Чернівецький національний університет
імені Юрія Федьковича

Цей навчальний посібник присвячено теорії та практиці застосування парадигми функціонального програмування в межах сучасної мови C#. У книзі детально розглянуто основні концепції: незмінність даних, функції вищого порядку, зіставлення зі зразком, дискриміновані об'єднання, монадичний потік та карринг. Видання містить численні практичні приклади, порівняння підходів, а також огляд популярних функціональних бібліотек NuGet.

Матеріал орієнтований на студентів комп'ютерних наук, викладачів, а також розробників, які прагнуть розширити свій інструментарій, підвищити надійність та лаконічність власного коду за допомогою сучасних декларативних технік.

Основні теми:

- Вступ до функціональної парадигми
- Першокласні функції, делегати та лямбди
- Зіставлення зі зразком та типи-записи в C#
- Функції вищого порядку та композиція
- Дискриміновані об'єднання та обробка помилок
- Каррінг, часткове застосування та мемоізація
- Практичні бібліотеки NuGet та комплексна симуляція

Ю. Я. Томка • ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ МОВОЮ C# • 2026

ЧЕРНІВЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ЮРІЯ ФЕДЬКОВИЧА

Кафедра комп'ютерних наук

ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ МОВОЮ C#

НАВЧАЛЬНИЙ ПОСІБНИК



Ю. Я. Томка
2026

Міністерство освіти і науки України

Чернівецький національний університет імені Юрія Федьковича

Ю.Я. Томка

ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ МОВОЮ C#

Навчальний посібник

видання електронне

Чернівці

ЧЕРНІВЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ЮРІЯ ФЕДЬКОВИЧА

2026

УДК 004.432.2
Т-56

Рекомендовано до друку кафедрою комп'ютерних наук
Чернівецького національного університету імені Юрія Федьковича
(Протокол № 3 від 07.10.2026)

Рецензенти:
Прохоров Г. В., доцент кафедри програмного забезпечення комп'ютерних систем Чернівецького національного університету імені Юрія Федьковича, канд. фіз.-мат. наук, доцент
Фриз М. Є., доцент кафедри комп'ютерних наук Тернопільського національного технічного університету імені Івана Пулюя, канд. техн. наук, доцент

Томка Ю.Я.

Т-56 Функціональне програмування мовою C# : навч. посіб. / Ю.Я. Томка. Чернівці : Чернівецький національний університет імені Юрія Федьковича, 2026. 332 с.

У навчальному посібнику систематично викладено концепції функціонального програмування та їх застосування в мові C#: математичні основи (функція як відображення, лямбда-числення, композиція), незмінність даних, функції вищого порядку, згортку й моноїди, алгебраїчні типи даних і дискриміновані об'єднання, функтори та монади, каррінг, рекурсію й мемоізацію, а також можливості мови C# 15 і платформи .NET 11. Усі приклади побудовано на наскрізній предметній області гостинності (HoReCa) й зібрано у відкритому репозиторії вихідного коду.

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

Вступ 11
1. Вступ до функціонального програмування 14
1.1. Поняття парадигми програмування 15
1.1.1. Імперативна парадигма 15
1.1.2. Декларативна парадигма 15
1.2. Основні властивості функціонального програмування 16
1.2.1. Незмінність 16
1.2.2. Функції вищого порядку 16
1.2.3. Перевага виразів над інструкціями 18
1.2.4. Прозорість посилань і чисті функції 21
1.2.5. Рекурсія 24
1.2.6. Зіставлення зі зразком 25
1.2.7. Відсутність спільного змінного стану 26
1.3. Математичні основи функціонального програмування 29
1.3.1. Функція як відображення 29
1.3.2. Лямбда-числення 31
1.3.3. Функції першого класу та замикання 32
1.3.4. Композиція функцій 33
1.3.5. Складові функціонального стилю в мові C# 35
1.4. Порівняння імперативного та декларативного підходів 36
1.5. Історичні передумови функціонального програмування 37
1.6. Функціональні та мультипарадигмові мови програмування 38
1.6.1. Чисто функціональні та функціональні мови 38
1.6.2. Доцільність попереднього вивчення чисто функціональної мови 38
1.6.3. Мова F# 39
1.6.4. Мультипарадигмові мови 39
1.7. Переваги функціонального програмування 40
1.7.1. Лаконічність 40
1.7.2. Тестованість 41
1.7.3. Надійність 41
1.7.4. Передбачуваність 41
1.7.5. Підтримка паралельних обчислень 41
1.7.6. Зменшення «шуму» в коді 42
1.8. Межі застосування функціонального підходу 42
1.9. Висновки 44
1.10. Контрольні запитання 44
1.11. Практичні завдання 45
Частина I. Функціональні можливості, наявні в мові C# 47
2. Функціональні засоби, доступні без додаткових бібліотек 48
2.1. Перший функціональний код 48

2.1.1. Нефункціональний запит до меню 48
2.1.2. Функціональний запит до меню 50
2.2. Програмування, орієнтоване на результат 51
2.3. Перелічувані послідовності та відкладене виконання 52
2.4. Перевага виразів над інструкціями 55
2.4.1. Метод Select 55
2.4.2. Агрегування: від багатьох значень до одного 59
2.4.3. Рекурсивна ітерація 61
2.5. Незмінність об'єктів у мові C# 63
2.6. Повний функціональний конвеєр обробки даних 66
2.7. Рекомендації щодо розвитку навичок функціонального програмування 69
2.8. Висновки 69
2.9. Контрольні запитання 70
2.10. Практичні завдання 70
3. Еволюція функціональних можливостей мови C# 72
3.1. Огляд розвитку мови 73
3.2. C# 1.0 і 2.0: делегати, анонімні методи та ітератори 75
3.3. C# 3.0: лямбда-вирази, методи-розширення та LINQ 76
3.4. C# 4.0-6.0: параметри за замовчуванням, async/await, члени-вирази 78
3.5. C# 7.0-7.3: кортежі, зіставлення зі зразком і локальні функції 79
3.5.1. Кортежі та деконструкція 79
3.5.2. Локальні функції та вирази throw 80
3.5.3. Зіставлення зі зразком: постановка задачі 81
3.5.4. Зіставлення зі зразком у C# 7 83
3.5.5. Структури лише для читання 84
3.6. C# 8.0: вирази switch і nullable reference types 85
3.6.1. Вирази switch і шаблони властивостей 85
3.6.2. Статичні локальні функції, індекси та діапазони 87
3.6.3. Типи-посилання з підтримкою null 88
3.7. C# 9.0: типи-записи, init і нові шаблони 89
3.7.1. Реляційні та логічні шаблони 89
3.7.2. Методи доступу init 90
3.7.3. Типи-записи 90
3.7.4. Статичні лямбда-вирази 93
3.8. C# 10: розширені шаблони та лямбда-вирази як значення 94
3.8.1. Розширені шаблони властивостей 94
3.8.2. Природний тип і тип результату лямбда-виразу 95
3.9. C# 11: шаблони списків і статичні абстрактні члени 95
3.9.1. Шаблони списків 95
3.9.2. Статичні абстрактні члени інтерфейсів 97

3.9.3. Обов'язкові члени 98
3.10. C# 12: первинні конструктори, вирази колекцій і псевдоніми типів 98
3.10.1. Первинні конструктори 98
3.10.2. Вирази колекцій і незмінні колекції 99
3.10.3. Параметри лямбда-виразів за замовчуванням 100
3.10.4. Псевдоніми будь-яких типів 101
3.11. C# 13: колекції в параметрах params 101
3.12. C# 14: члени-розширення і ключове слово field 102
3.12.1. Члени-розширення 102
3.12.2. Ключове слово field 102
3.13. C# 15: типи-об'єднання і закриті ієрархії 104
3.14. Перспективи розвитку: активні шаблони 104
3.15. Висновки 105
3.16. Контрольні запитання 106
3.17. Практичні завдання 107
4. Функціональні прийоми, що спрощують код 109
4.1. Делегати Func 109
4.1.1. Делегати в колекціях 110
4.1.2. Валідатор на основі масиву правил 111
4.1.3. Зіставлення зі зразком для ранніх версій C# 113
4.2. Безпечне використання словників 115
4.3. Розбір значень 117
4.4. Власні способи перебору 119
4.4.1. Аналіз пар сусідніх елементів 119
4.4.2. Ітерація до виконання умови 121
4.5. Висновки 123
4.6. Контрольні запитання 123
4.7. Практичні завдання 124
Частина II. Поглиблене вивчення функціонального програмування 126
5. Функції вищого порядку 127
5.1. Постановка задачі: звіт про інциденти 128
5.2. Параметризація функціями та thunk 130
5.3. Ланцюжки функцій: Map 132
5.4. Комбінатор Fork 134
5.5. Комбінатор Alt 136
5.6. Композиція функцій: Compose 137
5.7. Трансдуктори: Transduce 138
5.8. Спостереження за ланцюжком: Tap 139
5.9. Обробка винятків функціями вищого порядку 140
5.10. Обробка значень null: Unless 143

5.11. Оновлення перелічуваної послідовності 145
5.12. Згортка та моноїди 146
5.12.1. Згортка як узагальнення циклу 146
5.12.2. Моноїди 148
5.13. Дерева виразів 150
5.13.1. Код як дані 150
5.13.2. Обхід і побудова дерев виразів 151
5.14. Висновки 153
5.15. Контрольні запитання 154
5.16. Практичні завдання 155
6. Дискриміновані об'єднання 157
6.1. Обмеження ієрархії успадкування 157
6.2. Дискриміноване об'єднання в мові C# 159
6.2.1. Алгебраїчні типи даних 159
6.2.2. Моделювання об'єднання ієрархією записів 160
6.2.3. Закриті ієрархії (C# 15) 163
6.2.4. Типи-об'єднання (C# 15) 163
6.3. Моделювання форм імені гостя 165
6.4. Результат пошуку в базі даних 167
6.5. Операції без результату 169
6.6. Обробка введення користувача 170
6.7. Узагальнені об'єднання 172
6.7.1. Maybe 172
6.7.2. Result 174
6.7.3. Вибір між Maybe і Result 175
6.7.4. Either 176
6.7.5. Узагальнені об'єднання засобами C# 15 177
6.8. Висновки 179
6.9. Контрольні запитання 180
6.10. Практичні завдання 180
7. Функціональний потік 182
7.1. Maybe з функціями Map і Bind 182
7.1.1. Функція Map 183
7.1.2. Залізничне програмування 185
7.1.3. Налагодження ланцюжків 186
7.2. Відмінність між Map і Bind 187
7.3. Maybe і примітивні типи 189
7.4. Maybe і журналювання 189
7.5. Maybe та асинхронні операції 191
7.6. Вкладені Maybe 192
7.7. Функтори, аплікативні функтори та монади 192
7.7.1. Закони монад 193

7.7.2. Функтор 194
7.7.3. Аплікативний функтор 194
7.7.4. Співвідношення понять 196
7.7.5. Синтаксис запитів LINQ як монадична нотація 197
7.8. Монада Reader 199
7.9. Монада State 201
7.10. Монади, що вже використовуються в C# 204
7.11. Інші монадичні структури 206
7.12. Комплексний приклад: розрахунок вартості трансферу 206
7.13. Висновки 208
7.14. Контрольні запитання 209
7.15. Практичні завдання 209
8. Власна бібліотека монад 211
8.1. Проєктування бібліотеки монад 211
8.2. Монада Either 213
8.2.1. Структура типу 213
8.2.2. Функції Map і Bind 214
8.2.3. Синтаксис запитів LINQ 216
8.3. Exceptional: результат з винятком 217
8.3.1. Структура типу 217
8.3.2. Операції над Exceptional 218
8.3.3. Аплікативне застосування Apply 220
8.4. Монада Try: відкладене обчислення 221
8.4.1. Негайне і відкладене подання обчислень 221
8.4.2. Використання Try 221
8.5. Task<T> як монада 223
8.5.1. Функції Map і Bind для Task 223
8.5.2. Відновлення після помилок 225
8.6. Поєднання монад: Task і Either 227
8.6.1. Постановка задачі 227
8.6.2. Функції для поєднаного контексту 228
8.6.3. Реалізація бізнес-логіки 228
8.6.4. Обробка результату 230
8.7. Перевірка бібліотеки та типові помилки реалізації 231
8.7.1. Закони монад 231
8.7.2. Типові помилки реалізації 232
8.8. Висновки 234
8.9. Контрольні запитання 235
8.10. Практичні завдання 235
9. Каррінг і часткове застосування 237
9.1. Поняття каррінгу 237
9.2. Каррінг функцій з багатьма параметрами 238

9.3. Каррінг і функції вищого порядку 241
9.4. Реалізація каррінгу в .NET 242
9.5. Часткове застосування 243
9.6. Реалізація часткового застосування в .NET 244
9.7. Висновки 244
9.8. Контрольні запитання 245
9.9. Практичні завдання 245
10. Нескінченні цикли 247
10.1. Постановка задачі 247
10.2. Рекурсія 249
10.3. Трамплінінг 251
10.3.1. Прихований цикл: IterateUntil 251
10.3.2. Трамплін 252
10.4. Власні ітератори 254
10.4.1. Будова перелічувача 254
10.4.2. Власні перелічувачі 255
10.4.3. Ітератор послідовних станів 257
10.4.4. Використання ітераторів станів 258
10.5. Висновки 260
10.6. Контрольні запитання 260
10.7. Практичні завдання 261
11. Мемоізація 262
11.1. Постановка задачі: граф професійних зв'язків 262
11.2. Застосування мемоізації 265
11.3. Реалізація мемоізації в мові C# 266
11.4. Обмеження мемоізації 268
11.5. Висновки 269
11.6. Контрольні запитання 269
11.7. Практичні завдання 270
Частина III. Практичне застосування 271
12. Практичне функціональне програмування в C# 272
12.1. Постановка експерименту 272
12.2. Імперативне рішення як база порівняння 274
12.3. Визначений цикл: функціональні рішення 275
12.4. Невизначений цикл: функціональні рішення 276
12.4.1. Рекурсія 276
12.4.2. Метод IterateUntil 277
12.4.3. Власний перелічувач 278
12.5. Вплив операцій введення-виведення 279
12.6. Інтерпретація результатів 281
12.7. Практичні питання впровадження функціонального стилю 281
12.7.1. Ступінь функціональності кодової бази 281

12.7.2. Структура функціонального рішення 281
12.7.3. Спільне використання функціональних засобів 282
12.7.4. Обґрунтування функціонального підходу для команди 282
12.7.5. Доцільність проєктів на F# 283
12.7.6. Межі можливостей функціонального стилю 283
12.7.7. Методика функціонального розв'язання задачі 283
12.7.8. Фрагменти з граничними вимогами до продуктивності 283
12.8. Висновки 284
12.9. Контрольні запитання 284
12.10. Практичні завдання 285
13. Функціональні бібліотеки NuGet 286
13.1. OneOf 287
13.2. LanguageExt 288
13.2.1. Option 288
13.2.2. Either 289
13.2.3. Мемоізація 290
13.2.4. Reader 290
13.2.5. State 291
13.2.6. Оцінка бібліотеки LanguageExt 291
13.3. Functional.Maybe 292
13.4. CSharpFunctionalExtensions 293
13.4.1. Maybe 293
13.4.2. Result 294
13.4.3. Бібліотеки тверджень для модульних тестів 294
13.4.4. Оцінка бібліотеки CSharpFunctionalExtensions 295
13.5. Мова F# 295
13.6. Висновки 295
13.7. Контрольні запитання 295
13.8. Практичні завдання 296
14. Комплексний приклад: симуляція «Сезон закладу» 297
14.1. Опис симуляції 297
14.2. Архітектура застосунку 298
14.3. Реалізація 298
14.3.1. Взаємодія з користувачем 298
14.3.2. Правила симуляції 300
14.3.3. Стартові закупівлі 301
14.3.4. Ігровий цикл 304
14.3.5. Фаза «Прогноз погоди» 306
14.3.6. Фаза «Вибір дії» 308
14.3.7. Фази «Підсумки тижня», «Випадкова подія» і «Кінець тижня» 311
14.3.8. Композиція застосунку 313

14.4. Напрями розвитку симуляції 315
14.5. Висновки 315
14.6. Контрольні запитання 316
14.7. Практичні завдання 316
15. Висновки та напрями подальшого вивчення 318
15.1. Узагальнення матеріалу 318
15.2. Теоретичні основи для подальшого вивчення 319
15.2.1. Стратегії обчислення 319
15.2.2. Персистентні структури даних 320
15.2.3. Системи типів і параметричність 321
15.2.4. Теорія категорій 322
15.2.5. Обчислювальні ефекти 323
15.3. Напрями подальшого вивчення 324
15.3.1. Поглиблене вивчення функціонального C# 324
15.3.2. Вивчення мови F# 324
15.3.3. Чисто функціональні мови 324
15.4. Контрольні запитання 325
15.5. Практичні завдання 325
Глосарій 326
Список використаних джерел 330

Вступ

Функціональне програмування (ФП) протягом останнього десятиліття перетворилося з нішевого напряму на одну з провідних тенденцій розвитку програмної інженерії. Функціональні засоби з'являються в усіх поширених мовах програмування, а функціональні підходи до організації коду стали стандартом у задачах обробки даних, паралельних обчислень, контейнерних і безсерверних застосунків. Для екосистеми .NET ця тенденція особливо помітна: розвиток мови C# від версії до версії супроводжується розширенням функціональних можливостей, чому значною мірою сприяє мова F#, що використовує спільне з C# середовище виконання.

Водночас для розробника на C# опанування функціонального програмування не вимагає переходу на нову мову чи використання сторонніх бібліотек: значну частину функціонального стилю можна реалізувати стандартними засобами мови. Саме такий шлях покладено в основу посібника - від знайомих засобів C# до фундаментальних концепцій функціонального програмування, з постійним наголосом на їх практичній користі для промислової розробки. До таких переваг належать:

- чистіший, стисліший і зрозуміліший код;
- простіший супровід кодової бази;
- менша ймовірність необроблених виняткових і непередбачуваної поведінки;
- висока тестованість коду.

Адресат посібника. Посібник призначений для студентів закладів вищої освіти, що навчаються за спеціальностями галузі інформаційних технологій, а також для розробників програмного забезпечення, які володіють основами мови C# і прагнуть опанувати функціональний стиль програмування. Від читача очікується вміння самостійно створювати нескладні застосунки мовою C#; складніші питання платформи .NET пояснюються в тексті в міру їх появи.

Першоджерело. Структура і послідовність викладу посібника ґрунтуються на книзі S. J. Painter «Functional Programming with C#» (O'Reilly Media, 2023). Матеріал першоджерела адаптовано до вимог навчального видання: виклад переосмислено відповідно до академічного стилю, фактичні неточності першоджерела виправлено, відомості про мову та платформу актуалізовано відповідно до .NET 11 і C# 15. Порівняно з першоджерелом посібник доповнено теоретичними підрозділами (математичні основи функціонального програмування, алгебраїчні типи даних, згортка й моноїди, функтори та аплікативні функтори), матеріалом про дерева виразів, нові можливості мови C# 11-15 і вбудовані типи-об'єднання, а також глосарієм. Усі приклади програмного коду розроблено заново на основі єдиної предметної області, а результати

вимірювань продуктивності отримано власними експериментами. Матеріал розділу 8 про проєктування власної бібліотеки монад підготовлено з додатковим використанням книги C. Bueno «C# Functional: Monads from Zero to Hero» (2024).

Предметна область прикладів. Усі приклади посібника побудовано на єдиній предметній області - інформаційній системі мережі закладів гостинності HoReCa (Hotel, Restaurant, Café): готелю, ресторану та кав'ярні. Наскрізна предметна область дає змогу зосередитися на функціональних прийомах, не відволікаючись на щоразу нові контексти, і демонструє застосування функціонального програмування до реалістичних задач: обробки замовлень і рахунків, бронювання номерів, обліку складу, роботи з постачальниками тощо.

Структура посібника. Посібник складається з трьох частин і п'ятнадцяти розділів:

- розділ 1 вводить поняття функціонального програмування, його основні властивості, історію, переваги та межі застосування;
- частина I (розділи 2-4) присвячена функціональним засобам, доступним у стандартній мові C#: LINQ і відкладеному виконанню, еволюції функціональних можливостей мови від C# 1.0 до C# 15, прийомам з делегатами та перелічуваними послідовностями;
- частина II (розділи 5-11) розглядає поглиблені концепції: функції вищого порядку та комбінатори, дискриміновані об'єднання, монади, проєктування власної бібліотеки монад (*Either*, *Exceptional*, *Try*, монадичні операції для *Task*), каррінг і часткове застосування, невизначені цикли, мемоізацію;
- частина III (розділи 12-15) узагальнює матеріал і розглядає практичні аспекти: продуктивність функціонального коду, сторонні бібліотеки, побудову завершеного застосунку, напрями подальшого вивчення.

Наприкінці посібника наведено глосарій основних термінів з посиланнями на підрозділи, у яких їх розглянуто.

Кожен розділ містить мету та очікувані результати навчання, теоретичний матеріал із прикладами, висновки, контрольні запитання та практичні завдання. Розділи доцільно опрацьовувати послідовно, оскільки кожен наступний спирається на попередні; водночас підготовлений читач може розпочати з тієї частини, що відповідає рівню його підготовки.

Програмний код. Вихідний код усіх прикладів зібрано в репозиторії *FunctionalCSharp.HoReCa* - рішенні .NET 11, що містить спільну бібліотеку предметної області *HoReCa.Domain* та окремий проєкт для кожного розділу. Усі лістинги посібника є фрагментами коду, що компілюється, а весь консольний вивід отримано реальним виконанням програм. Файл

README.md репозиторію містить відповідність між номерами лістингів і файлами вихідного коду.

Умовні позначення. Лістинги програмного коду наводяться з нумерацією рядків і мають наскрізну в межах розділу нумерацію (наприклад, «Лістинг 5.3»). Консольний вивід програм виділено окремими блоками. Імена типів, методів, змінних і ключові слова мови в тексті виділено курсивом. Терміни під час першої появи виділено напівжирним шрифтом і супроводжено англійським відповідником.

Автори висловлюють подяку студентам, які брали участь у підготовці робочого перекладу першоджерела в межах індивідуальних навчально-дослідних завдань.

1. Вступ до функціонального програмування

Переважна більшість курсів програмування, зокрема курси з мов C#, Java, Python та Visual Basic, побудована на основі об'єктно-орієнтованої парадигми, яка нині є домінантною в промисловій розробці програмного забезпечення. Об'єктно-орієнтоване програмування (ООП) ґрунтується на поєднанні даних (властивостей) і поведінки (методів) у логічні одиниці - класи, що слугують шаблонами для створення об'єктів. До його ключових механізмів належать інкапсуляція, успадкування та поліморфізм.

Предметом цього посібника є альтернативний стиль побудови програм - **функціональне програмування** (functional programming, ФП). Попри зростання інтересу до нього протягом останнього десятиліття, функціональне програмування не є новим напрямом: його математичні засади були сформовані в період від кінця XIX століття до 1950-х років, а перші мови з підтримкою функціонального стилю з'явилися наприкінці 1950-х років. У посібнику розглядається реалізація принципів ФП засобами мови C# без потреби опановувати іншу мову програмування.

Під час вивчення матеріалу доцільно тимчасово абстрагуватися від звичних об'єктно-орієнтованих прийомів: функціональний підхід потребує іншого способу мислення, і пряме перенесення досвіду ООП нерідко ускладнює його засвоєння.

Мета розділу - сформувати уявлення про функціональне програмування як парадигму, визначити його основні властивості, історичні передумови, переваги та межі застосування.

Після опрацювання розділу студент повинен:

- знати відмінності між імперативною та декларативною парадигмами програмування;
- розуміти зміст понять незмінності, функції вищого порядку, виразу та інструкції, чистої функції, прозорості посилань, рекурсії, зіставлення зі зразком;
- уміти розпізнавати чисті та нечисті функції в коді мовою C#;
- орієнтуватися в історії розвитку функціонального програмування та в сучасних мовах, що його підтримують;
- обґрунтовувати доцільність застосування функціонального підходу для конкретного класу задач;
- знати математичне поняття функції, відмінність між тотальними й частковими функціями, основні конструкції лямбда-числення, поняття замикання та властивості композиції функцій.

Усі приклади посібника побудовано на єдиній предметній області - інформаційній системі закладів сфери гостинності HoReCa (Hotel, Restaurant, Café): готелю, ресторану та кав'ярні. Вихідний код прикладів

наведено в репозиторії *FunctionalCSharp.HoReCa*; приклади цього розділу містяться в проєкті *Ch01.Introduction*.

1.1. Поняття парадигми програмування

Функціональне програмування не є ні окремою мовою, ні бібліотекою, яку можна підключити до проєкту через менеджер пакетів NuGet. Воно є **парадигмою програмування** (programming paradigm) - сукупністю принципів, понять і прийомів, що визначають стиль побудови програм. Одна й та сама мова може підтримувати кілька парадигм, і в межах одного програмного продукту їх можна поєднувати, використовуючи сильні сторони кожної.

Теоретичні засади функціонального програмування сформувалися раніше, ніж об'єктно-орієнтоване програмування, і навіть раніше, ніж з'явилися перші електронні обчислювальні машини (див. підрозділ 1.5).

Існує значна кількість парадигм програмування, однак у контексті цього посібника достатньо розглянути дві найпоширеніші - імперативну та декларативну.

1.1.1. Імперативна парадигма

Імперативна парадигма (imperative paradigm) передбачає, що програма є послідовністю команд, які детально визначають, *як* саме слід досягти результату: які змінні зберігають проміжні значення, у якому порядку виконуються кроки, за яких умов змінюється хід виконання. Тривалий час ця парадигма була фактично єдиною; до неї належать процедурне та об'єктно-орієнтоване програмування. Саме в імперативному стилі здебільшого відбувається початкове навчання програмуванню.

1.1.2. Декларативна парадигма

Декларативна парадигма (declarative paradigm) зосереджується на описі того, *що* має бути отримано в результаті, а не на детальному алгоритмі отримання результату. Такі деталі, як порядок виконання окремих кроків, значною мірою покладаються на середовище виконання. До декларативної парадигми належить функціональне програмування, а також мова структурованих запитів SQL.

Аналогія з SQL є показовою. Під час написання запиту

SELECT ... FROM ... WHERE ... ORDER BY ...

розробник не визначає ні фізичного порядку виконання операцій (СУБД не обов'язково спочатку виконує вибірку, потім фільтрацію, а потім сортування), ні способу перетворення даних, а лише описує бажаний результат. Аналогічної мети прагне функціональний стиль у мові C#, тому фахівцям із досвідом роботи з реляційними базами даних засвоєння подальшого матеріалу зазвичай дається легше.

1.2. Основні властивості функціонального програмування

У цьому підрозділі розглядаються ключові властивості функціонального програмування та їхнє практичне значення для розробника.

1.2.1. Незмінність

Об'єкт, стан якого може змінюватися після створення, називають **змінним** (mutable). Об'єкт, стан якого після створення змінити неможливо, називають **незмінним** (immutable). У програмуванні принцип незмінності (immutability) означає, що значення змінної встановлюється під час її оголошення і надалі не змінюється. Якщо потрібне нове значення, створюється нова змінна, обчислена на основі попередньої. Саме так трактуються всі змінні у функціональному коді.

Такий спосіб роботи відрізняється від імперативного, проте він наближає програму до послідовного розв'язання задачі в математиці, де кожен крок спирається на результати попередніх і не змінює їх. Це сприяє кращій структурованості коду та підвищує його передбачуваність, а отже, і надійність.

У платформі .NET незмінними є, зокрема, типи *String* і *DateTime*. Операції, які на перший погляд «змінюють» рядок чи дату, насправді створюють новий екземпляр, а вихідний залишається незмінним. Саме тому багаторазове конкатенування рядків у циклі є неефективним: на кожній ітерації в пам'яті створюється новий рядок, а для накопичення тексту слід використовувати клас *StringBuilder*.

1.2.2. Функції вищого порядку

Функція вищого порядку (higher-order function) - це функція, яка приймає інші функції як параметри або повертає функцію як результат. Для того щоб функцію можна було передавати як значення, вона має бути подана у вигляді змінної - локальної змінної, параметра чи значення, що повертається. У мові C# для цього використовуються узагальнені делегатні типи *Func<T, TResult>* і *Action<T>*.

Обидва типи описують функцію, збережену як значення. Їхні узагальнені параметри визначають типи вхідних параметрів і (для *Func*) тип результату. Відмінність полягає в тому, що делегат *Action* не повертає значення (відповідає методу з типом результату *void*), тоді як в делегаті *Func* останній узагальнений параметр завжди визначає тип результату.

У лістингу 1.1 наведено два методи класу, що використовуються під час формування рахунку в ресторані.

Лістинг 1.1 - Формування рядка рахунку за допомогою методів класу

```
// Для ціни 185 і кількості 2 повертає рядок
// "185 грн × 2 = 370 грн"
public string ComposeLine(decimal price, int quantity)
{
    return price + " грн × " + quantity + " = "
        + (price * quantity) + " грн";
}

public void LogLine(string line)
 {
     _log.Add("додано до рахунку: " + line);
 }
```

Ті самі дії можна подати у вигляді значень делегатних типів (лістинг 1.2).

Лістинг 1.2 - Подання дій у вигляді делегатів Func та Action

```
public static readonly Func<decimal, int, string> ComposeLine =
    (price, quantity) =>
        $"{price} грн × {quantity} = {price * quantity} грн";

public static readonly Action<string> LogLine =
    line => Console.WriteLine($"додано до рахунку: {line}");
```

Виклик делегатів синтаксично не відрізняється від виклику звичайних методів (лістинг 1.3).

Лістинг 1.3 - Виклик делегатів

```
var line = ComposeLine(185m, 2);
LogLine(line);
```

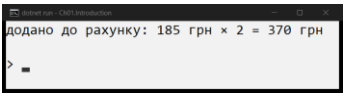


Рисунок 1.1 - Результат виконання прикладу «Функції як значення» (фрагмент)

Головна перевага делегатних типів полягає в тому, що вони зберігаються у змінних і можуть передаватися між частинами програми як звичайні дані - як параметри інших функцій або як результати, що повертаються. У лістингу 1.4 функція *FormatBill* є функцією вищого порядку: спосіб форматування рядка рахунку вона отримує ззовні як параметр *format*.

Лістинг 1.4 - Функція вищого порядку FormatBill

```
// Функція форматування передається як параметр
public static IEnumerable<string> FormatBill(
    IEnumerable<OrderLine> lines,
    Func<decimal, int, string> format) =>
    lines.Select(l => $"{l.Item.Name}: "
                      + format(l.Item.Price, l.Quantity));
```

Завдяки цьому одна й та сама функція формує рахунок у різних форматах залежно від переданого делегата (лістинг 1.5).

Лістинг 1.5 - Передавання різних функцій форматування

```
1. OrderLine[] order =
2. [
3.     new(SampleData.Item("U01"), 2),
4.     new(SampleData.Item("B02"), 3)
5. ];
6.
7. // Той самий рахунок у двох різних форматах
8. Func<decimal, int, string> brief =
9.     (price, quantity) => $"{price * quantity} грн";
10.
11.  foreach (var s in FormatBill(order, ComposeLine))
12.      Console.WriteLine(s);
13.  foreach (var s in FormatBill(order, brief))
14.      Console.WriteLine(s);
```

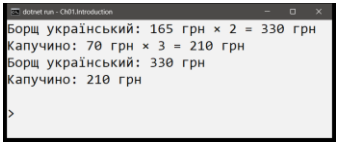


Рисунок 1.2 - Результат виконання прикладу «Функції як значення» (фрагмент)

Засобами функціонального програмування делегати можна **компонувати** (compose), створюючи складніші функції з менших, простіших функціональних блоків. Саме з цим пов'язана назва парадигми: програма будується з функцій як основних будівельних елементів. Тому на більшість запитань щодо способу побудови функціонального коду відповідь, як правило, полягає у виділенні, передаванні та композиції функцій.

Примітка. У термінології багатьох джерел розрізняють *функції* - фрагменти коду, що повертають значення, та *методи* (процедури), що значення не повертають (у мові C# - методи з типом результату *void*). Оскільки метод без результату має сенс лише за наявності побічного ефекту, у функціональному коді такі методи використовують лише там, де без них обійтися неможливо. Прикладом обґрунтованого застосування є журналювання (логування), яке є необхідним елементом промислового програмного забезпечення.

1.2.3. Перевага виразів над інструкціями

Вираз (expression) - це фрагмент коду, результатом обчислення якого є значення. У найпростішому випадку виразами є оголошення констант (лістинг 1.6).

Лістинг 1.6 - Найпростіші вирази

```
1. const int SeatsPerTable = 4;
2. const int SeatsInHall = SeatsPerTable * 12;
```

Вираз може залежати від вхідних значень, як у функції обчислення суми з урахуванням сервісного збору (лістинг 1.7).

Лістинг 1.7 - Вираз, що залежить від параметра

```
1. public static decimal AddServiceCharge(decimal amount) =>
2.     amount + amount * 0.10m;
```

Функція в лістингу 1.8 перевіряє умову, однак результатом її роботи є значення типу *bool*, тому вона також є виразом.

Лістинг 1.8 - Вираз логічного типу

```
1. public static bool IsLargeParty(int guests) => guests > 8;
```

Тернарний умовний оператор (скорочена форма інструкції *if*) також є виразом, якщо він використовується виключно для визначення значення, що повертається (лістинг 1.9).

Лістинг 1.9 - Тернарний оператор як вираз

```
1. public static string SeatingMessage(int guests) =>
2.     guests > 8
3.         ? "потрібен банкетний стіл"
4.         : "достатньо звичайного столика";
```

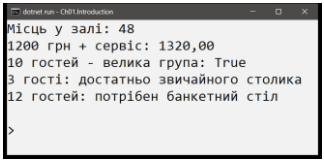


Рисунок 1.3 - Результат виконання прикладу «Вирази та інструкції» (фрагмент)

Інструкція (statement), навпаки, - це фрагмент коду, який не обчислює значення, а наказує виконати певну дію. Інструкції змінюють порядок виконання програми (ключові слова *if*, *for*, *foreach*, *while*) або викликають методи, що не повертають результату й діють через побічні ефекти (лістинг 1.10).

Лістинг 1.10 - Інструкція: виклик методу без результату

```
1. printer.PrintTicket("Стіл 7: борщ × 2");
```

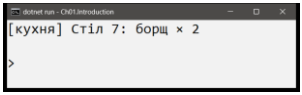


Рисунок 1.4 - Результат виконання прикладу «Вирази та інструкції» (фрагмент)

Для попереднього розрізнення виразів та інструкцій М. Земанн (M. Seemann) запропонував просте емпіричне правило: рядок коду, що містить знак присвоєння, імовірно, є виразом, оскільки він надає значення певній змінній; рядок без знака присвоєння напевно є інструкцією. Правило має обмеження - виклик іншої функції в правій частині присвоєння може спричиняти непередбачувані наслідки, - однак як орієнтир воно є корисним.

Програмування на основі виразів

Програмування на основі виразів (expression-based programming) можна порівняти з оформленням розв'язку математичної задачі: кожен рядок є завершеним обчисленням, що спирається на результати попередніх рядків і не змінює їх. Проміжні результати зберігаються незмінними протягом усього виконання функції, що, зокрема, спрощує налагодження: будь-яке попереднє значення можна переглянути з упевненістю, що його не було змінено на наступних кроках.

У лістингу 1.11 обчислення підсумку рахунку з урахуванням сервісного збору та знижки побудовано як послідовність виразів, кожен з яких має власне змістове ім'я.

Лістинг 1.11 - Обчислення підсумку рахунку на основі виразів

```
1. public static decimal CalculateTotal(
2.     decimal subtotal, decimal serviceRate, decimal discountRate)
3. {
4.     var service = subtotal * serviceRate;
5.     var gross = subtotal + service;
6.     var discount = gross * discountRate;
7.     var total = Math.Round(gross - discount, 2);
8.     return total;
9. }
```

Той самий результат можна отримати з повторним використанням однієї змінної (лістинг 1.12).

Лістинг 1.12 - Обчислення з повторним присвоєнням змінної

```
1. public static decimal CalculateTotalMutable(
2.     decimal subtotal, decimal serviceRate, decimal discountRate)
3. {
4.     var result = subtotal;
5.     result += result * serviceRate;
6.     result -= result * discountRate;
7.     result = Math.Round(result, 2);
8.     return result;
9. }
```



Рисунок 1.5 - Результат виконання прикладу «Програмування на основі виразів»

Обидві функції повертають однаковий результат, однак другий варіант гірше читається через відсутність змістових імен, а всі проміжні значення в ньому втрачаються: у разі помилки доведеться покроково відстежувати значення змінної *result* на кожному етапі. У першому варіанті вся послідовність обчислень залишається доступною для аналізу.

1.2.4. Прозорість посилань і чисті функції

Чиста функція (pure function) - це функція, яка має такі властивості:

1) не змінює нічого за своїми межами: не оновлює стан об'єктів, не записує файли, не звертається до зовнішніх ресурсів;

2) для однакових значень параметрів завжди повертає однаковий результат незалежно від стану системи;

3) не має неочікуваних побічних ефектів (side effects), до яких належить і генерування винятків.

Термін **прозорість посилань** (referential transparency) відображає наслідок цих властивостей: оскільки для однакових вхідних даних результат завжди однаковий, виклик функції в будь-якому обчисленні можна замінити значенням, яке вона повертає, без зміни поведінки програми (лістинг 1.13).

Лістинг 1.13 - Прозорість посилань

```
1. var lineTotal = (decimal price, int quantity) => price * quantity;
2. var borschForTwo = lineTotal(165m, 2);
```

Виклик *lineTotal(165m, 2)* завжди дає значення 330 без жодних побічних ефектів, тому в принципі його можна замінити константою 330. Саме ця властивість і називається прозорістю посилань.

У лістингу 1.14 наведено приклади чистих функцій.

Лістинг 1.14 - Чисті функції

```
1. public static decimal LineTotal(decimal price, int quantity) =>
2.     price * quantity;
3.
4. public static string Greet(string? guestName) =>
5.     "Вітаємо, " +
6.     (string.IsNullOrWhiteSpace(guestName)
7.         ? "шановний гостю!"
8.         : guestName + "!");
```

Функція *Greet* враховує випадок порожнього або відсутнього імені гостя, тому вона не генерує винятків і не змінює нічого поза своїми межами - вона лише обчислює та повертає нове значення.

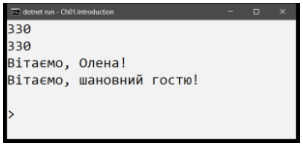


Рисунок 1.6 - Результат виконання прикладу «Чисті й нечисті функції» (фрагмент)

У лістингу 1.15 наведено нечисті варіанти аналогічних функцій.

Лістинг 1.15 - Нечисті функції

```
1. public class Waiter
2. {
3.     private decimal _total;
4.     public decimal Total => _total;
5.     public string GuestName { get; set; } = "";
6.
7.     public void Add(decimal amount) => _total += amount; // змінює стан
8.
9.     public string Greet() => "Вітаємо, " + GuestName + "!";
10.      // читає стан об'єкта замість параметра
11.  }
```

Обидва методи класу *Waiter* звертаються до даних об'єкта, що перебувають за межами самої функції. Метод *Add* змінює внутрішній стан об'єкта, а метод *Greet* отримує ім'я не з параметра, а зі змінної властивості, значення якої може бути будь-яким на момент виклику, зокрема порожнім. Тому жоден із цих методів не можна вважати чистим.

