyReport(DateTime month);
    void ExportToExcel(Report report);
    void SendBulkEmail(string message);
}
```

Визначте "ролі" в цьому інтерфейсі та запропонуйте розбиття на менші інтерфейси.

31. Студент каже: "Я реалізую IClinicService в класі ClinicFacade — він все одно все реалізовує, тому ISP не потрібен." Що не так у цьому міркуванні?
32. Поясніть концепцію "інтерфейс як роль". Наведіть приклади ролей ISaveable, IAuditable, ISendable. Як один клас може реалізовувати кілька ролей?
33. Розгляньте сценарій: є зовнішня бібліотека LegacyPatientSystem з методом GetAllData(), що повертає все одразу. Ваш код очікує IPatientReader з методом GetPatient(int id). Як адаптер вирішує цю проблему без порушення ISP?
34. Чи суперечить ISP принципу "єдиного місця для зміни"? Тобто: якщо я розбив на 5 інтерфейсів, чи не стане важче вносити зміни? Аргументуйте.
35. Коли ISP не потрібен? Назвіть ситуації, де один великий інтерфейс є кращим вибором ніж кілька дрібних.

20.6. Dependency Inversion Principle

36. DIP складається з двох частин. Сформулюйте обидві. Чому він називається "Inversion" (інверсія)? Що саме інвертується порівняно з "природнім" способом написання коду?
37. Розгляньте код:

```
C#
1  public class AppointmentService
2  {
3      private readonly SqlAppointmentRepository _repo = new
   SqlAppointmentRepository();
4      public void Book(Appointment a) => _repo.Save(a);
5  }
```

Назвіть всі проблеми цього дизайну. Перепишіть код відповідно до DIP.

38. Поясніть три методи Dependency Injection: через конструктор, через властивість, через метод. Для кожного наведіть приклад коду та вкажіть, коли він доцільний.
39. Розгляньте:

```
C#
1  public class ReportService
2  {
3      private IEmailSender _emailSender;
4      public void SetEmailSender(IEmailSender sender) { _emailSender = sender; }
5      public void Send(Report r) { _emailSender?.Send(r.ToString()); }
6  }
```

Який метод DI тут використовується? Яка ризик, що _emailSender може бути null під час виклику Send? Чи є Constructor Injection кращим варіантом?

40. Як DIP пов'язаний з тестованістю? Напишіть FakeAppointmentRepository : IAppointmentRepository, що зберігає дані в пам'яті, та покажіть юніт-тест для AppointmentService без підключення до бази даних.
41. Що таке IoC Container (контейнер інверсії залежностей)? Реалізуйте спрощений SimpleContainer з методами Register<TInterface, TImplementation>() та Resolve<T>(). Поясніть, як він знаходить потрібну реалізацію.
42. Поясніть, як SOLID як система підсилює себе: як SRP веде до менших класів → що спрощує DIP → що дозволяє реалізовувати OCP → що безпечно завдяки LSP → що стає можливим через

ISP. Наведіть конкретний приклад системи, де всі 5 принципів працюють разом.

43. "DIP означає, що треба скрізь писати інтерфейси" — правда чи ні? Де DIP є обов'язковим, а де надмірним? Наведіть 2 приклади кожного варіанту.

Розділ 21. Generic Host та Dependency Injection

21.1. Generic Host — архітектура та життєвий цикл додатку

Будь-який реальний додаток потребує більше, ніж просто виконання рядків коду зверху вниз. Йому потрібно запуститися, налаштуватись, підключитись до баз даних, логерів, черг повідомлень, дочекатись сигналу зупинки і правильно завершити роботу — зберегти стан, закрити з'єднання, завершити незакінчені операції. Якщо все це реалізовувати вручну у кожному проекті, код Main() перетвориться на величезний монолітний метод, а будь-яка зміна в порядку ініціалізації ризикує зламати всю систему.

Саме для вирішення цієї проблеми Microsoft у .NET Core 2.1 (2018) ввів концепцію **Generic Host** (Microsoft.Extensions.Hosting). Generic Host — це інфраструктурна оболонка, що бере на себе відповідальність за **управління життєвим циклом додатку**: запуск, виконання фонових сервісів, обробку сигналів завершення і коректний shutdown. До цього аналогічний механізм існував лише для ASP.NET Core (WebHost), але з появою Generic Host він став доступним для будь-якого типу додатків — консольних, Windows Services, worker services, мікросервісів.

Проблема: «розсипаний» Main без хоста

Розглянемо типову еволюцію консольного додатку для клінічної системи. Спочатку все здається простим:

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Threading;
 4
 5  // ── Проблема: Main без хоста ──────────────────────────────
 6  // Усе в одному місці. З кожним тижнем Main росте.
 7
 8  class DatabaseConnection
 9  {
10      public string ConnectionString { get; }
11      public DatabaseConnection(string cs) { ConnectionString = cs; }
12      public void Open()  { Console.WriteLine($"[DB] Відкрито:
    {ConnectionString}"); }
13      public void Close() { Console.WriteLine("[DB] З'єднання закрито"); }
14  }
```

```
15
16  class AppointmentService
17  {
18      private readonly DatabaseConnection _db;
19      private readonly bool _sendNotifications;
20
21      // AppointmentService сам створює залежності — порушення DIP
22      public AppointmentService()
23      {
24          _db = new DatabaseConnection("Server=prod;Database=clinic"); //
    hardcoded!
25          _sendNotifications = true;
26      }
27
28      public void BookAppointment(string patient, string doctor)
29      {
30          Console.WriteLine($"[Service] Бронювання: {patient} -> {doctor}");
31      }
32  }
33
34  class ReportService
35  {
36      // ReportService теж сам знає про рядок з'єднання
37      private readonly DatabaseConnection _db =
38          new DatabaseConnection("Server=prod;Database=clinic"); // дублювання!
39
40      public void GenerateMonthlyReport()
41      {
42          Console.WriteLine("[Report] Генерується місячний звіт...");
43      }
44  }
45
46  class Program
47  {
48      static void Main(string[] args)
49      {
50          // ─── Проблеми цього підходу ──────────────────────────────
51          // 1. Кожен сервіс сам створює свої залежності (немає DI)
52          // 2. Рядки конфігурації розкидані по коду
53          // 3. Немає єдиного місця для ініціалізації ресурсів
54          // 4. Немає обробки Ctrl+C / сигналів завершення
55          // 5. Не зрозумілий порядок shutdown (що закривати першим?)
56
57          Console.WriteLine("=== Запуск клінічної системи (БЕЗ хоста) ===");
58
59          var db = new DatabaseConnection("Server=prod;Database=clinic");
60          db.Open();
61
62          var appointmentService = new AppointmentService();
63          var reportService = new ReportService();
64
65          appointmentService.BookAppointment("Іваненко О.", "Лікар Петренко");
66          reportService.GenerateMonthlyReport();
67
68          // Немає обробки Ctrl+C — додаток просто завершиться або зависне
69          Console.WriteLine("Натисніть Enter для завершення...");
70          Console.ReadLine();
71
72          db.Close();
73          // А якщо між db.Open() та db.Close() трапиться виняток?
```

```
        // З'єднання залишиться відкритим назавжди.
    }
}
```

Цей код має щонайменше п'ять структурних проблем. Жодна з них не критична сама по собі — але разом вони перетворюють додаток на крихку конструкцію, яку важко розширювати й майже неможливо тестувати.

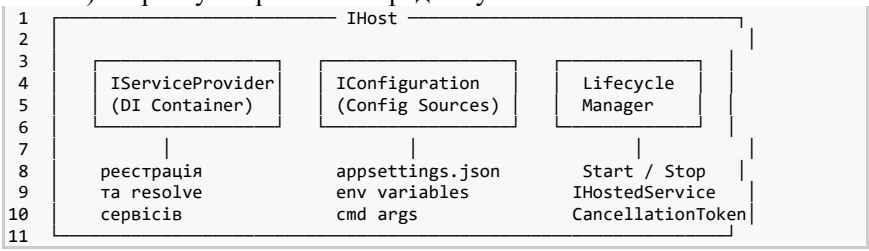
Що таке Generic Host

Generic Host — це об'єкт типу IHost, що виконує три ролі одночасно:

1. Service Container (DI-контейнер) — зберігає реєстрацію всіх сервісів і відповідає за їхнє створення та управління їхнім часом життя. Замість того, щоб кожен клас сам створював свої залежності через new, він отримує їх через конструктор — а контейнер знає, як їх побудувати.

2. Configuration Provider — агрегує конфігурацію з різних джерел: appsettings.json, змінні середовища, аргументи командного рядка, секрети — і надає єдиний уніфікований доступ через IConfiguration.

3. Lifecycle Manager — відстежує стан додатку, запускає фонові сервіси (IHostedService), перехоплює системні сигнали (SIGTERM, Ctrl+C) і гарантує коректний порядок зупинки компонентів.



Архітектура: Builder Pattern для хоста

Generic Host будується за **Builder Pattern**: спочатку конфігурується IHostBuilder, потім викликається .Build() і отримується IHost. Реальний API виглядає так:

```
C#
// Реальний код з Microsoft.Extensions.Hosting (не runnable в браузері)
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.DependencyInjection;

IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
```

```
 7      {
 8          // Реєструємо сервіси
 9          services.AddSingleton<DatabaseConnection>();
10          services.AddScoped<AppointmentService>();
11          services.AddTransient<ReportService>();
12      })
13      .Build();
14
15  await host.RunAsync();
```

Host.CreateDefaultBuilder() — це фабричний метод, що автоматично налаштовує:

- завантаження appsettings.json та appsettings.{Environment}.json
- читання змінних середовища: з префіксом DOTNET_ — для конфігурації самого хоста (середовище, кореневий каталог), а **всі** змінні середовища без префікса — для конфігурації застосунку (IConfiguration)
- User Secrets — лише в середовищі Development
- читання аргументів командного рядка
- логування в консоль, Debug та EventSource (на Windows — ще й EventLog)
- встановлення кореневого каталогу додатку (поточний каталог)
- у середовищі Development — перевірку контейнера під час Build() (ValidateOnBuild і ValidateScopes, див. 21.2–21.3)

Починаючи з .NET 7, поряд із цим «callback»-стилем є і лінійний API — Host.CreateApplicationBuilder(args), який повертає HostApplicationBuilder з властивостями Services, Configuration і Logging. Його налаштовують звичайними інструкціями без вкладених лямбд — так само, як WebApplication.CreateBuilder у веб-проєктах, — і саме його використовують сучасні шаблони Worker Service:

```
C#
1  var builder = Host.CreateApplicationBuilder(args);
2  builder.Services.AddSingleton<IPatientRepository, InMemoryPatientRepository>();
3  builder.Services.AddHostedService<AppointmentReminderService>();
4  using IHost host = builder.Build();
5  await host.RunAsync();
```

Обидва варіанти будують той самий IHost з однаковими налаштуваннями за замовчуванням; відрізняється лише стиль конфігурування.