Складніші випадки наведено в лістингу 1.16.

Лістинг 1.16 - Функції, чистота яких є сумнівною

```
1. public string Greet1() => "Вітаємо, " + GetGuestName() + "!";
2.
3. public string Greet2(GuestCard card)
4. {
5.     card.Greeted = true;
6.     return "Вітаємо, " + (card?.Name ?? "шановний гостю") + "!";
7. }
8.
9. public string Greet3(string? name) =>
10.      DateTime.Now.ToString("HH:mm") + " - Вітаємо, "
11.      + (name ?? "шановний гостю") + "!";
```

Жоден із цих методів, найімовірніше, не є чистим:

- метод *Greet1* залежить від іншого методу *GetGuestName*, поведінка якого невідома в точці виклику. Якщо він повертає константу, *Greet1* можна вважати чистим; якщо ж він звертається до бази даних, то

відсутність запису чи збій мережі можуть спричинити виняток, тобто неочікуваний побічний ефект;

- метод *Greet2* змінює об'єкт, переданий як параметр (встановлює властивість *Greeted*), що є явним побічним ефектом. Для ООП така практика є поширеною, але у функціональному програмуванні вона неприпустима;

- метод *Greet3* використовує властивість *DateTime.Now*, значення якої змінюється з кожним викликом, тому для однакових параметрів метод повертає різні результати.

Побічний ефект методу *Greet2* підтверджує виконання програми: значення властивості *Greeted* переданого об'єкта змінюється внаслідок виклику.

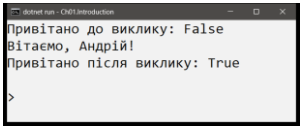


Рисунок 1.7 - Результат виконання прикладу «Чисті й нечисті функції» (фрагмент)

Усі три методи можна перетворити на чисті (лістинг 1.17). Залежність від зовнішнього джерела даних у *Greet1* передається через параметр-делегат *Func<string>*; оновлення картки гостя в *Greet2* виноситься в окрему функцію, яка повертає новий об'єкт; поточний час у *Greet3* передається як параметр.

Лістинг 1.17 - Перетворення функцій на чисті

```
1. // Джерело імені передається як функція (делегат)
2. public static string Greet1(Func<string> getGuestName) =>
3.     "Вітаємо, " + getGuestName() + "!";
4.
5. // Оновлення картки гостя винесено в окрему функцію,
6. // яка повертає нову картку, а не змінює вихідну
7. public static GuestCard MarkGreeted(GuestCard card) =>
8.     new() { Name = card.Name, Greeted = true };
9.
10.  public static string Greet2(GuestCard card) =>
11.      "Вітаємо, " + card.Name + "!";
12.
13.  // Поточний час передається як параметр
14.  public static string Greet3(string? name, DateTime moment) =>
15.      moment.ToString("HH:mm") + " - Вітаємо, "
16.      + (name ?? "шановний гостю") + "!";
```

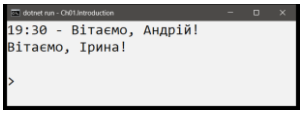


Рисунок 1.8 - Результат виконання прикладу «Чисті й нечисті функції» (фрагмент)

Прозорість посилань суттєво підвищує **тестованість** функціонального коду: результат чистої функції визначається лише її параметрами, тому для перевірки достатньо задати вхідні дані й порівняти результат з очікуваним. Водночас відмова від змінного стану потребує інших способів його відстеження, які розглядаються в наступних розділах.

Ступінь «чистоти» реальної програми завжди обмежений: взаємодія з користувачем, файловою системою, базою даних, мережею або сторонніми бібліотеками, що не дотримуються функціональної парадигми, неминуче породжує побічні ефекти. Тому в мові C# доводиться шукати компроміс. Архітектурний підхід до такого компромісу Г. Бернгардт (G. Bernhardt) назвав «функціональне ядро, імперативна оболонка» (Functional Core, Imperative Shell): бізнес-логіка зосереджується в ядрі, що складається з чистих функцій і незмінних даних, а всі взаємодії із зовнішнім світом виносяться в тонку зовнішню оболонку (рис. 1.9). Завдання розробника полягає в тому, щоб максимально розширити чисте ядро й мінімізувати оболонку.



Рисунок 1.9 - Архітектура «функціональне ядро, імперативна оболонка»

1.2.5. Рекурсія

Рекурсія (recursion) - це спосіб організації обчислень, за якого функція викликає саму себе для реалізації циклу з наперед невідомою кількістю повторень. Рекурсивні алгоритми використовуються, зокрема, для обходу

деревоподібних структур (наприклад, каталогів файлової системи) та в ефективних алгоритмах сортування.

Рекурсивна функція, як правило, складається з двох частин:

- **умови завершення** (base case), що визначає, чи досягнуто кінцевого стану (знайдено шукане значення, оброблено всі елементи тощо);
- **інструкції повернення**, яка залежно від умови або повертає кінцеве значення, або повторно викликає ту саму функцію з іншими аргументами.

У лістингу 1.18 наведено рекурсивну функцію, яка визначає кількість столиків, потрібних для розміщення групи гостей.

Лістинг 1.18 - Рекурсивне обчислення кількості столиків

```
1. // Скільки столиків потрібно, щоб розсадити гостей
2. public static int TablesNeeded(int guestsLeft, int seatsPerTable)
3. {
4.     if (guestsLeft <= 0)
5.         return 0;
6.     else
7.         return 1 + TablesNeeded(guestsLeft - seatsPerTable,
8.                                 seatsPerTable);
9. }
```

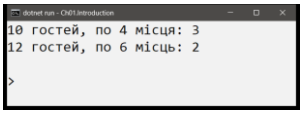


Рисунок 1.10 - Результат виконання прикладу «Рекурсія»

Приклад навмисно спрощено (у реальній програмі кількість столиків доцільніше обчислювати арифметично), однак він ілюструє важливу особливість: значення параметрів жодного разу не змінюються. Кожен рекурсивний виклик отримує нові аргументи, обчислені як вирази на основі власних параметрів. Це ще один прояв принципу незмінності.

Рекурсія є одним із засобів, якими функціональне програмування замінює цикли *while* і *foreach*. Водночас у мові C# глибока рекурсія пов'язана з ризиком переповнення стека та певними витратами продуктивності; ці питання детально розглядаються в розділі 10.

1.2.6. Зіставлення зі зразком

Зіставлення зі зразком (pattern matching) у мові C# - це розширення можливостей конструкції *switch*, яке дає змогу обирати шлях виконання залежно від типу об'єкта, значень його властивостей або комбінації кількох значень. Повноцінну реалізацію цього механізму у вигляді виразів *switch* (switch expressions) було додано у версії C# 8, і в кожній наступній версії мови його можливості розширюються. У мові F# цей механізм розвинений ще більше.

Зіставлення зі зразком дає змогу замінити громіздкі вкладені конструкції *if*. У лістингу 1.19 вартість проживання в номері готелю за одну ніч визначається залежно від категорії номера та дня тижня.

Лістинг 1.19 - Визначення тарифу вкладеними інструкціями if

```
1. public static decimal NightlyRateIf(RoomType type, bool isWeekend)
2. {
3.     if (type == RoomType.Suite)
4.     {
5.         if (isWeekend)
6.             return 4200m;
7.         else
8.             return 3600m;
9.     }
10.      if (type == RoomType.Superior)
11.          return 2400m;
12.      if (isWeekend)
13.          return 2100m;
14.      else
15.          return 1800m;
16.  }
```

Та сама логіка за допомогою виразу *switch* над кортежем подається значно компактніше й наочніше (лістинг 1.20).

Лістинг 1.20 - Визначення тарифу виразом switch

```
1. public static decimal NightlyRate(RoomType type, bool isWeekend) =>
2.     (type, isWeekend) switch
3.     {
4.         (RoomType.Suite, true)    => 4200m,
5.         (RoomType.Suite, _)       => 3600m,
6.         (RoomType.Superior, _)    => 2400m,
7.         (RoomType.Standard, true) => 2100m,
8.         _                         => 1800m
9.     };
```

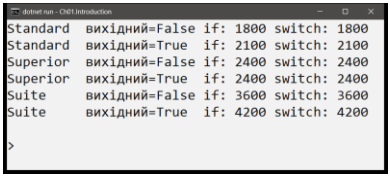


Рисунок 1.11 - Результат виконання прикладу «Зіставлення зі зразком»

Результати обох функцій збігаються для всіх комбінацій вхідних даних. Детально зіставлення зі зразком розглядається в розділі 3, а способи його імітації в ранніх версіях мови C# - у розділі 4.

1.2.7. Відсутність спільного змінного стану

Об'єктно-орієнтований код, як правило, містить **об'єкти стану**, що відображають певний реальний або віртуальний процес і періодично оновлюються, щоб відповідати поточному стану цього процесу. У

лістингу 1.21 клас *ShiftState* описує поточну зміну ресторану, а клас *ShiftRepository* змінює її стан.

Лістинг 1.21 - Змінний об'єкт стану зміни ресторану

```
1. public class ShiftState
2. {
3.     public int ShiftNumber { get; set; }
4.     public string ChefOnDuty { get; set; } = "";
5.     public int OrdersServed { get; set; }
6.     public decimal Revenue { get; set; }
7. }
8.
9. public class ShiftRepository
10.  {
11.      private readonly ShiftState _state;
12.
13.      public ShiftRepository(ShiftState initialState)
14.      {
15.          _state = initialState;
16.      }
17.
18.      public void RegisterOrder(decimal amount)
19.      {
20.          _state.OrdersServed++;
21.          _state.Revenue += amount;
22.      }
23.
24.      public void ChangeChef(string newChef)
25.      {
26.          _state.ShiftNumber++;
27.          _state.ChefOnDuty = newChef;
28.      }
29.  }
```

У функціональному програмуванні центрального об'єкта стану, властивості якого змінюються, не існує. Стан системи, строго кажучи, присутній, проте він є не змінною сутністю, а результатом послідовності перетворень. Такий підхід, зокрема, реалізовано в бібліотеці Redux для React, яка, своєю чергою, запозичила його з функціональної мови Elm: стан застосунку є незмінним об'єктом, а для його «оновлення» визначається функція, що отримує попередній стан і команду та повертає новий об'єкт стану. У мові C# такий підхід значно спростився з появою типів-записів (record types) у версії C# 9 (див. розділ 3).

У лістингу 1.22 метод заміни шеф-кухаря переписано у функціональному стилі: замість зміни наявного об'єкта створюється новий.

Лістинг 1.22 - Функція, що повертає новий стан

```
1. public static ShiftState ChangeChef(ShiftState oldState,
2.                                     string newChef)
3. {
4.     return new ShiftState
5.     {
6.         ShiftNumber = oldState.ShiftNumber + 1,
7.         ChefOnDuty = newChef,
8.         OrdersServed = oldState.OrdersServed,
9.         Revenue = oldState.Revenue
10.     };
11. }
```

Головною перевагою такого підходу є передбачуваність. У мові C# об'єкти класів передаються за посиланням, тому зміна об'єкта всередині методу відображається і за його межами. Отже, якщо об'єкт передано в метод як параметр, немає гарантії, що після виклику він залишиться незмінним, навіть якщо йому явно не присвоювалося нове значення (лістинг 1.23).

Лістинг 1.23 - Зміна об'єкта через спільне посилання

```
1. var morning = new ShiftState
2. {
3.     ShiftNumber = 1, ChefOnDuty = "Марченко"
4. };
5. var repository = new ShiftRepository(morning);
6. repository.RegisterOrder(520m);
7. repository.ChangeChef("Коваль");
8.
9. // Об'єкт morning змінився "ззовні"
10.  Console.WriteLine(
11.      $"morning: зміна {morning.ShiftNumber}, " +
12.      $"шеф {morning.ChefOnDuty}, виручка {morning.Revenue}");
```

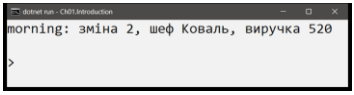


Рисунок 1.12 - Результат виконання прикладу «Стан без змін» (фрагмент)

Об'єкт *morning* змінився, хоча в коді, який його створив, жодних присвоєнь його властивостям не виконувалося. У функціональному варіанті (лістинг 1.24) вихідний об'єкт залишається незмінним, а новий стан отримується явно.

Лістинг 1.24 - Отримання нового стану без зміни попереднього

```
1. var first = new ShiftState
2. {
3.     ShiftNumber = 1, ChefOnDuty = "Марченко",
4.     OrdersServed = 1, Revenue = 520m
5. };
6. var second = ChangeChef(first, "Коваль");
7.
8. Console.WriteLine(
9.     $"first: зміна {first.ShiftNumber}, шеф {first.ChefOnDuty}");
10.  Console.WriteLine(
11.      $"second: зміна {second.ShiftNumber}, шеф {second.ChefOnDuty}");
```

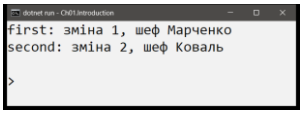


Рисунок 1.13 - Результат виконання прикладу «Стан без змін» (фрагмент)

У функціональному програмуванні будь-яка зміна даних виконується лише через явне створення нового значення, тому не виникає неоднозначності щодо того, чи було змінено той чи інший об'єкт.

1.3. Математичні основи функціонального програмування

Властивості, розглянуті в підрозділі 1.2, мають спільне джерело - математичне поняття функції. Функціональне програмування переносить у програмування не лише термін «функція», а й правила роботи з функціями, усталені в математиці: функції можна поєднувати, передавати як аргументи й отримувати як результат, а обчислення виконувати шляхом підставлення виразів. У цьому підрозділі коротко розглядаються поняття, на які спирається решта посібника.

1.3.1. Функція як відображення

У математиці **функція** f: A -> B - це відображення, яке кожному елементу множини A (**області визначення**, domain) ставить у відповідність **рівно один** елемент множини B (**області значень**, codomain). Наприклад, меню закладу задає функцію, що ставить у відповідність кожній страві її ціну: кожна страва має одну ціну, хоча різні страви можуть коштувати однаково. Відповідність, у якій одному елементу відповідає кілька значень, функцією не є (рис. 1.14).

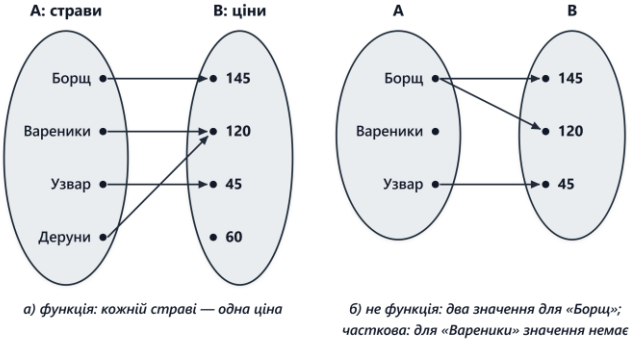


Рисунок 1.14 - Функція як відображення множин

У мові C# множинам відповідають типи, а запис f: A -> B - типу делегата *Func<A, B>*. Математична функція не має «побічних дій»: вона лише ставить у відповідність значення. Саме тому чиста функція (підрозділ 1.2.4) є програмним аналогом математичної функції, тоді як метод, результат якого залежить від зовнішнього стану, функцією в математичному сенсі не є.

Функцію, визначену для всіх елементів області визначення, називають **тотальною** (total), а визначену лише для частини з них - **частковою** (partial). Багато методів, що здаються функціями, насправді часткові: ділення не визначене для нульового дільника, *int.Parse* - для рядка, що не є числом, пошук у словнику - для відсутнього ключа. Часткова функція «повідомляє» про невизначеність винятком, тобто побічним ефектом, який не відображено в її типі. Функціональний підхід надає перевагу тотальним функціям, у яких можлива відсутність результату виражена типом результату (лістинги 1.25 і 1.26). Систематично цей прийом розвивається в розділах 6 і 7.

Лістинг 1.25 - Часткова та тотальна функції розподілу рахунку

```
1. // Часткова функція: для guests = 0 результат не визначено
2. public static decimal ShareOf(decimal total, int guests) =>
3.     total / guests;
4.
5. // Тотальна функція: визначена для всіх значень параметрів,
6. // відсутність результату виражено типом
7. public static decimal? SafeShareOf(decimal total, int guests) =>
8.     guests > 0 ? total / guests : null;
```

Лістинг 1.26 - Виклик часткової та тотальної функцій

```
1. Console.WriteLine(SafeShareOf(1200m, 4));
2. Console.WriteLine(SafeShareOf(1200m, 0) is null
3.     ? "Кількість гостей має бути додатною"
4.     : "?");
5. try
6. {
7.     Console.WriteLine(ShareOf(1200m, 0));
8. }
9. catch (DivideByZeroException)
10.  {
11.      Console.WriteLine("ShareOf: ділення на нуль");
12.  }
```

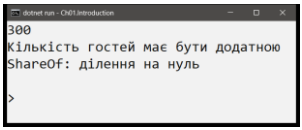


Рисунок 1.15 - Результат виконання прикладу «Часткові й тотальні функції»

1.3.2. Лямбда-числення

Лямбда-числення (λ-calculus), створене А. Черчем у 1930-х роках, - формальна система, що описує обчислення виключно за допомогою функцій. Попри простоту, вона є універсальною моделлю обчислень, рівносильною машині Тюрінга, і слугує теоретичною основою функціональних мов. Лямбда-числення містить лише три види виразів (термів):

1) **змінна**: x;

2) **абстракція** λx.M - функція з параметром x і тілом M; у мові C# їй відповідає лямбда-вираз *x => M*;

3) **застосування** M N - виклик функції M з аргументом N; у мові C# - *M(N)*.

Обчислення в лямбда-численні полягає в **β-редукції** (beta reduction): застосування абстракції до аргументу замінюється її тілом, у якому параметр замінено аргументом. Наприклад, (λx.x · 1,2) 100 -> 100 · 1,2 -> 120. Обчислення, таким чином, є послідовним переписуванням виразів за правилом підстановки, без змінних станів і присвоєнь.

У лямбда-численні кожна функція має рівно один параметр. Функцію кількох аргументів подають як функцію, що повертає функцію: λx.λy.x + y. Застосування до першого аргументу дає нову функцію: (λx.λy.x + y) 100 -> λy.100 + y. Цей прийом, названий **каррінгом** на честь Х. Каррі, детально розглядається в розділі 9. Відповідність між термами лямбда-числення та лямбда-виразами мови C# показано в лістингу 1.27.

Лістинг 1.27 - Терми лямбда-числення мовою C#

```
1. // λx.x - тотожна функція
2. Func<decimal, decimal> id = x => x;
3. // λx.x·1,2 - нарахування ПДВ
4. Func<decimal, decimal> vat = x => x * 1.2m;
5. // λx.λy.x+y - функція двох аргументів як ланцюжок функцій
6. Func<decimal, Func<decimal, decimal>> add = x => y => x + y;
7. // λx.λy.x - функція, що «запам'ятовує» перший аргумент
8. Func<string, Func<int, string>> constant = x => _ => x;
9.
10.  // Застосування (β-редукція): (λx.x·1,2) 100 -> 100·1,2 -> 120
11.  Console.WriteLine(vat(100m));
12.  Console.WriteLine(id(vat(100m)));
13.  // (λx.λy.x+y) 100 -> λy.100+y;  (λy.100+y) 15 -> 115
14.  var plus100 = add(100m);
15.  Console.WriteLine(plus100(15m));
16.  Console.WriteLine(constant("зачинено")(7));
```



Рисунок 1.16 - Результат виконання прикладу «Лямбда-вирази»

Модель обчислення як підстановки безпосередньо пов'язана з прозорістю посилань: якщо функції чисті, будь-який виклик можна замінити його результатом, і програму можна аналізувати як систему рівностей - так само, як математичну формулу. Таке **рівнянне міркування** (equational reasoning) є одним з основних практичних наслідків функціонального стилю.

1.3.3. Функції першого класу та замикання

Мова підтримує **функції першого класу** (first-class functions), якщо функції в ній є повноправними значеннями: їх можна зберігати в змінних і колекціях, передавати як аргументи, повертати як результат і створювати під час виконання програми. Функції вищого порядку (підрозділ 1.2.2) можливі лише за цієї умови. У мові C# функціями першого класу є делегати та лямбда-вирази.

Функція, створена всередині іншої функції, може використовувати змінні зовнішньої області видимості. Таку функцію разом із «захопленими» змінними називають **замиканням** (closure). У лістингу 1.28 функція *Discount* повертає нову функцію, що запам'ятовує розмір знижки; отримані функції зберігаються в словнику як звичайні значення (лістинг 1.29).

Лістинг 1.28 - Функція, що повертає замикання

```
// Повертає функцію, що «замикає» значення параметра rate
public static Func<decimal, decimal> Discount(decimal rate) =>
    price => price * (1 - rate);
```

Лістинг 1.29 - Словник функцій-знижок

```
// Функції - значення: зберігаються в колекції за ключем
var discounts = new Dictionary<string, Func<decimal, decimal>>
{
    ["Гість"] = Discount(0m),
    ["Постійний"] = Discount(0.05m),
    ["VIP"] = Discount(0.15m)
};
foreach (var (level, apply) in discounts)
    Console.WriteLine($"{level,-10} {apply(800m)}");
```

Замикання захоплює саму змінну, а не її значення на момент створення. Якщо захоплена змінна згодом змінюється, змінюється й результат замикання (лістинг 1.30): усі три функції повертають номер 4, оскільки звертаються до однієї змінної *number*, остаточне значення якої дорівнює 4.

Лістинг 1.30 - Захоплення змінюваної змінної

```
// Замикання захоплює змінну, а не її значення
var tables = new List<Func<string>>();
var number = 1;
for (var i = 0; i < 3; i++)
{
    tables.Add(() => $"столик {number}");
    number++;
}
Console.WriteLine(string.Join(", ", tables.Select(f => f())));
```

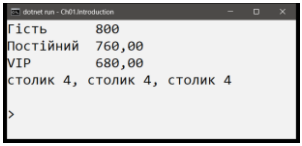


Рисунок 1.17 - Результат виконання прикладу «Замикання»

Ця поведінка є ще одним аргументом на користь незмінності: замикання над незмінними значеннями є чистими функціями, тоді як замикання над змінюваною змінною приховують залежність від стану.

1.3.4. Композиція функцій

Композицією функцій f: A -> B і g: B -> C називають функцію g ∘ f: A -> C, яка спочатку застосовує f, а потім g до її результату: (g ∘ f)(x) = g(f(x)). Композиція можлива лише тоді, коли тип результату першої функції збігається з типом параметра другої (рис. 1.18).

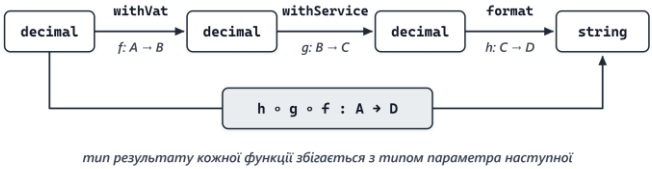


Рисунок 1.18 - Композиція функцій

Композиція має дві важливі властивості:

- **асоціативність**: h ∘ (g ∘ f) = (h ∘ g) ∘ f, тобто порядок групування не впливає на результат, і довгий ланцюжок можна будувати з частин у будь-якому порядку групування;

- **нейтральний елемент**: тотожна функція id(x) = x не змінює іншої функції під час композиції: f ∘ id = id ∘ f = f.

Ці властивості дають змогу будувати програму як конвеєр невеликих функцій, кожну з яких можна перевірити окремо. Реалізацію композиції та перевірку її властивостей мовою C# наведено в лістингах 1.31 і 1.32.

Лістинг 1.31 - Композиція функцій і тотожна функція

```
1. // (g ∘ f)(x) = g(f(x))
2. public static Func<A, C> Compose<A, B, C>(Func<B, C> g,
3.                                           Func<A, B> f) =>
4.     x => g(f(x));
5.
6. public static T Identity<T>(T x) => x;
```

Лістинг 1.32 - Побудова функції розрахунку рахунку композицією

```
1. Func<decimal, decimal> withVat = p => p * 1.2m;
2. Func<decimal, decimal> withService = p => p * 1.1m;
3. Func<decimal, string> format = p => $"{p:F2} грн";
4.
5. var bill = Compose(format, Compose(withService, withVat));
6. Console.WriteLine(bill(500m));
7.
8. // Асоціативність: h ∘ (g ∘ f) = (h ∘ g) ∘ f
9. var left = Compose(format, Compose(withService, withVat));
10.  var right = Compose(Compose(format, withService), withVat);
11.  Console.WriteLine(left(500m) == right(500m));
12.
13.  // Тотожна функція - нейтральний елемент композиції
14.  var same = Compose(Identity, withVat);
15.  Console.WriteLine(same(500m) == withVat(500m));
```



Рисунок 1.19 - Результат виконання прикладу «Композиція»

Ланцюжки методів LINQ (розділ 2), комбінатори (розділ 5) і монадичне зв'язування (розділ 7) є різновидами композиції функцій.

1.3.5. Складові функціонального стилю в мові C#

Мова C# не є чисто функціональною: вона дозволяє змінюваний стан, не гарантує оптимізації хвостової рекурсії й не перевіряє чистоту функцій. Функціональне програмування в C# є, отже, дисципліною, яку мова підтримує дедалі повніше, а не обов'язковою моделлю. Узагальнено його можна описати як побудову програми з композиції чистих функцій над незмінними даними, де можливі стани описуються типами, а побічні ефекти винесено на межі системи. Складові цього стилю та розділи посібника, у яких вони розглядаються, подано на рис. 1.20.

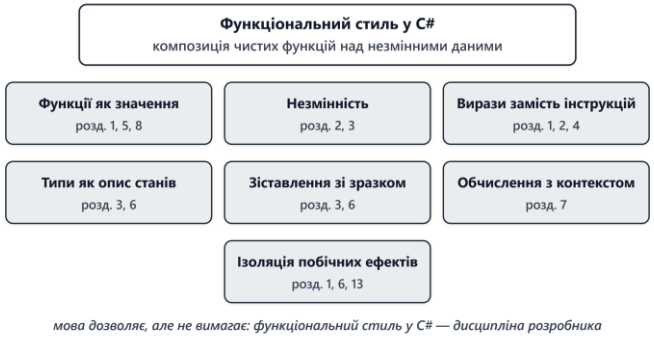


Рисунок 1.20 - Складові функціонального стилю в мові C#

1) **Функції як значення**: лямбда-вирази, делегати, функції вищого порядку, композиція, каррінг (розділи 1, 5, 9).

2) **Незмінність**: записи, вираз *with*, методи доступу *init*, незмінні колекції (розділи 2, 3).

3) **Вирази замість інструкцій**: вирази *switch*, тернарний оператор, конвеєри LINQ (розділи 1, 2, 4).

4) **Типи як опис можливих станів**: алгебраїчні типи, дискриміновані об'єднання, *Maybe* і *Result* замість *null* і винятків (розділи 3, 6).

5) **Зіставлення зі зразком**: розбір даних за формою з перевіркою повноти (розділи 3, 6).

6) **Обчислення з контекстом**: функтори й монади, синтаксис запитів LINQ, *async/await* (розділ 7).

7) **Ізоляція побічних ефектів**: чисте функціональне ядро та імперативна оболонка (розділи 1, 6, 14).

1.4. Порівняння імперативного та декларативного підходів

Відмінність між імперативною та декларативною парадигмами доцільно розглянути на завершеному прикладі. Нехай потрібно обчислити суму рахунку за столиком ресторану: врахувати лише позиції з ненульовою кількістю, а для компанії з шести й більше гостей додати сервісний збір у розмірі 10 %.

Імперативне рішення наведено в лістингу 1.33.

Лістинг 1.33 - Імперативне обчислення суми рахунку

```
1. public static decimal ImperativeBill(List<OrderLine> lines,
2.                                      int guests)
3. {
4.     decimal subtotal = 0;
5.     for (int i = 0; i < lines.Count; i++)
6.     {
7.         if (lines[i].Quantity > 0)
8.         {
9.             subtotal = subtotal
10.                 + lines[i].Item.Price * lines[i].Quantity;
11.          }
12.     }
13.
14.     decimal service = 0;
15.     if (guests >= 6)
16.     {
17.         service = subtotal * 0.10m;
18.     }
19.
20.     return subtotal + service;
21. }
```

Цей код є типовим прикладом імперативного стилю: він містить допоміжні змінні з коротким часом життя (*subtotal*, *service*, лічильник *i*), призначені для відстеження проміжного стану, і детально визначає порядок усіх дій. Програма нагадує інструкцію для виконавця, якому необхідно пояснити кожен крок.

Декларативне рішення тієї самої задачі наведено в лістингу 1.34.

Лістинг 1.34 - Декларативне обчислення суми рахунку

```
1. public static decimal DeclarativeBill(
2.     IEnumerable<OrderLine> lines, int guests) =>
3.     lines.Where(l => l.Quantity > 0)
4.          .Sum(l => l.Total)
5.     * (guests >= 6 ? 1.10m : 1.00m);
```

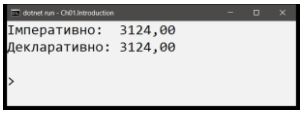


Рисунок 1.21 - Результат виконання прикладу «Рахунок столика»

У декларативному варіанті відсутні змінні для відстеження стану, цикли та інструкції *if*. Код описує результат: «відібрати позиції з ненульовою кількістю, підсумувати їхню вартість і помножити на коефіцієнт сервісного збору». Порядок виконання окремих операцій визначає бібліотека LINQ, яку детально розглянуто в розділі 2. Такий опис можна порівняти з інструкцією для виконавця, здатного самостійно визначити необхідні дії на основі опису мети.

1.5. Історичні передумови функціонального програмування

Функціональне програмування не є тимчасовою тенденцією: його теоретичні засади передують появі сучасних мов програмування і навіть обчислювальної техніки як такої. Це зумовлює доцільність його вивчення: більшість сучасних мов тією чи іншою мірою підтримує функціональні концепції, тому відповідні навички зберігають цінність незалежно від конкретної мови.

Функціональне програмування виросло з теоретичної математики, передусім з комбінаторної логіки та лямбда-числення. Найчастіше з ним пов'язують ім'я американського математика **Хаскелла Брукса Каррі** (Haskell Brooks Curry, 1900-1982), на честь якого названо, зокрема, мови програмування Haskell і Curry, а також функціональну техніку **каррінгу** (currying), що розглядається в розділі 9. Каррі досліджував **комбінаторну логіку** (combinatory logic) - математичну теорію, в якій функції записуються у вигляді лямбда-виразів і поєднуються для побудови складнішої логіки. Саме ця ідея становить фундамент функціонального програмування. Каррі спирався на праці своїх попередників:

- **Готлоб Фреге** (Gottlob Frege, 1848-1925, Німеччина) - імовірно, першим описав ідею, яку нині називають каррінгом;
- **Мойсей Шейнфінкель** (Moses Schönfinkel, 1888-1942) - уродженець Катеринослава (нині Дніпро), автор праць із комбінаторної логіки, що стали основою досліджень Каррі;
- **Алонзо Черч** (Alonzo Church, 1903-1995, США) - творець **лямбда-числення** (lambda calculus), автор терміна «лямбда-вираз», який використовується в мові C# та багатьох інших мовах.

Першими мовами програмування з функціональними можливостями стали:

- **IPL** (Information Processing Language), створена в 1956 році А. Ньюеллом (A. Newell), К. Шоу (C. Shaw) і Г. Саймоном (H. Simon);
- **LISP** (LISt Processor), створена в 1958 році Дж. Маккарті (J. McCarthy); мова та її діалекти (Common Lisp, Scheme, Clojure) використовуються й нині.

Жодна з цих мов не була «чисто» функціональною: подібно до C# і Java, вони поєднували функціональні та нефункціональні засоби, на відміну від сучасних чисто функціональних мов, таких як Haskell чи Elm.

1.6. Функціональні та мультипарадигмові мови програмування

Підтримку функціональної парадигми, тобто наявність засобів для написання коду у функціональному стилі, нині надає значна кількість мов. За ступенем такої підтримки їх поділяють на дві групи:

- **чисто функціональні мови** (pure functional languages) - призначені для написання виключно функціонального коду: усі змінні є незмінними, каррінг і функції вищого порядку підтримуються на рівні мови; окремі елементи ООП можуть бути доступні, але мають другорядне значення;
- **гібридні**, або **мультипарадигмові, мови** (multiparadigm languages) - надають засоби для програмування у двох і більше парадигмах, зазвичай функціональній та об'єктно-орієнтованій. Повна реалізація кожної з підтримуваних парадигм у таких мовах не гарантується: як правило, ООП підтримується повністю, а функціональна парадигма - частково.

1.6.1. Чисто функціональні та функціональні мови

Існує понад десяток функціональних мов. Серед найпоширеніших:

- **Haskell** - чисто функціональна мова з лінивими обчисленнями, що використовується, зокрема, у фінансовому секторі; її часто рекомендують як відправну точку для поглибленого вивчення функціональної парадигми;
- **Elm** - чисто функціональна мова для розроблення вебінтерфейсів; запропонована в ній архітектура оновлення інтерфейсу вплинула на низку інших проєктів, зокрема на екосистему React;
- **PureScript** - чисто функціональна мова, що компілюється в JavaScript і придатна як для клієнтської, так і для серверної частини застосунків (наприклад, у середовищі Node.js);
- **Elixir** - функціональна (але не чисто функціональна) мова загального призначення, що працює на віртуальній машині мови Erlang і широко застосовується для побудови розподілених систем.

1.6.2. Доцільність попереднього вивчення чисто функціональної мови

Нині об'єктно-орієнтована парадигма залишається домінантною в розробці програмного забезпечення, і функціональний підхід здебільшого опановують уже після неї. Існує думка, що фахівцеві з досвідом ООП

доцільно спочатку вивчити функціональне програмування в «чистому» вигляді, а потім переносити набуті знання в мову C#.

Такий шлях є цілком виправданим, однак не єдино можливим. Чисто функціональні мови не є складними - функціональне програмування оперує навіть меншою кількістю базових понять, ніж ООП, - проте вони суттєво відрізняються від звичних мов, і фахівцям із тривалим досвідом ООП адаптація до них дається важче. Крім того, чисто функціональні мови порівняно рідко використовуються в промисловій розробці, тому їх вивчення потребує значних зусиль, які не завжди мають безпосереднє практичне застосування.

Перевагою мови C# є одночасна підтримка об'єктно-орієнтованого та функціонального стилів. Обидві парадигми можуть співіснувати в одній кодовій базі, що дає змогу переходити від об'єктно-орієнтованого стилю до функціонального поступово, у прийнятному темпі. Такого поєднання парадигм чисто функціональні мови не допускають, хоча й надають функціональні можливості, недоступні в C#. Саме поступовий шлях опанування функціонального програмування в межах знайомої мови покладено в основу цього посібника.

1.6.3. Мова F#

Окремого розгляду заслуговує мова **F#** - функціонально-орієнтована мова платформи .NET. Вона не є чисто функціональною, однак значно ближча до неї, ніж C#: широкий набір функціональних можливостей доступний «з коробки», код на F# є лаконічним, а застосунки - продуктивними.

Рішення щодо використання F# у проєкті має насамперед організаційний характер. Розробників, які володіють мовою C#, на ринку праці значно більше, ніж фахівців з F#. Включення F# до кодової бази створює для команди залежність від наявності таких фахівців і ризик ускладнення супроводу відповідних модулів. Водночас цей ризик є помірним: F# - мова платформи .NET, вона компілюється в той самий проміжний код (Intermediate Language, IL), а проєкти на F# можна безпосередньо підключати до проєктів на C# в межах одного рішення (див. розділи 12 і 13).

Отже, опанування функціонального стилю в мові C# є раціональним першим кроком; мова F# може стати наступним етапом для тих, хто прагне глибше опанувати функціональну парадигму.

1.6.4. Мультипарадигмові мови

Строго кажучи, майже кожна мова, що не є чисто функціональною, певною мірою є гібридною, оскільки дає змогу реалізувати хоча б окремі елементи функціональної парадигми. Нижче розглянуто мови, в яких

функціональний стиль підтримується повністю або значною мірою як свідомо закладена розробниками можливість:

- **JavaScript** - мова з надзвичайно гнучкою моделлю, що дуже добре підтримує функціональний стиль; підходи до її функціонального використання систематизовано, зокрема, у працях Д. Крокфорда (D. Crockford);
- **Python** - підтримує функції вищого порядку, а стандартні модулі *itertools* і *functools* розширюють його функціональні можливості;
- **Java** - платформа Java має рівень підтримки функціональних можливостей, порівнянний із .NET, а мови на її основі (Scala, Clojure, Kotlin) надають значно ширші функціональні засоби;
- **F#** - функціонально-орієнтована мова платформи .NET, що підтримує взаємодію з бібліотеками на C#;
- **C#** - підтримка функціонального програмування розширюється з кожною версією мови.

Еволюцію функціональних можливостей мови C# доцільно розглянути докладніше. Першим кроком у цьому напрямі можна вважати анонімні методи та коваріантність делегатів у версії C# 2.0 (2005 рік). Визначальною подією стала поява у версії C# 3.0 (2007 рік) лямбда-виразів і технології **LINQ** (Language Integrated Query), яка глибоко ґрунтується на функціональній парадигмі й залишається одним з основних інструментів функціонального стилю в C# (див. розділ 2). Подальші версії додали кортежі та зіставлення зі зразком (C# 7), вирази *switch* (C# 8), типи-записи та ініціалізатори *init* (C# 9), шаблони списків і статичні абстрактні члени інтерфейсів (C# 11), члени-розширення (C# 14), типи-об'єднання та закриті ієрархії (C# 15). Команда розробників мови послідовно розширює підтримку функціонального програмування, чому значною мірою сприяє мова F#: нові можливості платформи .NET, запитувані для F#, стають доступними й для C#. Докладну хронологію наведено в розділі 3. Приклади цього посібника орієнтовані на платформу .NET 11 і мову C# 15.

1.7. Переваги функціонального програмування

1.7.1. Лаконічність

Функціональний код, як правило, значно лаконічніший за імперативний. Імперативний код зосереджений на низькорівневих деталях реалізації, і для того, щоб зрозуміти його призначення, нерідко доводиться детально аналізувати кожен рядок. Функціональний код орієнтований на опис того, *що* необхідно отримати, тоді як деталі оновлення змінних передаються середовищу виконання.

Ця властивість має і економічне значення. Розробникам часто доводиться супроводжувати успадковані кодові бази, і що більше часу витрачається на з'ясування призначення функції, то вищою є вартість супроводу. Функціональний код нерідко читається як опис мети природною мовою, що спрощує розуміння логіки програми та пошук помилок.

1.7.2. Тестованість

Функціональний код має високу тестованість: оскільки результат чистої функції визначається лише її параметрами, автоматизовані модульні тести для неї будуються просто й надійно. Функціональний стиль органічно поєднується з методологіями розробки через тестування (Test-Driven Development, TDD) та розробки через поведінку (Behavior-Driven Development, BDD), за якими автоматизовані тести пишуться до основного коду. Це сприяє покращенню структури програми та зменшенню кількості помилок у промисловій експлуатації.

1.7.3. Надійність

Надійність функціонального коду забезпечується не лише його тестованістю. Функціональне програмування пропонує структури, що запобігають виникненню помилок або, принаймні, не допускають неочікуваної поведінки програми в подальшому, спрощуючи точну діагностику проблеми. Зокрема, у функціональних мовах, як правило, відсутнє поняття порожнього посилання *null*: можлива відсутність значення виражається явно на рівні типу (у мові C# відповідні засоби розглядаються в розділах 3, 6 і 7). Це усуває цілий клас помилок і скорочує кількість необхідних тестів.

1.7.4. Передбачуваність

Функціональний код виконується послідовно від початку до кінця блоку, тоді як потік виконання процедурного коду ускладнюється циклами й розгалуженнями.

Належно побудований функціональний код не потребує блоків *try*/*catch*, які часто є джерелом непередбачуваного порядку виконання. Якщо блок *try* охоплює великий фрагмент коду і не пов'язаний тісно з відповідним блоком *catch*, точка перехоплення винятку та наслідки такого переривання потоку виконання стають важко прогнозованими. Неправильна обробка помилок можлива й у функціональному коді, проте сама природа парадигми цьому запобігає.

1.7.5. Підтримка паралельних обчислень

Протягом останніх років у розробці програмного забезпечення набули поширення дві технології, що інтенсивно використовують паралельну обробку даних:

- **контейнеризація** (Docker, Kubernetes тощо) - застосунок виконується не на традиційному сервері, а в ізольованому контейнері, що створюється сценарієм під час розгортання. Інфраструктура може запускати багато екземплярів одного застосунку, які обробляють спільне джерело вхідних даних (чергу повідомлень, запити користувачів), і динамічно змінювати їхню кількість залежно від навантаження;
- **безсерверні обчислення** (serverless), наприклад Azure Functions або AWS Lambda, - код розгортається не на вебсервері, а у вигляді окремих функцій у хмарному середовищі, що забезпечує автоматичне масштабування та дає змогу оптимізувати витрати на рівні окремих функцій.

Обидві технології передбачають одночасну роботу кількох екземплярів одного функціоналу над спільним джерелом даних. Основні проблеми паралельних обчислень пов'язані зі спільними ресурсами - станом у пам'яті або зовнішніми ресурсами. Функціональний код не має спільного змінного стану, тож такі конфлікти в ньому не виникають. Це суттєво спрощує побудову масштабованих систем.

1.7.6. Зменшення «шуму» в коді

В обробці звукових сигналів використовується поняття **відношення сигнал/шум** - відношення рівня корисного сигналу до рівня фонових завад. За аналогією в програмному коді «сигналом» можна вважати бізнес-логіку, тобто мету, якої досягає фрагмент коду, а «шумом» - допоміжний шаблонний код (boilerplate): оголошення циклів, лічильників, умовних інструкцій тощо.

Функціональний код містить значно менше шаблонних конструкцій, ніж процедурний, і тому має вище відношення «сигнал/шум». Це спрощує не лише роботу розробника, а й супровід та розвиток програмного продукту загалом.

1.8. Межі застосування функціонального підходу

Функціональна парадигма дає змогу розв'язати будь-яку задачу, яку розв'язують засобами інших парадигм, проте її переваги виявляються в різних предметних галузях неоднаково. На платформі .NET компроміси неминучі, оскільки базові класи та більшість сторонніх бібліотек розроблено відповідно до об'єктно-орієнтованої парадигми.

Функціональний підхід є найефективнішим у таких випадках:

- обробка та перетворення даних з високим ступенем передбачуваності (модулі обробки даних, функції перетворення даних з одного формату в інший, бізнес-логіка, що отримує дані від користувача чи з бази даних і передає їх для подальшого відображення);

- паралельні та асинхронні системи, зокрема ті, в яких кілька обробників одночасно читають спільну чергу повідомлень: за відсутності спільного стану конкуренція за ресурси практично неможлива; з тих самих причин функціональний стиль добре відповідає безсерверній архітектурі;

- критично важливі системи, для яких неприпустиме аварійне завершення роботи внаслідок необробленого винятку або некоректних вхідних даних.

Альтернативні підходи доцільно розглядати в таких ситуаціях:

- **взаємодія із зовнішнім середовищем** - операції введення-виведення, введення даних користувачем, звернення до сторонніх застосунків і вебсервісів. Такі операції неможливо реалізувати у вигляді чистих функцій, тому компроміс неминучий. Те саме стосується сторонніх бібліотек із NuGet і частини бібліотек .NET, які за своєю архітектурою не допускають функціонального використання (характерним прикладом є застарілий клас *SmtpClient*);

- **граничні вимоги до продуктивності** - якщо продуктивність є єдиним визначальним критерієм, що переважає навіть читабельність і модульність коду. Функціональний код на C# не є принципово повільним, проте не завжди є найшвидшим з можливих рішень. У більшості прикладних систем незначна втрата продуктивності компенсується зниженням витрат на розроблення, тестування й супровід; винятком є, наприклад, програми для пристроїв з обмеженими обчислювальними ресурсами. Питання продуктивності функціонального коду докладно розглянуто в розділі 12.

Слід також зазначити, що повністю реалізувати функціональну парадигму в мові C# неможливо: цьому перешкоджають, зокрема, вимоги зворотної сумісності мови та обмеження її системи типів. Тому метою посібника є не реалізація всіх елементів парадигми, а визначення меж можливого та практично доцільного, з урахуванням потреб супроводу реальних програмних продуктів.

Окремо слід згадати поняття **монади** (monad), яке нерідко вважають найскладнішим у функціональному програмуванні. Його формальне означення, що походить з теорії категорій, є досить абстрактним, проте для практичного використання монад формальний апарат не обов'язковий. Монади в посібнику вводяться поступово, на конкретних прикладах (розділи 6 і 7), після того як буде опановано необхідні базові поняття.

1.9. Висновки

1. Функціональне програмування є декларативною парадигмою програмування, математичні засади якої (комбінаторна логіка, лямбда-числення) сформувалися раніше, ніж з'явилися електронні обчислювальні машини.

2. Основними властивостями функціонального програмування є незмінність даних, використання функцій вищого порядку, перевага виразів над інструкціями, прозорість посилань (чисті функції), рекурсія, зіставлення зі зразком і відсутність спільного змінного стану.

3. Функціональне програмування спирається на математичне поняття функції як відображення, що кожному аргументу ставить у відповідність рівно одне значення. Тотальні функції визначені для всіх аргументів; часткові функції повідомляють про невизначеність винятками, тому функціональний підхід виражає можливу відсутність результату типом.

4. Лямбда-числення описує обчислення трьома видами термів (змінна, абстракція, застосування) і правилом β-редукції; лямбда-вирази мови C# є безпосередньою реалізацією абстракції. Функції першого класу та замикання дають змогу створювати функції під час виконання та поводитися з ними як зі значеннями.

5. Композиція функцій асоціативна й має нейтральний елемент - тотожну функцію; ці властивості дають змогу будувати програму як конвеєр невеликих перевірюваних функцій.

6. Імперативний код детально визначає, *як* досягти результату, тоді як декларативний описує, *що* має бути отримано; порядок виконання окремих кроків у декларативному коді визначає середовище виконання.

7. Мова C# є мультипарадигмовою: вона дає змогу поступово переходити від об'єктно-орієнтованого стилю до функціонального в межах однієї кодової бази, а її функціональні можливості розширюються з кожною версією.

8. Основними перевагами функціонального програмування є лаконічність, тестованість, надійність, передбачуваність, природна підтримка паралельних обчислень і висока частка бізнес-логіки в коді.

9. Найбільший ефект функціональний підхід дає в задачах обробки та перетворення даних, у паралельних і критично важливих системах; взаємодія із зовнішнім середовищем потребує компромісу, який доцільно реалізовувати за архітектурою «функціональне ядро, імперативна оболонка».

1.10. Контрольні запитання

1. Що таке парадигма програмування? Чому функціональне програмування не можна вважати мовою чи бібліотекою?

2. У чому полягає відмінність між імперативною та декларативною парадигмами? Наведіть приклади мов, що належать до кожної з них.

3. Що означає принцип незмінності? Які типи платформи .NET є незмінними?

4. Що таке функція вищого порядку? Яке призначення делегатних типів *Func<T, TResult>* і *Action<T>* та чим вони відрізняються?

5. Чим вираз відрізняється від інструкції? Сформулюйте емпіричне правило їх розрізнення та вкажіть його обмеження.

6. Які властивості має чиста функція? Що таке прозорість посилань?

7. Чому функція, що використовує *DateTime.Now*, не є чистою? Як її можна перетворити на чисту?

8. У чому полягає архітектурний підхід «функціональне ядро, імперативна оболонка»?

9. З яких частин складається рекурсивна функція? Які обмеження має рекурсія в мові C#?

10. Які переваги надає зіставлення зі зразком порівняно з вкладеними інструкціями *if*?

11. Чому передавання змінного об'єкта в метод знижує передбачуваність програми? Як цю проблему розв'язує функціональний підхід?

12. Назвіть науковців, роботи яких заклали теоретичні основи функціонального програмування, та перші мови з функціональними можливостями.

13. У яких задачах функціональний підхід є найефективнішим, а в яких доцільно використовувати інші парадигми?

14. Що таке функція з погляду математики? Чим тотальна функція відрізняється від часткової? Наведіть приклади часткових методів платформи .NET.

15. Які види термів містить лямбда-числення? Що таке β-редукція? Виконайте редукцію виразу (λx.λy.x · y) 3 4.

16. Що таке замикання? Чому замикання над змінюваною змінною може давати неочікуваний результат?

17. Що таке композиція функцій? Які її властивості та за якої умови композиція двох функцій можлива?

1.11. Практичні завдання

1. Реалізуйте метод формування рядка рахунку кав'ярні (назва напою, об'єм, ціна) у двох формах: як звичайний метод класу і як значення типу *Func<string, int, decimal, string>*. Напишіть функцію вищого порядку, яка отримує колекцію замовлень і функцію форматування та повертає колекцію рядків.

2. Проаналізуйте наведені нижче сигнатури методів готельної системи та визначте, які з них можуть бути чистими функціями; для нечистих запропонуйте чисті варіанти: *decimal StayCost(int nights)*, що використовує поле класу *_ratePerNight*; *bool IsLateCheckout()*, що використовує *DateTime.Now*; *void ApplyDiscount(Booking b)*, що змінює ціну в об'єкті *b*.

3. Перепишіть функцію визначення тарифу номера (лістинг 1.20), додавши категорію номера *Family* та сезонний коефіцієнт (високий / низький сезон). Використайте вираз *switch* над кортежем із трьох елементів і переконайтеся, що всі комбінації вхідних даних оброблено.

4. Реалізуйте рекурсивну функцію, яка визначає, за скільки днів буде вичерпано запас інгредієнта на складі ресторану за відомих початкового залишку та добової витрати. Значення параметрів у функції не повинні змінюватися.

5. Реалізуйте функціональний варіант класу стану бару (кількість відкритих рахунків, виручка, черговий бармен): функції реєстрації замовлення та зміни бармена мають повертати новий об'єкт стану. Продемонструйте, що попередній стан залишається незмінним.

6. Реалізуйте тотальні варіанти часткових функцій готельної системи: обчислення середньої вартості ночі за загальною сумою та кількістю ночей, розбір номера кімнати з рядка, пошук тарифу за кодом. Можливу відсутність результату подайте типом *T?*.

7. Побудуйте композицією невеликих функцій конвеєр формування рядка фіскального чека: нарахування знижки, нарахування ПДВ, округлення до копійок, форматування. Перевірте асоціативність композиції для побудованих функцій.

Частина І. Функціональні можливості, наявні в мові C#

2. Функціональні засоби, доступні без додаткових бібліотек

У цьому розділі розглядаються функціональні засоби, які можна застосовувати практично в будь-якому програмному продукті мовою C#, що перебуває в промисловій експлуатації. Більшість із них з'явилася ще у версії .NET Framework 3.5 разом із технологією LINQ і з того часу залишається незмінною частиною платформи. Окремі прийоми можуть здаватися відомими, проте саме вони становлять фундамент, на якому ґрунтується подальший матеріал посібника. Новіші можливості мови, орієнтовані на функціональне програмування, розглядаються в розділі 3, а нестандартні способи використання відомих засобів - у розділі 4.

Поширеним є уявлення, що функціональне програмування складніше за об'єктно-орієнтоване і вимагає тривалої попередньої підготовки. Насправді функціональна парадигма оперує меншою кількістю базових понять, ніж ООП. Труднощі її опанування зумовлені передусім звичкою до імперативного стилю, а не складністю самих понять. Крім того, розробник, що має досвід роботи з мовою C#, найімовірніше, вже використовує функціональні прийоми - часто не усвідомлюючи цього.

Мета розділу - показати, як стандартні засоби мови C# і бібліотеки LINQ дають змогу писати код у функціональному стилі без використання сторонніх бібліотек.

Після опрацювання розділу студент повинен:

- уміти замінювати імперативні цикли з накопиченням результату на декларативні запити LINQ;
- розуміти принцип відкладеного виконання перелічуваних послідовностей і його вплив на порядок обчислень;
- уміти застосовувати перевантажену форму методу *Select* з індексом, методи *Enumerable.Range*, *Enumerable.Repeat*, *SelectMany*, *Sum*, *Average* та *Aggregate*;
- розуміти обмеження рекурсії в мові C#;
- знати можливості та обмеження забезпечення незмінності об'єктів засобами мови C#;
- уміти будувати повний функціональний конвеєр обробки даних.

Приклади розділу містяться в проєкті *Ch02.WhatWeCanDoAlready* репозиторію *FunctionalCSharp.HoReCa*.

2.1. Перший функціональний код

2.1.1. Нефункціональний запит до меню

Розглядання функціонального стилю доцільно почати з імперативного прикладу, побудованого так, як зазвичай навчають на початковому етапі

вивчення мови C#. Метод у лістингу 2.1 отримує з умовного сховища повне меню ресторану і формує новий список, до якого копіює лише страви заданої категорії.

Лістинг 2.1 - Імперативний вибір страв заданої категорії

```
1. public static IEnumerable<MenuItem> GetDishesByCategory(
2.     Category category)
3. {
4.     var allDishes = GetAllDishes();
5.     var chosenDishes = new List<MenuItem>();
6.     foreach (var d in allDishes)
7.     {
8.         if (d.Category == category)
9.         {
10.              chosenDishes.Add(d);
11.          }
12.      }
13.      return chosenDishes;
14.  }
```

Цей код має кілька недоліків. По-перше, для простої за змістом операції написано значний обсяг коду. По-друге, створюється новий об'єкт-список, який існує протягом усього часу виконання методу. Якщо цей фрагмент є частиною довшого методу, обидві колекції - *allDishes* і *chosenDishes* - залишаються в області видимості, а отже, і в пам'яті, навіть коли вже не використовуються. Навіть якщо самі об'єкти не копіюються, у пам'яті зберігається зайвий набір посилань на них. По-третє, код жорстко визначає порядок операцій: коли виконувати перебір, коли додавати елемент до результату. За потреби в проміжних перетвореннях даних їх також довелося б задавати явно і зберігати в додаткових змінних.

Частину цих недоліків усуває використання інструкції *yield return* (лістинг 2.2).

Лістинг 2.2 - Вибір страв за допомогою yield return

```
1. public static IEnumerable<MenuItem> GetDishesByCategoryLazy(
2.     Category category)
3. {
4.     var allDishes = GetAllDishes();
5.     foreach (var d in allDishes)
6.     {
7.         if (d.Category == category)
8.         {
9.             yield return d;
10.          }
11.      }
12.  }
```

Такий метод не створює проміжного списку, а повертає елементи по одному в міру перебору результату. Однак порядок виконання операцій, як і раніше, повністю визначає програміст. Якщо подальший код не використає результат, робота виявиться виконаною даремно; якщо існує

ефективніший порядок обчислень, середовище виконання не може його застосувати. Необхідність детально описувати кожну дію є принциповою властивістю процедурного коду. Одним з основних завдань функціонального програмування є відмова від такої деталізації на користь декларативного опису.

2.1.2. Функціональний запит до меню

Функціональний варіант того самого запиту наведено в лістингу 2.3.

Лістинг 2.3 - Функціональний вибір страв заданої категорії

```
1. public static IEnumerable<MenuItem> GetDishesByCategory(
2.     IEnumerable<MenuItem> source, Category category) =>
3.     source.Where(d => d.Category == category);
```

Виклик методу показано в лістингу 2.4.

Лістинг 2.4 - Виклик функціонального запиту

```
1. var allDishes = SampleData.Menu;
2. var mains = GetDishesByCategory(allDishes, Category.Main);
```

Усі три варіанти повертають однаковий результат - коди основних страв меню:



Рисунок 2.1 - Результат виконання прикладу «Запит страв» (фрагмент)

Функціональний варіант побудовано на основі бібліотеки **LINQ** (Language Integrated Query) - набору методів-розширень для фільтрації, перетворення та агрегування колекцій даних, що є частиною мови C# з версії 3.0. Методи *Select*, *Where*, *All* та інші належать саме до LINQ. Ця бібліотека значною мірою реалізує функціональну парадигму, що можна показати на прикладі лістингу 2.3:

- **функції вищого порядку** - лямбда-вирази, що передаються в методи LINQ, є функціями, переданими як параметри;
- **незмінність** - LINQ не змінює вихідну колекцію, а повертає нову послідовність, побудовану на її основі;
- **вирази замість інструкцій** - інструкції *foreach* і *if* повністю усунуто;
- **прозорість посилань** - лямбда-вираз *d => d.Category == category* не має побічних ефектів. Мова цього не гарантує (у лямбда-виразі можна звернутися до змінної ззовні), тому відповідальність за чистоту лежить на розробникові. Крім того, джерело даних передається в метод як параметр, а не отримується всередині методу. Це спрощує тестування: для перевірки методу не потрібно створювати імітацію (mock) сховища даних, оскільки все необхідне метод отримує через власні параметри.

Отже, однорядковий метод у лістингу 2.3 демонструє суть функціонального підходу: над колекцією даних виконуються операції, задані функціями, у результаті чого створюється нова колекція на основі вихідної. Отриманий код є стислішим, легшим для читання і, як наслідок, для супроводу.

2.2. Програмування, орієнтоване на результат

Характерною рисою функціонального коду є зосередженість на кінцевому результаті, а не на процесі його отримання. У процедурному стилі складний об'єкт, як правило, створюється порожнім на початку блоку коду, а потім його властивості послідовно заповнюються. У лістингу 2.5 таким способом формується рахунок-фактура за бронюванням готельного номера: для корпоративного клієнта платником є компанія, для приватного - сам гість.

Лістинг 2.5 - Процедурне формування рахунку-фактури

```
1. public static Invoice BuildInvoice(BookingSource source)
2. {
3.     var invoice = new Invoice();
4.     invoice.Number = "INV-" + source.BookingId;
5.     invoice.Amount = source.Nights * source.NightlyRate;
6.     if (source.IsCorporate)
7.     {
8.         invoice.Payer = source.CompanyName;
9.         invoice.SendTo = source.CompanyEmail;
10.      }
11.      else
12.      {
13.          invoice.Payer = source.GuestName;
14.          invoice.SendTo = source.GuestEmail;
15.      }
16.      return invoice;
17.  }
```

Наведений приклад є коротким і зрозумілим, однак такий підхід легко призводить до ускладнення коду. У реальних програмних продуктах подібні методи нерідко розростаються: дані надходять із кількох джерел, потребують попередньої обробки й поєднання, з'являються вкладені інструкції *if*, і структура коду починає нагадувати генеалогічне дерево. Кожен рівень вкладеного розгалуження фактично подвоює кількість можливих шляхів виконання. Ситуацію погіршують численні інструкції *return*, розташовані в різних місцях методу. Зростає ризик того, що за певного поєднання умов властивість залишиться неініціалізованою або отримає неочікуване значення.

Уже в лістингу 2.5 властивості *Payer* і *SendTo* присвоюються у двох різних місцях. У великих програмних системах трапляються випадки, коли одна властивість визначається в багатьох місцях кількох класів і навіть за межами програмного коду, наприклад у збережених процедурах бази

даних. Супровід такого коду з часом ускладнюється, а швидкість внесення змін знижується.

Функціональний код доцільно будувати з невеликих стислих блоків, зосереджених на кінцевому результаті. Оскільки вирази моделюють кроки розв'язання математичної задачі, функціональний код слід писати як набір невеликих формул, кожна з яких точно визначає певне значення та всі величини, від яких воно залежить. Пошук місця, де визначається те чи інше значення, не повинен вимагати перегляду всієї кодової бази (лістинг 2.6).

Лістинг 2.6 - Формування рахунку-фактури, орієнтоване на результат

```
1. public static Invoice MakeInvoice(BookingSource source) =>
2.     new()
3.     {
4.         Number = "INV-" + source.BookingId,
5.         Amount = source.Nights * source.NightlyRate,
6.         Payer = source.IsCorporate
7.             ? source.CompanyName
8.             : source.GuestName,
9.         SendTo = source.IsCorporate
10.              ? source.CompanyEmail
11.              : source.GuestEmail
12.     };
```

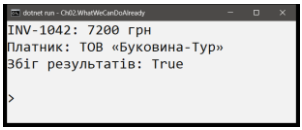


Рисунок 2.2 - Результат виконання прикладу «Рахунок-фактура»

Ознака *IsCorporate* перевіряється двічі, проте кожна властивість результату тепер визначається в одному місці - разом з усіма величинами, від яких вона залежить. Якщо обчислення певної властивості потребує складної логіки або довгого ланцюжка операцій LINQ, його доцільно винести в окрему функцію, зберігши при цьому єдину точку формування результату.

2.3. Перелічувані послідовності та відкладене виконання

Перелічувана послідовність (enumerable) - інтерфейс *IEnumerable<T>* - є найабстрактнішим поданням колекції даних у платформі .NET. Вона не обов'язково містить дані: це може бути лише збережений у пам'яті опис способу їх отримання. Перелічувана послідовність не «знає» навіть кількості своїх елементів, доки не буде виконано її повний перебір; їй відомий лише поточний елемент і спосіб переходу до наступного.

Такий механізм обчислень називають **лінивими обчисленнями** (lazy evaluation), або **відкладеним виконанням** (deferred execution). Перебір

забезпечує об'єкт-**перелічувач** (enumerator), що реалізує інтерфейс *IEnumerator<T>*: він надає доступ до поточного елемента і дає змогу перейти до наступного. Перебір можливий лише в одному напрямку, а визначити довжину послідовності без її повного перебору неможливо. Поведінку перелічувача можна визначати самостійно (див. розділ 10).

Для дослідження порядку обчислень у лістингу 2.7 визначено три функції, що відповідають етапам обслуговування замовлення: прийняття, приготування та подання. Кожна з них записує в журнал повідомлення з порядковим номером запису і повертає номер замовлення без змін.

Лістинг 2.7 - Функції етапів обслуговування з журналюванням

```
1. private static readonly List<string> log = [];
2.
3. public static int Accept(int orderId)
4. {
5.     log.Add($"{log.Count + 1}. прийнято №{orderId}");
6.     return orderId;
7. }
8.
9. public static int Cook(int orderId)
10.  {
11.      log.Add($"{log.Count + 1}. готується №{orderId}");
12.      return orderId;
13.  }
14.
15.  public static int Serve(int orderId)
16.  {
17.      log.Add($"{log.Count + 1}. подано №{orderId}");
18.      return orderId;
19.  }
```

У лістингу 2.8 ці функції послідовно застосовуються до трьох замовлень за допомогою ланцюжка методів *Select*.

Лістинг 2.8 - Ланцюжок викликів Select

```
1. var orders = new[] { 75, 22, 36 };
2.
3. var served = orders.Select(x => Accept(x))
4.     .Select(x => Cook(x))
5.     .Select(x => Serve(x))
6.     .ToArray();
```

Можна було б очікувати, що середовище виконання спочатку застосує функцію *Accept* до всіх трьох елементів масиву, сформує проміжний масив, потім застосує до нього функцію *Cook* і так далі. Проте журнал свідчить про інший порядок:

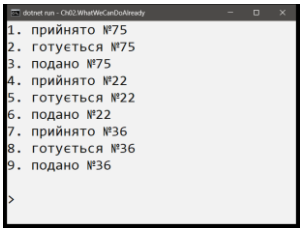


Рисунок 2.3 - Результат виконання прикладу «Ланцюжок Select»

Кожне замовлення проходить усі три етапи, перш ніж розпочинається обробка наступного. Такий самий порядок мав би цикл *for* або *foreach*, однак тут порядок операцій визначає середовище виконання, а не програміст: код лише описує необхідні операції й очікує результат. Точний порядок записів може залежати від коду, що виконує перебір (метод LINQ чи цикл *foreach*), проте принцип залишається незмінним: **перелічувана послідовність формує дані лише в момент, коли вони дійсно потрібні**. Значення має не місце визначення послідовності, а момент її перебору.

Цей принцип ілюструє лістинг 2.9: три послідовності визначено в окремих змінних, однак до виклику *ToArray()* жодна з функцій етапів не виконується.

Лістинг 2.9 - Відкладене виконання: визначення без перебору

```
1. var orders = new[] { 75, 22, 36 };
2.
3. var accepted = orders.Select(x => Accept(x));
4. var cooked = accepted.Select(x => Cook(x));
5. var served = cooked.Select(x => Serve(x));
6.
7. Console.WriteLine($"Записів до перебору: {log.Count}");
8. var result = served.ToArray();
9. Console.WriteLine($"Записів після перебору: {log.Count}");
```

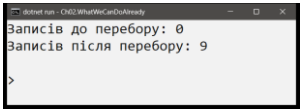


Рисунок 2.4 - Результат виконання прикладу «Відкладене виконання»

Змінні *accepted*, *cooked* і *served* є перелічуваними послідовностями, і жодна з них не містить даних до моменту перебору. Цю особливість легко спостерігати в налагоджувачі середовища розробки: після проходження рядка з визначенням послідовності її значення ще не обчислено.

Ситуація змінюється, якщо після кожного етапу примусово виконати перебір за допомогою методу *ToArray()* (лістинг 2.10).

Лістинг 2.10 - Примусове виконання кожного етапу

```
1. var orders = new[] { 75, 22, 36 };
2.
3. var accepted = orders.Select(x => Accept(x)).ToArray();
4. var cooked = accepted.Select(x => Cook(x)).ToArray();
5. var served = cooked.Select(x => Serve(x)).ToArray();
```

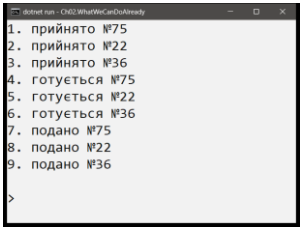


Рисунок 2.5 - Результат виконання прикладу «Примусове виконання етапів»

Тепер функція *Accept* застосовується до всіх замовлень до переходу на наступний етап. Тому, як правило, виклики *ToArray()* або *ToList()* доцільно відкладати якомога далі: операції залишаються невиконаними доти, доки результат дійсно не знадобиться, а якщо подальша логіка взагалі не потребуватиме даних, вони не будуть виконані зовсім. Для функціонального стилю перевагу варто надавати методу *ToArray()*, оскільки масив фіксованої довжини, на відміну від списку, не має методів додавання й видалення елементів.

Із цього правила є винятки. Матеріалізація результату виправдана, коли послідовність перебирається кілька разів (інакше всі обчислення повторюватимуться під час кожного перебору), а також з міркувань продуктивності. Невиконана послідовність не містить даних, але зберігає в пам'яті опис операцій. Надмірне нагромадження вкладених послідовностей, особливо в рекурсивних обчисленнях, може призвести до значних витрат пам'яті, зниження продуктивності і навіть до переповнення стека.

2.4. Перевага виразів над інструкціями

У цьому підрозділі розглядаються прийоми використання LINQ, які дають змогу уникати інструкцій *if*, *for*, *foreach*, *while* та зміни значень змінних. Не всі такі інструкції можна замінити стандартними засобами мови; способам розв'язання складніших випадків присвячено подальші розділи посібника.

2.4.1. Метод Select

Метод *Select* належить до найуживаніших методів LINQ, проте окремі його можливості, корисні для функціонального програмування, відомі порівняно мало.

Перша з таких можливостей уже продемонстрована в підрозділі 2.3: виклики *Select* можна об'єднувати в ланцюжки - безпосередньо один за одним або зберігаючи результат кожного виклику в окремій змінній. З погляду функціонального програмування обидва способи рівноцінні, і навіть виклик *ToArray()* після кожного етапу не порушує парадигми, якщо отримані масиви та їхні елементи надалі не змінюються.

Натомість від імперативного прийому, що полягає у створенні порожнього списку, переборі вихідної колекції циклом *foreach* і додаванні кожного нового елемента до списку, слід відмовлятися на користь методу *Select*: такий код коротший, легший для читання і не потребує змінного стану.

Доступ до індексу елемента

Нехай під час перетворення послідовності необхідно знати порядковий номер елемента. Наприклад, у лістингу 2.11 формується рейтинг страв за виручкою за зміну.

Лістинг 2.11 - Нумерація елементів за допомогою лічильника

```
1. var ranking = RestaurantData.ShiftSales
2.     .OrderByDescending(x => x.Revenue);
3.
4. var i = 1;
5. Console.WriteLine("Рейтинг страв за виручкою:");
6. foreach (var s in ranking)
7. {
8.     Console.WriteLine($"{i} - {s.Item.Name}");
9.     i++;
10.  }
```

Метод *Select* має перевантажену форму, яка надає доступ до індексу поточного елемента: для цього достатньо передати лямбда-вираз із двома параметрами, другий з яких є цілим числом - позицією елемента в послідовності (нумерація починається з нуля). Функціональний варіант наведено в лістингу 2.12.

Лістинг 2.12 - Нумерація елементів за допомогою Select з індексом

```
1. var ranking = RestaurantData.ShiftSales
2.     .OrderByDescending(x => x.Revenue);
3.
4. Console.WriteLine("Рейтинг страв за виручкою:");
5. var lines = ranking.Select((x, i) => $"{i + 1} - {x.Item.Name}");
6. Console.WriteLine(string.Join(Environment.NewLine, lines));
```

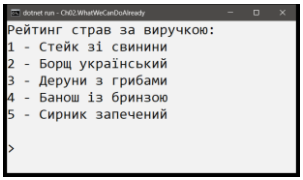


Рисунок 2.6 - Результат виконання прикладу «Індекс: функціонально»

Обидва варіанти дають однаковий результат і наочно ілюструють відмінність імперативного та декларативного підходів. В імперативному коді розробник самостійно створює змінну-лічильник і явно визначає місце її збільшення. Декларативний код не визначає ні місця зберігання індексу, ні способу його обчислення.

Примітка. Метод *string.Join* у лістингу 2.12 об'єднує колекцію рядків в один рядок. Це приклад **агрегування** (aggregation) - перетворення колекції елементів на одне значення, яке детально розглядається далі в цьому підрозділі.

Генерування послідовностей без вихідної колекції

Прийом з лістингу 2.12 застосовний, коли вихідна колекція вже існує. Якщо ж потрібно виконати певну кількість повторень за відсутності колекції, в імперативному стилі використовують цикл *for*. У функціональному стилі для цього застосовують статичні методи *Enumerable.Range* і *Enumerable.Repeat*, що генерують послідовність «з нічого».

Метод *Enumerable.Range* створює послідовність цілих чисел, що починається із заданого значення і містить задану кількість елементів, кожен з яких на одиницю більший за попередній (лістинг 2.13).

Лістинг 2.13 - Генерування послідовності методом Enumerable.Range

```
1. var a = Enumerable.Range(8, 5);
2. var s = string.Join(", ", a);
3. // s = "8, 9, 10, 11, 12": 5 елементів, починаючи з 8
```

До згенерованої послідовності можна застосовувати будь-які операції LINQ. У лістингу 2.14 формується таблиця вартості від однієї до п'яти порцій капучино.

Лістинг 2.14 - Таблиця вартості порцій

```
1. var cappuccino = SampleData.Item("B02");
2. var priceTable = Enumerable.Range(1, 5)
3.     .Select(n => $"{n} × капучино = {n * cappuccino.Price} грн");
4. var message = string.Join(Environment.NewLine, priceTable);
```

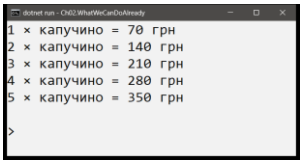


Рисунок 2.7 - Результат виконання прикладу «Range» (фрагмент)

Складніший випадок - перебір усіх елементів двовимірної структури. Нехай у готелі три поверхи по чотири номери, номер кімнати утворюється з номера поверху та порядкового номера на поверсі (101, 102, ...), а функція *GetRoomStatus* (лістинг 2.15) імітує звернення до сховища даних про зайнятість номерів.

Лістинг 2.15 - Імітація сховища даних про зайнятість номерів

```
1. // Імітація сховища: номери, зайняті на поточну дату
2. private static readonly HashSet<int> occupied =
3.     [101, 104, 202, 203, 301];
4.
5. public static string GetRoomStatus(int floor, int room) =>
6.     occupied.Contains(floor * 100 + room) ? "зайнятий" : "вільний";
```

Перелік усіх номерів і відбір вільних наведено в лістингу 2.16.

Лістинг 2.16 - Перебір усіх номерів готелю

```
1. var rooms = Enumerable.Range(1, 3)
2.     .SelectMany(floor => Enumerable.Range(1, 4)
3.         .Select(room => (Floor: floor, Room: room)));
4.
5. var free = rooms
6.     .Where(r => GetRoomStatus(r.Floor, r.Room) == "вільний")
7.     .Select(r => r.Floor * 100 + r.Room);
```

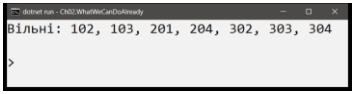


Рисунок 2.8 - Результат виконання прикладу «Сітка номерів»

Метод *Enumerable.Range(1, 3)* генерує номери поверхів [1, 2, 3]. Для кожного поверху вкладений виклик *Enumerable.Range(1, 4)* генерує номери кімнат, а метод *Select* перетворює кожну кімнату на кортеж (поверх, кімната). Метод *SelectMany* «вирівнює» отриману послідовність послідовностей в одну послідовність усіх можливих пар: (1, 1), (1, 2), ..., (3, 4). Далі відбираються вільні номери і обчислюються їхні тризначні номери.

Метод *Enumerable.Range* не дає змоги генерувати спадну послідовність. Якщо обхід номерів потрібно виконати у зворотному порядку - від верхнього поверху до нижнього, можна скористатися методом *Enumerable.Repeat*, який створює послідовність заданої довжини з однакових елементів (лістинг 2.17).

Лістинг 2.17 - Зворотний обхід номерів за допомогою Enumerable.Repeat

```
1. var roomsDown = Enumerable.Repeat(3, 3)
2.     .Select((floor, i) => floor - i)
3.     .SelectMany(floor => Enumerable.Repeat(4, 4)
4.         .Select((room, j) => (Floor: floor, Room: room - j)));
```

Виклик *Enumerable.Repeat(3, 3)* створює послідовність [3, 3, 3]. Перевантажена форма *Select* з індексом віднімає від кожного елемента його позицію: 3 - 0 = 3, 3 - 1 = 2, 3 - 2 = 1. Аналогічно генеруються номери кімнат 4, 3, 2, 1. Перші шість отриманих пар такі:

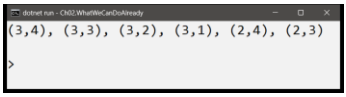


Рисунок 2.9 - Результат виконання прикладу «Repeat»

Той самий результат можна отримати і за допомогою методу *Reverse()*, застосованого до послідовності *Enumerable.Range*; прийом із *Enumerable.Repeat* і індексом корисний тоді, коли значення елемента має обчислюватися на основі його позиції.

2.4.2. Агрегування: від багатьох значень до одного

Розглянуті вище прийоми перетворюють N вхідних елементів на N вихідних. Іншим типовим призначенням циклів є **агрегування** - зведення множини елементів до одного значення: обчислення суми, кількості, середнього значення, інших статистичних показників або складніших узагальнень. У процедурному коді для цього використовують цикл і змінну стану, яка оновлюється на кожній ітерації (лістинг 2.18).

Лістинг 2.18 - Обчислення суми за допомогою циклу

```
1. var total = 0m;
2. foreach (var x in amounts)
3. {
4.     total += x;
5. }
```

Бібліотека LINQ містить для цього готовий метод (лістинг 2.19).

Лістинг 2.19 - Обчислення суми методом Sum

```
1. var totalLinq = amounts.Sum();
```

Методи *Sum* та *Average* мають і перевантажені форми, що приймають функцію вибору значення з елемента. Це дає змогу агрегувати властивість об'єктів без попереднього перетворення колекції (лістинг 2.20).

Лістинг 2.20 - Агрегування властивості об'єктів

```
1. var checks = RestaurantData.Checks;
2. var revenue = checks.Sum(x => x.Amount);
3. var averageCheck = checks.Average(x => x.Amount);
```

Стандартного методу обчислення медіани LINQ не містить (як і інших методів, яких немає в бібліотеці, наприклад моди чи стандартного відхилення). Медіану можна обчислити стислою функцією у функціональному стилі (лістинг 2.21): значення впорядковуються, і за парної кількості елементів медіаною є середнє двох центральних значень, а за непарної - центральне значення.

Лістинг 2.21 - Обчислення медіани

```
1. public static decimal Median(IEnumerable<decimal> values)
2. {
3.     bool IsEven(int number) => number % 2 == 0;
4.
5.     var sorted = values.OrderBy(x => x).ToArray();
6.     var count = sorted.Length;
7.     var median = IsEven(count)
8.         ? sorted.Skip(count / 2 - 1).Take(2).Average()
9.         : sorted.Skip(count / 2).First();
10.      return median;
11.  }
```

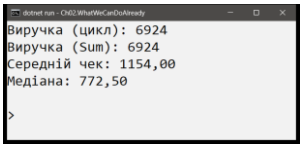


Рисунок 2.10 - Результат виконання прикладу «Агрегування»

Складніший випадок виникає, коли потрібно одночасно обчислити кілька агрегованих значень. Нехай необхідно визначити загальну кількість гостей і загальну виручку за всіма закритими рахунками зміни. Процедурний варіант наведено в лістингу 2.22.

Лістинг 2.22 - Обчислення двох підсумків в одному циклі

```
1. var totalGuests = 0;
2. var totalAmount = 0m;
3. foreach (var c in checks)
4. {
5.     totalGuests += c.Guests;
6.     totalAmount += c.Amount;
7. }
```

Два окремі виклики методу *Sum* призвели б до двох переборів колекції, що неефективно. Натомість можна скористатися методом **`Aggregate`**, який приймає два аргументи:

- **початкове значення** (seed) - вихідне значення акумулятора;

- **функцію агрегування** - функцію двох параметрів (поточного значення акумулятора та поточного елемента колекції), яка повертає нове значення акумулятора.

Початкове значення не обов'язково має бути простим типом: ним може бути й складний об'єкт. Для розглядуваної задачі достатньо кортежу з двох полів (лістинг 2.23).