Реалізуємо спрощений хост «з нуля»

Щоб зрозуміти, що відбувається всередині IHost, побудуємо власну спрощену реалізацію. Це найкращий спосіб зрозуміти будь-яку інфраструктурну абстракцію — побудувати її самостійно.

```C#
using System;
using System.Collections.Generic;
using System.Threading;

// ═══════════════════════════════════════════════════════
// КРОК 1: Сервіси та інтерфейси
// ═══════════════════════════════════════════════════════

interface IPatientRepository
{
    void Save(string patientName);
    List<string> GetAll();
}

interface INotificationService
{
    void Send(string recipient, string message);
}

class InMemoryPatientRepository : IPatientRepository
{
    private readonly List<string> _patients = new();

    public void Save(string patientName)
    {
        _patients.Add(patientName);
        Console.WriteLine($"  [Repo] Збережено пацієнта: {patientName}");
    }

    public List<string> GetAll() => _patients;
}

class ConsoleNotificationService : INotificationService
{
    public void Send(string recipient, string message)
    {
        Console.WriteLine($"  [Notify] -> {recipient}: {message}");
    }
}

class AppointmentService
{
    private readonly IPatientRepository _repo;
    private readonly INotificationService _notify;

    // Залежності отримуємо через конструктор — ніяких new усередині!
    public AppointmentService(
        IPatientRepository repo,
        INotificationService notify)
    {
        _repo   = repo;
```

```
        _notify = notify;
    }

    public void RegisterPatient(string name)
    {
        _repo.Save(name);
        _notify.Send(name, "Вас зареєстровано у клініці. Ласкаво просимо!");
    }

    public void PrintAllPatients()
    {
        var all = _repo.GetAll();
        Console.WriteLine($"  [Service] Усього пацієнтів:
{all.Count.ToString()}");
        foreach (var p in all)
            Console.WriteLine($"    - {p}");
    }
}

// ═══════════════════════════════════════════════════════
// КРОК 2: Спрощений DI-контейнер
// ═══════════════════════════════════════════════════════

class SimpleServiceCollection
{
    // Зберігаємо фабрики: тип → функція створення об'єкта
    private readonly Dictionary<Type, Func<SimpleServiceProvider, object>>
_factories = new();

    public void AddSingleton<TService, TImplementation>()
        where TImplementation : TService
    {
        object? instance = null;
        _factories[typeof(TService)] = provider =>
        {
            // Singleton: один екземпляр на весь час роботи
            instance ??= provider.Resolve(typeof(TImplementation));
            return instance!;
        };
        _factories[typeof(TImplementation)] = provider =>
            (TImplementation)Activator.CreateInstance(typeof(TImplementation),
                GetConstructorArgs(typeof(TImplementation), provider))!;
    }

    public void AddTransient<TService, TImplementation>()
        where TImplementation : TService
    {
        _factories[typeof(TService)] = provider =>
            // Transient: новий екземпляр при кожному запиті
            provider.Resolve(typeof(TImplementation));

        _factories[typeof(TImplementation)] = provider =>
            (TImplementation)Activator.CreateInstance(typeof(TImplementation),
                GetConstructorArgs(typeof(TImplementation), provider))!;
    }

    private object[] GetConstructorArgs(Type type, SimpleServiceProvider
provider)
    {
        var ctor = type.GetConstructors()[0];
```

```
109         var parameters = ctor.GetParameters();
110         var args = new object[parameters.Length];
111         for (int i = 0; i < parameters.Length; i++)
112             args[i] = provider.Resolve(parameters[i].ParameterType);
113         return args;
114     }
115 
116     public SimpleServiceProvider BuildServiceProvider()
117         => new SimpleServiceProvider(_factories);
118 }
119 
120 class SimpleServiceProvider
121 {
122     private readonly Dictionary<Type, Func<SimpleServiceProvider, object>>
    _factories;
123 
124     public SimpleServiceProvider(Dictionary<Type, Func<SimpleServiceProvider,
    object>> factories)
125         => _factories = factories;
126 
127     public object Resolve(Type type)
128     {
129         if (_factories.TryGetValue(type, out var factory))
130             return factory(this);
131         throw new InvalidOperationException($"Сервіс не зареєстрований:
    {type.Name}");
132     }
133 
134     public T Resolve<T>() => (T)Resolve(typeof(T));
135 }
136 
137 // ═══════════════════════════════════════════════════════════
138 // КРОК 3: Спрощений хост із lifecycle management
139 // ═══════════════════════════════════════════════════════════
140 
141 interface IHostedService
142 {
143     void Start();
144     void Stop();
145 }
146 
147 class SimpleHost
148 {
149     private readonly SimpleServiceProvider _services;
150     private readonly List<IHostedService> _hostedServices = new();
151     private readonly CancellationTokenSource _cts = new();
152 
153     public SimpleHost(SimpleServiceProvider services) => _services = services;
154 
155     public SimpleServiceProvider Services => _services;
156 
157     public void AddHostedService(IHostedService service)
158         => _hostedServices.Add(service);
159 
160     public void Run()
161     {
162         Console.WriteLine("[Host] Запуск хоста...");
163 
164         // Запуск усіх фонових сервісів
165         foreach (var svc in _hostedServices)
```

```
        {
            Console.WriteLine($"[Host] Запуск {svc.GetType().Name}");
            svc.Start();
        }

        Console.WriteLine("[Host] Хост запущено. Натисніть Ctrl+C або Enter для зупинки.");
        Console.ReadLine(); // У реальному хості тут — очікування CancellationToken

        Shutdown();
    }

    private void Shutdown()
    {
        Console.WriteLine("[Host] Отримано сигнал зупинки...");
        // Зупинка у зворотному порядку — важливо!
        for (int i = _hostedServices.Count - 1; i >= 0; i--)
        {
            Console.WriteLine($"[Host] Зупинка {_hostedServices[i].GetType().Name}");
            _hostedServices[i].Stop();
        }
        Console.WriteLine("[Host] Хост зупинено.");
    }
}

// ═══════════════════════════════════════════════════════════
// КРОК 4: Фоновий сервіс
// ═══════════════════════════════════════════════════════════

class AppointmentReminderService : IHostedService
{
    private readonly INotificationService _notify;

    public AppointmentReminderService(INotificationService notify)
        => _notify = notify;

    public void Start()
    {
        Console.WriteLine("  [Reminder] Сервіс нагадувань активовано");
        // У реальному коді тут запускається Timer або Task з циклом
        _notify.Send("Всі пацієнти з прийомом завтра",
                     "Нагадуємо про ваш прийом о 10:00");
    }

    public void Stop()
        => Console.WriteLine("  [Reminder] Сервіс нагадувань зупинено");
}

// ═══════════════════════════════════════════════════════════
// КРОК 5: Збираємо все разом — аналог Program.cs з Generic Host
// ═══════════════════════════════════════════════════════════

class Program
{
    static void Main()
    {
        // ─── Реєстрація сервісів (аналог ConfigureServices) ──────
        var services = new SimpleServiceCollection();
```

```
223         services.AddSingleton<IPatientRepository,
    InMemoryPatientRepository>();
224         services.AddTransient<INotificationService,
    ConsoleNotificationService>();
225         services.AddTransient<AppointmentService, AppointmentService>();
226
227         var provider = services.BuildServiceProvider();
228
229         // ── Будуємо хост ──────────────────────────────────────
230         var host = new SimpleHost(provider);
231
232         // Додаємо фоновий сервіс
233         var reminderNotify = provider.Resolve<INotificationService>();
234         host.AddHostedService(new AppointmentReminderService(reminderNotify));
235
236         // ── Використовуємо сервіси через DI ──────────────────
237         Console.WriteLine("\n=== Реєстрація пацієнтів ===");
238         var apptService = provider.Resolve<AppointmentService>();
239         apptService.RegisterPatient("Іваненко Олег");
240         apptService.RegisterPatient("Мельник Ганна");
241         apptService.RegisterPatient("Коваль Дмитро");
242
243         Console.WriteLine("\n=== Список пацієнтів ===");
244         apptService.PrintAllPatients();
245
246         // ── Демонстрація Singleton ────────────────────────────
247         Console.WriteLine("\n=== Перевірка Singleton (IPatientRepository)
    ===");
248         var repo1 = provider.Resolve<IPatientRepository>();
249         var repo2 = provider.Resolve<IPatientRepository>();
250         Console.WriteLine($"  repo1 == repo2: {object.ReferenceEquals(repo1,
    repo2).ToString()}");
251         // Singleton → той самий об'єкт
252     }
253 }
```

Ключові компоненти реального Generic Host

Розглянемо API, що ви побачите у реальному коді .NET-проектів:

IHost — центральний інтерфейс. Надає доступ до IServiceProvider через властивість Services. Методи StartAsync() / StopAsync() / RunAsync() керують lifecycle.

IHostBuilder — будівельник хоста. Надає fluent API для конфігурації:

— ConfigureServices(Action<IServiceCollection>) — реєстрація сервісів

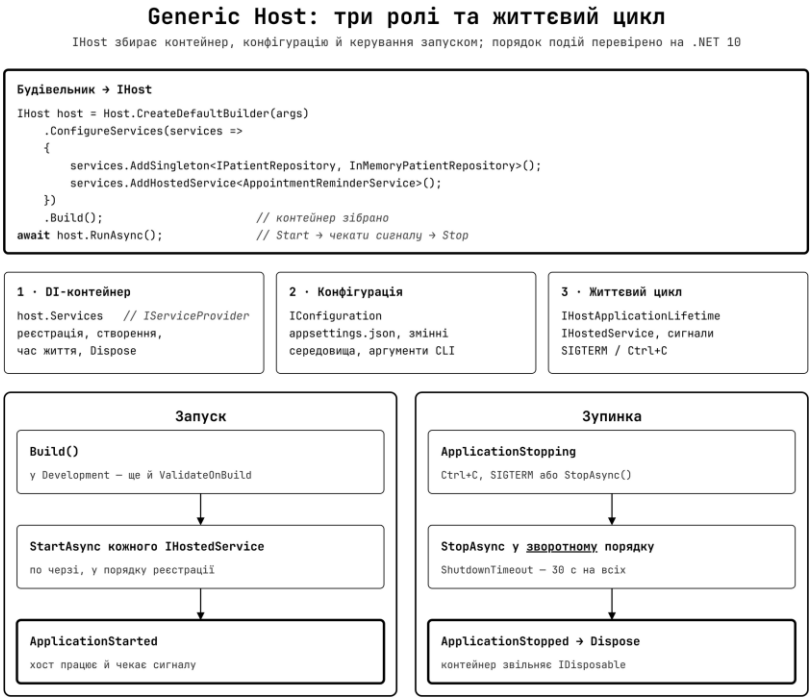
— ConfigureAppConfiguration(Action<IConfigurationBuilder>) — додаткові джерела конфігурації

— ConfigureLogging(Action<ILoggingBuilder>) — налаштування логування

— UseEnvironment(string) — встановлення середовища (Development, Production)

IHostedService — інтерфейс фонового сервісу. Два методи: StartAsync(CancellationToken) і StopAsync(CancellationToken). Хост запускає всі зареєстровані IHostedService при старті і зупиняє у зворотному порядку при shutdown.

BackgroundService — абстрактний базовий клас, що реалізує IHostedService. Достатньо перевизначити один метод: ExecuteAsync(CancellationToken stoppingToken) — і писати в ньому нескінченний цикл або очікування.



Generic Host — архітектура та потік запуску

Порядок старту та shutdown

Розуміння порядку є критичним для коректної роботи:

Запуск:

1. IHostBuilder.Build() — збирається DI-контейнер; у середовищі Development ще й перевіряється, чи всі зареєстровані сервіси можна створити (у Production ця перевірка вимкнена, і помилка реєстрації проявиться лише при першому запиті сервісу)
2. IHost.StartAsync() — послідовно викликає StartAsync() у кожного IHostedService у порядку реєстрації
3. Після старту всіх служб IHostApplicationLifetime сповіщає підписників ApplicationStarted; хост входить у стан «running» і чекає сигналу зупинки

Важлива деталь щодо BackgroundService: його StartAsync запускає ExecuteAsync і повертає керування, щойно той дійде до першого справжнього await. Тому довгий синхронний код на початку ExecuteAsync (до першого await) блокує запуск усіх наступних служб.

Shutdown (при Ctrl+C, SIGTERM або явному виклику StopAsync()):

4. IHostApplicationLifetime сповіщає підписників ApplicationStopping
5. IHost.StopAsync() — послідовно викликає StopAsync() у кожного IHostedService **у зворотному порядку**
6. IHostApplicationLifetime сповіщає підписників ApplicationStopped
7. DI-контейнер диспозиться — викликається Dispose() на всіх IDisposable-об'єктах, які створив кореневий контейнер (singleton-сервіси і transient-сервіси, отримані з кореня). Об'єкти, передані готовими через AddSingleton(instance), контейнер **не** звільняє — ними володіє той, хто їх створив

На весь shutdown хост дає обмежений час — HostOptions.ShutdownTimeout, за замовчуванням 30 секунд (до .NET 6 — 5 секунд). Служба, що не встигла завершити StopAsync, отримує скасування токена, і хост продовжує зупинку без неї.

Наведений порядок перевірено на .NET 10 для двох служб A і B, зареєстрованих саме в такому порядку: A start → B start → ApplicationStarted → ApplicationStopping → B stop → A stop → ApplicationStopped → Dispose.

Зворотний порядок shutdown є навмисним і важливим: перший зупинений сервіс може залежати від ресурсів, що надають пізніше зупинені сервіси. Наприклад, EmailSender (зупиняється першим) може надсилати Shutdown notification через SmtpClient (зупиняється останнім).

Generic Host vs WebApplication

Починаючи з .NET 6, для ASP.NET Core проектів з'явився WebApplication — ще більш спрощений API (var app = WebApplication.Create()). Він побудований поверх Generic Host і фактично є синтаксичним цукром для найпоширенішого сценарію — веб-додатків. Для консольних програм, Worker Services, мікросервісів без HTTP — Generic Host залишається правильним вибором.