Лістинг 2.23 - Обчислення двох підсумків методом Aggregate

```
1. var (guests, amount) = checks.Aggregate(
2.     (Guests: 0, Amount: 0m),
3.     (acc, c) => (acc.Guests + c.Guests, acc.Amount + c.Amount));
4.
5. var perGuest = Math.Round(amount / guests, 2);
```

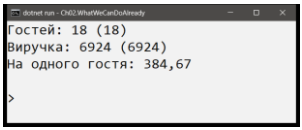


Рисунок 2.11 - Результат виконання прикладу «Aggregate»

Метод *Aggregate* є потужним інструментом, який доцільно ретельно опанувати. Він також ілюструє ще одне важливе для функціонального програмування поняття - рекурсію: результат кожного кроку агрегування стає вхідним значенням для наступного кроку.

2.4.3. Рекурсивна ітерація

Рекурсія лежить в основі багатьох функціональних способів перебору даних. У мові C# її використання має два суттєві обмеження:

- неправильно побудована рекурсія призводить до нескінченного циклу, що триває до примусового завершення програми або до вичерпання стека (виникає виняток *StackOverflowException*, який неможливо перехопити);
- рекурсія в мові C# потребує значно більше пам'яті, ніж інші способи ітерації, оскільки кожен виклик займає окремий кадр стека.

Способи подолання цих обмежень розглядаються в розділі 10; тут наведено найпростіший приклад. Нехай відома послідовність змін залишку інгредієнта на складі ресторану (поставки зі знаком «+», витрати зі знаком «-») і необхідно визначити позицію зміни, після якої залишок уперше став нульовим. Кінцеве значення залишку можна отримати методом *Aggregate*, однак у цій задачі потрібні проміжні значення, а перебір необхідно припинити, щойно умову буде виконано. Для складних об'єктів у реальних задачах дострокове припинення може суттєво підвищити продуктивність.

Імперативне рішення наведено в лістингу 2.24. Метод повертає значення -1, якщо залишок нульовий уже на початку; індекс зміни, після

якої залишок став нульовим; або *null*, якщо нульового залишку не досягнуто.

Лістинг 2.24 - Імперативний пошук моменту вичерпання запасу

```
1. public static int? FirstZeroImperative(int startingStock,
2.                                        int[] deltas)
3. {
4.     var current = startingStock;
5.     var i = -1;
6.     foreach (var d in deltas)
7.     {
8.         if (current == 0)
9.         {
10.              break;
11.          }
12.          i++;
13.          current += d;
14.      }
15.      return current == 0 ? i : null;
16.  }
```

Рекурсивне рішення наведено в лістингу 2.25.

Лістинг 2.25 - Рекурсивний пошук моменту вичерпання запасу

```
1. public static int? FirstZero(int[] deltas, int current, int i = -1) =>
2.     current == 0
3.         ? i
4.         : i + 1 >= deltas.Length
5.             ? null
6.             : FirstZero(deltas, current + deltas[i + 1], i + 1);
```

Для початкового залишку 10 кг і змін [+2, -12, +9] очікуваним результатом є індекс 1: після другої зміни залишок дорівнює 10 + 2 - 12 = 0, а третя зміна (+9) не повинна оброблятися (лістинг 2.26).

Лістинг 2.26 - Виклик функції пошуку

```
1. int[] deltas = [2, -12, 9];
2. var position = FirstZero(deltas, 10);
```

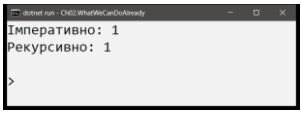


Рисунок 2.12 - Результат виконання прикладу «Пошук нульового залишку»

Рекурсивна функція є функціональною, однак має суттєвий недолік, пов'язаний із масштабуванням. Тіло функції - тернарний вираз: якщо залишок нульовий, повертається поточний індекс, інакше функція викликає саму себе з оновленими значеннями залишку та індексу. Третій виклик отримує результат, але перші два виклики залишаються незавершеними і зберігаються в стеку. Результат третього виклику повертається на рівень вище - другому виклику, той повертає його

першому, і лише перший виклик повертає остаточну відповідь. Трасування викликів наочно показує цей процес:

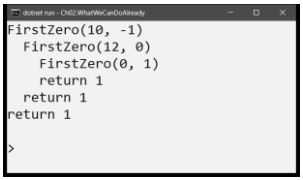


Рисунок 2.13 - Результат виконання прикладу «Стек рекурсивних викликів»

Для трьох елементів така поведінка не створює проблем, проте за сотень тисяч елементів стек може бути вичерпано. У чисто функціональних мовах, зокрема у F#, для цього передбачено **оптимізацію хвостової рекурсії** (tail call optimization): якщо рекурсивний виклик є останньою операцією функції, кадр стека поточного виклику повторно використовується, і пам'ять не накопичується. Середовище виконання .NET (CLR) підтримує хвостові виклики на рівні проміжного коду, однак компілятор мови C# їх не генерує. Прийоми, що дають змогу обійти це обмеження, розглядаються в розділі 10; до того часу рекурсію в мові C# слід застосовувати обережно.

2.5. Незмінність об'єктів у мові C#

Функціональний стиль у мові C# не обмежується бібліотекою LINQ. Іншою важливою його складовою є незмінність - неможливість змінити значення після його визначення. Нові засоби забезпечення незмінності, що з'явилися у версіях C# 8 і вище, розглядаються в розділі 3; тут ідеться про можливості, наявні в будь-якій версії платформи .NET.

У лістингу 2.27 наведено клас профілю гостя ресторану.

Лістинг 2.27 - Змінний клас профілю гостя

```
1. public class GuestProfile
2. {
3.     public string Name { get; set; } = "";
4.     public int Visits { get; set; }
5.     public DateTime FirstVisit { get; set; }
6.     public IEnumerable<decimal> Checks { get; set; } = [];
7.     public IList<string> Allergies { get; set; } = [];
8. }
```

Цей клас очевидно не є незмінним: будь-яку властивість можна замінити новим значенням через метод доступу *set*, а інтерфейс *IList<string>* надає методи додавання й видалення елементів.

Можна зробити методи доступу *set* закритими, так що об'єкт створюватиметься лише через конструктор із параметрами (лістинг 2.28).

Лістинг 2.28 - Клас профілю із закритими методами доступу set

```
1. public class GuestProfileV2
2. {
3.     public string Name { get; private set; }
4.     public int Visits { get; private set; }
5.     public DateTime FirstVisit { get; private set; }
6.     public IEnumerable<decimal> Checks { get; private set; }
7.     public IList<string> Allergies { get; private set; }
8.
9.     public GuestProfileV2(
10.          string name, int visits, DateTime firstVisit,
11.          IEnumerable<decimal> checks, IList<string> allergies)
12.      {
13.          Name = name;
14.          Visits = visits;
15.          FirstVisit = firstVisit;
16.          Checks = checks;
17.          Allergies = allergies;
18.      }
19.  }
```

Зовні класу замінити жодну властивість тепер неможливо. Усередині класу таку можливість формально збережено, проте її відсутність можна контролювати під час рецензування коду (code review). Властивості типів *string* і *DateTime* безпечні, оскільки ці типи незмінні; властивість типу *int* також безпечна, оскільки є значенням. Однак повної незмінності не досягнуто (лістинг 2.29).

Лістинг 2.29 - Зміна вмісту «незмінного» об'єкта

```
1. var checks = new List<decimal> { 640m, 905m };
2. var allergies = new List<string> { "горіхи" };
3. var profile = new GuestProfileV2("Олена", 2,
4.     new DateTime(2026, 3, 8), checks, allergies);
5.
6. profile.Allergies.Add("лактоза"); // через властивість
7. checks.Add(1480m);                // через зовнішнє посилання
8.
9. Console.WriteLine(string.Join(", ", profile.Allergies));
10.  Console.WriteLine(string.Join(", ", profile.Checks));
```

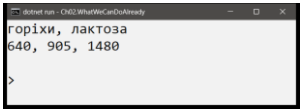


Рисунок 2.14 - Результат виконання прикладу «Незмінність» (фрагмент)

Вміст об'єкта змінився двома способами. Колекцію алергенів змінено через методи інтерфейсу *IList*, хоча саму властивість замінити неможливо. Список чеків змінено ззовні: властивість типу *IEnumerable<decimal>* на перший погляд захищена, але конструктору було передано об'єкт *List<decimal>*, посилання на який збереглося в коді, що викликав конструктор.

Проблема поширюється і на вкладені об'єкти (лістинг 2.30): усі властивості об'єкта *LoyaltyCard* залишаються змінними, якщо в його класі не застосовано той самий підхід.

Лістинг 2.30 - Змінний вкладений об'єкт

```
1. public class LoyaltyCard
2. {
3.     public string Number { get; set; } = "";
4.     public int Points { get; set; }
5. }
6.
7. public class GuestProfileV3
8. {
9.     public string Name { get; private set; }
10.      public LoyaltyCard Card { get; private set; }
11.
12.      public GuestProfileV3(string name, LoyaltyCard card)
13.      {
14.          Name = name;
15.          Card = card;
16.      }
17.  }
```

Частково ці проблеми розв'язує захисне копіювання вхідних колекцій і використання інтерфейсів лише для читання (лістинг 2.31): конструктор зберігає власні копії даних у вигляді масивів, доступних через інтерфейс *IReadOnlyList<T>*, а методи доступу *set* повністю відсутні.

Лістинг 2.31 - Захисне копіювання колекцій

```
1. public class GuestProfileV4
2. {
3.     public string Name { get; }
4.     public IReadOnlyList<decimal> Checks { get; }
5.     public IReadOnlyList<string> Allergies { get; }
6.
7.     public GuestProfileV4(string name,
8.         IEnumerable<decimal> checks, IEnumerable<string> allergies)
9.     {
10.          Name = name;
11.          Checks = checks.ToArray();       // власна копія даних
12.          Allergies = allergies.ToArray();
13.      }
14.  }
```



Рисунок 2.15 - Результат виконання прикладу «Незмінність» (фрагмент)

Проте навіть такий підхід не гарантує незмінності вкладених об'єктів довільних типів, зокрема класів платформи .NET і сторонніх бібліотек. Мова C# не має механізму, що забезпечував би незмінність усіх даних за замовчуванням, і з міркувань зворотної сумісності його поява

малоймовірна. Тому на практиці незмінність у функціональному коді на C# є насамперед **угодою**, прийнятою розробником або командою: створені об'єкти не змінюються, а будь-яка зміна даних виконується створенням нового об'єкта. Засоби мови, що полегшують дотримання цієї угоди, розглядаються в розділі 3.

2.6. Повний функціональний конвеєр обробки даних

Розглянуті прийоми доцільно поєднати в завершеному прикладі, що демонструє повний функціональний процес обробки даних. Нехай журнал замовлень ресторану зберігається у файлі формату CSV, а необхідно сформувати звіт, який для кожного дня містить загальну кількість замовлених порцій, кількість повернених (відхилених гостями) порцій і їх частку у відсотках. Розбір формату CSV свідомо спрощено: поля не містять лапок і ком; для промислових задач доцільно використовувати спеціалізовані бібліотеки.

Такий процес є типовим функціональним потоком: одне значення (текст файлу) розбивається на список, до списку застосовуються операції перетворення, після чого результат знову агрегується до одного значення (тексту звіту). Структуру CSV-файлу наведено в таблиці 2.1.

Таблиця 2.1 - Структура CSV-файлу журналу замовлень

Індекс	**Поле**	**Опис**
[0]	Day	Номер дня (ціле число)
[1]	Table	Номер столика
[2]	Waiter	Прізвище офіціанта
[3]	Dish	Назва страви
[4]	Portions	Кількість замовлених порцій
[5]	Returned	Кількість повернених порцій

Класи даних для записів журналу і рядків звіту наведено в лістингу 2.32. Методи доступу *init* (C# 9) дають змогу задавати властивості лише під час створення об'єкта.

Лістинг 2.32 - Класи запису журналу та рядка звіту

```
1. public class OrderRecord
2. {
3.     public int Day { get; init; }
4.     public int Table { get; init; }
5.     public string Waiter { get; init; } = "";
6.     public string Dish { get; init; } = "";
7.     public int Portions { get; init; }
8.     public int Returned { get; init; }
9. }
```

```
10.
11.  public class ReportLine
12.  {
13.      public int Day { get; init; }
14.      public int Portions { get; init; }
15.      public int Returned { get; init; }
16.      public decimal PercentageReturned { get; init; }
17.  }
```

Покрокова реалізація конвеєра наведена в лістингу 2.33.

Лістинг 2.33 - Покрокова побудова звіту за журналом замовлень

```
1. var text = File.ReadAllText(filePath);
2.
3. // Розбити текст файлу на рядки (кожен рядок - один запис)
4. var splitLines = text.Split(
5.     ['\r', '\n'], StringSplitOptions.RemoveEmptyEntries);
6.
7. // Розбити кожен рядок на масив полів за символом ','
8. var splitLinesAndFields = splitLines.Select(x => x.Split(','));
9.
10.  // Перетворити масив полів на об'єкт даних,
11.  // розібравши числові поля
12.  var parsedData = splitLinesAndFields.Select(x => new OrderRecord
13.  {
14.      Day = int.Parse(x[0]),
15.      Table = int.Parse(x[1]),
16.      Waiter = x[2],
17.      Dish = x[3],
18.      Portions = int.Parse(x[4]),
19.      Returned = int.Parse(x[5])
20.  });
21.
22.  // Згрупувати записи за днем
23.  var groupedByDay = parsedData.GroupBy(x => x.Day);
24.
25.  // Агрегувати кожну групу в кортеж із трьох полів:
26.  // D - день, P - усього порцій, R - повернено порцій
27.  var aggregatedLines = groupedByDay.Select(g =>
28.      g.Aggregate((D: g.Key, P: 0, R: 0),
29.          (acc, val) => (acc.D,
30.              acc.P + val.Portions,
31.              acc.R + val.Returned)));
32.
33.  // Перетворити кортежі на об'єкти звіту
34.  // з обчислюваним полем PercentageReturned
35.  var report = aggregatedLines.Select(x => new ReportLine
36.  {
37.      Day = x.D,
38.      Portions = x.P,
39.      Returned = x.R,
40.      PercentageReturned = Math.Round(100m * x.R / x.P, 1,
41.          MidpointRounding.AwayFromZero)
42.  });
43.
44.  // Відформатувати рядки звіту
45.  var reportTextLines = report.Select(x =>
46.      $"{x.Day,-5}{x.Portions,7}{x.Returned,11}" +
```

```
47.      $"{x.PercentageReturned,7:F1}");
48.
49.  var reportBody = string.Join(Environment.NewLine, reportTextLines);
50.  var reportHeader = "День  Порцій  Повернено      %";
51.
52.  // Звіт: заголовок, новий рядок, тіло звіту
53.  var finalReport =
54.      $"{reportHeader}{Environment.NewLine}{reportBody}";
```

Результат виконання:

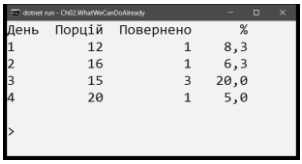


Рисунок 2.16 - Результат виконання прикладу «Звіт за CSV» (фрагмент)

Той самий конвеєр можна записати одним неперервним виразом із ланцюжка викликів (лістинг 2.34).

Лістинг 2.34 - Побудова звіту одним виразом

```
1. var reportTextLines = File.ReadAllText(filePath)
2.     .Split(['\r', '\n'], StringSplitOptions.RemoveEmptyEntries)
3.     .Select(x => x.Split(','))
4.     .GroupBy(x => int.Parse(x[0]))
5.     .Select(g => g.Aggregate((D: g.Key, P: 0, R: 0),
6.         (acc, val) => (acc.D,
7.             acc.P + int.Parse(val[4]),
8.             acc.R + int.Parse(val[5]))))
9.     .Select(x => $"{x.D,-5}{x.P,7}{x.R,11}" +
10.                 $"{Math.Round(100m * x.R / x.P, 1,
11.                     MidpointRounding.AwayFromZero),7:F1}");
12.
13.  var reportBody = string.Join(Environment.NewLine, reportTextLines);
14.  var reportHeader = "День  Порцій  Повернено      %";
15.  var finalReport =
16.      $"{reportHeader}{Environment.NewLine}{reportBody}";
```

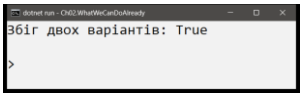


Рисунок 2.17 - Результат виконання прикладу «Звіт за CSV» (фрагмент)

Обидва підходи функціонально рівноцінні, і вибір між ними є питанням стилю. Покроковий варіант має дві переваги: імена проміжних змінних пояснюють зміст кожного кроку, слугуючи своєрідним коментарем; проміжні значення доступні для перегляду під час налагодження, що дає змогу визначити крок, на якому виникла помилка. Незалежно від обраного стилю головним критерієм має бути зрозумілість коду для всіх учасників команди.

2.7. Рекомендації щодо розвитку навичок функціонального програмування

Для закріплення розглянутих прийомів доцільно виконувати вправи, дотримуючись таких обмежень:

- усі змінні вважаються незмінними: після визначення значення змінної не змінюється, тобто кожна змінна фактично є константою;
- інструкції *if*, *for*, *foreach* і *while* не використовуються; умовна логіка допускається лише у вигляді тернарного виразу *умова ? значення1 : значення2*;
- функції, де це можливо, записуються у вигляді стислих виразів-тіл (expression-bodied members) і лямбда-виразів.

Для таких вправ придатні як завдання навчального курсу, так і задачі платформ змагального програмування (наприклад, Advent of Code чи Project Euler). Для швидкої перевірки фрагментів коду без створення повноцінного проєкту зручно використовувати інтерактивні засоби, такі як LINQPad або файлові програми C# (*dotnet run app.cs*), доступні починаючи з .NET 10.

2.8. Висновки

1. Бібліотека LINQ, що входить до складу мови C# з версії 3.0, реалізує ключові властивості функціонального програмування: функції вищого порядку, незмінність вихідних даних, використання виразів замість інструкцій.
2. Функціональний код зосереджується на результаті: кожне значення доцільно визначати в одному місці у вигляді виразу разом з усіма величинами, від яких воно залежить.
3. Перелічувані послідовності *IEnumerable<T>* реалізують відкладене виконання: дані формуються лише під час перебору, а порядок обчислень визначає середовище виконання. Матеріалізацію результату (*ToArray()*) доцільно відкладати, за винятком випадків багаторазового перебору та вимог продуктивності.
4. Перевантажена форма *Select* з індексом, методи *Enumerable.Range*, *Enumerable.Repeat* і *SelectMany* дають змогу замінити цикли з лічильниками декларативними виразами.
5. Агрегування виконується методами *Sum*, *Average*, *Count* та універсальним методом *Aggregate*, що дає змогу обчислювати кілька підсумків за один перебір.
6. Рекурсія є функціональною альтернативою циклам, проте в мові C# вона обмежена глибиною стека через відсутність оптимізації хвостових викликів.
7. Мова C# не забезпечує незмінність даних за замовчуванням; закриті методи доступу, інтерфейси лише для читання та захисне

копіювання лише частково розв'язують проблему, тому незмінність є насамперед угодою команди розробників.

8. Типовий функціональний конвеєр обробки даних має структуру «розбиття - перетворення - агрегування».

2.9. Контрольні запитання

1. Які недоліки має імперативний запит із формуванням нового списку в циклі *foreach*? Які з них усуває *yield return*?

2. Які властивості функціонального програмування реалізує бібліотека LINQ? Наведіть приклади.

3. Чому передавання джерела даних у метод як параметра спрощує тестування?

4. У чому полягає програмування, орієнтоване на результат? Які ризики має поступове заповнення властивостей об'єкта?

5. Що таке відкладене виконання? У який момент виконуються операції, задані методами LINQ?

6. Як зміниться порядок обчислень, якщо після кожного виклику *Select* викликати *ToArray()*? Коли матеріалізація результату є виправданою?

7. Як отримати порядковий номер елемента під час перетворення послідовності без змінної-лічильника?

8. Для чого призначені методи *Enumerable.Range*, *Enumerable.Repeat* і *SelectMany*?

9. Які параметри має метод *Aggregate*? У яких випадках він ефективніший за кілька викликів *Sum*?

10. Які обмеження має рекурсія в мові C#? Що таке оптимізація хвостової рекурсії?

11. Чому закритих методів доступу *set* недостатньо для забезпечення незмінності об'єкта?

12. Які етапи має типовий функціональний конвеєр обробки даних? Які переваги має покроковий запис конвеєра порівняно з одним виразом?

2.10. Практичні завдання

1. Перепишіть у функціональному стилі метод, який із переліку номерів готелю відбирає номери заданої категорії з вартістю, не вищою за вказану, та повертає їх номери в порядку зростання вартості.

2. Експериментально дослідіть відкладене виконання: додайте в ланцюжок із трьох етапів обслуговування фільтр *Where*, що відкидає замовлення з непарними номерами, і поясніть отриманий порядок записів у журналі. Перевірте, як зміниться журнал, якщо результат перебрати двічі.

3. Сформуйте за допомогою *Enumerable.Range* і *SelectMany* розклад бронювання конференц-залу: для кожного з п'яти робочих днів - годинні слоти з 9:00 до 18:00. Відберіть слоти, не зайняті за даними імітованого сховища.

4. Для колекції закритих рахунків обчисліть за один перебір (методом *Aggregate*) мінімальну, максимальну й середню суму рахунку та кількість рахунків, що перевищують 1000 грн.

5. Реалізуйте рекурсивну функцію, яка за послідовністю щоденних витрат мінібару номера визначає день, коли сумарні витрати вперше перевищать встановлений ліміт. Додайте трасування викликів і проаналізуйте глибину стека.

6. Реалізуйте функціональний конвеєр, який за CSV-файлом бронювань (дата заїзду, категорія номера, кількість ночей, ознака скасування) формує звіт за категоріями: кількість бронювань, кількість скасувань і частку скасувань у відсотках.

3. Еволюція функціональних можливостей мови C#

Мова C# з'явилася у 2002 році як об'єктно-орієнтована мова платформи .NET. Функціональні засоби додавалися до неї поступово: делегати першої версії вже дозволяли поводитися з методами як зі значеннями, версія C# 3.0 з лямбда-виразами й технологією LINQ зробила функціональний стиль повсякденною практикою, а починаючи з C# 7 (2017 рік) кожна нова версія розширює засоби зіставлення зі зразком і підтримки незмінності даних. Версія C# 15 (2026 рік) запровадила вбудовані дискриміновані об'єднання.

У розділі 2 розглядалися засоби, доступні практично в будь-якому програмному продукті мовою C#. Цей розділ простежує розвиток функціональних можливостей мови від першої до п'ятнадцятої версії. Кожен підрозділ присвячено одній версії або групі суміжних версій і містить приклади засобів, суттєвих для функціонального стилю. Засоби, що розглядаються докладно в інших розділах (LINQ, дерева виразів, *async/await*, дискриміновані об'єднання), тут лише позиціонуються в хронології з посиланням на відповідні розділи. Знання хронології має практичне значення: у промислових проєктах нерідко доводиться підтримувати код, що компілюється попередніми версіями мови, і розробник повинен розуміти, які засоби йому доступні.

Мета розділу - сформувати цілісне уявлення про еволюцію функціональних можливостей мови C# від версії 1.0 до версії 15 і навички їх застосування.

Після опрацювання розділу студент повинен:

- уміти використовувати кортежі для тимчасового групування даних у ланцюжках LINQ;
- знати, у яких версіях мови C# з'явилися основні функціональні засоби, і вміти обирати засоби, доступні в цільовій версії;
- знати можливості зіставлення зі зразком, що з'явилися у версіях C# 7-11, і вміти застосовувати вирази *switch*;
- розуміти відмінності між структурами лише для читання, методами доступу *init* і типами-записами;
- уміти створювати модифіковані копії записів за допомогою виразу *with*;
- розуміти призначення режиму nullable reference types і вміти обмежувати використання значення *null*;
- орієнтуватися в перспективних напрямах розвитку мови (дискриміновані об'єднання, активні шаблони);
- знати функціональні можливості версій C# 11-15: статичні абстрактні члени інтерфейсів, обов'язкові члени, первинні

конструктори, вирази колекцій, члени-розширення, ключове слово *field*, типи-об'єднання.

Приклади розділу містяться в проєктах *Ch03.ModernCSharp*, *Ch03.NullableWarnings* і *Ch03.ActivePatterns* (мова F#).

3.1. Огляд розвитку мови

Узагальнену хронологію функціональних можливостей мови C# наведено в таблиці 3.1. Версії мови тісно пов'язані з версіями платформи: компілятор за замовчуванням використовує версію мови, що відповідає цільовій платформі проєкту.

Таблиця 3.1 - Функціональні можливості мови C# за версіями

Версія (рік, платформа)	**Засоби мови**	**Значення для функціонального стилю**
C# 1.0 (2002, .NET Framework 1.0)	Делегати, події	Методи як значення: функції можна зберігати й передавати
C# 2.0 (2005, .NET Framework 2.0)	Узагальнення, анонімні методи, ітератори *yield*, типи *T?*	Функції без імені, замикання; відкладене обчислення послідовностей
C# 3.0 (2007, .NET Framework 3.5)	Лямбда-вирази, методи-розширення, анонімні типи, *var*, LINQ, дерева виразів	Функції вищого порядку, конвеєри обробки даних, код як дані
C# 4.0 (2010, .NET Framework 4)	Іменовані й необов'язкові параметри, узагальнена варіантність	Гнучкіші сигнатури функцій
C# 5.0 (2012, .NET Framework 4.5)	*async/await*	Композиція асинхронних обчислень
C# 6.0 (2015, .NET Framework 4.6)	Члени з тілом-виразом, *?.*, *nameof*, інтерполяція рядків	Стислі функції-вирази, безпечна робота з відсутніми значеннями
C# 7.0-7.3 (2017-2018)	Кортежі, деконструкція, зіставлення зі зразком, локальні	Групування значень без класів; розбір даних за формою

Версія (рік, платформа)	**Засоби мови**	**Значення для функціонального стилю**
	функції, вирази *throw*, *readonly struct*	
C# 8.0 (2019, .NET Core 3.0)	Вирази *switch*, шаблони властивостей, nullable reference types, статичні локальні функції, індекси й діапазони	Розгалуження як вираз; явне подання відсутності значення
C# 9.0 (2020, .NET 5)	Типи-записи, *init*, вираз *with*, реляційні й логічні шаблони, статичні лямбда-вирази	Незмінні дані з рівністю за значенням
C# 10 (2021, .NET 6)	Записи-структури, розширені шаблони властивостей, природний тип і явний тип результату лямбда-виразів	Лямбда-вирази як повноцінні значення
C# 11 (2022, .NET 7)	Шаблони списків, обов'язкові члени *required*, статичні абстрактні члени інтерфейсів	Розбір послідовностей; узагальнені алгебраїчні абстракції
C# 12 (2023, .NET 8)	Первинні конструктори, вирази колекцій, параметри лямбда-виразів за замовчуванням, псевдоніми будь-яких типів	Стисле створення незмінних колекцій і кортежних типів
C# 13 (2024, .NET 9)	Колекції в параметрах *params*	Незначне
C# 14 (2025, .NET 10)	Члени-розширення (*extension*), ключове слово *field*, присвоєння з умовою на *null*	Розширення типів функціональними операціями без зміни їх коду

Версія (рік, платформа)	**Засоби мови**	**Значення для функціонального стилю**
C# 15 (2026, .NET 11)	Типи-об'єднання (*union*), закриті ієрархії (*closed*), аргументи виразів колекцій	Вбудовані дискриміновані об'єднання з перевіркою повноти

Деякі засоби мови - події, динамічна типізація (*dynamic*, C# 4.0), мітки в інструкціях *break* і *continue* та оновлені правила безпечної роботи з пам'яттю (C# 15) - не стосуються функціонального стилю й у посібнику не розглядаються.

3.2. C# 1.0 і 2.0: делегати, анонімні методи та ітератори

Уже перша версія мови містила **делегати** (delegates) - типи, значеннями яких є посилання на методи з певною сигнатурою. Делегат є «типом функції»: змінна делегатного типу може зберігати метод, передаватися як аргумент і викликатися. Отже, функції вищого порядку були можливі в C# від початку (лістинги 3.1 і 3.2). Проте кожну функцію доводилося оголошувати окремим іменованим методом.

Лістинг 3.1 - Тип делегата і функція вищого порядку (C# 1.0)

```
1. // C# 1.0: тип делегата - «тип функції» з певною сигнатурою
2. public delegate decimal PriceRule(decimal price);
3.
4. public static class PriceRules
5. {
6.     public static decimal Regular(decimal price) => price;
7.     public static decimal Student(decimal price) => price * 0.9m;
8.
9.     // Функція вищого порядку: приймає правило як значення
10.      public static decimal Apply(PriceRule rule, decimal price) =>
11.          rule(price);
12.  }
```

Версія C# 2.0 (2005 рік) додала **анонімні методи** (anonymous methods) - функції без імені, що оголошуються безпосередньо в місці використання й можуть звертатися до локальних змінних зовнішнього методу, тобто є замиканнями (підрозділ 1.3.3). У лістингу 3.2 анонімний метод задає правило знижки «щасливої години».

Лістинг 3.2 - Іменований метод і анонімний метод як значення делегата

```
1. PriceRule rule = PriceRules.Student;   // група методів
2. Console.WriteLine(PriceRules.Apply(rule, 200m));
3.
4. // C# 2.0: анонімний метод - функція без імені
5. PriceRule happyHour = delegate (decimal p) { return p * 0.8m; };
6. Console.WriteLine(PriceRules.Apply(happyHour, 200m));
```

Тією самою версією запроваджено **узагальнення** (generics), без яких неможливі узагальнені функції вищого порядку. Колекція *List<T>* отримала методи *FindAll*, *ConvertAll*, *ForEach*, що приймали делегати, - прообраз майбутніх *Where* і *Select* (лістинг 3.3).

Лістинг 3.3 - Методи вищого порядку колекції List<T> (C# 2.0)

```
1. // C# 2.0: узагальнені колекції з методами вищого порядку
2. List<decimal> prices = new List<decimal>(new[] { 145m, 60m, 230m });
3. List<decimal> cheap = prices.FindAll(
4.     delegate (decimal p) { return p < 150m; });
5. List<string> labels = cheap.ConvertAll(
6.     delegate (decimal p) { return p + " грн"; });
7. Console.WriteLine(string.Join(", ", labels));
```

Третім функціональним засобом C# 2.0 стали **ітератори** (iterators): метод з інструкцією *yield return* повертає послідовність, елементи якої обчислюються лише під час перебору. Це **відкладене обчислення** (lazy evaluation) дає змогу описувати навіть нескінченні послідовності (лістинг 3.4). На ітераторах ґрунтується відкладене виконання запитів LINQ (підрозділ 2.3) і власні ітератори розділу 10.

Лістинг 3.4 - Нескінченна послідовність номерів замовлень

```
1. // C# 2.0: ітератор - відкладене обчислення послідовності
2. static IEnumerable<int> OrderNumbers(int first)
3. {
4.     for (var n = first; ; n++)
5.         yield return n;              // нескінченна послідовність
6. }
7.
8. foreach (var n in OrderNumbers(1001).Take(3))
9.     Console.Write(n + " ");
10.  Console.WriteLine();
```

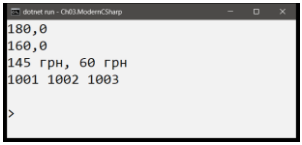


Рисунок 3.1 - Результат виконання прикладу «C# 1-2»

3.3. C# 3.0: лямбда-вирази, методи-розширення та LINQ

Версія C# 3.0 (2007 рік) є поворотним пунктом в історії мови. Вона запровадила **лямбда-вирази** - стислу форму анонімних функцій, назва якої походить від лямбда-числення (підрозділ 1.3.2). Лямбда-вираз може мати тіло-блок або тіло-вираз; типи параметрів компілятор, як правило, виводить із контексту. Одночасно в бібліотеці класів з'явилися узагальнені делегати *Func<...>* і *Action<...>*, що позбавили потреби оголошувати власні делегатні типи. Еволюцію запису однієї функції показано в лістингу 3.5.

Лістинг 3.5 - Анонімний метод і лямбда-вирази

```
1. decimal[] prices = [145m, 60m, 230m];
2.
3. // Та сама функція: анонімний метод (C# 2.0) і лямбда-вирази
4. Func<decimal, bool> a = delegate (decimal p) { return p < 150m; };
5. Func<decimal, bool> b = (decimal p) => { return p < 150m; };
6. Func<decimal, bool> c = p => p < 150m;
7.
8. Console.WriteLine($"{prices.Count(a)} {prices.Count(b)} " +
9.                   $"{prices.Count(c)}");
```

Методи-розширення (extension methods) дають змогу «дописувати» операції до наявних типів без зміни їх коду й викликати їх як методи екземпляра (лістинг 3.6). Саме методами-розширення інтерфейсу *IEnumerable<T>* реалізовано бібліотеку LINQ, а в цьому посібнику - більшість функціональних прийомів (розділи 4, 5, 7).

Лістинг 3.6 - Метод-розширення

```
1. // C# 3.0: метод-розширення «додає» операцію до наявного типу
2. public static decimal WithVat(this decimal price) => price * 1.2m;
```

Анонімні типи, неявна типізація змінних (*var*) і **синтаксис запитів** (*from ... where ... select*) завершують картину: усі вони потрібні для LINQ (лістинг 3.7). Лямбда-вираз, присвоєний змінній типу *Expression<...>*, компілюється в **дерево виразів** - цей засіб розглянуто в підрозділі 5.13.

Лістинг 3.7 - Запит LINQ з анонімним типом результату

```
1. var menu = new[]
2. {
3.     new { Name = "Борщ", Price = 145m },
4.     new { Name = "Узвар", Price = 45m }
5. };
6. // Синтаксис запитів, анонімний тип результату, var
7. var offer =
8.     from d in menu
9.     where d.Price > 50m
10.      select new { d.Name, Gross = d.Price.WithVat() };
11.  foreach (var o in offer)
12.      Console.WriteLine(o);
```

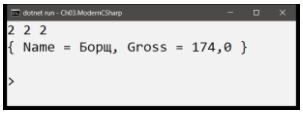


Рисунок 3.2 - Результат виконання прикладу «C# 3»

Технологію LINQ детально розглянуто в розділі 2, а синтаксис запитів як монадичну нотацію - у підрозділі 7.7.5.

3.4. C# 4.0-6.0: параметри за замовчуванням, async/await, члени-вирази

Версія C# 4.0 (2010 рік) додала **іменовані та необов'язкові параметри** (лістинги 3.8 і 3.9). Необов'язкові параметри частково розв'язують задачу, для якої у функціональному програмуванні використовують часткове застосування (розділ 9): функцію можна викликати, задавши лише частину аргументів.

Лістинг 3.8 - Функція з необов'язковими параметрами

```
1. // C# 4.0: необов'язкові та іменовані параметри
2. public static decimal Bill(decimal sum, decimal tip = 0.1m,
3.                            decimal discount = 0m) =>
4.     sum * (1 - discount) * (1 + tip);
```

Лістинг 3.9 - Виклики з іменованими аргументами

```
1. Console.WriteLine(Bill(1000m));
2. Console.WriteLine(Bill(1000m, discount: 0.05m));
3. Console.WriteLine(Bill(sum: 1000m, tip: 0m));
```

Версія C# 5.0 (2012 рік) запровадила **асинхронні функції** *async/await*. Асинхронна функція повертає задачу *Task<T>* - обіцянку результату в майбутньому, а оператор *await* «розгортає» результат і продовжує обчислення. З погляду функціонального програмування *Task<T>* є монадою, а *await* - її операцією зв'язування (підрозділ 7.10); композицію асинхронних функцій показано в лістингу 3.10.

Лістинг 3.10 - Композиція асинхронних функцій (C# 5.0)

```
1. // C# 5.0: async/await - композиція асинхронних функцій
2. static async Task<decimal> PriceAsync(string dish)
3. {
4.     await Task.Delay(10);                 // імітація звернення до БД
5.     return dish == "Борщ" ? 145m : 120m;
6. }
7.
8. static async Task<decimal> OrderAsync(string dish, int portions) =>
9.     (await PriceAsync(dish)) * portions;
```

Версія C# 6.0 (2015 рік) містила низку засобів, що зробили код стислішим і наблизили його до функціонального стилю: **члени з тілом-виразом** (=> для методів і властивостей), **умовний доступ** *?.*, що повертає *null* замість генерування винятку, інтерполяцію рядків і оператор *nameof* (лістинги 3.11 і 3.12).

Лістинг 3.11 - Члени з тілом-виразом

```
1. public sealed class Guest(string name, string? phone)
2. {
3.     public string Name { get; } = name;
4.     public string? Phone { get; } = phone;
5.     // C# 6.0: члени з тілом-виразом та інтерполяція рядків
6.     public string Card => $"{Name}: {Phone ?? "телефон невідомий"}";
7. }
```

Лістинг 3.12 - Умовний доступ і оператор nameof

```
1. Guest? guest = null;
2. // C# 6.0: умовний доступ ?. і nameof
3. Console.WriteLine(guest?.Phone?.Length ?? 0);
4. Console.WriteLine(new Guest("Олена", null).Card);
5. Console.WriteLine(nameof(Guest.Phone));
```

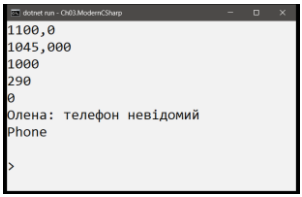


Рисунок 3.3 - Результат виконання прикладу «C# 4-6»

Члени з тілом-виразом закріпили в мові стиль, за якого функція є одним виразом (підрозділ 1.2.3); у цьому посібнику він використовується повсюдно.

3.5. C# 7.0-7.3: кортежі, зіставлення зі зразком і локальні функції

Версія C# 7 (2017 рік) започаткувала серію змін, що безпосередньо підтримують функціональний стиль. Їх можна поділити на дві групи: засоби **зіставлення зі зразком** і засоби, що полегшують дотримання **незмінності** даних. Обидві групи розвивалися в усіх наступних версіях, тому зіставлення зі зразком розглядається в підрозділах 3.5-3.9 послідовно, на одному наскрізному прикладі.

3.5.1. Кортежі та деконструкція

Кортеж (tuple) - це впорядкований набір значень, можливо різних типів, що об'єднуються без оголошення окремого класу. Синтаксична підтримка кортежів-значень (тип *System.ValueTuple*) з'явилася у версії C# 7. Кортежі зручні, коли кілька значень потрібно тимчасово зберегти разом, а потім одразу використати, - зокрема для передавання кількох значень між послідовними викликами *Select*, а також для повернення кількох значень із функції.

У лістингу 3.13 для кожного ідентифікатора бронювання за допомогою двох функцій пошуку отримуються дані бронювання та список супутників гостя. Обидва значення поєднуються в кортеж з іменованими елементами, після чого наступний виклик *Select* перетворює кортеж на рядок.