	Generic Host	**WebApplication**
Тип додатку	будь-який	ASP.NET Core
Підключення	NuGet-пакет Microsoft.Extensions.Hosting	спільний фреймворк Microsoft.AspNetCore.App (SDK Microsoft.NET.Sdk.Web)
HTTP pipeline	немає	є (middleware)
Запуск	host.RunAsync()	app.RunAsync()
IServiceCollection	ConfigureServices()	builder.Services

Підсумок

> Generic Host вирішує проблему «розсипаного Main» шляхом введення чіткої відповідальності: хост знає про lifecycle, конфігурацію і контейнер сервісів — а бізнес-код нічого цього не знає і не зобов'язаний знати. Це пряме втілення принципів SRP та DIP зі SOLID: кожна частина системи відповідає за одне, і залежності течуть через абстракції.

21.2. IServiceCollection — реєстрація та розпізнавання сервісів

У попередньому розділі ми побачили, що Generic Host бере на себе управління lifecycle і надає DI-контейнер. Сам контейнер будується в два окремі кроки: спочатку **реєстрація** (IServiceCollection), потім **розпізнавання** (IServiceProvider). Це навмисне розділення: список зареєстрованих сервісів формується один раз при запуску, а потім

контейнер стає незмінним — жоден код у runtime не може додати новий сервіс або змінити вже зареєстрований.

Розуміння того, що саме зберігається в колекції сервісів і як контейнер вирішує, який об'єкт повернути, — це ключ до ефективної роботи з будь-яким DI-фреймворком у .NET.

ServiceDescriptor: атомарна одиниця реєстрації

Кожна реєстрація в IServiceCollection — це один об'єкт ServiceDescriptor. Він містить три речі:

```
ServiceDescriptor
├── ServiceType   : Type      — який тип запитується (зазвичай інтерфейс)
├── ImplementationType / ImplementationFactory / ImplementationInstance
│                             — як створити об'єкт
└── Lifetime      : ServiceLifetime — Singleton | Scoped | Transient
```

Коли контейнер отримує запит на IPatientRepository, він шукає ServiceDescriptor з ServiceType == typeof(IPatientRepository), дивиться на lifetime, і за допомогою ImplementationType або фабрики (ImplementationFactory) створює або повертає вже існуючий екземпляр.

Три способи реєстрації

Реальний IServiceCollection надає три перевантажені методи, що відповідають трьом ситуаціям:

Реєстрація через тип — найчастіша форма:

```C#
services.AddSingleton<IPatientRepository, SqlPatientRepository>();
services.AddScoped<AppointmentService>();          // ServiceType == ImplementationType
services.AddTransient<INotificationService, SmtpNotificationService>();
```

Реєстрація через фабрику — коли потрібна додаткова логіка при створенні:

```C#
services.AddSingleton<IPatientRepository>(provider =>
{
    var config = provider.GetRequiredService<IConfiguration>();
    var connStr = config["Database:ConnectionString"];
    return new SqlPatientRepository(connStr);
});
```

Реєстрація через готовий екземпляр — тільки для Singleton (об'єкт уже існує):

```
C#
1  var sharedCache = new MemoryCache();
2  services.AddSingleton<ICache>(sharedCache);
```

Побудуємо власну IServiceCollection

Розберемо механізм реєстрації та розпізнавання детально — реалізуємо повноцінний варіант, що підтримує всі три типи реєстрації і правильно обробляє циклічні залежності:

```
C#
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Reflection;
 5
 6  // ═══════════════════════════════════════════════════════════
 7  // ServiceDescriptor — опис однієї реєстрації
 8  // ═══════════════════════════════════════════════════════════
 9
10  enum ServiceLifetime { Singleton, Scoped, Transient }
11
12  class ServiceDescriptor
13  {
14      public Type ServiceType        { get; }
15      public Type? ImplementationType { get; }
16      public Func<IServiceProvider, object>? Factory { get; }
17      public object? Instance        { get; }
18      public ServiceLifetime Lifetime { get; }
19
20      // Конструктор 1: через тип реалізації
21      public ServiceDescriptor(Type serviceType, Type implType, ServiceLifetime
    lifetime)
22      {
23          ServiceType        = serviceType;
24          ImplementationType = implType;
25          Lifetime           = lifetime;
26      }
27
28      // Конструктор 2: через фабрику
29      public ServiceDescriptor(Type serviceType,
30                               Func<IServiceProvider, object> factory,
31                               ServiceLifetime lifetime)
32      {
33          ServiceType = serviceType;
34          Factory     = factory;
35          Lifetime    = lifetime;
36      }
37
38      // Конструктор 3: готовий екземпляр (тільки Singleton)
39      public ServiceDescriptor(Type serviceType, object instance)
40      {
41          ServiceType = serviceType;
42          Instance    = instance;
43          Lifetime    = ServiceLifetime.Singleton;
44      }
45  }
```

```
46
47  // ═══════════════════════════════════════════════════════════
48  // IServiceCollection — колекція реєстрацій (незмінна після Build)
49  // ═══════════════════════════════════════════════════════════
50
51  interface IServiceCollection
52  {
53      void Add(ServiceDescriptor descriptor);
54      IServiceProvider BuildServiceProvider();
55  }
56
57  interface IServiceProvider
58  {
59      object? GetService(Type serviceType);
60  }
61
62  static class ServiceCollectionExtensions
63  {
64      public static void AddSingleton<TService, TImpl>(this IServiceCollection
    col)
65          where TImpl : TService
66          => col.Add(new ServiceDescriptor(typeof(TService), typeof(TImpl),
    ServiceLifetime.Singleton));
67
68      public static void AddSingleton<TService>(this IServiceCollection col,
69                                                Func<IServiceProvider,
    TService> factory)
70          where TService : class
71          => col.Add(new ServiceDescriptor(typeof(TService),
72                      p => factory(p), ServiceLifetime.Singleton));
73
74      public static void AddTransient<TService, TImpl>(this IServiceCollection
    col)
75          where TImpl : TService
76          => col.Add(new ServiceDescriptor(typeof(TService), typeof(TImpl),
    ServiceLifetime.Transient));
77
78      public static void AddScoped<TService, TImpl>(this IServiceCollection col)
79          where TImpl : TService
80          => col.Add(new ServiceDescriptor(typeof(TService), typeof(TImpl),
    ServiceLifetime.Scoped));
81
82      public static T GetRequiredService<T>(this IServiceProvider provider)
83      {
84          var result = provider.GetService(typeof(T));
85          if (result is null)
86              throw new InvalidOperationException($"Сервіс {typeof(T).Name} не
    зареєстровано.");
87          return (T)result;
88      }
89  }
90
91  // ═══════════════════════════════════════════════════════════
92  // ServiceCollection + ServiceProvider — реалізація
93  // ═══════════════════════════════════════════════════════════
94
95  class ServiceCollection : IServiceCollection
96  {
97      private readonly List<ServiceDescriptor> _descriptors = new();
98      private bool _built = false;
```

```
 99
100     public void Add(ServiceDescriptor descriptor)
101     {
102         if (_built)
103             throw new InvalidOperationException(
104                 "Не можна реєструвати сервіси після BuildServiceProvider().");
105         _descriptors.Add(descriptor);
106     }
107
108     public IServiceProvider BuildServiceProvider()
109     {
110         _built = true;
111         return new ServiceProvider(_descriptors);
112     }
113 }
114
115 class ServiceProvider : IServiceProvider
116 {
117     private readonly List<ServiceDescriptor> _descriptors;
118     private readonly Dictionary<Type, object> _singletons = new();
119     // Для виявлення циклічних залежностей
120     private readonly HashSet<Type> _resolving = new();
121
122     public ServiceProvider(List<ServiceDescriptor> descriptors)
123         => _descriptors = descriptors;
124
125     public object? GetService(Type serviceType)
126     {
127         var descriptor = _descriptors.LastOrDefault(d => d.ServiceType ==
    serviceType);
128         if (descriptor is null) return null;
129
130         return descriptor.Lifetime switch
131         {
132             ServiceLifetime.Singleton  => GetOrCreateSingleton(descriptor),
133             ServiceLifetime.Transient  => CreateInstance(descriptor),
134             ServiceLifetime.Scoped     => GetOrCreateSingleton(descriptor), //
    спрощено
135             _ => throw new NotSupportedException()
136         };
137     }
138
139     private object GetOrCreateSingleton(ServiceDescriptor d)
140     {
141         if (_singletons.TryGetValue(d.ServiceType, out var existing))
142             return existing;
143         var instance = CreateInstance(d);
144         _singletons[d.ServiceType] = instance;
145         return instance;
146     }
147
148     private object CreateInstance(ServiceDescriptor d)
149     {
150         // Готовий екземпляр
151         if (d.Instance is not null) return d.Instance;
152
153         // Фабрика
154         if (d.Factory is not null) return d.Factory(this);
155
156         // Тип реалізації — знаходимо конструктор і вирішуємо аргументи
```

```
        var implType = d.ImplementationType!;

        // Захист від циклічних залежностей
        if (_resolving.Contains(implType))
            throw new InvalidOperationException(
                $"Циклічна залежність при розпізнаванні {implType.Name}");

        _resolving.Add(implType);
        try
        {
            var ctor = implType.GetConstructors()
                               .OrderByDescending(c =>
    c.GetParameters().Length)
                               .First();

            var args = ctor.GetParameters()
                           .Select(p => GetService(p.ParameterType)
                                        ?? throw new
    InvalidOperationException(
                                            $"Немає реєстрації для
    {p.ParameterType.Name} " +
                                            $"(потрібен {implType.Name})"))
                           .ToArray();

            return Activator.CreateInstance(implType, args)!;
        }
        finally
        {
            _resolving.Remove(implType);
        }
    }
}

// ═══════════════════════════════════════════════════════════
// Доменні класи — клінічна система
// ═══════════════════════════════════════════════════════════

interface IPatientRepository
{
    void Save(string name);
    List<string> GetAll();
}

interface IAppointmentRepository
{
    void Save(string patient, string doctor, DateTime time);
    List<string> GetAll();
}

interface INotificationService
{
    void Notify(string to, string message);
}

class InMemoryPatientRepository : IPatientRepository
{
    private readonly List<string> _data = new();
    public void Save(string name) { _data.Add(name); }
    public List<string> GetAll() => _data;
}
```

```
214
215 class InMemoryAppointmentRepository : IAppointmentRepository
216 {
217     private readonly List<string> _data = new();
218     public void Save(string p, string d, DateTime t)
219         => _data.Add($"{p} -> {d} o {t:HH:mm}");
220     public List<string> GetAll() => _data;
221 }
222
223 class ConsoleNotificationService : INotificationService
224 {
225     public void Notify(string to, string message)
226         => Console.WriteLine($"  [Email] {to}: {message}");
227 }
228
229 class AppointmentService
230 {
231     private readonly IPatientRepository _patients;
232     private readonly IAppointmentRepository _appointments;
233     private readonly INotificationService _notifications;
234
235     public AppointmentService(
236         IPatientRepository patients,
237         IAppointmentRepository appointments,
238         INotificationService notifications)
239     {
240         _patients      = patients;
241         _appointments  = appointments;
242         _notifications = notifications;
243     }
244
245     public void RegisterPatient(string name)
246     {
247         _patients.Save(name);
248         _notifications.Notify(name, "Реєстрацію підтверджено");
249     }
250
251     public void BookAppointment(string patient, string doctor, DateTime time)
252     {
253         _appointments.Save(patient, doctor, time);
254         _notifications.Notify(patient, $"Запис до {doctor} о {time:HH:mm}
    підтверджено");
255     }
256
257     public void PrintSummary()
258     {
259         var pts = _patients.GetAll();
260         var apts = _appointments.GetAll();
261         Console.WriteLine($"  Пацієнтів: {pts.Count.ToString()}, Записів:
    {apts.Count.ToString()}");
262         foreach (var a in apts) Console.WriteLine($"    {a}");
263     }
264 }
265
266 // ═══════════════════════════════════════════════════════════
267 // ДЕМОНСТРАЦІЯ
268 // ═══════════════════════════════════════════════════════════
269
270 class Program
271 {
```

```
    static void Main()
    {
        Console.WriteLine("=== Реєстрація сервісів ===");

        var collection = new ServiceCollection();

        // Реєстрація через тип
        collection.AddSingleton<IPatientRepository,
InMemoryPatientRepository>();
        collection.AddSingleton<IAppointmentRepository,
InMemoryAppointmentRepository>();
        collection.AddSingleton<INotificationService,
ConsoleNotificationService>();
        collection.AddTransient<AppointmentService, AppointmentService>();

        Console.WriteLine("  [OK] IPatientRepository     →
InMemoryPatientRepository (Singleton)");
        Console.WriteLine("  [OK] IAppointmentRepository →
InMemoryAppointmentRepository (Singleton)");
        Console.WriteLine("  [OK] INotificationService   →
ConsoleNotificationService (Singleton)");
        Console.WriteLine("  [OK] AppointmentService     → AppointmentService
(Transient)");

        // ── Build: після цього реєстрація заблокована ──────────
        Console.WriteLine("\n=== Build ServiceProvider ===");
        var provider = collection.BuildServiceProvider();
        Console.WriteLine("  [OK] ServiceProvider побудовано");

        // ── Спроба зареєструвати після Build ────────────────────
        try
        {
            collection.AddSingleton<IPatientRepository,
InMemoryPatientRepository>();
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine($"  [X] Очікувана помилка: {ex.Message}");
        }

        // ── Resolve та використання ─────────────────────────────
        Console.WriteLine("\n=== Resolve та використання ===");
        var service = provider.GetRequiredService<AppointmentService>();

        service.RegisterPatient("Іваненко Олег");
        service.RegisterPatient("Мельник Ганна");
        service.BookAppointment("Іваненко Олег", "Лікар Петренко", new
DateTime(2026, 6, 10, 10, 0, 0));
        service.BookAppointment("Мельник Ганна", "Лікар Сидоренко", new
DateTime(2026, 6, 11, 14, 30, 0));

        Console.WriteLine("\n=== Підсумок ===");
        service.PrintSummary();

        // ── Перевірка Singleton: завжди той самий об'єкт ────────
        Console.WriteLine("\n=== Singleton: перевірка ідентичності ===");
        var r1 = provider.GetRequiredService<IPatientRepository>();
        var r2 = provider.GetRequiredService<IPatientRepository>();
        Console.WriteLine($"  r1 === r2: {object.ReferenceEquals(r1,
r2).ToString()}");
```

```
321
322         // ── Перевірка Transient: завжди новий об'єкт ──────────────
323         Console.WriteLine("\n=== Transient: перевірка ідентичності ===");
324         var s1 = provider.GetRequiredService<AppointmentService>();
325         var s2 = provider.GetRequiredService<AppointmentService>();
326         Console.WriteLine($"  s1 === s2: {object.ReferenceEquals(s1,
    s2).ToString()}");
327     }
328 }
```