Лістинг 3.13 - Використання кортежу в ланцюжку Select

```
1. var bookingIds = new[] { 4665, 6718, 7101 };
2.
3. // Кожен ідентифікатор перетворюється на кортеж
4. // із двох властивостей: бронювання та супутники гостя
5. var bookingsWithCompanions = bookingIds.Select(x => (
6.     booking: GetBooking(x),
7.     companions: GetCompanions(x)));
8.
9. // x - кортеж, який перетворюється на рядок
10.  var details = bookingsWithCompanions.Select(x =>
11.      $"{x.booking.GuestName}, {x.booking.Room}, " +
12.      $"супутників: {x.companions.Length}");
```

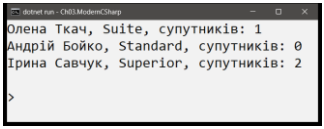


Рисунок 3.4 - Результат виконання прикладу «Кортежі»

Кортеж можна **деконструювати** - розкласти на окремі змінні одним оператором; значення, які не потрібні, позначаються символом **відкидання** (discard) _. Повернення кортежу з функції та його деконструкція в місці виклику замінюють *out*-параметри, що порушують функціональний стиль (лістинг 3.14).

Лістинг 3.14 - Деконструкція кортежу та відкидання

```
1. // C# 7.0: деконструкція кортежу та відкидання (_)
2. (string Dish, decimal Price, int Kcal) Lookup() =>
3.     ("Деруни", 110m, 450);
4.
5. var (dish, price, _) = Lookup();
6. Console.WriteLine($"{dish}: {price} грн");
7. Console.WriteLine(AveragePrice([145m, 60m, 230m]));
```

3.5.2. Локальні функції та вирази throw

Локальні функції (local functions) оголошуються всередині іншого методу й доступні лише в ньому. Вони дають змогу виділити допоміжну, зокрема рекурсивну, функцію, не засмічуючи клас. Ключове слово *throw* стало можливим використовувати у виразах, тож перевірку аргументів можна записати як частину виразу (лістинг 3.15).

Лістинг 3.15 - Рекурсивна локальна функція і вираз throw

```
1. public static decimal AveragePrice(IReadOnlyList<decimal> prices)
2. {
3.     // C# 7.0: вираз throw у виразі та локальна функція
4.     _ = prices.Count > 0
5.         ? true
6.         : throw new ArgumentException("Порожнє меню");
7.     return Sum(0, 0m) / prices.Count;
8.
9.     decimal Sum(int i, decimal acc) =>
10.          i == prices.Count ? acc : Sum(i + 1, acc + prices[i]);
11.  }
```

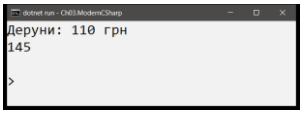


Рисунок 3.5 - Результат виконання прикладу «C# 7»

3.5.3. Зіставлення зі зразком: постановка задачі

Інструкція *switch* існує в мовах програмування дуже давно, однак її можливості обмежені порівнянням значення з константами. Функціональне програмування розвинуло цю ідею до механізму **зіставлення зі зразком** (pattern matching), який дає змогу обирати гілку обчислень залежно від типу, структури та значень властивостей об'єкта. У мові C# цей механізм з'явився у версії C# 7 і з того часу розширювався майже в кожній версії мови.

Для порівняння підходів розглядається задача нарахування бонусів у програмі лояльності мережі закладів HoReCa. Правила нарахування є умовними і мають навчальний характер:

- для базового рахунку бонус дорівнює сумі витрат, помноженій на ставку нарахування;
- золотий рахунок із витратами до 10 000 грн включно обробляється як базовий;
- для золотого рахунку з витратами понад 10 000 грн застосовується ставка, збільшена на додаткову ставку;
- для корпоративного рахунку окремо обліковуються витрати в готелі, і бонус нараховується на обидві суми;
- рахунок партнера-організатора банкетів додатково отримує фіксований бонус за проведений захід.

Відповідні класи наведено в лістингу 3.16.

Лістинг 3.16 - Класи рахунків програми лояльності

```
1. // Базовий рахунок програми лояльності:
2. // бонус = сума витрат × ставка нарахування
3. public class LoyaltyAccount
4. {
5.     public decimal Spent { get; set; }
```

```
6.     public decimal BonusRate { get; set; }
7. }
8.
9. // Золотий рівень: за витрат понад 10 000 грн
10.  // додається підвищена ставка
11.  public class GoldAccount : LoyaltyAccount
12.  {
13.      public decimal ExtraRate { get; set; }
14.  }
15.
16.  // Корпоративний рахунок: окремо обліковуються
17.  // витрати в готелі
18.  public class CorporateAccount : LoyaltyAccount
19.  {
20.      public decimal HotelSpent { get; set; }
21.  }
22.
23.  // Рахунок партнера-організатора банкетів:
24.  // фіксований бонус за кожен проведений захід
25.  public class PartnerAccount : LoyaltyAccount
26.  {
27.      public decimal EventBonus { get; set; }
28.  }
```

Процедурна реалізація функції нарахування бонусу наведена в лістингу 3.17.

Лістинг 3.17 - Процедурна реалізація нарахування бонусу

```
1. public static decimal CalculateBonus(LoyaltyAccount acc)
2. {
3.     // Якщо фактичний тип об'єкта - GoldAccount
4.     if (acc.GetType() == typeof(GoldAccount))
5.     {
6.         // приведення типу для доступу до ExtraRate
7.         var gold = (GoldAccount)acc;
8.         if (gold.Spent > 10000)
9.         {
10.              return gold.Spent * (gold.BonusRate + gold.ExtraRate);
11.          }
12.      }
13.      // Якщо фактичний тип - CorporateAccount
14.      if (acc.GetType() == typeof(CorporateAccount))
15.      {
16.          var corp = (CorporateAccount)acc;
17.          return corp.Spent * corp.BonusRate
18.                 + corp.HotelSpent * corp.BonusRate;
19.      }
20.      // Якщо фактичний тип - PartnerAccount
21.      if (acc.GetType() == typeof(PartnerAccount))
22.      {
23.          var partner = (PartnerAccount)acc;
24.          return partner.Spent * partner.BonusRate
25.                 + partner.EventBonus;
26.      }
27.      // Особливі правила не застосовуються
28.      return acc.Spent * acc.BonusRate;
29.  }
```

Як і більшість процедурного коду, ця функція громіздка, а для розуміння її логіки потрібне уважне прочитання. Перевірка типу через порівняння *GetType() == typeof(...)* до того ж враховує лише точний тип об'єкта і не спрацьовує для похідних класів, що, як буде показано далі, призводить до помилок. За додавання нових правил у процесі експлуатації складність функції швидко зростатиме.

Об'єктно-орієнтованим розв'язанням цієї задачі є використання інтерфейсу або поліморфізму - абстрактного базового класу з віртуальним методом нарахування бонусу. Проте в такому разі логіка розподіляється між багатьма класами замість того, щоб бути зосередженою в одній функції, яку легко прочитати. Далі розглядається, як цю задачу розв'язують засоби зіставлення зі зразком різних версій мови.

3.5.4. Зіставлення зі зразком у C# 7

Версія C# 7 запропонувала два засоби. Перший - оператор *is* із шаблоном типу (type pattern), який перевіряє тип об'єкта й одночасно оголошує змінну відповідного типу, до якої автоматично приводиться значення (лістинг 3.18).

Лістинг 3.18 - Перевірка типу оператором is

```
1. public static decimal CalculateBonus(LoyaltyAccount acc)
2. {
3.     if (acc is GoldAccount gold)
4.     {
5.         if (gold.Spent > 10000)
6.         {
7.             return gold.Spent * (gold.BonusRate + gold.ExtraRate);
8.         }
9.     }
10.      if (acc is CorporateAccount corp)
11.      {
12.          return corp.Spent * corp.BonusRate
13.                 + corp.HotelSpent * corp.BonusRate;
14.      }
15.      if (acc is PartnerAccount partner)
16.      {
17.          return partner.Spent * partner.BonusRate
18.                 + partner.EventBonus;
19.      }
20.      return acc.Spent * acc.BonusRate;
21.  }
```

Такий код компактніший і позбавлений явних приведень типів. Другий засіб - **перемикання за типом** (type switch) в інструкції *switch* з уточненням умови за допомогою ключового слова *when* (лістинг 3.19).

Лістинг 3.19 - Перемикання за типом в інструкції switch

```
1. public static decimal CalculateBonusSwitch(LoyaltyAccount acc)
2. {
3.     switch (acc)
```

```
4.     {
5.         case GoldAccount gold when gold.Spent > 10000:
6.             return gold.Spent * (gold.BonusRate + gold.ExtraRate);
7.         case CorporateAccount corp:
8.             return corp.Spent * corp.BonusRate
9.                    + corp.HotelSpent * corp.BonusRate;
10.          case PartnerAccount partner:
11.             return partner.Spent * partner.BonusRate
12.                    + partner.EventBonus;
13.          default:
14.             return acc.Spent * acc.BonusRate;
15.     }
16. }
```

3.5.5. Структури лише для читання

Структури (*struct*) у мові C# є типами-значеннями: під час передавання у функцію передається копія структури, а оригінал залишається незмінним. Для функціонального стилю це суттєва перевага, оскільки об'єктно-орієнтований прийом передавання об'єкта в метод для його зміни у функціональному програмуванні неприйнятний. Проте до версії C# 7.2 властивості структури можна було змінювати, тобто структури не були незмінними.

Модифікатор *readonly* у визначенні структури (C# 7.2) вимагає, щоб усі її поля були доступні лише для читання; спроба оголосити властивість із методом доступу *set* спричиняє помилку компіляції. Відповідно, усі властивості мають ініціалізуватися в конструкторі (лістинг 3.20).

Лістинг 3.20 - Структура лише для читання

```
1. public readonly struct SignatureDish
2. {
3.     public string Name { get; }
4.     public string Chef { get; }
5.     public IEnumerable<string> Ingredients { get; }
6.
7.     public SignatureDish(string name, string chef,
8.                          IEnumerable<string> ingredients)
9.     {
10.         Name = name;
11.         Chef = chef;
12.         Ingredients = ingredients;
13.     }
14. }
```

Недоліком такого підходу є необхідність оновлювати конструктор після додавання кожної нової властивості. Крім того, модифікатор *readonly* не поширюється на вміст об'єктів-посилань, що зберігаються в полях структури. У лістингу 3.21 поле *Ingredients* має тип *IList<string>*, а конструктору передано масив.

Лістинг 3.21 - Структура лише для читання з полем типу IList

```
1. public readonly struct SignatureDishV2
2. {
```

```
3.     public readonly string Name;
4.     public readonly string Chef;
5.     public readonly IList<string> Ingredients;
6.
7.     public SignatureDishV2(string name, string chef,
8.                            IList<string> ingredients)
9.     {
10.          Name = name;
11.          Chef = chef;
12.          Ingredients = ingredients;
13.      }
14.  }
```

Код компілюється, однак спроба додати елемент до колекції під час виконання спричиняє виняток (лістинг 3.22).

Лістинг 3.22 - Спроба змінити колекцію в структурі

```
1. var banosh = new SignatureDishV2(
2.     "Банош із бринзою",
3.     "Марченко",
4.     new[] { "кукурудзяна крупа", "сметана", "бринза" });
5.
6. try
7. {
8.     banosh.Ingredients.Add("шкварки");
9. }
10.  catch (NotSupportedException ex)
11.  {
12.      Console.WriteLine($"Виняток: {ex.GetType().Name}");
13.  }
```

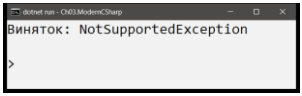


Рисунок 3.6 - Результат виконання прикладу «readonly struct і список»

Виняток виникає не через модифікатор *readonly*, а тому, що масив має фіксовану довжину і через інтерфейс *IList* не підтримує додавання елементів. Якби конструктору було передано об'єкт *List<string>*, елемент було б успішно додано. Отже, модифікатор *readonly* чітко виражає намір розробника і запобігає очевидним змінам, проте не гарантує незмінності вкладених об'єктів.

3.6. C# 8.0: вирази switch і nullable reference types

3.6.1. Вирази switch і шаблони властивостей

У версії C# 8 з'явилися **вирази `switch`** (switch expressions) і **шаблони властивостей** (property patterns). Вираз *switch* повертає значення, тому може становити тіло стислої функції; гілка за замовчуванням позначається символом відкидання _. Шаблон властивостей задається у фігурних дужках і дає змогу перевіряти значення властивостей об'єкта (лістинг 3.23).

Лістинг 3.23 - Вираз switch і шаблони властивостей (C# 8)

```
public static decimal CalculateBonus(LoyaltyAccount acc) =>
    acc switch
    {
        GoldAccount gold when gold.Spent > 10000 =>
            gold.Spent * (gold.BonusRate + gold.ExtraRate),
        CorporateAccount corp =>
            corp.Spent * corp.BonusRate
            + corp.HotelSpent * corp.BonusRate,
        PartnerAccount { EventBonus: 0 } partner =>
              partner.Spent * partner.BonusRate,
          PartnerAccount partner =>
              partner.Spent * partner.BonusRate + partner.EventBonus,
          _ => acc.Spent * acc.BonusRate
     };
```

Оскільки *switch* став виразом, функцію на його основі можна зберегти в делегаті *Func* і передавати як функцію вищого порядку. Нехай у готелі необхідно визначити, як змінилася категорія номера під час заселення порівняно із забронюваною: підвищилася (upgrade), знизилася (downgrade) чи залишилася тією самою. Можливі результати описано переліченням у лістингу 3.24.

Лістинг 3.24 - Перелічення результатів порівняння категорій

```
public enum RoomChange
{
    Upgrade,     // гість отримує номер вищої категорії
    Downgrade,   // номер нижчої категорії
    Same         // категорія збігається
}
```

Функцію порівняння подано у вигляді делегата, тілом якого є вираз *switch* над кортежем (лістинг 3.25). Перша гілка використовує шаблон відкидання з умовою *when* для випадку збігу категорій.

Лістинг 3.25 - Функція порівняння категорій номерів

```
public static readonly Func<RoomType, RoomType, RoomChange>
    CompareRooms = (booked, assigned) => (booked, assigned) switch
    {
        _ when booked == assigned => RoomChange.Same,
        (RoomType.Standard, RoomType.Superior) => RoomChange.Upgrade,
        (RoomType.Standard, RoomType.Suite) => RoomChange.Upgrade,
        (RoomType.Superior, RoomType.Suite) => RoomChange.Upgrade,
        _ => RoomChange.Downgrade
    };
```

Збережену в змінній логіку можна передати туди, де вона потрібна, наприклад як параметр функції формування звіту про заселення (лістинг 3.26).

Лістинг 3.26 - Функція вищого порядку для формування звіту

```
1. public static string FormatCheckIns(
2.     IEnumerable<(RoomType Booked, RoomType Assigned)> checkIns,
3.     Func<RoomType, RoomType, RoomChange> compare) =>
4.     string.Join(Environment.NewLine,
5.         checkIns.Select((x, i) =>
6.             $"Заселення {i + 1}: {compare(x.Booked, x.Assigned)}"));
```

Під час модульного тестування функції *FormatCheckIns* реальну логіку порівняння можна підмінити довільною тестовою функцією; сама логіка порівняння тестується окремо (лістинг 3.27).

Лістинг 3.27 - Виклик із реальною та підміненою функцією

```
1. (RoomType, RoomType)[] checkIns =
2. [
3.     (RoomType.Standard, RoomType.Suite),
4.     (RoomType.Suite, RoomType.Suite),
5.     (RoomType.Suite, RoomType.Superior)
6. ];
7. Console.WriteLine(FormatCheckIns(checkIns, CompareRooms));
8.
9. // Для тестування форматування реальну логіку
10.  // можна підмінити довільною функцією
11.  Console.WriteLine(FormatCheckIns(checkIns,
12.      (_, _) => RoomChange.Same));
```

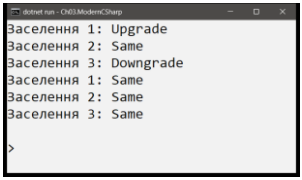


Рисунок 3.7 - Результат виконання прикладу «Зміна категорії номера»

3.6.2. Статичні локальні функції, індекси та діапазони

Модифікатор *static* для локальної функції забороняє їй захоплювати змінні зовнішнього методу. Це гарантований компілятором захист від прихованої залежності від стану: статична локальна функція залежить лише від своїх параметрів. Індекси від кінця (*^1*) та діапазони (*..*) дають змогу отримувати фрагменти масивів без циклів і проміжних змінних (лістинг 3.28).

Лістинг 3.28 - Статична локальна функція та діапазони

```
1. decimal[] week = [1200m, 950m, 1430m, 1100m, 1600m, 2100m, 1900m];
2.
3. // C# 8.0: індекси й діапазони
4. var weekend = week[^2..];
5. var workdays = week[..^2];
6.
7. // C# 8.0: статична локальна функція не може захопити
8. // зовнішніх змінних - гарантія відсутності прихованого стану
9. static decimal Avg(decimal[] xs) => xs.Sum() / xs.Length;
10.
11.  Console.WriteLine($"{Avg(workdays):F1} / {Avg(weekend):F1}");
```

3.6.3. Типи-посилання з підтримкою null

Режим **nullable reference types**, запроваджений у версії C# 8, не є новим типом даних, а є режимом аналізу коду компілятором. Він вмикається в файлі проєкту параметром <*Nullable*>*enable*</*Nullable*> (у шаблонах проєктів сучасних версій .NET увімкнений за замовчуванням).

Увімкнення режиму не змінює згенерованого коду, але додає попередження компілятора в ситуаціях, коли змінній або властивості посилального типу може бути присвоєно значення *null*. Лістинг 3.29 містить запис, властивості якого не ініціалізовано.

Лістинг 3.29 - Запис із неініціалізованими властивостями

```
1. public record GuestRequest
2. {
3.     public string GuestName { get; init; }
4.     public string Wishes { get; init; }
5. }
```

Спроба присвоїти такій властивості значення *null* (лістинг 3.30) також порушує вимоги режиму.

Лістинг 3.30 - Присвоєння null властивості, що не допускає null

```
1. public static GuestRequest Anonymous(GuestRequest request) =>
2.     request with { GuestName = null };
```

Під час збирання проєкту компілятор видає попередження CS8618 для кожної неініціалізованої властивості та CS8625 для присвоєння *null*:

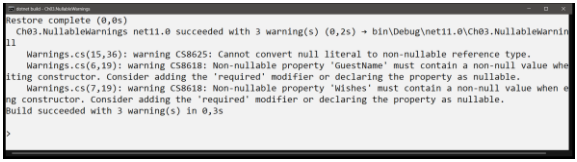


Рисунок 3.8 - Попередження компілятора під час збирання проєкту Ch03.NullableWarnings

Це лише попередження, тож код компілюється і виконується. Проте вони спрямовують розробника до коду, у якому ймовірність винятковой ситуації *NullReferenceException* значно нижча. Уникати значення *null* доцільно незалежно від парадигми програмування. Його автор, британський учений Ч. Е. Р. Гоар (C. A. R. Hoare), який запровадив порожнє посилання в мові ALGOL W у 1965 році, згодом назвав це рішення своєю «помилкою вартістю в мільярд доларів». Значення *null* ускладнює кодову базу і є джерелом помилок, тому попередження компілятора варто сприймати як вимогу.

Якщо відсутність значення є допустимою за змістом, тип властивості позначається символом *?* (лістинг 3.31).

Лістинг 3.31 - Запис із властивостями, що допускають null

```
1. public record GuestRequest
2. {
3.     public string GuestName { get; init; } = "";
4.     public string? Wishes { get; init; }     // побажання необов'язкові
5.     public DateOnly? ArrivalDate { get; init; }
6. }
```

Значення, що може бути відсутнім, доцільно обробляти безпосередньо на межі системи - там, де дані надходять від користувача чи сторонньої бібліотеки, - і перетворювати на безпечнішу структуру, не допускаючи поширення *null* у решті коду (лістинг 3.32). Систематичні засоби такого перетворення розглядаються в розділах 6 і 7.

Лістинг 3.32 - Обробка відсутнього значення на межі системи

```
1. // Значення, що може бути відсутнім, обробляється на межі
2. // системи й далі не поширюється
3. public static string DescribeWishes(GuestRequest request) =>
4.     string.IsNullOrWhiteSpace(request.Wishes)
5.         ? "без особливих побажань"
6.         : request.Wishes.Trim();
```

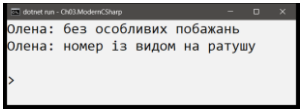


Рисунок 3.9 - Результат виконання прикладу «Nullable»

3.7. C# 9.0: типи-записи, init і нові шаблони

3.7.1. Реляційні та логічні шаблони

Версія C# 9 додала **реляційні шаблони** (relational patterns: <, <=, >, >=), **логічні шаблони** (*and*, *or*, *not*) та можливість використовувати шаблон типу без оголошення змінної, якщо її властивості не потрібні. Ці засоби помітно скорочують шаблонний код. У лістингу 3.33 додано нові правила: золотий рахунок має дві категорії з різними додатковими ставками, для

рахунку партнера враховується статус (призупинені рахунки бонусів не отримують), а для закритого рахунку бонус не нараховується.

Лістинг 3.33 - Реляційні та логічні шаблони (C# 9)

```
1. public static decimal CalculateBonus(LoyaltyAccount acc) =>
2.     acc switch
3.     {
4.         GoldAccount { Spent: > 10000 and <= 20000 } g =>
5.             g.Spent * (g.BonusRate + g.ExtraRate),
6.         GoldAccount { Spent: > 20000 } g =>
7.             g.Spent * (g.BonusRate + g.ExtraRate * 1.25m),
8.         CorporateAccount c =>
9.             c.Spent * c.BonusRate + c.HotelSpent * c.BonusRate,
10.          PartnerAccountV2 { Status: not AccountStatus.Suspended } p =>
11.              p.Spent * p.BonusRate + p.EventBonus,
12.          ClosedAccount => 0,
13.          _ => acc.Spent * acc.BonusRate
14.      };
```

3.7.2. Методи доступу init

У версії C# 9 з'явився новий метод доступу автоматичних властивостей - **`init`**. Властивість із методами доступу *get* і *init* можна встановити лише під час створення об'єкта (в ініціалізаторі об'єкта), після чого її значення змінити неможливо. Це дає змогу створювати незмінні структури й класи без громіздких конструкторів (лістинги 3.34 і 3.35).

Лістинг 3.34 - Структура з методами доступу init

```
1. public readonly struct SignatureDishV3
2. {
3.     public string Name { get; init; }
4.     public string Chef { get; init; }
5.     public IEnumerable<string> Ingredients { get; init; }
6. }
```

Лістинг 3.35 - Створення об'єкта з ініціалізатором

```
1. var banosh = new SignatureDishV3
2. {
3.     Name = "Банош із бринзою",
4.     Chef = "Марченко",
5.     Ingredients = ["кукурудзяна крупа", "сметана", "бринза"]
6. };
7. // banosh.Name = "Банош"; // помилка компіляції CS8852
```

Спроба присвоїти властивості нове значення після створення об'єкта спричиняє помилку компіляції CS8852. Проблема змінності вкладених колекцій і об'єктів, однак, зберігається.

3.7.3. Типи-записи

Типи-записи (record types), що з'явилися у версії C# 9, є одним із найважливіших для функціонального програмування засобів мови. Запис

(*record*, або *record class*) є посилальним типом, для якого компілятор автоматично генерує:

- **рівність за значенням** (value equality) - два записи рівні, якщо рівні значення всіх їхніх властивостей;
- метод *ToString* з переліком властивостей;
- підтримку **виразу `with`** для створення модифікованих копій.

У версії C# 10 додано **записи-структури** (*record struct*), що є типами-значеннями з тими самими можливостями. Звичайний запис при цьому залишається посилальним типом.

Перевага записів виявляється тоді, коли потрібно створити модифіковану версію об'єкта. Нехай на основі фірмової страви «Банош із бринзою» необхідно створити нову позицію меню, що відрізняється лише назвою. Для структури лише для читання довелося б явно скопіювати всі властивості (лістинг 3.36).

Лістинг 3.36 - Створення модифікованої копії структури

```
1. var banosh = new SignatureDishV3
2. {
3.     Name = "Банош із бринзою",
4.     Chef = "Марченко",
5.     Ingredients = ["кукурудзяна крупа", "сметана", "бринза"]
6. };
7.
8. var banoshWithMushrooms = new SignatureDishV3
9. {
10.      Name = $"{banosh.Name} і білими грибами",
11.      Chef = banosh.Chef,
12.      Ingredients = banosh.Ingredients
13.  };
```

Такий код відповідає функціональній парадигмі, проте потребує значного обсягу шаблонного коду, особливо для об'єктів стану з багатьма властивостями, що мають регулярно оновлюватися. Запис, оголошений у лістингу 3.37, разом із виразом *with* (лістинг 3.38) розв'язує цю проблему.

Лістинг 3.37 - Оголошення типу-запису

```
1. public record Dish
2. {
3.     public string Name { get; init; } = "";
4.     public string Chef { get; init; } = "";
5.     public IEnumerable<string> Ingredients { get; init; } = [];
6. }
```

Лістинг 3.38 - Створення модифікованої копії записом with

```
1. var banosh = new Dish
2. {
3.     Name = "Банош із бринзою",
4.     Chef = "Марченко",
5.     Ingredients = ["кукурудзяна крупа", "сметана", "бринза"]
6. };
7.
8. var banoshWithMushrooms = banosh with
9. {
10.      Name = $"{banosh.Name} і білими грибами"
11.  };
```

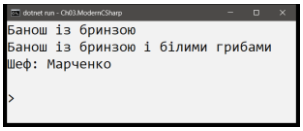


Рисунок 3.10 - Результат виконання прикладу «Записи та with» (фрагмент)

Вираз *with* створює новий запис, копіюючи всі властивості вихідного та замінюючи лише вказані. Вихідний запис залишається незмінним. Такий прийом є основою функціонального керування станом: поточний стан системи зберігається в записі, а кожна дія повертає його модифіковану копію. Рівність записів визначається значеннями властивостей, а не посиланнями (лістинг 3.39).

Лістинг 3.39 - Рівність записів за значенням

```
1. var copy = banosh with { };
2. Console.WriteLine($"copy == banosh: {copy == banosh}");
3. Console.WriteLine(
4.     $"ReferenceEquals: {ReferenceEquals(copy, banosh)}");
```

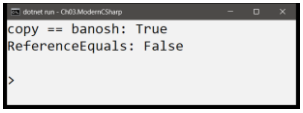


Рисунок 3.11 - Результат виконання прикладу «Записи та with» (фрагмент)

Запис можна оголосити й у стислій **позиційній формі** (positional record) - одним рядком із переліком параметрів (лістинг 3.40). Компілятор генерує для нього первинний конструктор, властивості з методами доступу *init* і метод деконструкції.

Лістинг 3.40 - Позиційний запис

```
1. public record DishCard(string Name, string Chef,
2.                        IEnumerable<string> Ingredients);
```

Екземпляр позиційного запису створюється викликом конструктора з аргументами в порядку оголошення або з іменованими аргументами в довільному порядку (лістинг 3.41).

Лістинг 3.41 - Створення та деконструкція позиційного запису

```
1. var card = new DishCard(
2.     "Банош із бринзою",
3.     "Марченко",
4.     ["кукурудзяна крупа", "сметана", "бринза"]);
5.
6. var sameCard = new DishCard(
7.     Ingredients: ["кукурудзяна крупа", "сметана", "бринза"],
8.     Chef: "Марченко",
9.     Name: "Банош із бринзою");
10.
11.  var (name, chef, _) = card; // деконструкція
```

Слід враховувати, що рівність записів є **поверхневою** (shallow): властивості-колекції порівнюються за посиланням, а не за вмістом (лістинг 3.42).

Лістинг 3.42 - Поверхнева рівність записів

```
1. // Рівність записів порівнює значення властивостей;
2. // колекції порівнюються за посиланням
3. Console.WriteLine($"card == sameCard: {card == sameCard}");
4. var sameIngredients = card with { Name = card.Name };
5. Console.WriteLine(
6.     $"card == sameIngredients: {card == sameIngredients}");
```

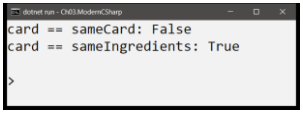


Рисунок 3.12 - Результат виконання прикладу «Позиційний запис» (фрагмент)

Записи *card* і *sameCard* не рівні, оскільки їхні властивості *Ingredients* посилаються на різні масиви з однаковим вмістом. Запис, отриманий виразом *with*, рівний вихідному, оскільки посилається на той самий масив.

Вибір між класичним і позиційним синтаксисом оголошення записів є питанням стилю; у більшості випадків вони рівноцінні.

3.7.4. Статичні лямбда-вирази

Подібно до статичних локальних функцій, модифікатор *static* перед лямбда-виразом (C# 9) забороняє захоплення зовнішніх змінних. Статичний лямбда-вираз гарантовано не є замиканням над змінним станом (підрозділ 1.3.3), а компілятор може не створювати для нього окремого об'єкта. Перший рядок лістингу 3.44 містить статичний лямбда-вираз.

3.8. C# 10: розширені шаблони та лямбда-вирази як значення

3.8.1. Розширені шаблони властивостей

У версії C# 10 з'явилися **розширені шаблони властивостей** (extended property patterns), що дають змогу звертатися до властивостей вкладених об'єктів через крапку. У лістингу 3.43 для рахунків партнерів, власниками яких є працівники мережі, фіксований бонус за захід зменшується вдвічі.

Лістинг 3.43 - Розширені шаблони властивостей (C# 10)

```
1. public static decimal CalculateBonus(LoyaltyAccount acc) =>
2.     acc switch
3.     {
4.         GoldAccount { Spent: > 10000 and <= 20000 } g =>
5.             g.Spent * (g.BonusRate + g.ExtraRate),
6.         GoldAccount { Spent: > 20000 } g =>
7.             g.Spent * (g.BonusRate + g.ExtraRate * 1.25m),
8.         CorporateAccount c =>
9.             c.Spent * c.BonusRate + c.HotelSpent * c.BonusRate,
10.          PartnerAccountV2 { Status: AccountStatus.Suspended } => 0,
11.          PartnerAccountV2 { Owner.IsEmployee: true } p =>
12.              p.Spent * p.BonusRate + p.EventBonus / 2,
13.          PartnerAccountV2 p =>
14.              p.Spent * p.BonusRate + p.EventBonus,
15.          ClosedAccount => 0,
16.          _ => acc.Spent * acc.BonusRate
17.      };
```

Формально тіло функції складається з одного виразу, хоча для зручності читання його розбито на багато рядків. Аналогічна логіка, реалізована інструкціями *if*, зайняла б у кілька разів більше коду. Результати обчислення бонусів усіма розглянутими версіями функції для набору тестових рахунків подано нижче (стовпці: процедурна реалізація, C# 7, C# 9, C# 10).

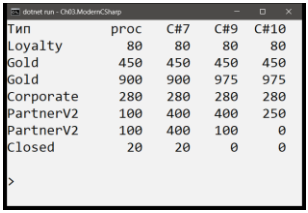


Рисунок 3.13 - Результат виконання прикладу «Бонуси лояльності»

Результати свідчать про таке. Для рахунку типу *PartnerAccountV2* (похідного від *PartnerAccount*) процедурна реалізація повертає неправильне значення 100: порівняння *GetType() == typeof(PartnerAccount)* не враховує успадкування, і об'єкт обробляється гілкою за замовчуванням. Оператор *is*

і шаблони типів враховують ієрархію класів. Відмінності між стовпцями C# 7, C# 9 і C# 10 для золотого рахунку з витратами 30 000 грн, призупиненого партнера, працівника-партнера й закритого рахунку зумовлені новими правилами, доданими в цих версіях функції.

3.8.2. Природний тип і тип результату лямбда-виразу

До версії C# 10 лямбда-вираз не мав власного типу: його можна було присвоїти лише змінній явно вказаного делегатного типу (*Func*<...>, *Action*<...>). Тепер компілятор визначає **природний тип** (natural type) лямбда-виразу, тож допустимий запис *var f = (decimal x) => x * 2;*. Якщо тип результату неможливо вивести (наприклад, вираз повертає *null* в одній з гілок), його вказують явно перед списком параметрів (лістинг 3.44).

Лістинг 3.44 - Статичний лямбда-вираз (C# 9) і явний тип результату (C# 10)

```
1. // C# 9.0: статичний лямбда-вираз (без захоплення змінних)
2. Func<decimal, decimal> half = static x => x / 2;
3. // C# 10: явний тип результату лямбда-виразу
4. var parse = int? (string s) => int.TryParse(s, out var n) ? n : null;
5.
6. Console.WriteLine(half(1430m));
7. Console.WriteLine($"{parse("4")} {parse("четверо") is null}");
```



Рисунок 3.14 - Результат виконання прикладу «C# 8-10»

Тією самою версією запроваджено **записи-структури** (*record struct*), розглянуті в підрозділі 3.7.3.

3.9. C# 11: шаблони списків і статичні абстрактні члени

3.9.1. Шаблони списків

У версії C# 11 з'явилися **шаблони списків** (list patterns), які дають змогу зіставляти вміст масиву або списку зі зразком і деконструювати його елементи в окремі змінні. Сфера застосування цього засобу вужча, проте в задачах розбору даних він виявляється надзвичайно ефективним.

Нехай кіоск самообслуговування ресторану приймає текстові команди, у яких дієслово-команда стоїть на першому місці: «ЗАМОВИТИ УЗВАР 2», «СКАСУВАТИ ЗАМОВЛЕННЯ 17», «РАХУНОК». Розбір таких команд за допомогою шаблонів списків наведено в лістингу 3.45.

Лістинг 3.45 - Розбір команд за допомогою шаблонів списків

```
1. // Розбір текстової команди кіоску самообслуговування
```

```
2. public static string Execute(string input) =>
3.     input.ToUpperInvariant().Split(' ',
4.         StringSplitOptions.RemoveEmptyEntries) switch
5.     {
6.         ["ЗАМОВИТИ", var dish, var qty]
7.             when int.TryParse(qty, out var n) => Order(dish, n),
8.         ["ЗАМОВИТИ", .. var rest] => Order(string.Join(' ', rest), 1),
9.         ["СКАСУВАТИ", "ЗАМОВЛЕННЯ", var id] => Cancel(id),
10.          ["РАХУНОК"] => "формування рахунку",
11.          [] => "порожня команда",
12.          _ => "невідома команда"
13.      };
```

Шаблон зрізу .. у виразі означає «будь-яка кількість елементів», а змінна після нього (.. *var rest*) отримує всі елементи, не зіставлені явно. Порядок гілок має значення: шаблон *["ЗАМОВИТИ", var dish, var qty]* з умовою перевіряється раніше, ніж загальніший шаблон *["ЗАМОВИТИ", .. var rest]*.

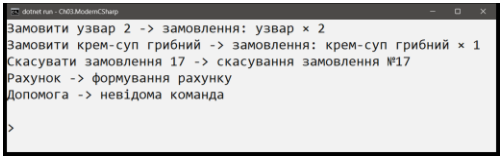


Рисунок 3.15 - Результат виконання прикладу «Команди кіоску»

Ще одне застосування шаблонів списків - розбір повного імені гостя на ім'я, прізвище та довільну кількість додаткових імен (лістинги 3.46 і 3.47).

Лістинг 3.46 - Клас для зберігання складових імені гостя

```
1. public class GuestName
2. {
3.     public string FirstName { get; init; } = "";
4.     public IEnumerable<string> MiddleNames { get; init; } = [];
5.     public string LastName { get; init; } = "";
6. }
```

Лістинг 3.47 - Виділення додаткових імен шаблоном списку

```
1. var input = "Анна Марія Олександра Коваль".Split(' ');
2.
3. var guest = new GuestName
4. {
5.     FirstName = input.First(),
6.     MiddleNames = input is [_, .. var middle, _] ? middle : [],
7.     LastName = input.Last()
8. };
```

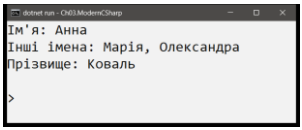


Рисунок 3.16 - Результат виконання прикладу «Розбір імені»

Шаблон *[_, .. var middle, _]* пропускає перший і останній елементи масиву, а всі елементи між ними поміщає в змінну *middle*.

3.9.2. Статичні абстрактні члени інтерфейсів

Починаючи з версії C# 11, інтерфейс може оголошувати **статичні абстрактні члени** (static abstract members): статичні властивості, методи й оператори, які зобов'язаний реалізувати тип, що підтримує інтерфейс. Узагальнений метод, параметр-тип якого обмежено таким інтерфейсом, може звертатися до цих членів через параметр-тип, наприклад *T.Empty*.

Цей засіб дає змогу виразити в мові C# алгебраїчні абстракції функціонального програмування. Найпростішою з них є **моноїд** (monoid) - множина значень з асоціативною бінарною операцією та нейтральним елементом (докладніше див. підрозділ 5.12). Прикладами моноїдів є числа з додаванням і нулем, рядки з конкатенацією й порожнім рядком, а також складені статистичні показники. У лістингу 3.48 інтерфейс *IMonoid<T>* вимагає нейтрального елемента *Empty* та оператора +; його реалізують грошова сума *Money* і статистика зміни *ShiftStats* (кількість чеків, виручка, найбільший чек).

Лістинг 3.48 - Інтерфейс моноїду зі статичними абстрактними членами

```
1. // Моноїд: нейтральний елемент і асоціативна операція поєднання
2. public interface IMonoid<T> where T : IMonoid<T>
3. {
4.     static abstract T Empty { get; }
5.     static abstract T operator +(T left, T right);
6. }
7.
8. public readonly record struct Money(decimal Amount) : IMonoid<Money>
9. {
10.      public static Money Empty => new(0m);
11.      public static Money operator +(Money a, Money b) =>
12.          new(a.Amount + b.Amount);
13.  }
14.
15.  // Статистика зміни: кількість чеків, виручка, найбільший чек
16.  public readonly record struct ShiftStats(int Checks, decimal Revenue,
17.      decimal MaxCheck) : IMonoid<ShiftStats>
18.  {
19.      public static ShiftStats Empty => new(0, 0m, 0m);
20.      public static ShiftStats operator +(ShiftStats a, ShiftStats b) =>
21.          new(a.Checks + b.Checks, a.Revenue + b.Revenue,
22.              Math.Max(a.MaxCheck, b.MaxCheck));
23.      public static ShiftStats Of(decimal check) => new(1, check,
     check);
24.  }
```

Для будь-якого моноїду достатньо однієї узагальненої функції згортки (лістинг 3.49): вона починає з нейтрального елемента й послідовно поєднує всі значення.

Лістинг 3.49 - Узагальнена згортка моноїду

```
1. // Одна узагальнена згортка для будь-якого моноїду
2. public static T Concat<T>(this IEnumerable<T> items)
3.     where T : IMonoid<T> =>
4.     items.Aggregate(T.Empty, (acc, x) => acc + x);
```

Одна й та сама функція обчислює і загальну виручку, і всі показники зміни, а для порожньої послідовності повертає нейтральний елемент без додаткових перевірок (лістинг 3.50).

Лістинг 3.50 - Згортка чеків зміни

```
1. decimal[] checks = [420m, 1250m, 380m, 960m];
2.
3. var total = checks.Select(c => new Money(c)).Concat();
4. var stats = checks.Select(ShiftStats.Of).Concat();
5. var none = Array.Empty<ShiftStats>().Concat();
6.
7. Console.WriteLine($"Виручка: {total.Amount} грн");
8. Console.WriteLine($"Зміна: {stats}");
9. Console.WriteLine($"Порожня зміна: {none}");
```

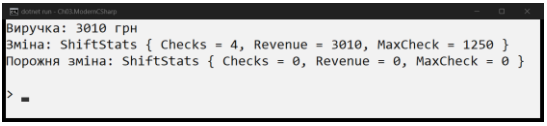


Рисунок 3.17 - Результат виконання прикладу «Моноїд»

До появи статичних абстрактних членів нейтральний елемент і операцію поєднання доводилося передавати узагальненій функції окремими параметрами, як у методі *Aggregate*.