Множинна реєстрація одного інтерфейсу

Важлива особливість: один інтерфейс можна зареєструвати кілька разів з різними реалізаціями. При запиті через GetService<T>() повертається **остання** реєстрація. При запиті через GetServices<T>() (або IEnumerable<T>) — **всі** реєстрації. Це використовується у патернах типу «ланцюжок обробників» або «список валідаторів»:

```
C#
1  // Всі валідатори для AppointmentRequest
2  services.AddTransient<IAppointmentValidator, TimeSlotValidator>();
3  services.AddTransient<IAppointmentValidator, DoctorAvailabilityValidator>();
4  services.AddTransient<IAppointmentValidator, PatientInsuranceValidator>();
5
6  // У AppointmentService:
7  // IEnumerable<IAppointmentValidator> validators  ← отримуємо всі три
```

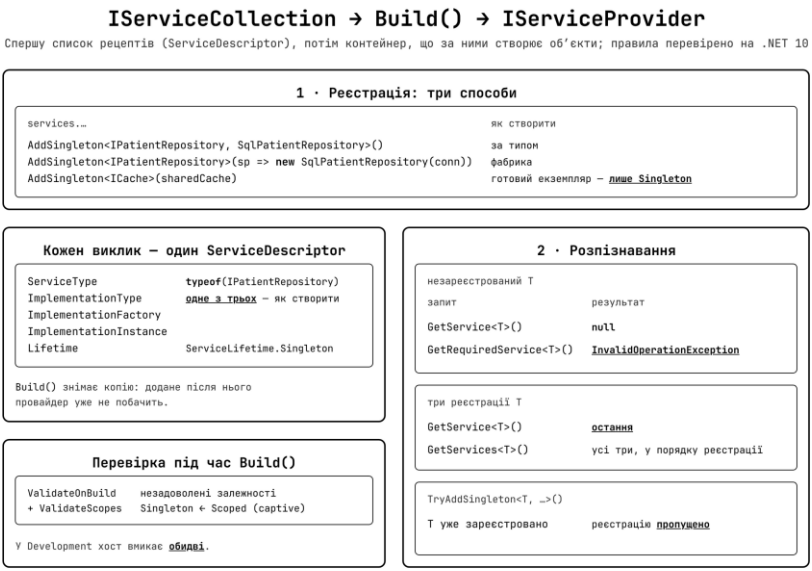
TryAdd: реєстрація «якщо ще не зареєстровано»

Методи TryAddSingleton, TryAddScoped, TryAddTransient реєструють сервіс **тільки якщо** такого ServiceType ще немає в колекції. Це корисно для бібліотек, що хочуть надати реалізацію «за замовчуванням», але не перезаписувати ту, що встановив користувач:

```
C#
1  // Program.cs: користувач спершу реєструє свою реалізацію
2  services.AddSingleton<IPatientRepository, CustomPatientRepository>();
3
4  // …потім викликає метод розширення бібліотеки, усередині якого:
5  services.TryAddSingleton<IPatientRepository, DefaultPatientRepository>();
6  // → IPatientRepository вже є, TryAdd нічого не додає: у колекції один
   дескриптор
```

Порядок тут принциповий. Якщо поміняти рядки місцями (спершу TryAdd бібліотеки, потім Add користувача), TryAdd спрацює — колекція ще порожня, — і в ній опиняться **обидва** дескриптори. GetService<IPatientRepository>() поверне CustomPatientRepository (остання реєстрація), але GetServices<IPatientRepository>() чи IEnumerable<IPatientRepository> у конструкторі отримають обидві

реалізації, і DefaultPatientRepository все ж буде створено. Обидва сценарії перевірено на .NET 10.



Реєстрація та розпізнавання сервісів у IServiceCollection

Перевірка на валідність реєстрацій

Реальний ServiceProvider у .NET підтримує опцію ValidateOnBuild = true. При увімкненні вона перевіряє при BuildServiceProvider():

— чи не є якийсь зареєстрований сервіс невирішуваним (залежність не зареєстрована)

— чи немає «captive dependencies» (Singleton залежить від Scoped — помилка, бо Scoped живе коротше) — **але лише разом з ValidateScopes = true**. Сам ValidateOnBuild перевіряє тільки, чи можна побудувати кожен сервіс; область видимості перевіряє ValidateScopes, яка також забороняє брати Scoped-сервіс з кореневого провайдера.

```
C#
1  var provider = services.BuildServiceProvider(new ServiceProviderOptions
2  {
3      ValidateOnBuild = true,   // незадоволені залежності — виняток уже тут
4      ValidateScopes  = true    // Singleton ← Scoped — теж виняток під час
   Build()
```

```
5  });
```

Перевірка на .NET 10: з одним лише ValidateOnBuild пара «Singleton залежить від Scoped» будується без помилки; з обома опціями BuildServiceProvider кидає AggregateException з описом проблемного дескриптора.

У Development-середовищі Generic Host автоматично вмикає обидві перевірки. В Production — вимикає заради швидшого старту, тож помилки реєстрації варто ловити тестом, що будує контейнер з обома опціями.

Підсумок

> IServiceCollection — це не просто список об'єктів. Це **специфікація** того, як система будується. Розділення реєстрації (IServiceCollection) і розпізнавання (IServiceProvider) — принципово важливе архітектурне рішення: воно гарантує, що склад системи визначається один раз на старті, а не довільно змінюється під час роботи. Це робить додаток передбачуваним, тестованим і правильно сконструйованим.

Антипатерн: Service Locator

Service Locator — це патерн, де клас отримує залежності не через конструктор, а **самостійно запитуючи їх у контейнера** (IServiceProvider) всередині своїх методів. Це антипатерн, що руйнує всі переваги DI.

```
C#
 1  // ПОГАНО: Service Locator — AppointmentService тягне IServiceProvider і сам
    витягує залежності
 2  class AppointmentService
 3  {
 4      private readonly IServiceProvider _sp; // зберігаємо контейнер!
 5
 6      public AppointmentService(IServiceProvider sp) => _sp = sp;
 7
 8      public void Book(string patientId, string doctorId)
 9      {
10          // Залежності приховані — видно тільки зсередини методу
11          var repo     = _sp.GetRequiredService<IPatientRepository>(); // ←
    Service Locator
12          var schedule = _sp.GetRequiredService<IDoctorSchedule>();    // ←
    Service Locator
13          var logger   = _sp.GetRequiredService<ILogger>();            // ←
    Service Locator
14
15          // ... логіка
```

```
    }
}

// ПРАВИЛЬНО: явні залежності через конструктор (Dependency Injection)
class AppointmentServiceCorrect
{
    private readonly IPatientRepository _repo;
    private readonly IDoctorSchedule    _schedule;
    private readonly ILogger            _logger;

    // Усі залежності видно у публічному API класу
    public AppointmentServiceCorrect(
        IPatientRepository repo,
        IDoctorSchedule    schedule,
        ILogger            logger)
    {
        _repo     = repo;
        _schedule = schedule;
        _logger   = logger;
    }

    public void Book(string patientId, string doctorId)
    {
        _logger.Log($"Записую {patientId} до {doctorId}");
        // ...
    }
}
```

Чому Service Locator — антипатерн:

Проблема	**Service Locator**	**DI через конструктор**
Залежності	Приховані в методах	**Явні** в конструкторі
Тестування	Потрібен реальний або складний mock IServiceProvider	**Достатньо mock-ів** конкретних інтерфейсів
Помилки	Виникають в рантаймі (GetRequiredService кидає)	Виявляються **при старті** (якщо ValidateOnBuild = true)
Принцип	Порушує DIP — клас знає про контейнер	Відповідає DIP — клас знає лише свої абстракції

Коли IServiceProvider все ж доречний:

— У Program.cs/Startup.cs при конфігурації — це root composition root, де контейнер і повинен бути присутній

— У IServiceScopeFactory.CreateScope() у фонових сервісах (Singleton, що потребує Scoped-сервісів)

— У middleware або factory-методах, де тип сервісу визначається динамічно

У всіх інших місцях — **ін'єктуйте через конструктор**, а не через IServiceProvider.

21.3. Часи життя сервісів — Singleton, Scoped, Transient

Вибір правильного часу життя сервісу (ServiceLifetime) — одне з найважливіших архітектурних рішень при роботі з DI-контейнером. Неправильний вибір не завжди призводить до видимої помилки одразу, але може спричинити тонкі баги: загальний стан там, де його не повинно бути, витоки пам'яті, баги в багатопотоковому середовищі або ObjectDisposedException у найнеочікуванішому місці.

.NET DI підтримує три часи життя. Їхні назви коротко описують правило:

Lifetime	Екземпляр	Коли створюється	Коли знищується
Singleton	один на контейнер	перший запит	Dispose() кореневого контейнера (зупинка додатку)
Scoped	один на «область» (запит)	початок scope	кінець scope
Transient	новий при кожному запиті	кожен GetService<T>()	IDisposable — разом зі scope, що його створив; інші — збирач сміття, коли зникнуть посилання

Singleton

Singleton — контейнер створює рівно один екземпляр і повертає його при кожному запиті. Цей екземпляр живе весь час роботи додатку.

Коли використовувати:

— Об'єкти, що є дорогими для ініціалізації і можна безпечно розділяти між потоками: кеш, пул з'єднань, клієнт HTTP, конфігурація

— Сервіси без стану, що однаково поводяться при будь-якому виклику

— Реєстрація «готового екземпляра»: services.AddSingleton<IConfiguration>(config). Такий об'єкт створили ви, а не контейнер, тому контейнер його й не звільняє: Dispose() для нього при зупинці **не** викликається

Небезпеки:

— Singleton повинен бути **потокобезпечним** (thread-safe), бо кілька потоків можуть одночасно звертатись до нього

— Singleton **не може залежати від Scoped-сервісів** — captive dependency (про це далі)

— Якщо Singleton зберігає змінний стан, може виникати «забруднення» між запитами

Scoped

Scoped — контейнер створює один екземпляр на «область» (scope). У контексті ASP.NET Core один scope = один HTTP-запит. У Worker Service scope створюється вручну.

Коли використовувати:

— DbContext (Entity Framework) — класичний приклад: один контекст на запит означає спільний трекер змін і спільне з'єднання для всіх репозиторіїв у межах одного HTTP-запиту, тож один виклик SaveChanges() зберігає всі зміни запиту разом (сам SaveChanges виконується в транзакції). Автоматично в одну транзакцію весь запит не загортається — для кількох SaveChanges її відкривають явно (Database.BeginTransaction())

— UnitOfWork — паттерн, що агрегує кілька репозиторіїв і спільний контекст

— Будь-який сервіс, що повинен мати спільний стан у межах одного запиту, але незалежний від інших

Небезпеки:

— Scoped-сервіс, якого запитали з кореневого провайдера (поза scope), кидає InvalidOperationException — **якщо** увімкнено ValidateScopes (у Generic Host — у Development). Без перевірки

контейнер мовчки створить його в кореневій області, і він фактично стане Singleton — помилку буде важко помітити

— Якщо захопити scoped-сервіс у Singleton — він ніколи не буде знищений після запиту

Transient

Transient — новий екземпляр при кожному запиті. Якщо об'єкт не реалізує IDisposable, контейнер не зберігає на нього посилання — ним опікується збирач сміття. Якщо ж реалізує — контейнер запам'ятовує його у списку scope, що його створив, щоб викликати Dispose() при завершенні цього scope.

Коли використовувати:

— Легкі сервіси без стану, що коштують мало для ініціалізації

— Сервіси, де важлива ізоляція між різними частинами коду (наприклад, ланцюжки обробників подій)

— Валідатори, мапери, дрібні обчислювачі — об'єкти, яким не треба нічого пам'ятати між викликами

(Часто тут згадують ILogger<T>, але це не так: AddLogging() реєструє відкритий generic ILogger<> як **Singleton** — кожен тип T отримує свій логер, але один на весь контейнер; перевірено на .NET 10.)

Небезпеки:

— Якщо Transient-сервіс реалізує IDisposable і його отримують з **кореневого** провайдера, контейнер тримає посилання на кожен такий екземпляр аж до власного Dispose() при зупинці додатку. Кожен запит додає новий об'єкт у список — це витік пам'яті й ресурсів. Disposable-transient слід брати зі scope (CreateScope()), тоді вони звільняються разом із ним

Детальна демонстрація всіх трьох

```C#
using System;
using System.Collections.Generic;
using System.Linq;

// ═══════════════════════════════════════════════════════════
// Спрощений контейнер з підтримкою Scope
// ═══════════════════════════════════════════════════════════

enum ServiceLifetime { Singleton, Scoped, Transient }

record ServiceDescriptor(
    Type ServiceType,
```

```
    Type ImplementationType,
    ServiceLifetime Lifetime);

class Container
{
    private readonly List<ServiceDescriptor> _descriptors = new();
    private readonly Dictionary<Type, object> _singletons = new();
    private bool _built;

    public void AddSingleton<TService, TImpl>() where TImpl : TService
        => Register<TService, TImpl>(ServiceLifetime.Singleton);

    public void AddScoped<TService, TImpl>() where TImpl : TService
        => Register<TService, TImpl>(ServiceLifetime.Scoped);

    public void AddTransient<TService, TImpl>() where TImpl : TService
        => Register<TService, TImpl>(ServiceLifetime.Transient);

    private void Register<TService, TImpl>(ServiceLifetime lt) where TImpl :
TService
    {
        if (_built) throw new InvalidOperationException("Container is already
built.");
        _descriptors.Add(new ServiceDescriptor(typeof(TService),
typeof(TImpl), lt));
    }

    public Scope CreateScope()
    {
        _built = true;
        return new Scope(_descriptors, _singletons);
    }

    public TService Resolve<TService>() => CreateScope().Resolve<TService>();
}

class Scope : IDisposable
{
    private readonly List<ServiceDescriptor> _descriptors;
    private readonly Dictionary<Type, object> _singletons; // розділяється між
scope
    private readonly Dictionary<Type, object> _scoped = new(); // тільки для
цього scope
    private readonly List<IDisposable> _toDispose = new();
    private bool _disposed;

    public Scope(List<ServiceDescriptor> descriptors, Dictionary<Type, object>
singletons)
    {
        _descriptors = descriptors;
        _singletons  = singletons;
    }

    public T Resolve<T>() => (T)Resolve(typeof(T));

    public object Resolve(Type t)
    {
        var d = _descriptors.LastOrDefault(x => x.ServiceType == t)
                ?? throw new InvalidOperationException($"Не зареєстровано:
{t.Name}");
```

```
66
67          return d.Lifetime switch
68          {
69              ServiceLifetime.Singleton  => GetOrCreate(_singletons, d),
70              ServiceLifetime.Scoped     => GetOrCreate(_scoped, d),
71              ServiceLifetime.Transient  => Create(d),
72              _ => throw new NotSupportedException()
73          };
74      }
75
76      private object GetOrCreate(Dictionary<Type, object> cache,
    ServiceDescriptor d)
77      {
78          if (!cache.TryGetValue(d.ServiceType, out var inst))
79          {
80              inst = Create(d);
81              cache[d.ServiceType] = inst;
82          }
83          return inst;
84      }
85
86      private object Create(ServiceDescriptor d)
87      {
88          var ctor = d.ImplementationType.GetConstructors()
89                      .OrderByDescending(c => c.GetParameters().Length).First();
90          var args = ctor.GetParameters()
91                         .Select(p => Resolve(p.ParameterType))
92                         .ToArray();
93          var inst = Activator.CreateInstance(d.ImplementationType, args)!;
94          if (inst is IDisposable disposable) _toDispose.Add(disposable);
95          return inst;
96      }
97
98      public void Dispose()
99      {
100         if (_disposed) return;
101         _disposed = true;
102         // Знищення у зворотному порядку створення
103         for (int i = _toDispose.Count - 1; i >= 0; i--)
104             _toDispose[i].Dispose();
105         Console.WriteLine("  [Scope] Scope завершено та очищено");
106     }
107 }
108
109 // ═══════════════════════════════════════════════════════════
110 // Сервіси для демонстрації часів життя
111 // ═══════════════════════════════════════════════════════════
112
113 // Лічильник викликів — допомагає відстежити кількість екземплярів
114 static class InstanceCounter
115 {
116     private static int _singletonCount;
117     private static int _scopedCount;
118     private static int _transientCount;
119
120     public static int NextSingleton() => ++_singletonCount;
121     public static int NextScoped()    => ++_scopedCount;
122     public static int NextTransient() => ++_transientCount;
123 }
124
```

```
// ── Singleton сервіс ──────────────────────────────────────────

interface IAppConfiguration { string Get(string key); int InstanceId { get; } }

class AppConfiguration : IAppConfiguration
{
    public int InstanceId { get; } = InstanceCounter.NextSingleton();
    private readonly Dictionary<string, string> _settings = new()
    {
        ["DatabaseUrl"] = "Server=prod;Database=clinic",
        ["MaxConnections"] = "100"
    };
    public string Get(string key) => _settings.TryGetValue(key, out var v) ? v : "";
    public AppConfiguration()
        => Console.WriteLine($"  [NEW Singleton] AppConfiguration #{InstanceId.ToString()} створено");
}

// ── Scoped сервіс ─────────────────────────────────────────────

interface IUnitOfWork : IDisposable
{
    int InstanceId { get; }
    void SaveChanges();
}

class ClinicUnitOfWork : IUnitOfWork
{
    public int InstanceId { get; } = InstanceCounter.NextScoped();
    private readonly List<string> _pendingOperations = new();

    public ClinicUnitOfWork()
        => Console.WriteLine($"  [NEW Scoped]    ClinicUnitOfWork #{InstanceId.ToString()} створено");

    public void RegisterOperation(string op) => _pendingOperations.Add(op);
    public void SaveChanges()
    {
        Console.WriteLine($"  [UoW #{InstanceId.ToString()}] Зберігаю {_pendingOperations.Count.ToString()} операцій");
        _pendingOperations.Clear();
    }
    public void Dispose()
        => Console.WriteLine($"  [DISPOSE Scoped] ClinicUnitOfWork #{InstanceId.ToString()} знищено");
}

// ── Transient сервіс ──────────────────────────────────────────

interface IPatientValidator { bool Validate(string name); int InstanceId { get; } }

class PatientValidator : IPatientValidator
{
    public int InstanceId { get; } = InstanceCounter.NextTransient();
    private readonly IAppConfiguration _config;

    public PatientValidator(IAppConfiguration config)
```

```
178     {
179         _config = config;
180         Console.WriteLine($"  [NEW Transient]  PatientValidator
    #{InstanceId.ToString()} створено");
181     }
182 
183     public bool Validate(string name) => !string.IsNullOrWhiteSpace(name) &&
    name.Length >= 2;
184 }
185 
186 // ─── AppointmentService — залежить від усіх трьох ──────────────
187 
188 class AppointmentService
189 {
190     private readonly IAppConfiguration _config;   // Singleton
191     private readonly IUnitOfWork _uow;            // Scoped
192     private readonly IPatientValidator _validator; // Transient
193 
194     public AppointmentService(
195         IAppConfiguration config,
196         IUnitOfWork uow,
197         IPatientValidator validator)
198     {
199         _config    = config;
200         _uow       = uow;
201         _validator = validator;
202     }
203 
204     public void BookAppointment(string patient, string doctor)
205     {
206         if (!_validator.Validate(patient))
207         {
208             Console.WriteLine($"    [!] Невалідне ім'я пацієнта: {patient}");
209             return;
210         }
211         var db = _config.Get("DatabaseUrl");
212         Console.WriteLine($"    Запис [{patient} -> {doctor}] через {db}");
213     }
214 }
215 
216 // ═══════════════════════════════════════════════════════════
217 // ДЕМОНСТРАЦІЯ
218 // ═══════════════════════════════════════════════════════════
219 
220 class Program
221 {
222     static void Main()
223     {
224         var container = new Container();
225         container.AddSingleton<IAppConfiguration, AppConfiguration>();
226         container.AddScoped<IUnitOfWork, ClinicUnitOfWork>();
227         container.AddTransient<IPatientValidator, PatientValidator>();
228         container.AddTransient<AppointmentService, AppointmentService>();
229 
230         // ─── Scope 1: перший «запит» ──────────────────────────
231         Console.WriteLine("\n╔══ Scope 1 (запит 1) ══╗");
232         using (var scope1 = container.CreateScope())
233         {
234             var uow1a = scope1.Resolve<IUnitOfWork>();
235             var uow1b = scope1.Resolve<IUnitOfWork>();
```

```
            Console.WriteLine($"  uow1a == uow1b (Scoped): {object.ReferenceEquals(uow1a, uow1b).ToString()}");

            var val1a = scope1.Resolve<IPatientValidator>();
            var val1b = scope1.Resolve<IPatientValidator>();
            Console.WriteLine($"  val1a == val1b (Transient): {object.ReferenceEquals(val1a, val1b).ToString()}");

            var cfg1 = scope1.Resolve<IAppConfiguration>();
            Console.WriteLine($"  Config InstanceId: {cfg1.InstanceId.ToString()}");
        }

        // ── Scope 2: другий «запит» ──────────────────────────
        Console.WriteLine("\n╔══ Scope 2 (запит 2) ══╗");
        using (var scope2 = container.CreateScope())
        {
            var uow2 = scope2.Resolve<IUnitOfWork>();
            var cfg2 = scope2.Resolve<IAppConfiguration>();

            Console.WriteLine($"  UoW InstanceId scope2: {uow2.InstanceId.ToString()}");
            Console.WriteLine($"  Config InstanceId scope2: {cfg2.InstanceId.ToString()} (той самий Singleton!)");

            scope2.Resolve<AppointmentService>().BookAppointment("Іваненко Олег", "Лікар Петренко");
            scope2.Resolve<AppointmentService>().BookAppointment("!", "Лікар Сидоренко");
        }

        // ── Підсумок ─────────────────────────────────────────
        Console.WriteLine("\n=== Підсумок екземплярів ===");
        Console.WriteLine("  AppConfiguration: створено 1 раз (Singleton)");
        Console.WriteLine("  ClinicUnitOfWork: 1 на scope → створено 2 рази (Scoped)");
        Console.WriteLine("  PatientValidator: новий кожен раз (Transient)");
    }
}
```

Captive Dependency — найпоширеніша помилка

Captive dependency — ситуація, коли сервіс з довшим часом життя тримає посилання на сервіс з коротшим часом життя, «захоплюючи» його і не даючи знищитися.

Класичний приклад:

```
1  Singleton(AppointmentService)
2    └─ Scoped(ClinicUnitOfWork)   ← ПРОБЛЕМА!
```

AppointmentService живе весь час роботи додатку. Він отримав ClinicUnitOfWork при першому запиті. Але ClinicUnitOfWork мав жити тільки один запит — після його завершення він не знищується, бо Singleton тримає на нього посилання. При другому запиті AppointmentService продовжує використовувати **старий**

ClinicUnitOfWork від першого запиту, замість нового. Якщо UnitOfWork зберігає відкрите з'єднання з базою даних або транзакцію — це буде або витік ресурсів, або баг зі спільним станом між запитами.