3.9.3. Обов'язкові члени

Методи доступу *init* (підрозділ 3.7.2) забороняють змінювати властивість після створення об'єкта, проте не змушують її задати. Модифікатор `**required**` усуває цей недолік: об'єкт неможливо створити, не задавши в ініціалізаторі всіх обов'язкових властивостей, - порушення виявляє компілятор. Приклад використання *required* разом із ключовим словом *field* версії C# 14 наведено в підрозділі 3.12.2.

3.10. C# 12: первинні конструктори, вирази колекцій і псевдоніми типів

3.10.1. Первинні конструктори

Версія C# 12 поширила **первинні конструктори** (primary constructors), раніше доступні лише для записів, на всі класи та структури. З погляду функціонального стилю цей засіб потребує обережності. Параметр первинного конструктора класу не є властивістю чи полем лише

для читання: це змінна, доступна в усіх членах класу, і будь-який метод може змінити її значення. У лістингу 3.51 клас лічильника талонів змінює параметр *next* під час кожного виклику, тоді як запис *TicketRange* створює незмінні властивості й повертає новий діапазон.

Лістинг 3.51 - Первинний конструктор класу і позиційний запис

```
1. // Параметр первинного конструктора класу - змінна, а не поле
2. // лише для читання: метод може непомітно змінювати стан
3. public sealed class TicketCounter(int next)
4. {
5.     public int Issue() => next++;
6. }
7.
8. // Позиційний запис створює незмінні властивості
9. public sealed record TicketRange(int First, int Count)
10.  {
11.      public TicketRange Next() => this with { First = First + Count };
12.  }
```

Лістинг 3.52 - Використання лічильника і діапазону талонів

```
1. var counter = new TicketCounter(100);
2. Console.WriteLine($"{counter.Issue()}, {counter.Issue()}");
3.
4. var range = new TicketRange(100, 10);
5. var next = range.Next();
6. Console.WriteLine($"{range} -> {next}");
```

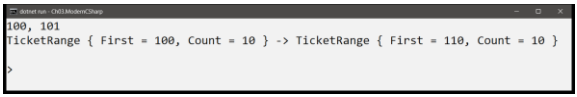


Рисунок 3.18 - Результат виконання прикладу «Первинний конструктор»

Отже, для незмінних даних доцільно використовувати позиційні записи; первинні конструктори класів зручні передусім для впровадження залежностей.

3.10.2. Вирази колекцій і незмінні колекції

Вирази колекцій (collection expressions, C# 12) - запис *[a, b, c]* з оператором розгортання .. - створюють колекцію будь-якого придатного типу, зокрема незмінні колекції простору імен *System.Collections.Immutable*. Операції над незмінною колекцією, як-от *Add*, не змінюють її, а повертають нову колекцію. У версії C# 15 першим елементом виразу колекції може бути *with(...)* - аргументи конструктора або фабричного методу колекції, наприклад порівнювач рядків. Для довідників, що після заповнення лише читаються, платформа .NET 8 і новіші пропонує **заморожені колекції** (*FrozenDictionary*, *FrozenSet*), оптимізовані для швидкого пошуку (лістинг 3.53).

Лістинг 3.53 - Незмінні та заморожені колекції

```
// Вирази колекцій створюють і незмінні колекції
ImmutableArray<string> starters = ["Салат", "Паштет"];
var extended = starters.Add("Брускета");

// Аргументи виразу колекції (C# 15): порівнювач без
// урахування регістру
HashSet<string> allergens =
    [with(StringComparer.OrdinalIgnoreCase),
     "Глютен", "ГЛЮТЕН", "Лактоза"];

// Заморожений словник: незмінний і оптимізований для читання
var prices = new Dictionary<string, decimal>
{
    ["Борщ"] = 145m, ["Вареники"] = 120m
}.ToFrozenDictionary();

Console.WriteLine($"{starters.Length} -> {extended.Length}");
Console.WriteLine($"Алергенів: {allergens.Count}");
Console.WriteLine($"Вареники: {prices["Вареники"]} грн");
```

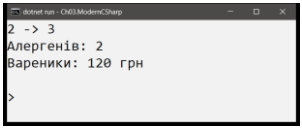


Рисунок 3.19 - Результат виконання прикладу «Незмінні колекції»

Масив *starters* після виклику *Add* залишився з двома елементами; множина алергенів містить два елементи, оскільки рядки «Глютен» і «ГЛЮТЕН» за заданим порівнювачем рівні.

3.10.3. Параметри лямбда-виразів за замовчуванням

Версія C# 12 дозволила параметри лямбда-виразів зі значеннями за замовчуванням (лістинг 3.54).

Лістинг 3.54 - Лямбда-вираз з параметром за замовчуванням

```
// Природний тип лямбда-виразу та параметр за замовчуванням
var withService = (decimal sum, decimal rate = 0.1m) =>
    sum * (1 + rate);

Console.WriteLine(withService(1000m));
Console.WriteLine(withService(1000m, 0.15m));
Console.WriteLine(withService.GetType().Name);
```

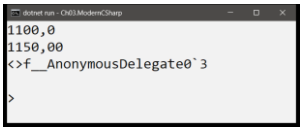


Рисунок 3.20 - Результат виконання прикладу «Природний тип лямбди»

Останній рядок виводу показує, що для лямбда-виразу з параметром за замовчуванням компілятор створює анонімний тип делегата: стандартні делегати *Func* не підтримують значень параметрів за замовчуванням.

3.10.4. Псевдоніми будь-яких типів

Директива *using* дозволяє тепер оголошувати псевдонім для будь-якого типу, зокрема кортежу. Це дає змогу надати змістовну назву типу-добутку (підрозділ 6.2.1) без оголошення класу чи запису (лістинг 3.55 і 3.56).

Лістинг 3.55 - Псевдонім кортежного типу

```
1. // C# 12: псевдонім для будь-якого типу, зокрема кортежу
2. using Charge = (decimal Amount, string Currency);
```

Лістинг 3.56 - Функція над значеннями кортежного типу

```
1. static Charge Add(Charge a, Charge b) =>
2.     a.Currency == b.Currency
3.         ? (a.Amount + b.Amount, a.Currency)
4.         : throw new InvalidOperationException("Різні валюти");
5.
6. Charge room = (2400m, "грн");
7. Charge spa = (900m, "грн");
8. Console.WriteLine(Add(room, spa));
```

3.11. C# 13: колекції в параметрах params

Версія C# 13 (2024 рік) не містила засобів, суттєвих для функціонального стилю. Модифікатор *params* тепер можна застосовувати не лише до масивів, а й до будь-якого типу колекції, зокрема *ReadOnlySpan<T>* та *IEnumerable<T>*, що зменшує кількість проміжних масивів під час виклику функцій зі змінною кількістю аргументів (лістинг 3.57).

Лістинг 3.57 - Параметр params типу ReadOnlySpan

```
1. // C# 13: params для будь-якої колекції, зокрема ReadOnlySpan
2. static decimal Total(params ReadOnlySpan<decimal> items)
3. {
4.     var sum = 0m;
5.     foreach (var x in items) sum += x;
6.     return sum;
7. }
8.
9. Console.WriteLine(Total(145m, 120m, 45m));
```



Рисунок 3.21 - Результат виконання прикладу «C# 12-13»

3.12. C# 14: члени-розширення і ключове слово field

3.12.1. Члени-розширення

Методи-розширення (C# 3.0) - основа LINQ і більшості функціональних прийомів посібника: вони дають змогу «дописувати» операції до наявних типів. Версія C# 14 узагальнила цей механізм до **членів-розширення** (extension members): у блоці *extension(...)* статичного класу оголошуються не лише методи, а й властивості, статичні члени та оператори для вказаного типу-отримувача. Блок може бути узагальненим. У лістингу 3.58 до будь-якої функції *Func<TIn, TOut>* додано операцію композиції *Then*, а до послідовності сум - властивості *Total* та *IsEmpty*.

Лістинг 3.58 - Члени-розширення для функцій і послідовностей

```
1. public static class FunctionExtensions
2. {
3.     // Члени-розширення (C# 14): блок extension оголошує
4.     // методи й властивості для вже наявного типу
5.     extension<TIn, TOut>(Func<TIn, TOut> f)
6.     {
7.         // Композиція: спочатку f, потім g
8.         public Func<TIn, TNext> Then<TNext>(Func<TOut, TNext> g) =>
9.             x => g(f(x));
10.     }
11.
12.     extension(IEnumerable<decimal> amounts)
13.     {
14.         public decimal Total => amounts.Sum();
15.         public bool IsEmpty => !amounts.Any();
16.     }
17. }
```

Композиція функцій, що розглядатиметься в розділі 5, записується тепер як ланцюжок викликів безпосередньо на делегатах (лістинг 3.59).

Лістинг 3.59 - Композиція функцій і властивості-розширення

```
1. Func<decimal, decimal> withVat = p => p * 1.2m;
2. Func<decimal, decimal> withTip = p => p * 1.1m;
3. var finalPrice = withVat.Then(withTip).Then(Math.Round);
4.
5. decimal[] bill = [180m, 95m, 240m];
6. Console.WriteLine($"Сума рахунку: {bill.Total}");
7. Console.WriteLine($"До сплати: {finalPrice(bill.Total)}");
8. Console.WriteLine($"Порожній: {Array.Empty<decimal>().IsEmpty}");
```

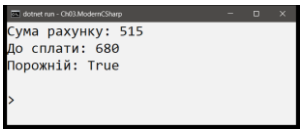


Рисунок 3.22 - Результат виконання прикладу «Члени-розширення»

3.12.2. Ключове слово field

Модифікатор *required* (підрозділ 3.9.3) гарантує, що обов'язкові властивості буде задано під час створення об'єкта. Ключове слово **`field`** (C# 14) надає доступ до поля, яке компілятор створює для автоматичної властивості, і дає змогу додати до методу доступу перетворення чи перевірку значення без явного оголошення поля (лістинг 3.60).

Лістинг 3.60 - Обов'язкові властивості з перевіркою значення

```
1. public sealed class MenuPosition
2. {
3.     // required: об'єкт не можна створити без цих властивостей;
4.     // field: доступ до автоматично створеного поля (C# 14)
5.     public required string Name
6.     {
7.         get;
8.         init => field = value.Trim();
9.     }
10.
11.      public required decimal Price
12.      {
13.          get;
14.          init => field = value >= 0
15.              ? value
16.              : throw new ArgumentOutOfRangeException(nameof(Price));
17.      }
18.  }
```

Поєднання *required*, *init* і перевірки в методі доступу гарантує, що об'єкт від моменту створення перебуває в коректному стані й надалі не змінюється (лістинг 3.61). Це відповідає функціональному принципу «некоректний стан неможливо подати».

Лістинг 3.61 - Створення позицій меню

```
1. var soup = new MenuPosition { Name = "  Борщ  ", Price = 145m };
2. Console.WriteLine($"[{soup.Name}] {soup.Price} грн");
3. try
4. {
5.     _ = new MenuPosition { Name = "Узвар", Price = -5m };
6. }
7. catch (ArgumentOutOfRangeException e)
8. {
9.     Console.WriteLine($"Відхилено: {e.ParamName}");
10.  }
```

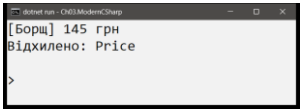


Рисунок 3.23 - Результат виконання прикладу «required і field»

Версія C# 14 також дозволила присвоєння з умовою на *null* (*order?.Status* = ...). Цей засіб спрощує імперативний код зі змінюваними об'єктами, проте у функціональному стилі, де об'єкти не змінюються, практично не застосовується.

3.13. C# 15: типи-об'єднання і закриті ієрархії

Дискримінований об'єднання (discriminated union) - це тип, значення якого належить до одного з кількох заздалегідь визначених варіантів. Тривалий час мова C# не мала для нього вбудованої підтримки, і об'єднання доводилося моделювати ієрархіями записів або бібліотеками NuGet, зокрема OneOf (див. розділ 13). Версія C# 15 запровадила **типи-об'єднання** (*union*) і **закриті ієрархії** (модифікатор *closed*), для яких компілятор перевіряє повноту виразів *switch*. Теоретичні основи й обидва способи - класичне моделювання та вбудовані засоби - докладно розглянуто в розділі 6.

Версія C# 15 також дозволила передавати аргументи конструктора колекції у виразі колекції (*with(...)*, див. лістинг 3.53).

3.14. Перспективи розвитку: активні шаблони

Активні шаблони (active patterns) - механізм мови F#, що розширює зіставлення зі зразком: у лівій частині гілки, тобто в самому зразку, можна використовувати функцію, визначену користувачем. У лістингу 3.62 активний шаблон *Price* розпізнає рядок, що містить невід'ємну ціну в гривнях.

Лістинг 3.62 - Активний шаблон мовою F#

```
// Активний шаблон: розпізнає рядок із ціною у гривнях
let (|Price|_|) (input: string) =
    let text = input.Replace("грн", "").Trim()
    match Decimal.TryParse(text, NumberStyles.Number,
                           CultureInfo.InvariantCulture) with
    | true, value when value >= 0m -> Some value
    | _ -> None

let describe input =
    match input with
    | Price p -> sprintf "ціна %M грн" p
    | _ -> "некоректне значення"
```

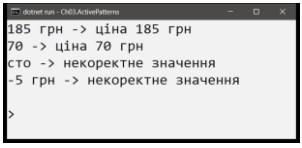


Рисунок 3.24 - Результат виконання прикладу «Активні шаблони»

Якщо рядок вдається розібрати, шаблон повертає значення *Some value*, і гілка *Price p* обирається зі зв'язуванням ціни зі змінною *p*; інакше обирається гілка за замовчуванням. Синтаксис мови F# не є предметом цього посібника; для його вивчення доцільно звернутися до спеціалізованої літератури. Оскільки мови C# і F# використовують спільне середовище

виконання, перенесення подібних механізмів у C# у майбутньому є можливим.

3.15. Висновки

1. Функціональні засоби мови C# формувалися поступово: делегати (C# 1.0) зробили методи значеннями, анонімні методи, узагальнення й ітератори (C# 2.0) - замикання та відкладене обчислення, лямбда-вирази, методи-розширення й LINQ (C# 3.0) - функціональний стиль повсякденною практикою, *async/await* (C# 5.0) - композицію асинхронних обчислень, члени з тілом-виразом (C# 6.0) - стиль «функція як вираз».

2. Починаючи з версії C# 7, кожна версія мови розширює підтримку функціонального програмування. Нові засоби поділяються на дві групи: зіставлення зі зразком і засоби забезпечення незмінності. Версія C# 15 завершила цю лінію розвитку вбудованими дискримінованими об'єднаннями.

3. Кортежі дають змогу тимчасово групувати значення без оголошення окремих класів, зокрема в ланцюжках LINQ.

4. Зіставлення зі зразком еволюціонувало від оператора *is* і перемикання за типом (C# 7) до виразів *switch* і шаблонів властивостей (C# 8), реляційних і логічних шаблонів (C# 9), розширених шаблонів властивостей (C# 10) і шаблонів списків (C# 11). Вираз *switch* можна зберегти в делегаті й використовувати як функцію вищого порядку.

5. Перевірка типу через *GetType()* не враховує успадкування, тоді як шаблони типів його враховують.

6. Структури лише для читання, методи доступу *init* і типи-записи полегшують дотримання незмінності, однак не поширюють її на вкладені об'єкти й колекції.

7. Типи-записи забезпечують рівність за значенням і створення модифікованих копій виразом *with*, що робить їх основним засобом подання незмінного стану. Рівність записів є поверхневою.

8. Режим nullable reference types дає змогу на етапі компіляції виявляти можливі присвоєння *null*; значення, що можуть бути відсутніми, слід обробляти на межі системи.

9. Версії C# 11-15 додали засоби, що безпосередньо підтримують функціональний стиль: статичні абстрактні члени інтерфейсів (узагальнені алгебраїчні абстракції на зразок моноїду), обов'язкові члени та ключове слово *field* (коректний незмінний стан від моменту створення), вирази колекцій і незмінні колекції, природний тип лямбда-виразів, члени-розширення, а також вбудовані типи-об'єднання й закриті ієрархії (C# 15).

10. Параметри первинних конструкторів класів є змінюваними змінними; для незмінних даних слід використовувати позиційні записи.

3.16. Контрольні запитання

1. Які функціональні засоби з'явилися в мові C# до версії 7? Яке значення мала версія C# 3.0?

2. Чим анонімний метод C# 2.0 відрізняється від лямбда-виразу C# 3.0? Чому обидва є замиканнями?

3. Що таке ітератор *yield return*? Як він пов'язаний із відкладеним виконанням запитів LINQ?

4. Яку гарантію дають статичні локальні функції та статичні лямбда-вирази?

5. Що таке кортеж? У яких випадках використання кортежу доцільніше за оголошення класу?

6. Які недоліки має процедурна перевірка типів за допомогою *GetType()* та явних приведень?

7. Які засоби зіставлення зі зразком з'явилися у версії C# 7?

8. Чим вираз *switch* відрізняється від інструкції *switch*? Чому вираз *switch* можна використовувати як функцію вищого порядку?

9. Що таке реляційні й логічні шаблони? У якій версії мови вони з'явилися?

10. Що таке розширені шаблони властивостей? Наведіть приклад.

11. Для чого призначені шаблони списків? Що означає шаблон зрізу ..?

12. Які гарантії дає модифікатор *readonly* для структури? Чому він не забезпечує незмінності вкладених колекцій?

13. Чим метод доступу *init* відрізняється від *set* і від закритого методу доступу *set*?

14. Які члени компілятор генерує для типу-запису? Чим запис відрізняється від запису-структури?

15. Як працює вираз *with*? Чому рівність записів називають поверхневою?

16. Яке призначення режиму nullable reference types? Чи змінює він поведінку програми під час виконання?

17. Що таке статичні абстрактні члени інтерфейсів? Як за їх допомогою виразити поняття моноїду?

18. Чим модифікатор *required* доповнює метод доступу *init*? Для чого призначене ключове слово *field*?

19. Чому параметри первинного конструктора класу не забезпечують незмінності?

20. Чим члени-розширення C# 14 відрізняються від методів-розширення? Наведіть приклад їх використання для композиції функцій.

21. Які засоби для дискримінованих об'єднань з'явилися у версії C# 15?

22. Що таке дискриміновані об'єднання та активні шаблони? Які засоби доступні для них у мовах C# і F#?

3.17. Практичні завдання

1. Сформуйте за допомогою кортежів звіт для адміністратора готелю: для кожного номера на поверсі - категорія, прізвище гостя (якщо номер зайнятий) і кількість ночей до виїзду. Використайте ланцюжок методів *Select* без оголошення допоміжних класів.

2. Реалізуйте функцію розрахунку вартості доставки замовлення ресторану за допомогою виразу *switch*: врахуйте тип замовлення (самовивіз, кур'єр, таксі), відстань (реляційні шаблони) та статус клієнта (логічні шаблони).

3. Додайте до ієрархії рахунків лояльності новий тип *StudentAccount* зі знижкою на бонусну ставку, а до функції з лістингу 3.43 - відповідну гілку з розширеним шаблоном властивостей. Перевірте, як новий тип обробляється процедурною версією функції, і поясніть результат.

4. Реалізуйте розбір команд голосового помічника номера готелю («ПРИБИРАННЯ ЗАРАЗ», «ПРИБИРАННЯ О 14», «РУМ-СЕРВІС КАВА 2», «ВИЇЗД») за допомогою шаблонів списків.

5. Опишіть стан столика ресторану (номер, кількість гостей, список позицій замовлення, статус) у вигляді позиційного запису. Реалізуйте функції додавання позиції, зміни статусу та закриття рахунку, кожна з яких повертає новий запис, створений виразом *with*.

6. Увімкніть у навчальному проєкті режим nullable reference types і усуньте всі попередження компілятора, не використовуючи оператор *!*.

7. Реалізуйте моноїд *InventoryDelta* (зміна залишків складу: словник «інгредієнт - кількість»), у якому операція поєднання підсумовує кількості однакових інгредієнтів. За допомогою узагальненої функції *Concat* обчисліть підсумкову зміну залишків за день за списком поставок і списань.

8. Оголосіть позицію прайс-листа постачальника з обов'язковими властивостями (назва, одиниця виміру, ціна, мінімальна партія), перевіркою значень у методах доступу з ключовим словом *field* і нормалізацією назви. Переконайтеся, що об'єкт з некоректними значеннями створити неможливо.

9. Перепишіть фрагмент коду в стилі C# 2.0 (методи *FindAll* і *ConvertAll* з анонімними методами) засобами C# 3.0 (LINQ і лямбда-

вирази) та C# 12 (вирази колекцій). Порівняйте обсяг і читабельність трьох варіантів.

4. Функціональні прийоми, що спрощують код

У попередніх розділах розглядалися функціональні засоби мови C# у тому вигляді, у якому їх передбачили розробники мови. У цьому розділі показано, як за допомогою делегатів, перелічуваних послідовностей і методів-розширень можна вийти за межі стандартних можливостей мови, позбутися значної частини шаблонного коду та реалізувати окремі сучасні функціональні засоби в ранніх версіях C#.

Розглядаються три групи прийомів:

- **делегати `Func` у колекціях** - розміщення функцій у перелічуваних послідовностях та обробка їх засобами LINQ;
- **делегати `Func` як фільтри** - функції, що слугують проміжним шаром між кодом і джерелом даних;
- **власні способи перебору** - використання перелічувача (enumerator) для реалізації нестандартної поведінки ітерації.

Мета розділу - сформувати навички застосування делегатів, перелічуваних послідовностей і методів-розширень для побудови лаконічного декларативного коду.

Після опрацювання розділу студент повинен:

- уміти зберігати функції в колекціях і обробляти їх методами LINQ;
- уміти будувати валідатори на основі масивів правил;
- уміти реалізовувати спрощене зіставлення зі зразком засобами C# 7 і раніших версій;
- уміти створювати безпечні функції пошуку на основі словників і функції розбору значень зі значеннями за замовчуванням;
- розуміти роботу перелічувача та вміти реалізовувати власні методи перебору пар сусідніх елементів і ітерації до виконання умови.

Приклади розділу містяться в проєкті *Ch04.WorkSmart*.

4.1. Делегати Func

Делегатний тип *Func* описує функцію, збережену як значення змінної: визначаються типи параметрів і тип результату, а виклик не відрізняється від виклику звичайної функції (лістинг 4.1).

Лістинг 4.1 - Функція привітання гостя у вигляді делегата

```
1. private static readonly Func<Guest, DateOnly, string> Greet =
2.     (guest, today) => $"{today:dd.MM.yyyy}: вітаємо, {guest.Name}!";
```

Останній узагальнений параметр у кутових дужках визначає тип результату, усі попередні - типи параметрів. Делегат у лістингу 4.1 приймає об'єкт гостя та дату і повертає рядок. Надалі делегати *Func*

використовуються в посібнику постійно, тому впевнене володіння ними є необхідною передумовою засвоєння подальшого матеріалу.

4.1.1. Делегати в колекціях

Делегати найчастіше використовують як параметри функцій, проте їх можна також розміщувати в перелічуваних послідовностях. Найочевидніше застосування такого прийому - застосування набору функцій до одних і тих самих даних. У лістингу 4.2 з одного об'єкта бронювання формується кілька рядків його опису, які потім агрегуються в один рядок методом *string.Join*.

Лістинг 4.2 - Опис бронювання набором функцій

```
1. private static readonly IEnumerable<Func<Booking, string>> descriptors =
2. [
3.     x => $"Бронювання №{x.Id}",
4.     x => $"Гість: {x.GuestName}",
5.     x => $"Номер: {x.Room}, гостей: {x.Guests}",
6.     x => $"Заїзд: {x.CheckIn:dd.MM.yyyy}, ночей: {x.Nights}",
7.     x => $"Виїзд: {x.CheckIn.AddDays(x.Nights):dd.MM.yyyy}"
8. ];
9.
10.  public static string DescribeBooking(Booking booking) =>
11.      string.Join(Environment.NewLine,
12.                  descriptors.Select(x => x(booking)));
```

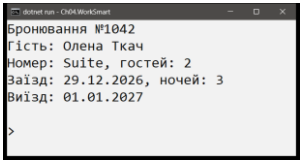


Рисунок 4.1 - Результат виконання прикладу «Func у колекціях» (фрагмент)

Порівняно з послідовним формуванням тексту за допомогою *StringBuilder* такий підхід має кілька переваг. По-перше, набір функцій можна формувати динамічно, обираючи правила відображення залежно від умов. По-друге, перелічувана послідовність підтримує відкладене виконання (див. розділ 2): елементи обчислюються по одному і лише тоді, коли до них доходить черга. Якщо обчислення елементів потребує звернення до зовнішніх систем або значних ресурсів, це дає змогу уникнути зайвої роботи. Наприклад, метод *Any* припиняє перебір, щойно знаходить перший елемент, що задовольняє умову, і решта елементів не обчислюється. По-третє, такий код простий у супроводі: додавання нового рядка опису зводиться до додавання ще одного елемента масиву, а сама структура стримує розробників від розміщення в цьому місці складної логіки.

4.1.2. Валідатор на основі масиву правил

Функції перевірки коректності даних (валідатори) в імперативному стилі зазвичай мають вигляд послідовності інструкцій *if* з достроковим поверненням результату. У лістингу 4.3 наведено перевірку заявки на бронювання номера.

Лістинг 4.3 - Імперативна перевірка заявки на бронювання

```
1. public static bool IsRequestValidImperative(BookingRequest r,
2.                                             DateOnly today)
3. {
4.     if (string.IsNullOrWhiteSpace(r.GuestName))
5.         return false;
6.     if (!r.Email.Contains('@'))
7.         return false;
8.     if (r.CheckIn < today)
9.         return false;
10.      if (r.Nights < 1 || r.Nights > 30)
11.          return false;
12.      if (r.Guests < 1)
13.          return false;
14.      if (r.Guests > Capacity(r.Room))
15.          return false;
16.      return true;
17.  }
```

Для порівняно простого набору правил написано багато однотипного коду, а додавання кожного нового правила потребує ще двох рядків, з яких змістовним є лише один. Той самий валідатор можна побудувати як масив функцій, кожна з яких перевіряє одне правило і повертає значення типу *bool* (лістинг 4.4).

Лістинг 4.4 - Валідатор на основі масиву функцій

```
1. public static bool IsRequestValidArray(BookingRequest request,
2.                                        DateOnly today) =>
3.     new Func<BookingRequest, bool>[]
4.     {
5.         x => !string.IsNullOrWhiteSpace(x.GuestName),
6.         x => x.Email.Contains('@'),
7.         x => x.CheckIn >= today,
8.         x => x.Nights is >= 1 and <= 30,
9.         x => x.Guests >= 1,
10.          x => x.Guests <= Capacity(x.Room)
11.      }.All(rule => rule(request));
```

Метод LINQ *All* застосовує передану функцію до всіх елементів масиву. Якщо хоча б для одного елемента результат дорівнює *false*, перебір припиняється достроково (наступні правила не обчислюються) і метод повертає *false*; якщо всі правила виконуються - *true*. Інструкції *if* і дострокові повернення, які в лістингу 4.3 доводилося писати явно, тепер неявно закладено в саму структуру.

Цей прийом доцільно узагальнити у вигляді методів-розширень (лістинг 4.5).

Лістинг 4.5 - Узагальнені методи-розширення для валідації

```
1. public static class ValidationExtensions
2. {
3.     public static bool IsValid<T>(this T @this,
4.                                   params Func<T, bool>[] rules) =>
5.         rules.All(rule => rule(@this));
6.
7.     public static bool IsInvalid<T>(this T @this,
8.                                    params Func<T, bool>[] rules) =>
9.         rules.Any(rule => rule(@this));
10.  }
```

Узагальнений метод *IsValid* ще більше скорочує валідатор і може використовуватися для перевірки об'єктів будь-якого типу (лістинг 4.6).

Лістинг 4.6 - Валідатор на основі методу IsValid

```
1. public static bool IsRequestValid(BookingRequest request,
2.                                   DateOnly today) =>
3.     request.IsValid(
4.         x => !string.IsNullOrWhiteSpace(x.GuestName),
5.         x => x.Email.Contains('@'),
6.         x => x.CheckIn >= today,
7.         x => x.Nights is >= 1 and <= 30,
8.         x => x.Guests >= 1,
9.         x => x.Guests <= Capacity(x.Room));
```

Метод *IsInvalid*, побудований на основі *Any*, дає змогу записувати правила в інверсній формі - як умови помилки, ближче до вихідного імперативного коду (лістинг 4.7).

Лістинг 4.7 - Валідатор на основі методу IsInvalid

```
1. public static bool IsRequestValidInverse(BookingRequest request,
2.                                          DateOnly today) =>
3.     !request.IsInvalid(
4.         x => string.IsNullOrWhiteSpace(x.GuestName),
5.         x => !x.Email.Contains('@'),
6.         x => x.CheckIn < today,
7.         x => x.Nights is < 1 or > 30,
8.         x => x.Guests < 1,
9.         x => x.Guests > Capacity(x.Room));
```

Результати перевірки чотирьох заявок (коректної, з перевищенням місткості номера, з некоректною адресою електронної пошти та з датою заїзду в минулому) усіма чотирма варіантами валідатора збігаються:

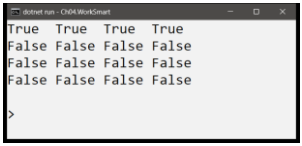


Рисунок 4.2 - Результат виконання прикладу «Валідація заявки»

Слід звернути увагу, що поточна дата передається у валідатор як параметр, а не отримується всередині через *DateTime.Now*. Завдяки цьому валідатор є чистою функцією (див. підрозділ 1.2.4).

4.1.3. Зіставлення зі зразком для ранніх версій C#

Вбудоване зіставлення зі зразком (див. розділ 3) недоступне в ранніх версіях мови. Проте його спрощений аналог можна реалізувати власними засобами, починаючи з C# 7. Такий аналог значно поступається вбудованим засобам за виразністю, але надає частину їхніх переваг.

Нехай ресторан надає знижку на банкет залежно від суми замовлення: до 10 000 грн включно знижка не надається; до 30 000 грн - 5 %; до 60 000 грн - 10 %; понад 60 000 грн - 15 %. Імперативна реалізація наведена в лістингу 4.8.

Лістинг 4.8 - Імперативний розрахунок знижки на банкет

```
1. // Знижка на банкет залежно від суми замовлення
2. public static decimal ApplyDiscount(decimal amount)
3. {
4.     if (amount <= 10000)
5.         return amount;
6.     else if (amount <= 30000)
7.         return amount * 0.95m;
8.     else if (amount <= 60000)
9.         return amount * 0.90m;
10.      else
11.          return amount * 0.85m;
12.  }
```

Мета полягає в тому, щоб записати ту саму логіку у вигляді, наближеному до виразу *switch* (лістинг 4.9).

Лістинг 4.9 - Розрахунок знижки методом Match

```
1. amount.Match(
2.     (x => x <= 10000, x => x),
3.     (x => x <= 30000, x => x * 0.95m),
4.     (x => x <= 60000, x => x * 0.90m)
5. ).DefaultMatch(x => x * 0.85m);
```

Методу *Match* передається масив кортежів, кожен з яких містить два лямбда-вирази: перший визначає, чи відповідає вхідне значення зразку, другий задає перетворення значення в разі відповідності. Завершальний виклик *DefaultMatch* визначає перетворення за замовчуванням, якщо жоден зразок не підійшов. Умови в лівій частині кортежів можуть бути як завгодно складними, зокрема викликами окремих функцій.

Реалізацію методу *Match* наведено в лістингу 4.10. Метод повертає проміжний об'єкт *MatchValueOrDefault*, який зберігає ознаку наявності збігу, отримане значення та вихідне значення. Метод *DefaultMatch* повертає отримане значення, якщо збіг відбувся, або застосовує функцію за замовчуванням до вихідного значення.

Лістинг 4.10 - Реалізація методу Match

```
1. public static class MatchExtensions
2. {
3.     public static MatchValueOrDefault<TInput, TOutput>
4.         Match<TInput, TOutput>(
5.             this TInput @this,
6.             params (Func<TInput, bool> IsMatch,
7.                     Func<TInput, TOutput> Transform)[] cases)
8.     {
9.         foreach (var c in cases)
10.              if (c.IsMatch(@this))
11.                  return new(true, c.Transform(@this), @this);
12.          return new(false, default!, @this);
13.      }
14.  }
15.
16.  public sealed class MatchValueOrDefault<TInput, TOutput>
17.  {
18.      private readonly bool hasMatch;
19.      private readonly TOutput value;
20.      private readonly TInput original;
21.
22.      public MatchValueOrDefault(bool hasMatch, TOutput value,
23.                                 TInput original)
24.      {
25.          this.hasMatch = hasMatch;
26.          this.value = value;
27.          this.original = original;
28.      }
29.
30.      public TOutput DefaultMatch(Func<TInput, TOutput> defaultCase) =>
31.          hasMatch ? value : defaultCase(original);
32.  }
```

Явна ознака збігу *hasMatch* є принципово важливою. Якби відсутність збігу визначалася порівнянням результату зі значенням за замовчуванням для типу (*default*), то коректний результат, що дорівнює нулю, помилково вважався б відсутністю збігу. Крім того, кортеж є типом-значенням, тож метод *FirstOrDefault* у разі відсутності збігу повертав би кортеж із порожніми посиланнями на функції, виклик яких спричинив би виняток *NullReferenceException*.

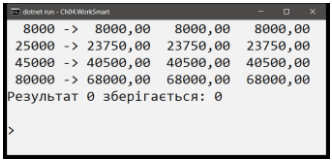


Рисунок 4.3 - Результат виконання прикладу «Зіставлення для старих версій C#»

Результати всіх трьох реалізацій збігаються, а останній рядок підтверджує, що нульовий результат зіставлення обробляється коректно.

Порівняно з вбудованими засобами сучасних версій C# такий підхід суттєво обмежений: зіставлення за типом об'єкта не підтримується, а синтаксис менш виразний. Проте він дає змогу позбутися значного обсягу шаблонного коду. У версіях C#, що не підтримують кортежів, замість них можна використати структуру *KeyValuePair<TKey, TValue>* (лістинги 4.11 і 4.12), хоча синтаксис створення таких пар є громіздким, і доцільність подібного рішення залежить від конкретної ситуації.

Лістинг 4.11 - Метод Match на основі KeyValuePair

```
1. public static class KeyValueMatchExtensions
2. {
3.     public static MatchValueOrDefault<TInput, TOutput>
4.         MatchKv<TInput, TOutput>(
5.             this TInput @this,
6.             params KeyValuePair<Func<TInput, bool>,
7.                                 Func<TInput, TOutput>>[] cases)
8.     {
9.         foreach (var c in cases)
10.              if (c.Key(@this))
11.                  return new(true, c.Value(@this), @this);
12.          return new(false, default!, @this);
13.      }
14.  }
```

Лістинг 4.12 - Розрахунок знижки з використанням KeyValuePair

```
1. amount.MatchKv(
2.     new KeyValuePair<Func<decimal, bool>, Func<decimal, decimal>>(
3.         x => x <= 10000, x => x),
4.     new KeyValuePair<Func<decimal, bool>, Func<decimal, decimal>>(
5.         x => x <= 30000, x => x * 0.95m),
6.     new KeyValuePair<Func<decimal, bool>, Func<decimal, decimal>>(
7.         x => x <= 60000, x => x * 0.90m)
8. ).DefaultMatch(x => x * 0.85m);
```

4.2. Безпечне використання словників

Функції можна використовувати не лише для перетворення даних, а й як **фільтри** - додатковий шар між розробником і вихідним джерелом даних або функціональності. Цей прийом дає змогу усунути недолік словників *Dictionary<TKey, TValue>*.

Словники є ефективним засобом пошуку даних за ключем і дають змогу замінити громіздкий шаблонний код простими зверненнями за індексом. Проте звернення до словника за відсутнім ключем спричиняє виняток *KeyNotFoundException*, який доводиться обробляти (лістинг 4.13).

Лістинг 4.13 - Звернення до словника за відсутнім ключем

```
1. var roomTypes = new[]
2. {
3.     ("STD", "Стандарт"),
4.     ("SUP", "Покращений"),
5.     ("STE", "Люкс")
6. }.ToDictionary(x => x.Item1, x => x.Item2);
```

```
7.
8. try
9. {
10.      var info = $"Категорія FAM: {roomTypes["FAM"]}";
11.  }
12.  catch (KeyNotFoundException ex)
13.  {
14.      Console.WriteLine(ex.GetType().Name);
15.  }
```



Рисунок 4.4 - Результат виконання прикладу «Словник: виняток» (фрагмент)

Безпечне звернення потребує попередньої перевірки наявності ключа, наприклад за допомогою методу *TryGetValue* (лістинг 4.14), що знову повертає шаблонний код.

Лістинг 4.14 - Звернення до словника з перевіркою наявності ключа

```
1. var familyName = roomTypes.TryGetValue("FAM", out var name)
2.     ? name
3.     : "невідома категорія";
```

Функціональне розв'язання полягає в перетворенні словника на функцію пошуку за допомогою методів-розширень (лістинг 4.15).

Лістинг 4.15 - Методи-розширення для безпечного пошуку

```
1. public static class DictionaryExtensions
2. {
3.     public static Func<TKey, TValue?> ToSafeLookup<TKey, TValue>(
4.         this IDictionary<TKey, TValue> @this) =>
5.         key => @this.TryGetValue(key, out var value) ? value : default;
6.
7.     public static Func<TKey, TValue> ToSafeLookup<TKey, TValue>(
8.         this IDictionary<TKey, TValue> @this, TValue defaultValue) =>
9.         key => @this.TryGetValue(key, out var value)
10.              ? value
11.              : defaultValue;
12.  }
```

Використання цих методів наведено в лістингу 4.16.

Лістинг 4.16 - Безпечний пошук у словнику

```
1. var roomTypes = new[]
2. {
3.     ("STD", "Стандарт"),
4.     ("SUP", "Покращений"),
5.     ("STE", "Люкс")
6. }.ToDictionary(x => x.Item1, x => x.Item2)
7.  .ToSafeLookup("невідома категорія");
8.
9. var suite = $"Категорія STE: {roomTypes("STE")}";
10.  var family = $"Категорія FAM: {roomTypes("FAM")}";
```

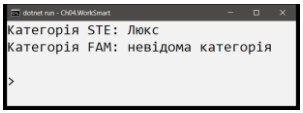


Рисунок 4.5 - Результат виконання прикладу «Словник: безпечний пошук»

Звернення до значень тепер виконується за допомогою круглих, а не квадратних дужок, оскільки змінна *roomTypes* вже не є словником - вона є функцією. Методи-розширення повертають функції, які утримують вихідний словник в області видимості, доки існують самі (такі функції називають **замиканнями**, closures). Функція відіграє роль фільтра між словником і рештою коду: виняток у разі відсутності ключа більше не виникає, а повертається значення за замовчуванням для типу або значення, задане розробником.

Недоліком цього прийому є те, що результат уже не є словником: його не можна доповнювати чи обробляти методами LINQ. Тому він доцільний у тих випадках, коли такі операції не потрібні.

4.3. Розбір значень

Ще одним поширеним джерелом громіздкого шаблонного коду є перетворення рядкових значень на значення інших типів. Типовим прикладом є зчитування налаштувань програми з конфігураційного файлу, у якому всі значення зберігаються як рядки, можуть бути відсутніми або некоректними. У лістингу 4.17 так зчитуються налаштування касової системи ресторану.

Лістинг 4.17 - Імперативне зчитування налаштувань

```
1. public static PosSettings GetSettingsImperative()
2. {
3.     var settings = new PosSettings();
4.
5.     AppSettings.TryGetValue("PrintRetries", out var retriesStr);
6.     if (int.TryParse(retriesStr, out var retries))
7.         settings.PrintRetries = retries;
8.     else
9.         settings.PrintRetries = 3;
10.
11.      AppSettings.TryGetValue("KitchenOpensAt", out var hourStr);
12.      if (int.TryParse(hourStr, out var hour))
13.          settings.KitchenOpensAt = hour;
14.      else
15.          settings.KitchenOpensAt = 7;
16.
17.      AppSettings.TryGetValue("AlertEmail", out var email);
18.      if (string.IsNullOrWhiteSpace(email))
19.          settings.AlertEmail = "manager@example.com";
20.      else
21.          settings.AlertEmail = email;
```

```
22.
23.      AppSettings.TryGetValue("ServerName", out var server);
24.      if (string.IsNullOrWhiteSpace(server))
25.          settings.ServerName = "pos-main";
26.      else
27.          settings.ServerName = server;
28.
29.      return settings;
30.  }
```

Значний обсяг однотипного коду приховує мету програми, а додавання кожного нового параметра потребує ще п'яти-шести рядків. Повторювану структуру можна винести в методи-розширення (лістинг 4.18).

Лістинг 4.18 - Методи-розширення для розбору значень

```
1. public static class ParsingExtensions
2. {
3.     public static int ToIntOrDefault(this object? @this,
4.                                      int defaultValue = 0) =>
5.         int.TryParse(@this?.ToString(), out var parsed)
6.             ? parsed
7.             : defaultValue;
8.
9.     public static string ToStringOrDefault(this object? @this,
10.                                           string defaultValue = "")
    =>
11.         @this?.ToString() is { } s && !string.IsNullOrWhiteSpace(s)
12.             ? s
13.             : defaultValue;
14.  }
```

Код зчитування налаштувань набуває вигляду, орієнтованого на результат (лістинг 4.19).

Лістинг 4.19 - Зчитування налаштувань, орієнтоване на результат

```
1. public static PosSettings GetSettings() =>
2.     new()
3.     {
4.         PrintRetries = Setting("PrintRetries").ToIntOrDefault(3),
5.         KitchenOpensAt = Setting("KitchenOpensAt").ToIntOrDefault(7),
6.         AlertEmail = Setting("AlertEmail")
7.             .ToStringOrDefault("manager@example.com"),
8.         ServerName = Setting("ServerName")
9.             .ToStringOrDefault("pos-main")
10.      };
```

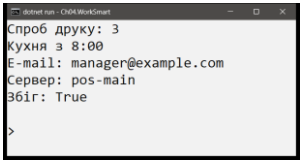


Рисунок 4.6 - Результат виконання прикладу «Розбір налаштувань»

Тепер з першого погляду зрозуміло, які налаштування зчитуються і які значення використовуються за замовчуванням, а новий параметр додається одним рядком. Для типів, відмінних від *int* і *string*, достатньо реалізувати відповідні додаткові методи-розширення.

4.4. Власні способи перебору

Перелічувані послідовності широко використовуються в коді, проте механізм, що забезпечує їх перебір, можна застосовувати безпосередньо для реалізації нестандартної поведінки ітерації. Це дає змогу суттєво скоротити код складних циклів.

Перебір перелічуваної послідовності забезпечує об'єкт-**перелічувач** (enumerator), що реалізує інтерфейс *IEnumerator<T>*. Саме він використовується циклом *foreach*. Перелічувач має два основні члени:

- властивість **`Current`** повертає поточний елемент; її можна викликати багаторазово, доки не виконано перехід до наступного елемента. До першого виклику *MoveNext()* значення *Current* не визначене;

- метод **`MoveNext()`** виконує перехід до наступного елемента і повертає *true*, якщо такий елемент існує, або *false*, якщо досягнуто кінця послідовності (або послідовність порожня). Перший виклик *MoveNext()* встановлює перелічувач на перший елемент.

4.4.1. Аналіз пар сусідніх елементів

Нехай для розміщення групи гостей потрібно визначити, чи є серед вільних столиків ресторану два з послідовними номерами (такі столики розташовані поруч і їх можна зсунути). Імперативне рішення наведено в лістингу 4.20.

Лістинг 4.20 - Імперативний пошук сусідніх вільних столиків

```
1. // Чи є серед вільних столиків два з сусідніми номерами
2. public static bool HasAdjacentFreeTables(IEnumerable<int> free)
3. {
4.     var sorted = free.OrderBy(x => x).ToArray();
5.     for (var i = 0; i < sorted.Length - 1; i++)
6.     {
7.         if (sorted[i] + 1 == sorted[i + 1])
8.             return true;
9.     }
10.      return false;
11.  }
```

Функціональне рішення потребує методу-розширення, який отримує перелічувач послідовності і керує перебором. Щоб уникнути імперативного циклу, використовується рекурсія (лістинг 4.21).

Лістинг 4.21 - Метод-розширення AnyPair

```
1. public static partial class PairExtensions
```

```
2. {
3.     // Чи існує пара сусідніх елементів, що задовольняє умову
4.     public static bool AnyPair<T>(this IEnumerable<T> @this,
5.                                   Func<T, T, bool> evaluator)
6.     {
7.         using var enumerator = @this.GetEnumerator();
8.         var hasElements = enumerator.MoveNext();
9.         return hasElements
10.              && AnyPair(enumerator, evaluator, enumerator.Current);
11.     }
12.
13.     private static bool AnyPair<T>(IEnumerator<T> enumerator,
14.                                    Func<T, T, bool> evaluator,
15.                                    T previous)
16.     {
17.         var moreItems = enumerator.MoveNext();
18.         return moreItems
19.             && (evaluator(previous, enumerator.Current)
20.                 || AnyPair(enumerator, evaluator,
    enumerator.Current));
21.     }
22. }
```

Відкритий метод *AnyPair* отримує перелічувач і переходить до першого елемента. Закритий метод приймає перелічувач, функцію перевірки та попередній елемент, переходить до наступного елемента й застосовує функцію перевірки до пари «попередній - поточний». Якщо елементи закінчилися, повертається *false*; якщо пара задовольняє умову - *true*; інакше виконується рекурсивний виклик для решти послідовності. Перебір припиняється одразу після знаходження першої відповідної пари. Функціональна версія пошуку сусідніх столиків наведена в лістингу 4.22.

Лістинг 4.22 - Функціональний пошук сусідніх вільних столиків

```
1. public static bool HasAdjacentFreeTablesF(IEnumerable<int> free) =>
2.     free.OrderBy(x => x)
3.         .AnyPair((prev, curr) => curr == prev + 1);
```

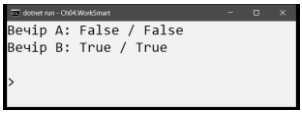


Рисунок 4.7 - Результат виконання прикладу «Сусідні елементи» (фрагмент)

За тією самою схемою реалізується метод *AllPairs*, що перевіряє, чи всі пари сусідніх елементів задовольняють умову (лістинг 4.23). Відмінність полягає лише в умовах продовження перебору та дострокового завершення: перебір припиняється, щойно знайдено пару, яка умову не задовольняє.

Лістинг 4.23 - Метод-розширення AllPairs

```
1. public static partial class PairExtensions
```

```
2. {
3.     // Чи всі пари сусідніх елементів задовольняють умову
4.     public static bool AllPairs<T>(this IEnumerable<T> @this,
5.                                    Func<T, T, bool> evaluator)
6.     {
7.         using var enumerator = @this.GetEnumerator();
8.         var hasElements = enumerator.MoveNext();
9.         return !hasElements
10.              || AllPairs(enumerator, evaluator, enumerator.Current);
11.     }
12.
13.     private static bool AllPairs<T>(IEnumerator<T> enumerator,
14.                                     Func<T, T, bool> evaluator,
15.                                     T previous)
16.     {
17.         var moreItems = enumerator.MoveNext();
18.         return !moreItems
19.             || (evaluator(previous, enumerator.Current)
20.                 && AllPairs(enumerator, evaluator,
    enumerator.Current));
21.     }
22. }
```

Прикладом застосування є контроль температурного режиму холодильної камери кухні: різниця між послідовними вимірюваннями не повинна перевищувати 2 °C (лістинг 4.24).

Лістинг 4.24 - Контроль стабільності температури холодильної камери

```
1. // Температура холодильної камери (°C) з інтервалом 1 год;
2. // стрибок понад 2 °C між вимірами є порушенням
3. decimal[] fridgeLog = [3.1m, 3.4m, 2.9m, 5.6m, 3.2m];
4. var stable = fridgeLog.AllPairs(
5.     (prev, curr) => Math.Abs(curr - prev) <= 2m);
```

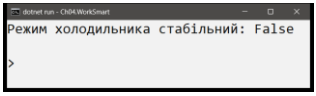


Рисунок 4.8 - Результат виконання прикладу «Сусідні елементи» (фрагмент)

4.4.2. Ітерація до виконання умови

Прийом, розглянутий у цьому пункті, є функціональною заміною циклу *while*. Нехай моделюється робота кухні протягом зміни: страви з черги замовлень готуються по черзі, доки черга не стане порожньою. Імперативна реалізація зі змінним станом і циклом *while* наведена в лістингу 4.25.

Лістинг 4.25 - Імперативне моделювання роботи кухні

```
1. // Імперативний варіант: змінний стан і цикл while
2. public static (int Minute, int Done) RunImperative()
3. {
4.     var queue = new Queue<(string Dish, int Minutes)>(Initial().Queue);
5.     var done = new List<string>();
```

```
6.      var minute = 0;
7.      while (queue.Count > 0)
8.      {
9.          var (dish, minutes) = queue.Dequeue();
10.           minute += minutes;
11.           done.Add($"{dish} ({minute} хв)");
12.       }
13.       return (minute, done.Count);
14.   }
```

За змістом потрібна функція, подібна до методу *Aggregate*, яка, однак, не перебирає наперед відому колекцію, а повторює крок оновлення стану, доки не буде виконано умову завершення. Таку функцію реалізовано у вигляді рекурсивного методу-розширення (лістинг 4.26). Він також виграв би від оптимізації хвостової рекурсії; способи її імітації розглядаються в розділі 10.

Лістинг 4.26 - Метод-розширення AggregateUntil

```
1. public static class IterationExtensions
2. {
3.     public static T AggregateUntil<T>(this T @this,
4.                                       Func<T, bool> endCondition,
5.                                       Func<T, T> update) =>
6.         endCondition(@this)
7.             ? @this
8.             : AggregateUntil(update(@this), endCondition, update);
9. }
```

Стан кухні подається незмінним записом (лістинг 4.27), а один крок моделювання - чистою функцією, що повертає новий стан (лістинг 4.28).

Лістинг 4.27 - Незмінний стан кухні

```
1. // Стан кухні: черга замовлень, готові страви, поточна хвилина
2. public sealed record KitchenState(
3.     IReadOnlyList<(string Dish, int Minutes)> Queue,
4.     IReadOnlyList<string> Done,
5.     int Minute);
```

Лістинг 4.28 - Крок моделювання: приготування наступної страви

```
1. // Один крок: приготувати першу страву з черги
2. public static KitchenState CookNext(KitchenState s) =>
3.     Cook(s, s.Queue[0], s.Minute + s.Queue[0].Minutes);
4.
5. private static KitchenState Cook(KitchenState s,
6.     (string Dish, int Minutes) next, int finishedAt) =>
7.     s with
8.     {
9.         Queue = s.Queue.Skip(1).ToArray(),
10.          Minute = finishedAt,
11.          Done = [.. s.Done, $"{next.Dish} ({finishedAt} хв)"]
12.      };
```

Цикл *while* повністю замінюється одним викликом (лістинг 4.29).

Лістинг 4.29 - Моделювання зміни методом AggregateUntil

```
1. var endState = Initial().AggregateUntil(
2.     s => s.Queue.Count == 0,
3.     CookNext);
```

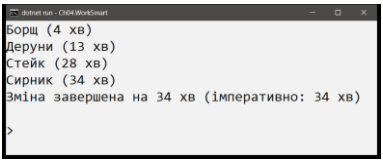


Рисунок 4.9 - Результат виконання прикладу «Ітерація до умови»

Наведене рішення є функціональним, хоча й не досконалим: у реальних задачах крок оновлення стану може включати взаємодію з користувачем або зовнішніми системами. Функціональні способи організації такої взаємодії розглядаються в розділі 14.

4.5. Висновки

1. Делегати *Func* можна розміщувати в колекціях і обробляти методами LINQ; це дає змогу застосовувати набір правил до одних даних, формувати набір правил динамічно та використовувати відкладене виконання.

2. Валідатор на основі масиву функцій-правил і методу *All* замінює послідовність інструкцій *if* з достроковим поверненням; узагальнені методи-розширення *IsValid* і *IsInvalid* придатні для перевірки об'єктів будь-якого типу.

3. Спрощене зіставлення зі зразком можна реалізувати в ранніх версіях C# за допомогою кортежів (або *KeyValuePair*) і проміжного об'єкта з явною ознакою збігу.

4. Функції-замикання можуть слугувати фільтрами між кодом і джерелом даних, зокрема для безпечного пошуку в словниках без генерування винятків.

5. Методи-розширення для розбору значень зі значеннями за замовчуванням дають змогу записувати зчитування налаштувань у вигляді, орієнтованому на результат.

6. Безпосереднє використання перелічувача *IEnumerator<T>* разом із рекурсією дає змогу реалізовувати власні способи перебору, зокрема аналіз пар сусідніх елементів і ітерацію до виконання умови (заміну циклу *while*).

4.6. Контрольні запитання

1. Як визначається тип результату делегата *Func*? Чим *Func* відрізняється від *Action*?

2. Які переваги має формування тексту за допомогою колекції функцій порівняно з використанням *StringBuilder*?

3. Як працює метод *All*? Чому валідатор на його основі не обчислює зайвих правил?

4. Чим відрізняються методи-розширення *IsValid* і *IsInvalid*? Як один із них можна виразити через інший?

5. Як реалізувати спрощене зіставлення зі зразком без вбудованої підтримки мови? Навіщо потрібен проміжний об'єкт *MatchValueOrDefault*?

6. Чому для визначення відсутності збігу не можна порівнювати результат зі значенням *default*?

7. Що таке замикання? Як замикання використовується для безпечного пошуку в словнику?

8. Які обмеження має перетворення словника на функцію пошуку?

9. Які члени має інтерфейс *IEnumerator<T>*? Що станеться під час звернення до *Current* до першого виклику *MoveNext()*?

10. Як працює метод *AnyPair*? У який момент припиняється перебір?

11. Яку імперативну конструкцію замінює метод *AggregateUntil*? Які обмеження має його рекурсивна реалізація?

4.7. Практичні завдання

1. Сформуйте картку гостя готелю за допомогою колекції функцій-описувачів; додайте правило, яке включає рядок про програму лояльності лише для учасників цієї програми (динамічне формування колекції).

2. Реалізуйте за допомогою методу *IsValid* перевірку позиції меню: назва не порожня і не довша за 60 символів, ціна додатна, вихід страви в грамах у межах від 50 до 1000, категорія визначена, а для напоїв обов'язково вказано об'єм.

3. За допомогою методу *Match* реалізуйте визначення туристичного збору залежно від кількості ночей і віку гостя (діти до 18 років звільнені від збору). Перевірте коректність обробки нульового результату.

4. Створіть безпечну функцію пошуку ціни позиції меню за кодом на основі словника; для відсутніх кодів функція має повертати 0 і записувати повідомлення до журналу.

5. Реалізуйте метод-розширення *CountPairs*, який підраховує кількість пар сусідніх елементів, що задовольняють умову. Застосуйте його для підрахунку випадків, коли завантаженість готелю зростала більш ніж на 10 % від доби до доби.

6. За допомогою методу *AggregateUntil* змоделюйте прибирання номерів готелю покоївкою: кожен номер потребує певної кількості

хвилин; моделювання завершується, коли прибрано всі номери або вичерпано тривалість зміни.

Частина II. Поглиблене вивчення функціонального програмування

5. Функції вищого порядку

Друга частина посібника виходить за межі стандартних засобів мови C# і розглядає функціональні концепції, що потребують певного теоретичного підґрунтя. Виклад, як і раніше, будується поступово й орієнтується на практичну користь для повсякденної розробки програмного забезпечення.

Поняття функції вищого порядку вже вводилося в підрозділі 1.2.2. Функції вищого порядку постійно використовуються в бібліотеці LINQ. Наприклад, у лістингу 5.1 метод *Where* отримує лямбда-вираз, який відбирає прізвища працівників ресторану, що починаються з літери, розташованої в абетці після «М».

Лістинг 5.1 - Передавання лямбда-виразу в метод Where

```
1. var staff = new[]
2. {
3.     "Марченко", "Коваль", "Ткачук",
4.     "Бойко", "Савчук", "Литвин", "Олійник"
5. };
6.
7. var secondHalf = staff.Where(x => x.First() > 'M');
```

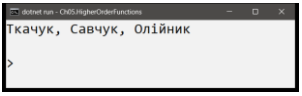


Рисунок 5.1 - Результат виконання прикладу «Основи» (фрагмент)

Лямбда-вираз *x => x.First() > 'M'* є скороченим записом анонімної функції; у повній формі вона могла б мати вигляд іменованого методу (лістинг 5.2).

Лістинг 5.2 - Еквівалентний іменований метод

```
1. private static bool IsAfterM(string name) => name.First() > 'M';
```

Отже, функцію передано як параметр іншій функції, усередині якої вона й виконується. Другий різновид функцій вищого порядку повертає функцію як результат (лістинг 5.3).

Лістинг 5.3 - Функція, що повертає функцію

```
1. public static Func<decimal, decimal> MakeAddFee(decimal fee) =>
2.     price => price + fee;
```

Вираз містить дві стрілки: функція *MakeAddFee* приймає розмір збору *fee* і повертає нову функцію, у якій значення *fee* зафіксоване на момент виклику *MakeAddFee* (лістинг 5.4).

Лістинг 5.4 - Створення та виклик функції нарахування доставки

```
1. var withDelivery = MakeAddFee(60m);
2. var total = withDelivery(475m);
3. // total = 535
```

Отже, **функція вищого порядку** має принаймні одну з двох властивостей:

- приймає функцію як параметр;
- повертає функцію як результат.

У мові C# для цього використовуються делегати *Func* (для функцій, що повертають значення) і *Action* (для функцій без результату). Попри простоту ідеї, функції вищого порядку можуть суттєво змінити структуру кодової бази. У розділі розглядаються способи їх практичного застосування, а також **комбінатори** (combinators) - функції вищого порядку, що поєднують інші функції для побудови складнішої поведінки. Назва походить від комбінаторної логіки (див. підрозділ 1.4).

Мета розділу - сформувати навички використання функцій вищого порядку та комбінаторів для усунення дублювання коду, побудови ланцюжків перетворень і обробки помилок.

Після опрацювання розділу студент повинен:

- уміти усувати дублювання коду шляхом параметризації функцій іншими функціями;
- розуміти поняття thunk і відкладеного обчислення;
- уміти реалізовувати та застосовувати функції *Map*, *Fork*, *Alt*, *Compose*, *Transduce*, *Tap*;
- уміти інкапсулювати обробку винятків і перевірки на *null* у функціях вищого порядку;
- уміти «оновлювати» елементи перелічуваних послідовностей без зміни вихідних даних;
- уміти виражати обробку послідовностей через згортку, знати закони моноїду та їх значення для паралельних обчислень;
- розуміти відмінність між делегатом і деревом виразів та вміти аналізувати й будувати дерева виразів.

Приклади розділу містяться в проєкті *Ch05.HigherOrderFunctions*; загальні методи-розширення, що використовуються і в наступних розділах, винесено в бібліотеку *HoReCa.Domain*.

5.1. Постановка задачі: звіт про інциденти

Нехай потрібно реалізувати функцію, яка отримує зі сховища даних журнал інцидентів обслуговування в закладах мережі, підраховує кількість інцидентів кожної категорії та надсилає звіт керівництву. Якщо даних немає, надсилається окреме попередження. Функція може мати вигляд, наведений у лістингу 5.5.

Лістинг 5.5 - Функція формування звіту за категоріями інцидентів

```
1. public void SendIncidentCategorySummary()
2. {
3.     try
4.     {
```

```
5.          var incidents = store.GetIncidents();
6.          var summary = incidents.GroupBy(x => x.Category)
7.              .Select(x => (Key: x.Key, Count: x.Count()));
8.          var report = new Report(
9.              "Інциденти за категоріями",
10.               summary.Select(x => new ReportItem(
11.                   x.Key, x.Count.ToString())).ToArray());
12.
13.           if (!report.Rows.Any())
14.               communication.SendNoDataWarning();
15.           else
16.               communication.SendReport(report);
17.       }
18.       catch (Exception e)
19.       {
20.           log.Error(e, "Помилка в " +
21.               nameof(SendIncidentCategorySummary) + ": " + e.Message);
22.       }
23.   }
```

Згодом керівництво замовляє другий звіт - розподіл інцидентів за рівнем серйозності. Найпростішим на перший погляд рішенням є копіювання першої функції зі зміною назви, заголовка звіту та властивості групування (лістинг 5.6).

Лістинг 5.6 - Друга функція, отримана копіюванням

```
1. public void SendIncidentSeveritySummary()
2. {
3.     try
4.     {
5.         var incidents = store.GetIncidents();
6.         var summary = incidents.GroupBy(x => x.Severity.ToString())
7.             .Select(x => (Key: x.Key, Count: x.Count()));
8.         var report = new Report(
9.             "Інциденти за рівнем серйозності",
10.              summary.Select(x => new ReportItem(
11.                  x.Key, x.Count.ToString())).ToArray());
12.
13.          if (!report.Rows.Any())
14.              communication.SendNoDataWarning();
15.          else
16.              communication.SendReport(report);
17.      }
18.      catch (Exception e)
19.      {
20.          log.Error(e, "Помилка в " +
21.              nameof(SendIncidentCategorySummary) + ": " + e.Message);
22.      }
23.  }
```

Такий підхід має кілька суттєвих недоліків. По-перше, модульні тести, написані для першої функції, не покривають другої: покриття нової функції тестами дорівнює нулю, а копіювання тестів ще більше збільшує обсяг коду. По-друге, підхід не масштабується: якщо знадобляться десятки подібних звітів, кодова база розростеться на тисячі рядків. По-третє, і це

найнебезпечніше, разом із кодом копіюються й помилки. Так, у лістингу 5.6 у повідомленні журналу залишилася назва першої функції (*nameof(SendIncidentCategorySummary)*), тож у разі збою журнал вказуватиме на хибне місце. Будь-яку помилку, виявлену в одній копії, доведеться виправляти в усіх копіях, а тестування збільшиться пропорційно кількості копій.

Рішення щодо способів економії часу розробки доцільно оцінювати з погляду всього життєвого циклу програмного продукту, який нерідко становить десятиліття.

5.2. Параметризація функціями та thunk

Розв'язання полягає в тому, щоб залишити змінну частину обчислення «незаповненою» і передавати її ззовні у вигляді функції. Функцію-шаблон, у якій частину логіки задано параметром-делегатом, можна образно назвати «функцією з отвором»: отвір заповнює код, що її викликає. Перероблений варіант звітів наведено в лістингу 5.7.

Лістинг 5.7 - Узагальнена функція формування звіту

```
1. public void SendIncidentCategorySummary() =>
2.     GenerateSummary(x => x.Category, "Інциденти за категоріями");
3.
4. public void SendIncidentSeveritySummary() =>
5.     GenerateSummary(x => x.Severity.ToString(),
6.                     "Інциденти за рівнем серйозності");
7.
8. public void GenerateSummary(Func<Incident, string> summarySelector,
9.                              string reportName)
10. {
11.     try
12.     {
13.         var incidents = store.GetIncidents();
14.         var summary = incidents.GroupBy(summarySelector)
15.             .Select(x => (Key: x.Key, Count: x.Count()));
16.         var report = new Report(
17.             reportName,
18.             summary.Select(x => new ReportItem(
19.                 x.Key, x.Count.ToString())).ToArray());
20.
21.         if (!report.Rows.Any())
22.             communication.SendNoDataWarning();
23.         else
24.             communication.SendReport(report);
25.     }
26.     catch (Exception e)
27.     {
28.         log.Error(e, "Помилка в " + nameof(GenerateSummary) +
29.             ", звіт: " + reportName + ", " + e.Message);
30.     }
31. }
```

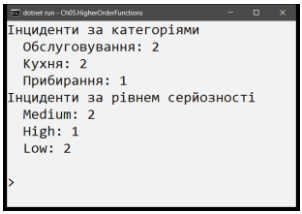


Рисунок 5.2 - Результат виконання прикладу «Звіт про інциденти» (фрагмент)

Переваги нового варіанта очевидні. Кожен новий звіт додається одним рядком коду, що безпосередньо відображає його відмінність від решти. Після тестування узагальненої функції покриття тестами нових звітів залишається майже повним, адже вони відрізняються лише назвою та функцією групування. Виправлення помилок і вдосконалення базової функції автоматично поширюються на всі звіти.

Узагальнену функцію доцільно зробити частиною відкритого інтерфейсу класу (лістинг 5.8).

Лістинг 5.8 - Інтерфейс із параметром-функцією

```
1. public interface IGenerateReports
2. {
3.     void GenerateSummary(Func<Incident, string> summarySelector,
4.                          string reportName);
5. }
```

Тоді для отримання звіту за новою ознакою не потрібно змінювати ні інтерфейс, ні клас, що його реалізує: код-споживач самостійно визначає ознаку групування, наприклад заклад, у якому стався інцидент.

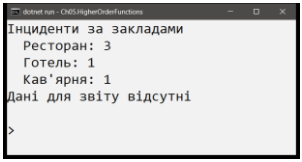


Рисунок 5.3 - Результат виконання прикладу «Звіт про інциденти» (фрагмент)

Ідею можна розвинути, додавши параметри типу *Func<ReportItem, string>* для довільного форматування рядків або *Action* для власного журналювання. Слід зазначити, що такий підхід цілком відповідає **принципу відкритості/закритості** (open/closed principle) з набору принципів об'єктно-орієнтованого проєктування SOLID: модуль відкритий для розширення, але закритий для змін. Це свідчить, що в мові C# функціональний та об'єктно-орієнтований підходи можуть ефективно доповнювати один одного.

У літературі з функціонального програмування поширений термін **thunk** - так називають функцію без параметрів, що інкапсулює **відкладене обчислення** (deferred computation): обчислення описується заздалегідь, а виконується лише на вимогу. У мові C# thunk подається делегатом *Func<T>*. У лістингу 5.9 функція отримує thunk для завантаження інцидентів і викликає його лише тоді, коли звіт справді потрібен.

Лістинг 5.9 - Відкладене завантаження даних за допомогою thunk

```
1. // Thunk - функція без параметрів, що інкапсулює
2. // відкладене обчислення
3. public static string SummarizeLater(
4.     Func<IEnumerable<Incident>> loadIncidents,
5.     bool reportRequested) =>
6.     reportRequested
7.         ? $"Інцидентів: {loadIncidents().Count()}"
8.         : "Звіт не запитувався, дані не завантажувалися";
```

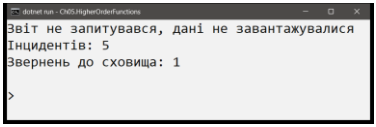


Рисунок 5.4 - Результат виконання прикладу «Thunk»

Хоча функцію *SummarizeLater* викликано двічі, звернення до сховища відбулося лише один раз, оскільки в першому випадку звіт не запитувався.

5.3. Ланцюжки функцій: Map

Функція **`Map`**, яку в різних мовах і бібліотеках називають також *chain* або *pipe*, є однією з найкорисніших у функціональному програмуванні. Слід зазначити, що функціональні конструкції нерідко мають кілька назв залежно від мови програмування; у посібнику по можливості вказуються альтернативні назви.

Нехай кухня ресторану отримала конвекційну піч, панель керування якої відображає температуру в градусах Фаренгейта, а технологічні карти страв складено в градусах Цельсія. Перерахунок виконується за формулою: від значення віднімається 32, результат множиться на 5 і ділиться на 9. Обчислення можна записати одним рядком (лістинг 5.10), однак такий запис важко читати.

Лістинг 5.10 - Перерахунок температури одним виразом

```
1. public static string FahrenheitToCelsiusOneLine(decimal tempF) =>
2.     Math.Round((tempF - 32) * 5 / 9, 1) + " °C";
```

Покроковий запис (лістинг 5.11) читається краще, проте створює змінні, які використовуються лише один раз. У великому методі такі змінні залишаються в області видимості до кінця методу, утримуючи пам'ять (що особливо помітно для складних об'єктів), і захаращують код.

Лістинг 5.11 - Покроковий перерахунок із проміжними змінними

```
1. public static string FahrenheitToCelsiusSteps(decimal tempF)
2. {
3.     var a = tempF - 32;
4.     var b = a * 5;
5.     var c = b / 9;
6.     var d = Math.Round(c, 1);
7.     var returnValue = d + " °C";
8.     return returnValue;
9. }
```

Функція *Map* подібна до методу *Select*, але застосовується не до кожного елемента послідовності, а до об'єкта в цілому - будь-якого об'єкта. Параметр лямбда-виразу позначає сам об'єкт; якщо застосувати *Map* до колекції, параметр позначатиме всю колекцію, а не її елементи (лістинг 5.12).

Лістинг 5.12 - Перерахунок температури за допомогою Map

```
1. public static string FahrenheitToCelsius(decimal tempF) =>
2.     tempF.Map(x => x - 32)
3.          .Map(x => x * 5)
4.          .Map(x => x / 9)
5.          .Map(x => Math.Round(x, 1))
6.          .Map(x => x + " °C");
```

Функціональність збігається з покроковим варіантом, однак одноразові змінні відсутні: результат кожного кроку передається наступному лямбда-виразу і після цього може бути звільнений збирачем сміття. Реалізація функції *Map* надзвичайно проста (лістинг 5.13).

Лістинг 5.13 - Реалізація функції Map

```
1. public static TOut Map<TIn, TOut>(this TIn @this,
2.                                   Func<TIn, TOut> f) =>
3.     f(@this);
```

Попри стислість, функція *Map* дає змогу переписувати тіла методів із кількома послідовними перетвореннями у вигляді одного виразу. Розширені варіанти *Map* з обробкою помилок розглядаються в розділі 7.

Якщо тип даних на всіх кроках не змінюється, можна використати спрощений варіант *Map*, що приймає масив перетворень і застосовує їх послідовно за допомогою методу *Aggregate* (лістинги 5.14 і 5.15).

Лістинг 5.14 - Варіант Map з масивом перетворень

```
1. public static T Map<T>(this T @this,
2.                        params Func<T, T>[] transformations) =>
3.     transformations.Aggregate(@this, (acc, f) => f(acc));
```

Лістинг 5.15 - Використання Map з масивом перетворень

```
1. public static decimal FahrenheitToCelsiusValue(decimal tempF) =>
2.     tempF.Map(
3.         x => x - 32,
4.         x => x * 5,
5.         x => x / 9,
6.         x => Math.Round(x, 1));
```

Усі чотири варіанти дають однаковий результат:

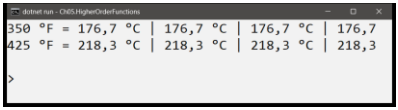


Рисунок 5.5 - Результат виконання прикладу «Map»

5.4. Комбінатор Fork

Комбінатор розгалуження (fork combinator; трапляється також назва converge) обробляє одне значення кількома незалежними способами, а потім поєднує отримані результати в одне кінцеве значення. Схема його роботи така:

1) обирається вихідне значення;

2) значення передається кільком функціям-«зубцям» (prongs), кожна з яких незалежно обчислює певний результат;

3) функція поєднання (join) об'єднує результати зубців у кінцевий результат.

Реалізація для двох зубців наведена в лістингу 5.16. Окремі узагальнені параметри для кожного зубця дають змогу повертати значення різних типів.

Лістинг 5.16 - Комбінатор Fork із двома зубцями

```
1. public static partial class ForkExtensions
2. {
3.     public static TOut Fork<TIn, T1, T2, TOut>(
4.         this TIn @this,
5.         Func<TIn, T1> f1,
6.         Func<TIn, T2> f2,
7.         Func<T1, T2, TOut> join)
8.     {
9.         var p1 = f1(@this);
10.          var p2 = f2(@this);
11.          var result = join(p1, p2);
12.          return result;
13.      }
14.  }
```

У лістингу 5.17 комбінатор обчислює середній чек: один зубець обчислює суму, другий - кількість рахунків, а функція поєднання ділить одне на інше.

Лістинг 5.17 - Обчислення середнього чека комбінатором Fork

```
1. var checks = RestaurantData.Checks.Select(c => c.Amount);
2. var average = checks.Fork(
3.     x => x.Sum(),
4.     x => x.Count(),
5.     (sum, count) => Math.Round(sum / count, 2));
6. // average = 1154
```

Аналогічно обчислюється виручка на одного гостя (лістинг 5.18).

Лістинг 5.18 - Обчислення виручки на одного гостя

```
1. var perGuest = RestaurantData.Checks.Fork(
2.     x => x.Sum(c => c.Amount),
3.     x => x.Sum(c => c.Guests),
4.     (revenue, guests) => Math.Round(revenue / guests, 2));
```

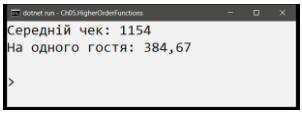


Рисунок 5.6 - Результат виконання прикладу «Fork» (фрагмент)

Для довільної кількості зубців, що повертають значення одного типу, комбінатор реалізується з масивом функцій (лістинг 5.19).

Лістинг 5.19 - Комбінатор Fork з довільною кількістю зубців

```
1. public static partial class ForkExtensions
2. {
3.     public static TEnd Fork<TStart, TMiddle, TEnd>(
4.         this TStart @this,
5.         Func<IEnumerable<TMiddle>, TEnd> join,
6.         params Func<TStart, TMiddle>[] prongs)
7.     {
8.         var intermediate = prongs.Select(p => p(@this));
9.         var result = join(intermediate);
10.          return result;
11.      }
12.  }
```

Такий варіант зручно використовувати для формування текстового опису складного об'єкта (лістинг 5.20): об'єкт закладу не є колекцією, тож метод *Select* до нього застосувати неможливо, а комбінатор фактично перетворює один об'єкт на набір значень, до якого застосовуються операції над списками.

Лістинг 5.20 - Формування опису закладу комбінатором Fork

```
1. var venue = new Venue("«Під ратушею»", "Чернівці", 64, 680m);
2. var description = venue.Fork(
3.     lines => string.Join(Environment.NewLine, lines),
4.     x => $"Заклад: {x.Name}",
5.     x => $"Місто: {x.City}",
6.     x => $"Посадкових місць: {x.Seats}",
7.     x => $"Середній чек: {x.AverageCheck} грн");
```

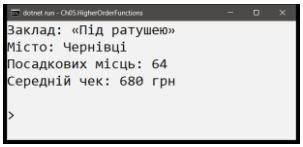


Рисунок 5.7 - Результат виконання прикладу «Fork» (фрагмент)

5.5. Комбінатор Alt

Комбінатор альтернативи (alt combinator; також or, alternate, alternation) поєднує кілька функцій, що досягають однієї мети різними способами, і застосовує їх по черзі, доки одна з них не поверне значення: «спробувати спосіб A; якщо не вдалося - спосіб B; якщо не вдалося - спосіб C». Реалізація наведена в лістингу 5.21.

Лістинг 5.21 - Реалізація комбінатора Alt

```
1. public static class AltExtensions
2. {
3.     public static TOut? Alt<TIn, TOut>(
4.         this TIn @this, params Func<TIn, TOut?>[] args) =>
5.         args.Select(f => f(@this))
6.             .FirstOrDefault(x => x is not null);
7. }
```

Хоча метод *Select* формально перетворює весь масив функцій на значення, завдяки відкладеному виконанню метод *FirstOrDefault* припиняє перебір, щойно отримано перше значення, відмінне від *null*, тож наступні функції не викликаються. На відміну від методу *First*, метод *FirstOrDefault* не генерує винятку, якщо жодна функція не повернула значення.

Типовий сценарій застосування - пошук даних у кількох джерелах із різним пріоритетом. У лістингу 5.22 бронювання шукається спочатку у власній базі даних закладу, потім на зовнішній платформі бронювання і, нарешті, у локальній резервній копії, що використовується в разі збою мережі.

Лістинг 5.22 - Пошук бронювання в кількох джерелах

```
1. public static Reservation? FindReservation(string code) =>
2.     code.Alt(
3.         FromDb,          // власна база даних закладу
4.         FromPlatform,    // зовнішня платформа бронювання
5.         FromCsv);        // локальна резервна копія
```

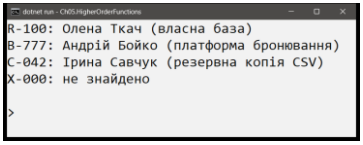


Рисунок 5.8 - Результат виконання прикладу «Alt»

5.6. Композиція функцій: Compose

Характерною рисою функціональних мов є можливість будувати складну функцію з простіших. Будь-який процес поєднання функцій називають **композицією** (composition). Найпростіший спосіб композиції в C# - ланцюжок викликів *Map*, збережений у змінній-делегаті (лістинг 5.23).

Лістинг 5.23 - Функції перерахунку температури, побудовані з Map

```
1. var fahrenheitToCelsius = (decimal x) =>
2.     x.Map(x => x - 32)
3.      .Map(x => x * 5)
4.      .Map(x => x / 9)
5.      .Map(x => Math.Round(x, 1))
6.      .Map(x => $"{x} градусів");
7.
8. var celsiusToFahrenheit = (decimal x) =>
9.     x.Map(x => x * 9)
10.       .Map(x => x / 5)
11.       .Map(x => x + 32)
12.       .Map(x => Math.Round(x, 1))
13.       .Map(x => $"{x} градусів");
```

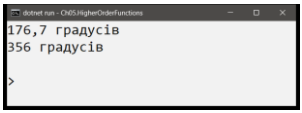


Рисунок 5.9 - Результат виконання прикладу «Композиція через Map»

Змінні *fahrenheitToCelsius* і *celsiusToFahrenheit* є функціями, побудованими з п'яти простих лямбда-виразів. Останні два кроки в обох функціях (округлення та форматування) збігаються. Повторення можна усунути за допомогою функції `**Compose**`, яка поєднує дві функції в нову функцію (лістинг 5.24).