Правило: час життя залежності не може бути коротшим за час життя того, хто від неї залежить.

```
1  Singleton  → може залежати від: Singleton (+ Transient – із застереженням)
2  Scoped     → може залежати від: Singleton, Scoped, Transient
3  Transient  → може залежати від: Singleton, Scoped, Transient
```

Контейнер забороняє (при ValidateScopes) лише одну комбінацію — Singleton, що залежить від Scoped. Залежність Singleton від Transient дозволена, але transient-об'єкт буде створено один раз і він житиме стільки ж, скільки Singleton. Це нормально, якщо він без стану і потокобезпечний, і є прихованою помилкою, якщо розрахований на короткий час життя (наприклад, тримає з'єднання).

C#

```
using System;

// ─── Демонстрація Captive Dependency ──────────────────────────

interface IRequestContext { string RequestId { get; } }

class RequestContext : IRequestContext
{
    public string RequestId { get; } = Guid.NewGuid().ToString("N")[..8];
    public RequestContext() => Console.WriteLine($"  [NEW] RequestContext RequestId={RequestId}");
}

class SingletonService
{
    private readonly IRequestContext _ctx; // Scoped захоплений у Singleton!

    public SingletonService(IRequestContext ctx)
    {
        _ctx = ctx;
        Console.WriteLine($"  [Singleton] Отримав ctx.RequestId={ctx.RequestId}");
    }

    public void DoWork()
    {
        // Завжди повертає RequestId з ПЕРШОГО запиту, навіть під час другого!
        Console.WriteLine($"  [DoWork] RequestId={_ctx.RequestId} (завжди той самий!)");
    }
}

// Ручна симуляція captive dependency:
class Program
{
    static void Main()
    {
```

```
        Console.WriteLine("=== Ілюстрація Captive Dependency ===\n");

        // «Перший запит» — RequestContext створюється
        var requestCtx1 = new RequestContext();
        var singleton   = new SingletonService(requestCtx1);

        Console.WriteLine("\n--- Запит 1 ---");
        singleton.DoWork(); // Виводить RequestId першого запиту

        // «Другий запит» — новий RequestContext, але Singleton вже має старий!
        var requestCtx2 = new RequestContext();
        Console.WriteLine($"\n--- Запит 2 --- (новий
ctx.RequestId={requestCtx2.RequestId})");
        singleton.DoWork(); // Виводить RequestId ПЕРШОГО запиту — баг!

        Console.WriteLine("\nВисновок: SingletonService тримає застарілий
RequestContext.");
        Console.WriteLine("Рішення: ніколи не ін'єктуйте Scoped у Singleton.");
        Console.WriteLine("Альтернатива: IServiceScopeFactory — отримати Scoped
через фабрику.");
    }
}
```

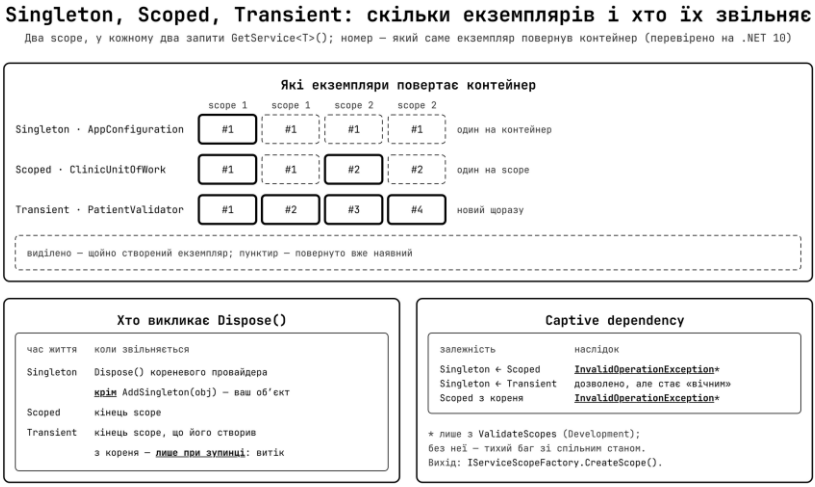
IServiceScopeFactory: правильний вихід

Якщо Singleton справді потребує scoped-сервісу (наприклад, BackgroundService обробляє один запис з черги і потребує DbContext), правильне рішення — ін'єктувати IServiceScopeFactory і самостійно керувати scope:

```
C#
class BackgroundProcessor // Singleton (IHostedService)
{
    private readonly IServiceScopeFactory _scopeFactory;

    public BackgroundProcessor(IServiceScopeFactory scopeFactory)
        => _scopeFactory = scopeFactory;

    public async Task ProcessAsync()
    {
        // Новий scope для кожної одиниці роботи
        using var scope = _scopeFactory.CreateScope();
        var dbContext =
scope.ServiceProvider.GetRequiredService<ClinicDbContext>();
        // dbContext живе тільки в межах цього using-блоку
        // Після виходу scope.Dispose() → dbContext.Dispose()
    }
}
```



Порівняння часів життя сервісів: Singleton, Scoped, Transient

Підсумок: як обирати lifetime

Singleton — якщо сервіс без стану або з незмінним станом, потокобезпечний, дорогий для ініціалізації.

Scoped — якщо сервіс має стан, специфічний для одного запиту (транзакція, UoW, контекст бази даних). Стандартний вибір для Entity Framework DbContext.

Transient — якщо сервіс легкий, без стану і не потребує спільності між різними споживачами. Стандартний вибір для дрібних утилітарних класів та валідаторів.

Коли маєте сумніви — починайте з **Transient**. Це найбезпечніший вибір з точки зору ізоляції, хоча й найдорожчий з точки зору алокацій. Перейдіть на Singleton або Scoped тільки тоді, коли це продиктовано конкретними вимогами до стану або продуктивності.

21.4. Options Pattern — типізована конфігурація з IOptions<T>

Конфігурація — невід'ємна частина будь-якого реального додатку. Рядок підключення до бази даних, адреса SMTP-сервера, максимальна кількість записів у черзі, тайм-аути — усе це параметри, що змінюються між середовищами (Development, Staging, Production) і не повинні бути

захардкоджені у коді. У .NET для роботи з конфігурацією існує ціла екосистема, центром якої є **Options Pattern**.

Options Pattern — це узгоджений підхід до читання конфігурації через типізовані класи. Замість читання рядків через IConfiguration["Section:Key"] скрізь по коду, ви оголошуєте POCO-клас (Plain Old C# Object), прив'язуєте його до секції конфігурації один раз при старті — і далі отримуєте через DI вже готовий, типізований, перевірений об'єкт.

Проблема: IConfiguration по всьому коду

Розглянемо типову еволюцію роботи з конфігурацією без Options Pattern:

```C#
// Антипаттерн: рядок ключа розкиданий по коду
class AppointmentService
{
    private readonly IConfiguration _config;
    public AppointmentService(IConfiguration config) { _config = config; }

    public void Book(...)
    {
        var maxPerDay = int.Parse(_config["Appointments:MaxPerDay"]); // magic string!
        var timeout   = TimeSpan.FromSeconds(
double.Parse(_config["Appointments:TimeoutSeconds"])); // дублювання!
        // ...
    }
}

class ReportService
{
    private readonly IConfiguration _config;
    public void Generate()
    {
        var maxPerDay = int.Parse(_config["Appointments:MaxPerDay"]); // ще раз дублювання
        // Якщо назву ключа зміните — помилка лише в runtime, не в compile time!
    }
}
```

Ця практика має щонайменше чотири вади:

8. **Magic strings** — будь-яка помилка в ключі виявляється лише у runtime
9. **Дублювання** — один ключ читається в десяти місцях
10. **Немає типізації** — IConfiguration повертає string?, потрібне ручне парсування

11. **Важко тестувати** — для тесту AppointmentService потрібно мокати весь IConfiguration

Options Pattern: рішення

Options Pattern пропонує три кроки:

1. Оголошення класу налаштувань:

```
C#
1  class AppointmentOptions
2  {
3      public int MaxPerDay    { get; set; } = 20;
4      public int TimeoutSeconds { get; set; } = 30;
5      public string AllowedStatuses { get; set; } = "Scheduled,Confirmed";
6  }
```

2. Реєстрація і прив'язка до секції:

```
C#
1  services.Configure<AppointmentOptions>(
2      configuration.GetSection("Appointments"));
```

3. Ін'єкція через IOptions<T>:

```
C#
 1  class AppointmentService
 2  {
 3      private readonly AppointmentOptions _opts;
 4      public AppointmentService(IOptions<AppointmentOptions> opts)
 5          => _opts = opts.Value;
 6
 7      public void Book(...)
 8      {
 9          if (_opts.MaxPerDay < 1) throw new ...;
10          // Типізовано, з автодоповненням, без magic strings
11      }
12  }
```

Три варіанти IOptions

Інтерфейс	Оновлення	Scope	Використання
IOptions<T>	ні (Singleton)	будь-який	типові налаштування
IOptionsSnapshot<T>	при кожному Scoped	Scoped	перезавантаження при зміні файлу

Інтерфейс	Оновлення	Scope	Використання
IOptionsMonitor<T>	реактивно (колбек)	Singleton	hot reload, повідомлення про зміни

IOptions<T> реєструється як Singleton: значення обчислюється один раз — при першому зверненні до .Value — і далі не змінюється, навіть якщо файл перезавантажено. IOptionsSnapshot<T> — Scoped: якщо файл конфігурації змінився між двома HTTP-запитами, другий запит отримає нові значення. IOptionsMonitor<T> — Singleton, але надає метод OnChange(Action<T>) для реакції на зміни в реальному часі.

Експеримент на .NET 10 (файл appsettings.json з reloadOnChange: true, значення MaxPerDay змінено з 20 на 50 під час роботи): IOptions<T>.Value і далі повертає 20; IOptionsSnapshot<T> у вже відкритому scope — 20, у новому scope — 50; IOptionsMonitor<T>.CurrentValue — одразу 50, а колбек OnChange отримав нове значення. Звідси й правило вибору: IOptions — для налаштувань, що не змінюються без перезапуску; IOptionsSnapshot — для Scoped-сервісів, яким потрібне узгоджене значення в межах запиту (у Singleton його ін'єктувати не можна — captive dependency); IOptionsMonitor — для Singleton і фонових служб, що мають реагувати на зміни. Іменовані налаштування (Get(name)) доступні лише через IOptionsSnapshot і IOptionsMonitor; IOptions<T>.Value завжди повертає варіант за замовчуванням.

Реалізуємо Options Pattern «з нуля»

C#

```
using System;
using System.Collections.Generic;

// ═══════════════════════════════════════════════════════════════
// Options Pattern — повна самописна реалізація
// ═══════════════════════════════════════════════════════════════

// ── 1. Аналог IOptions<T> ──────────────────────────────────
interface IOptions<T> where T : class
{
    T Value { get; }
}

class Options<T> : IOptions<T> where T : class
```

```
{
    public T Value { get; }
    public Options(T value) => Value = value;
}

// ─── 2. Аналог IOptionsSnapshot<T> ──────────────────────────
// Snapshot читає конфіг щоразу заново (у нас — з Dictionary)

interface IOptionsSnapshot<T> where T : class
{
    T Value { get; }
}

// ─── 3. Аналог IOptionsMonitor<T> ───────────────────────────

interface IOptionsMonitor<T> where T : class
{
    T CurrentValue { get; }
    IDisposable OnChange(Action<T> listener);
}

// ─── 4. Джерело конфігурації (аналог appsettings.json) ──────

class ConfigurationSource
{
    private readonly Dictionary<string, string> _values;
    private readonly List<Action> _changeListeners = new();

    public ConfigurationSource(Dictionary<string, string> values)
        => _values = values;

    public string? Get(string key)
        => _values.TryGetValue(key, out var v) ? v : null;

    public void Set(string key, string value)
    {
        _values[key] = value;
        // Сповіщаємо слухачів про зміну
        foreach (var listener in _changeListeners) listener();
    }

    public void OnChange(Action callback) => _changeListeners.Add(callback);
}

// ─── 5. Прив'язка: конфіг → клас налаштувань ────────────────

class ConfigurationBinder
{
    public static T Bind<T>(ConfigurationSource source, string prefix) where T
: new()
    {
        var obj = new T();
        var props = typeof(T).GetProperties();
        foreach (var prop in props)
        {
            var key = string.IsNullOrEmpty(prefix)
                ? prop.Name
                : $"{prefix}:{prop.Name}";
            var raw = source.Get(key);
            if (raw is null) continue;
```

```

            object? converted = prop.PropertyType switch
            {
                var t when t == typeof(string)  => raw,
                var t when t == typeof(int)     => int.Parse(raw),
                var t when t == typeof(double)  => double.Parse(raw),
                var t when t == typeof(bool)    => bool.Parse(raw),
                var t when t == typeof(TimeSpan)=> TimeSpan.Parse(raw),
                _ => null
            };

            if (converted is not null) prop.SetValue(obj, converted);
        }
        return obj;
    }
}

// ═══════════════════════════════════════════════════════════════
// Класи налаштувань (POCO — Plain Old C# Object)
// ═══════════════════════════════════════════════════════════════

class DatabaseOptions
{
    public string ConnectionString { get; set; } =
"Server=localhost;Database=clinic";
    public int MaxConnections      { get; set; } = 10;
    public int CommandTimeoutSec   { get; set; } = 30;
}

class SmtpOptions
{
    public string Host     { get; set; } = "smtp.localhost";
    public int    Port     { get; set; } = 587;
    public bool   UseSsl   { get; set; } = true;
    public string From     { get; set; } = "clinic@example.com";
}

class AppointmentOptions
{
    public int MaxPerDayPerDoctor { get; set; } = 20;
    public int ReminderHoursBefore { get; set; } = 24;
    public bool AutoConfirm { get; set; } = false;
}

// ═══════════════════════════════════════════════════════════════
// Реалізація IOptionsMonitor з підтримкою OnChange
// ═══════════════════════════════════════════════════════════════

class LiveOptionsMonitor<T> : IOptionsMonitor<T> where T : class, new()
{
    private readonly ConfigurationSource _source;
    private readonly string _prefix;
    private T _current;
    private readonly List<Action<T>> _listeners = new();

    public LiveOptionsMonitor(ConfigurationSource source, string prefix)
    {
        _source  = source;
        _prefix  = prefix;
        _current = ConfigurationBinder.Bind<T>(_source, _prefix);
```

```
134
135         // При зміні конфігурації — перечитуємо і сповіщаємо
136         _source.OnChange(() =>
137         {
138             _current = ConfigurationBinder.Bind<T>(_source, _prefix);
139             foreach (var l in _listeners) l(_current);
140         });
141     }
142
143     public T CurrentValue => _current;
144
145     public IDisposable OnChange(Action<T> listener)
146     {
147         _listeners.Add(listener);
148         return new Unsubscriber(() => _listeners.Remove(listener));
149     }
150
151     private class Unsubscriber : IDisposable
152     {
153         private readonly Action _remove;
154         public Unsubscriber(Action remove) => _remove = remove;
155         public void Dispose() => _remove();
156     }
157 }
158
159 // ═══════════════════════════════════════════════════════════
160 // Сервіси, що використовують Options
161 // ═══════════════════════════════════════════════════════════
162
163 class DatabaseService
164 {
165     private readonly DatabaseOptions _opts;
166
167     public DatabaseService(IOptions<DatabaseOptions> opts)
168         => _opts = opts.Value;
169
170     public void Connect()
171     {
172         Console.WriteLine($"  [DB] Підключення до {_opts.ConnectionString}");
173         Console.WriteLine($"  [DB]
    MaxConnections={_opts.MaxConnections.ToString()}, " +
174                           $"Timeout={_opts.CommandTimeoutSec.ToString()}s");
175     }
176 }
177
178 class NotificationService
179 {
180     private readonly SmtpOptions _opts;
181
182     public NotificationService(IOptions<SmtpOptions> opts)
183         => _opts = opts.Value;
184
185     public void SendReminder(string to, string message)
186     {
187         var ssl = _opts.UseSsl ? "SSL" : "plain";
188         Console.WriteLine($"  [SMTP] {_opts.From} → {to} " +
189                           $"via {_opts.Host}:{_opts.Port.ToString()}
    ({ssl})");
190         Console.WriteLine($"         Повідомлення: {message}");
191     }
```

```
}

class AppointmentService
{
    private readonly IOptionsMonitor<AppointmentOptions> _monitor;

    public AppointmentService(IOptionsMonitor<AppointmentOptions> monitor)
    {
        _monitor = monitor;
        // Реагуємо на зміни конфігурації в реальному часі
        _monitor.OnChange(opts =>
            Console.WriteLine($"  [!] Конфіг оновлено:
MaxPerDay={opts.MaxPerDayPerDoctor.ToString()}"));
    }

    public void Book(string patient, string doctor)
    {
        var opts = _monitor.CurrentValue;
        var confirm = opts.AutoConfirm ? "автоматично підтверджено" : "очікує
підтвердження";
        Console.WriteLine($"  [Appt] {patient} → {doctor} | {confirm} | " +
                          $"нагадування за
{opts.ReminderHoursBefore.ToString()}год");
    }
}

// ═══════════════════════════════════════════════════════════
// Аналог ConfigureServices + appsettings.json
// ═══════════════════════════════════════════════════════════

class Program
{
    static void Main()
    {
        // ─── «appsettings.json» у вигляді Dictionary ──────────────
        var source = new ConfigurationSource(new Dictionary<string, string>
        {
            ["Database:ConnectionString"]  = "Server=prod-
db;Database=clinic_prod",
            ["Database:MaxConnections"]    = "50",
            ["Database:CommandTimeoutSec"] = "60",
            ["Smtp:Host"]    = "smtp.clinic.ua",
            ["Smtp:Port"]    = "465",
            ["Smtp:UseSsl"]  = "true",
            ["Smtp:From"]    = "no-reply@clinic.ua",
            ["Appointments:MaxPerDayPerDoctor"]  = "15",
            ["Appointments:ReminderHoursBefore"] = "48",
            ["Appointments:AutoConfirm"]         = "false",
        });

        // ─── Реєстрація Options (аналог services.Configure<T>) ───
        IOptions<DatabaseOptions> dbOpts = new Options<DatabaseOptions>(
            ConfigurationBinder.Bind<DatabaseOptions>(source, "Database"));

        IOptions<SmtpOptions> smtpOpts = new Options<SmtpOptions>(
            ConfigurationBinder.Bind<SmtpOptions>(source, "Smtp"));

        IOptionsMonitor<AppointmentOptions> apptMonitor =
            new LiveOptionsMonitor<AppointmentOptions>(source,
"Appointments");
```

```
247
248         // ── Використання сервісів ──────────────────────────────
249         Console.WriteLine("=== Запуск сервісів ===\n");
250
251         var dbService    = new DatabaseService(dbOpts);
252         var notifyService = new NotificationService(smtpOpts);
253         var apptService  = new AppointmentService(apptMonitor);
254
255         dbService.Connect();
256
257         Console.WriteLine();
258         apptService.Book("Іваненко Олег",  "Лікар Петренко");
259         apptService.Book("Мельник Ганна",  "Лікар Сидоренко");
260
261         Console.WriteLine();
262         notifyService.SendReminder("Іваненко Олег",
263             "Нагадуємо про ваш прийом завтра о 10:00");
264
265         // ── Симуляція hot reload ────────────────────────────────
266         Console.WriteLine("\n=== Зміна конфігурації (hot reload) ===");
267         source.Set("Appointments:MaxPerDayPerDoctor", "25");
268         source.Set("Appointments:AutoConfirm",        "true");
269
270         Console.WriteLine();
271         apptService.Book("Коваль Дмитро", "Лікар Петренко");
272
273         // ── Перевірка IOptions (Singleton — не оновлюється) ─────
274         Console.WriteLine("\n=== IOptions vs IOptionsMonitor ===");
275         var staticOpts = new Options<AppointmentOptions>(
276             ConfigurationBinder.Bind<AppointmentOptions>(source,
    "Appointments"));
277
278         Console.WriteLine($"  IOptions MaxPerDay (зафіксовано):
    {staticOpts.Value.MaxPerDayPerDoctor.ToString()}");
279         Console.WriteLine($"  IOptionsMonitor MaxPerDay (live):
    {apptMonitor.CurrentValue.MaxPerDayPerDoctor.ToString()}");
280     }
281 }
```

Валідація налаштувань

Реальний IOptions підтримує валідацію через Data Annotations або кастомний IValidateOptions<T>. Це дозволяє перевірити конфігурацію при запуску і не запускати додаток з некоректними налаштуваннями:

```
C#
 1  // NuGet: Microsoft.Extensions.Options.DataAnnotations (для
    ValidateDataAnnotations)
 2  using System.ComponentModel.DataAnnotations;
 3
 4  // У реєстрації:
 5  builder.Services.AddOptions<DatabaseOptions>()
 6          .Bind(builder.Configuration.GetSection("Database"))
 7          .ValidateDataAnnotations()     // перевірка через атрибути
 8          .ValidateOnStart();            // перевірити під час host.StartAsync(),
    а не при першому .Value
 9
```

```
class DatabaseOptions
{
    [Required]
    [MinLength(10)]
    public string ConnectionString { get; set; } = "";

    [Range(1, 1000)]
    public int MaxConnections { get; set; } = 10;
}
```

Без ValidateOnStart() валідація теж працює, але «ліниво»: OptionsValidationException вилетить лише тоді, коли хтось уперше звернеться до .Value, — можливо, через години роботи. З ValidateOnStart() той самий виняток кидає вже host.StartAsync() (перевірено на .NET 10 з порожньою секцією Database), і некоректно налаштований додаток просто не стартує.

Це дозволяє уникнути ситуацій, коли додаток запускається, але падає через відсутнє або некоректне значення конфігурації через кілька хвилин роботи.

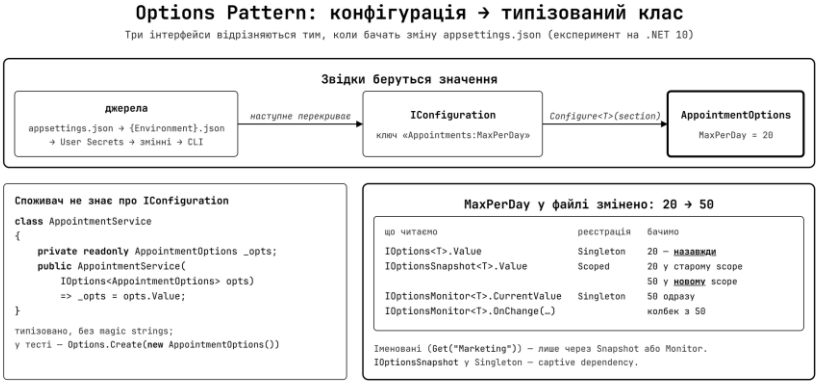
Named Options

Іноді потрібні кілька різних екземплярів одного класу налаштувань. Типовий приклад — два SMTP-сервери: один для транзакційних листів, інший для маркетингових:

C#
```
services.Configure<SmtpOptions>("Transactional",
    configuration.GetSection("Smtp:Transactional"));
services.Configure<SmtpOptions>("Marketing",
    configuration.GetSection("Smtp:Marketing"));

// Використання:
class EmailService
{
    private readonly SmtpOptions _transactional;
    private readonly SmtpOptions _marketing;

    public EmailService(IOptionsSnapshot<SmtpOptions> opts)
    {
        _transactional = opts.Get("Transactional");
        _marketing     = opts.Get("Marketing");
    }
}
```



Options Pattern — IOptions, IOptionsSnapshot, IOptionsMonitor

Де зберігати конфігурацію

Generic Host автоматично завантажує конфігурацію з декількох джерел у порядку зростаючого пріоритету (кожне наступне перекриває попереднє):

12. appsettings.json — базова конфігурація для всіх середовищ
13. appsettings.{Environment}.json — специфічна для середовища (наприклад, appsettings.Development.json)
14. Секрети (User Secrets) — для чутливих даних у Development
15. Змінні середовища — стандартний спосіб конфігурації в контейнерах (Docker/Kubernetes)
16. Аргументи командного рядка — найвищий пріоритет, зручно для тестів та CI/CD

Чутливі дані (паролі, API-ключі, рядки з'єднань) **ніколи** не повинні потрапляти до appsettings.json, що зберігається у git. Для Production використовують змінні середовища або секретні сховища (Azure Key Vault, AWS Secrets Manager тощо).

Підсумок

> Options Pattern вирішує проблему розсипаної конфігурації так само, як DI вирішує проблему розсипаних залежностей: збирає все в одному місці, дає типізацію і перевірку при запуску, і надає через DI вже готовий об'єкт — без magic strings і ручного парсування. Разом Generic Host, IServiceCollection і Options

> Pattern утворюють повну інфраструктуру для побудови добре структурованих, конфігурованих і тестованих .NET-додатків.

Питання для самоконтролю — Розділ 21. Generic Host та Dependency Injection

21.1. Generic Host

1. Назвіть три ролі Generic Host в .NET-застосунку. Які проблеми він вирішує порівняно зі стандартним підходом "все в Main()"? Що відбувається без хоста при спробі реалізувати DI, конфігурацію та логування вручну?
2. Розгляньте код:

```
C#
1  var host = Host.CreateDefaultBuilder(args)
2      .ConfigureServices(services =>
3      {
4          services.AddSingleton<IPatientService, PatientService>();
5          services.AddHostedService<SchedulerWorker>();
6      })
7      .Build();
8  await host.RunAsync();
```

Що автоматично налаштовує Host.CreateDefaultBuilder? Що робить Build()? Чому RunAsync() блокує до отримання Ctrl+C?

3. Що таке IHostedService? Назвіть два методи інтерфейсу та поясніть їх параметр CancellationToken. Як він відрізняється від BackgroundService? У яких випадках краще реалізовувати IHostedService напряму?
4. Опишіть порядок запуску та зупинки хоста: коли викликаються StartAsync кожного IHostedService? У якому порядку викликаються StopAsync? Що відбувається з Singleton-ами, що реалізують IDisposable?
5. Реалізуйте BackgroundService-клас HeartbeatWorker, що кожні 30 секунд записує в лог рядок "alive" і коректно завершується при зупинці хоста (обробляє CancellationToken у циклі).
6. Порівняйте Generic Host і WebApplication (мінімальний ASP.NET Core). Що спільного і чим вони відрізняються? Коли варто використовувати Generic Host без ASP.NET Core?

7. Що таке fluent API в контексті IHostBuilder? Поясніть методи ConfigureServices, ConfigureAppConfiguration, ConfigureLogging. Чи можна викликати ConfigureServices двічі в ланцюжку і що відбудеться?

21.2. IServiceCollection

8. Що таке ServiceDescriptor? Назвіть три поля, що він містить. Які три форми реєстрації сервісів існують в IServiceCollection? Наведіть приклад кожної.
9. Розгляньте код:

```
C#
1  services.AddSingleton<ILogger, FileLogger>();
2  services.AddSingleton<ILogger, ConsoleLogger>();
3
4  var logger = provider.GetService<ILogger>();
5  var allLoggers = provider.GetServices<ILogger>();
```

Що поверне GetService<ILogger>()? Що поверне GetServices<ILogger>()? Яким чином використати allLoggers для відправки логу всім реалізаціям одночасно?

10. Поясніть різницю між AddSingleton, TryAddSingleton та AddSingleton з перевіркою. Коли TryAddSingleton захищає від подвійної реєстрації? Наведіть сценарій, де це важливо.
11. Що відбувається після виклику BuildServiceProvider()? Чи можна реєструвати нові сервіси після Build? Що таке "фіксація колекції" (locking)?
12. Що таке ValidateOnBuild? Яку помилку він виловлює ще на старті? Напишіть приклад неправильної реєстрації (де залежність не зареєстрована), що ValidateOnBuild знайде при запуску.
13. Що таке антипаттерн Service Locator? Порівняйте з Constructor DI за критеріями: явність залежностей, тестованість, відстеження залежностей. Наведіть приклад коду обома способами та поясніть, чому один кращий.
14. Коли IServiceProvider у коді є прийнятним (не антипаттерном)? Назвіть 3 легітимних сценарії і поясніть кожен.

21.3. Service Lifetimes

15. Порівняйте три lifetime-и: Singleton, Scoped, Transient. Заповніть таблицю: скільки екземплярів на процес/scope/resolve, чи потрібна thread-safety, типовий use case.
16. Розгляньте код:

```
C#
1  services.AddSingleton<MySingletonService>();
2  services.AddScoped<MyDbContext>();
3  services.AddTransient<MyHelperService>();
```

MySingletonService у конструкторі приймає MyDbContext. Що станеться? Яка ця проблема називається і чому вона небезпечна?

17. Поясніть "captive dependency" (захоплена залежність) детально: чому Singleton, що тримає Scoped, є помилкою? Що конкретно відбувається зі Scoped-сервісом у цьому випадку (чи він знищується, чи живе нескінченно)?
18. Що таке IServiceScopeFactory? Напишіть реалізацію Singleton-сервісу ReportGeneratorService, що використовує IServiceScopeFactory для правильного доступу до Scoped-залежності IReportRepository всередині методу GenerateReport().
19. Правило "lifetime залежності не може бути коротшим за lifetime споживача" — поясніть усі заборонені комбінації (Singleton→Scoped, Singleton→Transient, Scoped→Transient). Які з них дозволені і чому?
20. Розгляньте сценарій:

```
C#
1  services.AddTransient<IEmailSender, SmtpEmailSender>();
```

SmtpEmailSender реалізує IDisposable і відкриває TCP-з'єднання. Яка проблема виникне, якщо IEmailSender резолвиться з кореневого IServiceProvider (не зі scope)? Як це виправити?

21. Для кожного lifetime наведіть конкретний приклад класу в медичній системі (клінічна ІС), де він є природним вибором: Singleton → ?, Scoped → ?, Transient → ?. Обґрунтуйте кожен вибір.
22. Напишіть клас InstanceCounter з полем public int Id { get; }, що у конструкторі отримує унікальний ID через Interlocked.Increment. Зареєструйте його як Singleton, Scoped і

Transient в одному тестовому застосунку та продемонструйте різницю в ID між трьома resolve-ами.

21.4. Options Pattern

23. Що таке проблема "магічних рядків" при роботі з IConfiguration? Чому configuration["Database:ConnectionString"] є поганою практикою? Як Options Pattern вирішує цю проблему?
24. Поясніть три кроки впровадження Options Pattern: POCO клас, реєстрація в DI, отримання через конструктор. Напишіть повний приклад для DatabaseOptions з полями ConnectionString, CommandTimeout, MaxRetries.
25. Порівняйте IOptions<T>, IOptionsSnapshot<T> і IOptionsMonitor<T> за: lifetime, коли перечитує config, підтримка hot reload, використання в Singleton/Scoped. Коли кожен з них є правильним вибором?
26. Розгляньте код:

```
C#
1  public class SmtpService
2  {
3      private readonly SmtpOptions _opts;
4      public SmtpService(IOptions<SmtpOptions> options)
5      {
6          _opts = options.Value;
7      }
8  }
```

Якщо SmtpService зареєстрований як Singleton і config змінюється під час роботи — чи побачить SmtpService нові значення? Що треба змінити, щоб побачив?

27. Що таке Named Options? Напишіть приклад реєстрації двох різних конфігурацій SmtpOptions ("primary" і "backup") та отримання потрібної через IOptionsMonitor<SmtpOptions>.Get("primary").
28. Як валідувати Options за допомогою Data Annotations? Напишіть клас AppointmentOptions з атрибутами [Required], [Range] і покажіть, як підключити ValidateDataAnnotations() та ValidateOnStart() так, щоб помилки конфігурації виявлялися на старті.
29. Поясніть пріоритет джерел конфігурації (від нижчого до вищого): appsettings.json, appsettings.{Environment}.json, User

Secrets, змінні середовища, аргументи командного рядка. Навіщо саме такий порядок? Наведіть реальний сценарій, де кожен рівень корисний.

30. Дизайн задача: потрібно реалізувати перемикання між production і staging режимами без перекомпіляції коду, де staging конфігурація перекриває тільки окремі поля. Опишіть рекомендовану стратегію з файлами appsettings.json і appsettings.Staging.json, змінними середовища та чому Options Pattern тут кращий за IConfiguration напряму.

ЕЛЕКТРОННИЙ СУПРОВІД ВИДАННЯ

Друкований посібник має природне обмеження: код у ньому можна прочитати, але не можна запустити. Для дисципліни, у якій розуміння формується саме через експеримент (змінити рядок, перезапустити програму, побачити повідомлення компілятора), це суттєва втрата. Тому паралельно з підготовкою посібника автором розроблено вебресурс курсу «ООР C# Course» за адресою https://tomka.space/. Сайт не замінює книгу, а доповнює її: теоретичний матеріал поєднано в ньому з інтерактивними прикладами, циклом лабораторних робіт і допоміжними практикумами.

Теоретична частина. Розділ «Лекції» відтворює структуру посібника. Він охоплює 21 тематичний розділ: від ролі платформи .NET і основ мови до принципів SOLID, Generic Host та впровадження залежностей. Загалом це 168 теоретичних блоків, і кожен розділ завершується питаннями для самоконтролю. Сторінки мають автоматичний зміст, навігацію між підрозділами, світлу й темну теми оформлення та адаптовані для роботи як із комп'ютера, так і з мобільного пристрою.

Виконуваний код. Головна відмінність електронної версії полягає у вбудованому середовищі запуску C#. Значна частина лістингів супроводжується посиланням, що відкриває приклад у браузері. Там його можна виконати, змінити й виконати знову без встановлення .NET SDK чи середовища розробки. Це дає змогу відтворювати експерименти, описані в тексті, безпосередньо під час читання. Наприклад, можна простежити порядок виклику конструкторів у ланцюжку успадкування або переконатися, що звернення до protected-члена ззовні класу не компілюється.

Лабораторний практикум. Розділ «Лабораторні» містить 22 роботи, побудовані навколо єдиного наскрізного проєкту: консольної інформаційної системи медичної клініки. Перші дві роботи є ізольованими тренажерами. Починаючи з третьої, кожна нова тема курсу реалізується як реальна функціональність системи, у такій послідовності:

- класи пацієнтів і лікарів;
- інкапсуляція з валідацією;
- ієрархії медичних записів;
- рахунки через інтерфейси;
- узагальнені репозиторії;
- рефлексія й атрибути;
- файлове збереження та події;

- LINQ-звіти;
- текстовий інтерфейс на Spectre.Console;
- база даних на EF Core;
- асинхронність;
- архітектура на основі SOLID і DI-контейнера.

Кожна робота має єдину структуру. Вона містить мету, контекст, перелік мовних засобів, дозволених на цьому етапі, задачі зі специфікаціями та прикладами очікуваного виводу. До задач додано поетапні підказки з посиланнями на документацію Microsoft, контрольний перелік перевірки перед здачею та питання для самоперевірки. Для кожної задачі наведено таблицю адаптації до одного з шести альтернативних доменів (готель, ресторан, університет, прокат авто, бібліотека, спортзал). Це дає змогу індивідуалізувати завдання без втрати методичної цілісності.

Практики професійної розробки. Окремий Git-воркшоп навчає роботи із системою контролю версій в обсязі, потрібному для курсу. Схема така: репозиторій, гілка на кожну лабораторну, коміт на кожну задачу, злиття в основну гілку. Упродовж семестру студент формує на GitHub власну історію розвитку проєкту, яку викладач може переглянути покроково. Сторінки «Карта розвитку» та «Еволюція проєкту» показують траєкторію курсу загалом. Для кожної лабораторної там наведено нові поняття, нові файли, загальну кількість файлів і типів у проєкті, а також приклади застосування відповідних механізмів у промисловій розробці.

Рекомендації щодо використання. Посібник і сайт розраховані на спільне використання:

- теоретичний матеріал доцільно опрацьовувати за посібником, а приклади коду запускати й модифікувати в електронній версії;
- перед кожною лабораторною варто переглянути відповідні підрозділи лекцій;
- питання для самоконтролю зручно використовувати для підготовки до модульного контролю.

Викладачам, які застосовують посібник у власних курсах, сайт може слугувати готовим методичним комплексом лабораторного практикуму.

Електронний ресурс розроблено й підтримується автором самостійно. Він оновлюється разом із розвитком платформи .NET, тож може оперативно відображати зміни мови та бібліотек, яких друковане видання врахувати не може.

Томка Юрій Ярославович

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ МОВОЮ C#

Навчальний посібник
Електронне мережеве навчальне видання

Комп'ютерне верстання та макетування: Томка Ю. Я.

Спосіб поширення: відкритий доступ в електронному репозиторії та на платформі дистанційного навчання Чернівецького національного університету імені Юрія Федьковича.

Усі права захищені. Жодна частина цього електронного видання не може бути відтворена, збережена в пошукових системах або передана в будь-якій формі чи будь-якими засобами без попереднього письмового дозволу автора та закладу вищої освіти.

Видавець та виготовлювач:
Чернівецький національний університет імені Юрія Федьковича
58012, м. Чернівці, вул. Коцюбинського, 2

© Чернівецький національний університет імені Юрія Федьковича, 2026
© Томка Ю. Я., 2026