Лістинг 5.24 - Реалізація функції Compose

```
1. public static Func<TIn, TNewOut> Compose<TIn, TOldOut, TNewOut>(
2.     this Func<TIn, TOldOut> @this,
3.     Func<TOldOut, TNewOut> f) =>
4.     x => f(@this(x));
```

На відміну від *Map*, функція *Compose* повертає не значення, а новий делегат. Використання наведено в лістингу 5.25.

Лістинг 5.25 - Усунення повторення за допомогою Compose

```
1. var formatDegrees = (decimal x) =>
2.     x.Map(x => Math.Round(x, 1))
3.      .Map(x => $"{x} градусів");
4.
5. var fahrenheitToCelsius = (decimal x) =>
6.     x.Map(x => x - 32)
7.      .Map(x => x * 5)
8.      .Map(x => x / 9);
```

```
9. var fToC = fahrenheitToCelsius.Compose(formatDegrees);
10.
11.  var celsiusToFahrenheit = (decimal x) =>
12.      x.Map(x => x * 9)
13.       .Map(x => x / 5)
14.       .Map(x => x + 32);
15.  var cToF = celsiusToFahrenheit.Compose(formatDegrees);
```

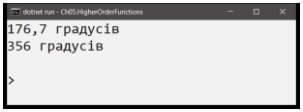


Рисунок 5.10 - Результат виконання прикладу «Compose»

Функціональність збігається з попереднім варіантом, однак форматування визначене в одному місці, і будь-які його зміни автоматично поширюються на обидві функції. Мова C# має обмеження: методи-розширення не можна застосовувати безпосередньо до лямбда-виразів, а лише до змінних типу делегата. Саме тому ланцюжки *Map* у лістингу 5.25 спочатку присвоюються змінним і лише потім поєднуються функцією *Compose*.

Такий спосіб повторного використання коду нагадує успадкування в ООП, проте застосовується на рівні окремих функцій, потребує значно менше шаблонного коду і зберігає споріднені фрагменти поруч, а не розподіляє їх між класами та файлами.

5.7. Трансдуктори: Transduce

Функція *Compose* діє на об'єкт у цілому і не призначена для операцій над елементами колекцій. Композицію операцій над колекцією можна побудувати, вкладаючи *Select* і *Where* в кожну функцію, але такий код громіздкий (лістинг 5.26). У прикладі до цін позицій меню додається ПДВ 20 % і сервісний збір 10 %, після чого відбираються позиції, вартість яких перевищує 200 грн.

Лістинг 5.26 - Композиція операцій над колекцією за допомогою Compose

```
1. decimal[] prices = [45m, 120m, 165m, 210m, 340m];
2.
3. var addVat = (IEnumerable<decimal> x) => x.Select(y => y * 1.2m);
4. var addVatAndService = addVat.Compose(x => x.Select(y => y * 1.1m));
5. var above200 = addVatAndService.Compose(x => x.Where(y => y > 200));
6. var message = above200.Compose(x => string.Join("; ", x));
7.
8. Console.WriteLine("Результат: " + message(prices));
```

Трансдуктор (transducer) поєднує операції над списками (*Select*, *Where*) з агрегуванням і описує всі етапи функціонального потоку:

- **фільтрація** (*Where*) - зменшення кількості елементів;
- **перетворення** (*Select*) - зміна форми елементів;

- **агрегування** (*Aggregate*) - зведення колекції до одного значення.

Одна з можливих реалізацій наведена в лістингу 5.27: функція *Transduce* приймає функцію-перетворювач (довільну комбінацію *Select* і *Where*) та функцію-агрегатор, а *ToTransducer* створює з них новий делегат для повторного використання.

Лістинг 5.27 - Реалізація трансдуктора

```
1. public static class TransduceExtensions
2. {
3.     public static TFinal Transduce<TIn, TFiltered, TFinal>(
4.         this IEnumerable<TIn> @this,
5.         Func<IEnumerable<TIn>, IEnumerable<TFiltered>> transformer,
6.         Func<IEnumerable<TFiltered>, TFinal> aggregator) =>
7.         aggregator(transformer(@this));
8.
9.     public static Func<IEnumerable<TIn>, TOut>
10.          ToTransducer<TIn, TMid, TOut>(
11.          this Func<IEnumerable<TIn>, IEnumerable<TMid>> @this,
12.          Func<IEnumerable<TMid>, TOut> aggregator) =>
13.          x => aggregator(@this(x));
14.  }
```

Застосування трансдуктора до тієї самої задачі наведено в лістингах 5.28 і 5.29.

Лістинг 5.28 - Застосування функції Transduce

```
1. var transformer = (IEnumerable<decimal> x) => x
2.     .Select(y => y * 1.2m)
3.     .Select(y => y * 1.1m)
4.     .Where(y => y > 200);
5. var aggregator = (IEnumerable<decimal> x) => string.Join("; ", x);
6.
7. var output = prices.Transduce(transformer, aggregator);
```

Лістинг 5.29 - Створення делегата-трансдуктора

```
1. var transducer = transformer.ToTransducer(aggregator);
2. var output2 = transducer(prices);
```

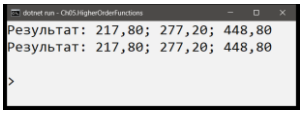


Рисунок 5.11 - Результат виконання прикладу «Transduce»

Отриманий делегат можна застосовувати до будь-яких колекцій цін: він виконує всі необхідні перетворення та фільтрацію і зводить колекцію до одного значення.

5.8. Спостереження за ланцюжком: Tap

Поширеним зауваженням щодо ланцюжків функцій є неможливість журналювання проміжних значень, якщо тільки один із кроків не викликає

окрему функцію з журналюванням. Функція **`Tap`** дає змогу «підключитися» до ланцюжка в будь-якій точці та виконати довільну дію з поточним значенням, не змінюючи його і не порушуючи потоку обчислень (лістинг 5.30).

Лістинг 5.30 - Реалізація функції Tap

```
1. public static T Tap<T>(this T @this, Action<T> action)
2. {
3.     action(@this);
4.     return @this;
5. }
```

Функція передає поточне значення в делегат *Action*, де, наприклад, виконується журналювання, і повертає те саме значення без змін (лістинг 5.31).

Лістинг 5.31 - Журналювання проміжного значення за допомогою Tap

```
1. var fahrenheitToCelsius = (decimal x) =>
2.     x.Map(x => x - 32)
3.      .Map(x => x * 5)
4.      .Map(x => x / 9)
5.      .Tap(x => Console.WriteLine($"[журнал] до округлення: {x}"))
6.      .Map(x => Math.Round(x, 1))
7.      .Map(x => $"{x} градусів");
8.
9. var output = fahrenheitToCelsius(350m);
```



Рисунок 5.12 - Результат виконання прикладу «Tap»

5.9. Обробка винятків функціями вищого порядку

Досконаліші функціональні засоби обробки помилок розглядаються в розділах 6 і 7. Тут наведено прості прийоми, які можна реалізувати кількома рядками коду.

У функціональному коді, що дотримується принципів чистоти й незмінності, помилки теоретично неможливі. Проте на межі системи, у взаємодії із зовнішнім середовищем, вони неминучі. Нехай ресторан отримує перелік товарів постачальника через вебсервіс, який може бути недоступним через мережеві чи апаратні збої, що не залежать від розробника. Об'єктно-орієнтований спосіб обробки такої ситуації наведено в лістингу 5.32.

Лістинг 5.32 - Обробка помилки зовнішнього сервісу блоком try/catch

```
1. public class SupplierClientOop(SupplierApi api, ILog log)
2. {
3.     public IEnumerable<string> GetProducts(int categoryId)
4.     {
```

```
        try
        {
            var returnValue = api.GetProductsByCategory(categoryId);
            return returnValue;
        }
          catch (Exception e)
          {
              log.Error(e, "Не вдалося отримати товари постачальника");
              return [];
          }
      }
  }
```

Такий код має два недоліки. По-перше, захисний шаблонний код займає більшу частину методу. По-друге, блоки *try/catch* порушують лінійний порядок виконання, переносячи керування в місце, яке може бути важко знайти. Ще менш бажаним є повторне генерування винятку для обробки на вищому рівні: невідомо, де і чи взагалі його буде перехоплено, а необроблений виняток може аварійно завершити програму. Крім того, ніщо не заважає з часом ускладнити такий код вкладеними викликами.

Альтернативою є функція *Map* із вбудованою обробкою винятків, що повертає результат або значення за замовчуванням (лістинги 5.33 і 5.34).

Лістинг 5.33 - Функція MapWithTryCatch

```
public static class TryCatchExtensions
{
    public static TOut? MapWithTryCatch<TIn, TOut>(this TIn @this,
                                                   Func<TIn, TOut> f)
    {
        try
        {
            return f(@this);
        }
          catch
          {
              return default;
          }
      }
  }
```

Лістинг 5.34 - Використання MapWithTryCatch

```
public IEnumerable<string> GetProductsSilently(int categoryId) =>
    categoryId.MapWithTryCatch(api.GetProductsByCategory) ?? [];
```

Цей варіант має суттєвий недолік: виняток «поглинається» без запису до журналу. Розв'язанням може бути метод-розширення, що отримує об'єкт журналу і повертає функцію, у яку вбудовано блок *try/catch* із журналюванням (лістинги 5.35 і 5.36).

Лістинг 5.35 - Функція CreateTryCatch з журналюванням

```
public static class TryCatchLoggerExtensions
{
    public static Func<Func<TIn, TOut>, TOut?>
```

```
4.          CreateTryCatch<TIn, TOut>(
5.          this TIn @this, ILog log) =>
6.          f =>
7.          {
8.              try
9.              {
10.                 return f(@this);
11.             }
12.             catch (Exception e)
13.             {
14.                 log.Error(e, "Виникла помилка: " + e.Message);
15.                 return default;
16.             }
17.         };
18.  }
```

Лістинг 5.36 - Використання CreateTryCatch

```
1. public IEnumerable<string> GetProductsLogged(int categoryId)
2. {
3.     var tryCatch = categoryId
4.         .CreateTryCatch<int, IEnumerable<string>>(log);
5.     var result = tryCatch(api.GetProductsByCategory) ?? [];
6.     return result;
7. }
```

Повідомлення в журналі в цьому разі є загальним, оскільки метод-розширення не знає контексту виклику. Якщо потрібне повідомлення, специфічне для місця виклику, можна повертати об'єкт-результат із даними про успішність виконання (лістинг 5.37).

Лістинг 5.37 - Об'єкт результату виконання

```
1. public sealed class ExecutionResult<T>
2. {
3.     public T? Result { get; init; }
4.     public Exception? Error { get; init; }
5. }
6.
7. public static class ExecutionResultExtensions
8. {
9.     public static ExecutionResult<TOut> MapWithResult<TIn, TOut>(
10.         this TIn @this, Func<TIn, TOut> f)
11.     {
12.         try
13.         {
14.             return new ExecutionResult<TOut> { Result = f(@this) };
15.         }
16.         catch (Exception e)
17.         {
18.             return new ExecutionResult<TOut> { Error = e };
19.         }
20.     }
21. }
```

Цей підхід також не позбавлений недоліків: успішний результат містить непотрібну властивість *Error*, а невдалий - непотрібну властивість *Result*, що

суперечить духу **принципу розділення інтерфейсів** (interface segregation principle) SOLID. Крім того, перевірку наявності помилки доводиться виконувати в коді, що викликає функцію. Цю перевірку можна інкапсулювати в методі-розширенні *OnError* (лістинг 5.38), який виконує обробник у разі помилки та повертає результат.

Лістинг 5.38 - Метод-розширення OnError

```
1. public static class OnErrorExtensions
2. {
3.     public static T? OnError<T>(this ExecutionResult<T> @this,
4.                                 Action<Exception> errorHandler)
5.     {
6.         if (@this.Error is not null)
7.             errorHandler(@this.Error);
8.         return @this.Result;
9.     }
10.  }
```

Остаточний варіант (лістинг 5.39) зобов'язує розробника явно визначити обробку помилки та дає змогу задати повідомлення з контекстом виклику.

Лістинг 5.39 - Обробка помилки за допомогою OnError

```
1. public IEnumerable<string> GetProducts(int categoryId) =>
2.     categoryId.MapWithResult(api.GetProductsByCategory)
3.         .OnError(e => log.Error(e,
4.             $"Категорія {categoryId}: постачальник недоступний"))
5.     ?? [];
```

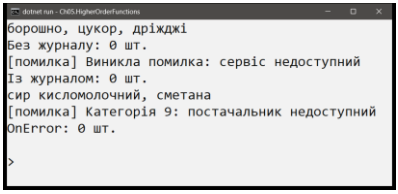


Рисунок 5.13 - Результат виконання прикладу «Try/Catch»

Досконаліший спосіб подання результату, що може бути успішним або помилковим, розглядається в розділі 6.

5.10. Обробка значень null: Unless

Функція **`Unless`** приймає умову та дію і виконує дію лише тоді, коли умова хибна - тобто «завжди, крім випадку, коли умова виконується» (лістинг 5.40). Найпоширеніше її застосування - перевірка на *null*.

Лістинг 5.40 - Реалізація функції Unless

```
1. public static class UnlessExtensions
2. {
3.     public static void Unless<T>(this T @this,
4.                                  Func<T, bool> condition,
5.                                  Action<T> action)
6.     {
7.         if (!condition(@this))
8.             action(@this);
9.     }
10.  }
```

Функція не повертає результату: якби дію було подано делегатом *Func*, залишалося б невизначеним, що повертати у випадку, коли умова виконується і дія не виконується.

У лістингу 5.41 ресторан надсилає гостям SMS-нагадування про бронювання. Якщо номер телефону гостя відсутній, виникає виняток *NullReferenceException*, і обробка решти гостей переривається.

Лістинг 5.41 - Надсилання нагадувань без перевірки на null

```
1. public static void SendRemindersUnsafe(IEnumerable<GuestContact> guests)
2. {
3.     foreach (var g in guests)
4.     {
5.         SendSms(g.Phone!.Trim(), "Чекаємо на вас о 19:00");
6.         LogReminder(g);
7.     }
8. }
```

Використання функції *Unless* (лістинг 5.42) запобігає винятку: для гостя без номера телефону SMS просто не надсилається, а решта обробки продовжується.

Лістинг 5.42 - Надсилання нагадувань із функцією Unless

```
1. public static void SendReminders(IEnumerable<GuestContact> guests)
2. {
3.     foreach (var g in guests)
4.     {
5.         g.Unless(
6.             x => x.Phone is null,
7.             x => SendSms(x.Phone!.Trim(), "Чекаємо на вас о 19:00"));
8.         LogReminder(g);
9.     }
10.  }
```

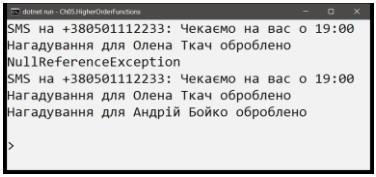


Рисунок 5.14 - Результат виконання прикладу «Unless»

Систематичніші способи запобігання помилкам, пов'язаним із *null*, розглядаються в розділах 6 і 7.

5.11. Оновлення перелічуваної послідовності

Наостанок розглядається прийом «оновлення» елемента перелічуваної послідовності без зміни будь-яких даних. Оскільки перелічувані послідовності обчислюються ліниво, виклик *Select* здебільшого не ініціює обчислень і може слугувати фільтром між джерелом даних і місцем, де дані перебираються. Методи заміни елемента за позицією наведено в лістингу 5.43.

Лістинг 5.43 - Методи заміни елемента за позицією

```
1. public static class ReplaceExtensions
2. {
3.     public static IEnumerable<T> ReplaceAt<T>(
4.         this IEnumerable<T> @this, int location, T replacement) =>
5.         @this.Select((x, i) => i == location ? replacement : x);
6.
7.     public static IEnumerable<T> ReplaceAt<T>(
8.         this IEnumerable<T> @this, int location,
9.         Func<T, T> replacement) =>
10.          @this.Select((x, i) => i == location ? replacement(x) : x);
11.  }
```

Повернута послідовність посилається на вихідну й отримує значення з неї, за винятком елемента із заданим індексом, який замінюється новим значенням (лістинг 5.44).

Лістинг 5.44 - Заміна позиції на дошці меню

```
1. var board = new[] { "Борщ", "Банош", "Деруни", "Сирник" };
2.
3. var updated = board.ReplaceAt(1, "Боrрач");
4. var text = string.Join(", ", updated);
5. // text = "Борщ, Боrрач, Деруни, Сирник"
```

Вихідний масив залишається незмінним, а фактична заміна відбувається лише під час виклику *string.Join*, коли значення дійсно потрібні:

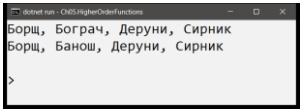


Рисунок 5.15 - Результат виконання прикладу «ReplaceAt» (фрагмент)

Перевантажений варіант приймає функцію, що обчислює нове значення на основі старого (лістинг 5.45).

Лістинг 5.45 - Заміна з використанням попереднього значення

```
1. var withNote = board.ReplaceAt(2, x => x + " (сезонна страва)");
```

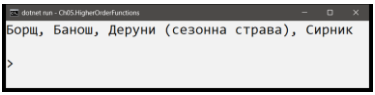


Рисунок 5.16 - Результат виконання прикладу «ReplaceAt» (фрагмент)

Якщо позиція елемента невідома або потрібно оновити кілька елементів, використовується функція *ReplaceWhen*, що визначає елементи для заміни за умовою (лістинг 5.46).

Лістинг 5.46 - Метод заміни елементів за умовою

```
1. public static class ReplaceWhenExtensions
2. {
3.     public static IEnumerable<T> ReplaceWhen<T>(
4.         this IEnumerable<T> @this,
5.         Func<T, bool> shouldReplace,
6.         Func<T, T> replacement) =>
7.         @this.Select(x => shouldReplace(x) ? replacement(x) : x);
8. }
```

У лістингу 5.47 до всіх страв меню, що не містять інгредієнтів тваринного походження, додається мітка «веганська». Нові записи створюються виразом *with*, а вихідні залишаються без змін.

Лістинг 5.47 - Додавання мітки до страв за умовою

```
1. var tagged = menu.ReplaceWhen(
2.     x => !x.Ingredients.Intersect(animal).Any(),
3.     x => x with { Tags = [.. x.Tags, "веганська"] });
```

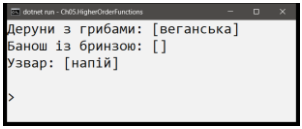


Рисунок 5.17 - Результат виконання прикладу «ReplaceWhen»

Функція *ReplaceWhen* дає змогу замінити численні інструкції *if* простішими й передбачуванішими операціями.

5.12. Згортка та моноїди

5.12.1. Згортка як узагальнення циклу

Більшість розглянутих функцій вищого порядку перетворюють послідовність на іншу послідовність. Не менш поширеною є задача **зведення** послідовності до одного значення: суми, кількості, максимуму, звіту. В імперативному коді для цього використовують цикл зі змінною-акумулятором. Функціональним аналогом такого циклу є **згортка** (fold, у LINQ - *Aggregate*) - функція вищого порядку, що приймає початкове значення акумулятора z і функцію кроку f та послідовно поєднує акумулятор з кожним елементом:

fold(f, z, [x1, x2, ..., xn]) = f(...f(f(z, x1), x2)..., xn).

Для порожньої послідовності результатом є початкове значення z. Схему обчислення показано на рис. 5.18.

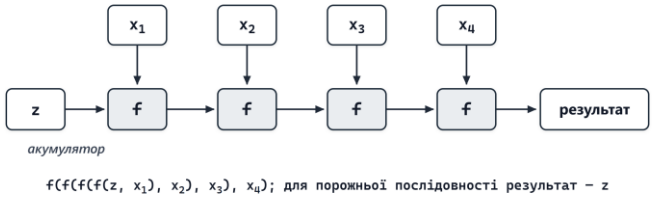


Рисунок 5.18 - Схема згортки зліва

Згортку легко визначити рекурсивно, без змінних і циклів (лістинг 5.48): якщо елементи вичерпано, результатом є акумулятор; інакше згортка продовжується з новим акумулятором f(z, x).

Лістинг 5.48 - Рекурсивне визначення згортки

```
1. // Згортка зліва: fold(f, z, [x1, x2, ..., xn]) =
2. //   f(...f(f(z, x1), x2)..., xn)
3. public static TAcc Fold<T, TAcc>(this IEnumerable<T> items,
4.     TAcc seed, Func<TAcc, T, TAcc> step)
5. {
6.     using var e = items.GetEnumerator();
7.     return Go(seed);
8.
9.     TAcc Go(TAcc acc) => e.MoveNext() ? Go(step(acc, e.Current)) : acc;
10.  }
```

Значення згортки полягає в її універсальності: будь-яку функцію, що обробляє послідовність поелементно, можна виразити через згортку, обравши відповідні акумулятор і функцію кроку. Зокрема, через згортку визначаються підрахунок, сума, а також відображення (*Select*) і фільтрація (*Where*) (лістинг 5.49). Згортку тому розглядають як фундаментальну схему рекурсії над послідовностями, а цикл з акумулятором - як її імперативну реалізацію.

Лістинг 5.49 - Типові операції як окремі випадки згортки

```
1. // Типові операції над послідовністю - окремі випадки згортки
2. public static int CountF<T>(this IEnumerable<T> xs) =>
3.     xs.Fold(0, (n, _) => n + 1);
4.
5. public static decimal SumF(this IEnumerable<decimal> xs) =>
6.     xs.Fold(0m, (s, x) => s + x);
7.
8. public static IEnumerable<TOut> MapF<T, TOut>(
9.     this IEnumerable<T> xs, Func<T, TOut> f) =>
10.      xs.Fold(Enumerable.Empty<TOut>(), (acc, x) => acc.Append(f(x)));
11.
12.  public static IEnumerable<T> FilterF<T>(this IEnumerable<T> xs,
```

```
13.      Func<T, bool> p) =>
14.      xs.Fold(Enumerable.Empty<T>(),
15.          (acc, x) => p(x) ? acc.Append(x) : acc);
```

Лістинг 5.50 - Використання операцій, визначених через згортку

```
1. var prices = Menu.Select(d => d.Price).ToArray();
2. Console.WriteLine($"Кількість: {Menu.CountF()}");
3. Console.WriteLine($"Сума цін: {prices.SumF()}");
4. Console.WriteLine("Веганські: " + string.Join(", ",
5.     Menu.FilterF(d => d.IsVegan).MapF(d => d.Name)));
6.
7. // Найдорожча страва: акумулятор - поточний «переможець»
8. var top = Menu.Aggregate((best, d) => d.Price > best.Price ? d : best);
9. Console.WriteLine($"Найдорожча: {top.Name}");
```

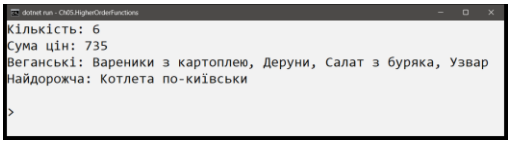


Рисунок 5.19 - Результат виконання прикладу «Згортка»

Перевантаження *Aggregate* без початкового значення використовує як акумулятор перший елемент послідовності; для порожньої послідовності воно генерує виняток, тобто є частковою функцією (підрозділ 1.3.1).

5.12.2. Моноїди

Згортка особливо проста й надійна, якщо акумулятор і елементи мають один тип, а операція їх поєднання має дві властивості. Множина значень M з бінарною операцією ⊕ та елементом e утворює **моноїд** (monoid), якщо виконуються закони:

1) **асоціативність**: (a ⊕ b) ⊕ c = a ⊕ (b ⊕ c) для будь-яких a, b, c;
2) **нейтральний елемент**: e ⊕ a = a ⊕ e = a для будь-якого a.

Моноїдами є цілі числа з додаванням і нулем, з множенням і одиницею, рядки з конкатенацією й порожнім рядком, послідовності з *Concat* і порожньою послідовністю, логічні значення з *&&* і *true*, функції вигляду T -> T з композицією й тотожною функцією (підрозділ 1.3.4). Записи, усі поля яких є моноїдами, також утворюють моноїд з покомпонентним поєднанням (лістинг 5.51).

Лістинг 5.51 - Підсумки кухні як моноїд

```
1. // Підсумок кухні як моноїд: асоціативне поєднання + нейтральний елемент
2. public readonly record struct KitchenTotals(int Dishes, decimal Revenue,
3.     int Calories)
4. {
5.     public static readonly KitchenTotals Empty = new(0, 0m, 0);
6.
7.     public static KitchenTotals Of(MenuDish d) =>
8.         new(1, d.Price, d.Calories);
9.
```

```
10.      public KitchenTotals Combine(KitchenTotals other) =>
11.          new(Dishes + other.Dishes, Revenue + other.Revenue,
12.              Calories + other.Calories);
13.  }
```

Закони моноїду мають безпосередній практичний наслідок. Нейтральний елемент є природним початковим значенням згортки й результатом для порожньої послідовності. Асоціативність дає змогу розбити послідовність на частини довільно, згорнути їх незалежно - зокрема паралельно, на різних ядрах чи серверах - і поєднати часткові результати: результат від розбиття не залежить (лістинг 5.52, рис. 5.20). На цій властивості ґрунтуються модель MapReduce та агрегування в розподілених системах обробки даних.

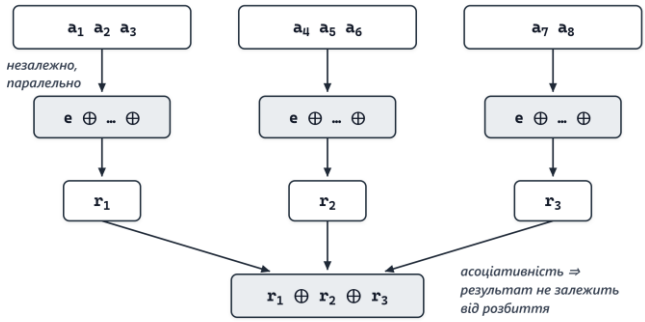


Рисунок 5.20 - Паралельна згортка моноїду частинами

Лістинг 5.52 - Послідовна та паралельна згортка

```
1. var orders = Enumerable.Range(0, 1200)
2.     .Select(i => Folds.Menu[i * 7 % Folds.Menu.Length])
3.     .ToArray();
4.
5. // Послідовна згортка
6. var sequential = orders.Select(KitchenTotals.Of)
7.     .Aggregate(KitchenTotals.Empty, (a, b) => a.Combine(b));
8.
9. // Згортка частинами: завдяки асоціативності частини можна
10.  // обробляти незалежно (зокрема паралельно) і поєднати потім
11.  var parallel = orders.Chunk(100).AsParallel()
12.      .Select(part => part.Select(KitchenTotals.Of)
13.          .Aggregate(KitchenTotals.Empty, (a, b) => a.Combine(b)))
14.      .Aggregate(KitchenTotals.Empty, (a, b) => a.Combine(b));
15.
16.  Console.WriteLine(sequential);
17.  Console.WriteLine($"Результати рівні: {sequential == parallel}");
```

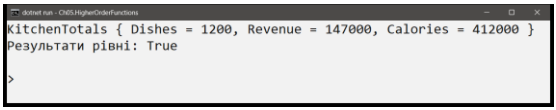


Рисунок 5.21 - Результат виконання прикладу «Моноїд і паралельна згортка»

Операція обчислення середнього значення не є асоціативною: середнє середніх частин у загальному випадку не дорівнює середньому всієї послідовності. Тому в моноїді зберігають суму й кількість, а середнє обчислюють наприкінці. Узагальнену реалізацію моноїду засобами мови C# 11 розглянуто в підрозділі 3.9.2.

5.13. Дерева виразів

5.13.1. Код як дані

Лямбда-вираз у мові C# може бути перетворений на дві принципово різні сутності. Якщо його присвоєно делегату (*Func*<...>), компілятор створює виконуваний код: такий делегат можна викликати, але неможливо дізнатися, які саме обчислення він виконує. Якщо ж лямбда-вираз присвоєно змінній типу *Expression*<*Func*<...>>, компілятор створює **дерево виразів** (expression tree) - структуру даних, вузли якої описують операції, операнди, виклики й параметри лямбда-виразу (лістинг 5.53, рис. 5.22).

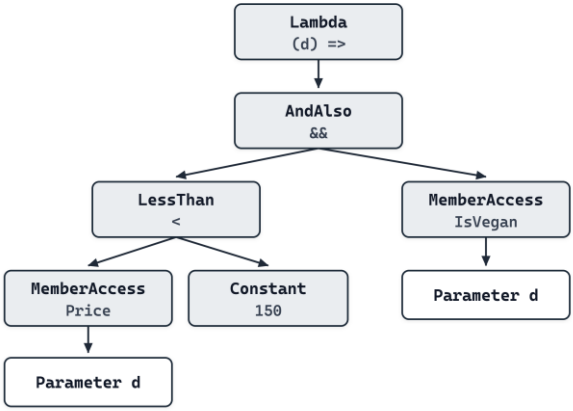


Рисунок 5.22 - Дерево виразу d => d.Price < 150 && d.IsVegan

Лістинг 5.53 - Делегат і дерево виразів для одного лямбда-виразу

```
1. // Делегат: скомпільований код, який можна лише виконати
```

```
2. Func<MenuDish, bool> asCode = d => d.Price < 150 && d.IsVegan;
3.
4. // Дерево виразів: той самий лямбда-вираз як структура даних
5. Expression<Func<MenuDish, bool>> asData =
6.     d => d.Price < 150 && d.IsVegan;
7.
8. Console.WriteLine(asCode.GetType().Name);
9. Console.WriteLine(asData.Body);
10.  Console.WriteLine(asData.Body.NodeType);
11.  var and = (BinaryExpression)asData.Body;
12.  Console.WriteLine($"ліворуч: {and.Left}, праворуч: {and.Right}");
13.
14.  // Дерево можна скомпілювати у звичайний делегат
15.  var compiled = asData.Compile();
16.  Console.WriteLine(Folds.Menu.Count(compiled));
```

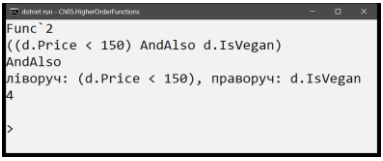


Рисунок 5.23 - Результат виконання прикладу «Дерево виразів»

Ідея подання програми у вигляді даних, які інша програма може аналізувати, перетворювати й виконувати, походить від мови LISP і називається **гомоїконічністю** (homoiconicity). Для функціонального програмування вона має особливе значення: функція стає не лише значенням, яке можна передати й викликати, а й значенням, яке можна дослідити. Дерева виразів незмінні; «зміна» дерева, як і зміна запису, полягає у створенні нового дерева.

Саме дерева виразів лежать в основі інтерфейсу *IQueryable<T>* та технологій LINQ to SQL і Entity Framework Core. Метод *Where* для *IQueryable<T>* приймає не делегат, а *Expression<Func<T, bool>>*; постачальник даних обходить дерево умови й перекладає його мовою запитів бази даних, тож фільтрація виконується на сервері бази даних, а не в пам'яті програми.

5.13.2. Обхід і побудова дерев виразів

Обхід дерева виразів зручно записувати виразом *switch* з шаблонами типів вузлів (лістинг 5.54): кожен вид вузла перекладається окремою гілкою, а складені вузли обробляються рекурсивно. Це спрощена модель того, що виконує постачальник даних Entity Framework Core.

Лістинг 5.54 - Переклад дерева виразу на умову SQL

```
1. // Обхід дерева: переклад умови на мову SQL
2. public static string ToSql(Expression e) => e switch
3. {
4.     LambdaExpression l => ToSql(l.Body),
5.     BinaryExpression b =>
```

```
        $"({ToSql(b.Left)} {SqlOperator(b.NodeType)} {ToSql(b.Right)})",
    MemberExpression { Type: var t } m when t == typeof(bool) =>
        $"{m.Member.Name} = 1",
    MemberExpression m => m.Member.Name,
      UnaryExpression { NodeType: ExpressionType.Not } u =>
          $"NOT {ToSql(u.Operand)}",
      ConstantExpression { Value: string s } => $"'{s}'",
      ConstantExpression c => Convert.ToString(c.Value,
          System.Globalization.CultureInfo.InvariantCulture) ?? "NULL",
      _ => throw new NotSupportedException(e.NodeType.ToString())
};

private static string SqlOperator(ExpressionType t) => t switch
{
    ExpressionType.AndAlso => "AND",
    ExpressionType.OrElse => "OR",
    ExpressionType.LessThan => "<",
    ExpressionType.LessThanOrEqual => "<=",
    ExpressionType.GreaterThan => ">",
    ExpressionType.GreaterThanOrEqual => ">=",
    ExpressionType.Equal => "=",
    ExpressionType.NotEqual => "<>",
    _ => throw new NotSupportedException(t.ToString())
};
```

Дерева виразів можна не лише аналізувати, а й будувати під час виконання. Типова задача - динамічний фільтр, умови якого обирає користувач. Функція *And* (лістинг 5.55) поєднує два предикати в новий. Оскільки кожен лямбда-вираз має власний параметр, тіло другого виразу переписується так, щоб воно посилалося на параметр першого; для цього використовується стандартний клас обходу *ExpressionVisitor*. Результат є чистою функцією від двох дерев: вихідні дерева не змінюються.

Лістинг 5.55 - Поєднання предикатів у дереві виразів

```
// Поєднання умов: тіла обох лямбда-виразів переносяться
// в нове дерево зі спільним параметром
public static Expression<Func<T, bool>> And<T>(
    this Expression<Func<T, bool>> left,
    Expression<Func<T, bool>> right)
{
    var p = left.Parameters[0];
    var rightBody = new ReplaceParameter(right.Parameters[0], p)
        .Visit(right.Body);
    return Expression.Lambda<Func<T, bool>>(
        Expression.AndAlso(left.Body, rightBody), p);
}

private sealed class ReplaceParameter(ParameterExpression from,
    ParameterExpression to) : ExpressionVisitor
{
    protected override Expression VisitParameter(
        ParameterExpression n) => n == from ? to : n;
}
```

Фільтр кіоску самообслуговування будується згорткою обраних відвідувачем умов: початковим значенням є умова за ціною, а кожна обрана умова додається функцією *And* (лістинг 5.56). Отримане дерево можна перекласти на SQL або скомпілювати в делегат методом *Compile*.

Лістинг 5.56 - Динамічна побудова фільтра меню

```
// Фільтр будується з умов, обраних відвідувачем у кіоску
Expression<Func<MenuDish, bool>> filter = d => d.Price <= 150;
var choices = new[] { "vegan", "light" };
foreach (var c in choices)
    filter = c switch
    {
        "vegan" => filter.And(d => d.IsVegan),
        "light" => filter.And(d => d.Calories < 400),
        _ => filter
    };

Console.WriteLine("WHERE " + ToSql(filter));
var found = Folds.Menu.Where(filter.Compile()).Select(d => d.Name);
Console.WriteLine(string.Join(", ", found));
```

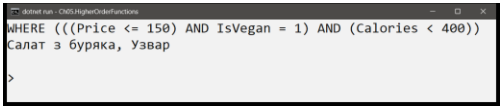


Рисунок 5.24 - Результат виконання прикладу «Дерево виразів у SQL»

Дерева виразів підтримують не всі можливості мови: у них не можна використовувати, зокрема, інструкції, присвоєння, вирази *switch* і шаблони. Компіляція дерева методом *Compile* є відносно дорогою операцією, тому скомпільовані делегати доцільно кешувати (див. розділ 11).

5.14. Висновки

1. Функція вищого порядку приймає функцію як параметр або повертає функцію як результат; у мові C# для цього використовуються делегати *Func* і *Action*.

2. Параметризація функцій іншими функціями усуває дублювання коду, підвищує покриття тестами і відповідає принципу відкритості/закритості.

3. Thunk - функція без параметрів, що інкапсулює відкладене обчислення і виконується лише на вимогу.

4. Функція *Map* застосовує перетворення до об'єкта в цілому і дає змогу записувати послідовні перетворення одним виразом без одноразових змінних.

5. Комбінатор *Fork* обробляє одне значення кількома незалежними функціями з подальшим поєднанням результатів; комбінатор *Alt* застосовує альтернативні функції по черзі до отримання першого результату.

6. Функція *Compose* поєднує функції в нову функцію, а трансдуктор поєднує фільтрацію, перетворення та агрегування колекції.

7. Функція *Tap* дає змогу виконувати побічні дії (зокрема журналювання) у ланцюжку без зміни значень.

8. Обробку винятків і перевірки на *null* можна інкапсулювати у функціях вищого порядку (*MapWithTryCatch*, *OnError*, *Unless*), позбувшись шаблонного коду.

9. Функції *ReplaceAt* і *ReplaceWhen* «оновлюють» елементи послідовності без зміни вихідних даних, використовуючи відкладене виконання.

10. Згортка (*Aggregate*) зводить послідовність до одного значення й узагальнює цикл з акумулятором; через неї виражаються підрахунок, сума, відображення та фільтрація.

11. Моноїд - множина з асоціативною операцією та нейтральним елементом. Нейтральний елемент є природним початковим значенням згортки, а асоціативність дає змогу згортати послідовність частинами, зокрема паралельно.

12. Лямбда-вираз, присвоєний змінній типу *Expression*<*Func*<...>>, подається деревом виразів - незмінною структурою даних, яку можна аналізувати, перетворювати, перекладати іншою мовою (зокрема SQL) і компілювати. На деревах виразів ґрунтуються *IQueryable*<*T*> та Entity Framework Core.

5.15. Контрольні запитання

1. Які дві властивості визначають функцію вищого порядку? Наведіть приклади з бібліотеки LINQ.

2. Які проблеми виникають під час копіювання коду для створення подібних функцій? Як їх розв'язує параметризація функціями?

3. Як параметризація функціями пов'язана з принципом відкритості/закритості?

4. Що таке thunk? Як він реалізується в мові C#?

5. Чим функція *Map* відрізняється від методу *Select*? Які переваги має ланцюжок *Map* порівняно з одноразовими змінними?

6. Як працює комбінатор *Fork*? Для яких задач він придатний?

7. Чому комбінатор *Alt* не виконує зайвих функцій? Чому в його реалізації використано *FirstOrDefault*, а не *First*?

8. Чим *Compose* відрізняється від *Map*? Чому перед викликом *Compose* лямбда-вираз потрібно присвоїти змінній?

9. З яких етапів складається трансдуктор?

10. Для чого призначена функція *Tap*?

11. Які недоліки мають блоки *try/catch* з погляду передбачуваності коду? Які переваги та недоліки мають функції *MapWithTryCatch*, *CreateTryCatch* і *OnError*?

12. Чому функція *Unless* не повертає значення?

13. Чому функція *ReplaceAt* не змінює вихідних даних? У який момент відбувається фактична заміна?

14. Що таке згортка? Як через згортку виразити методи *Count*, *Select* і *Where*?

15. Сформулюйте закони моноїду. Наведіть три приклади моноїдів. Чому обчислення середнього значення не утворює моноїду?

16. Чому асоціативність операції дає змогу виконувати згортку паралельно?

17. Чим дерево виразів відрізняється від делегата, створеного з того самого лямбда-виразу? Що таке гомоїконічність?

18. Яку роль відіграють дерева виразів у роботі *IQueryable<T>* та Entity Framework Core?

5.16. Практичні завдання

1. Реалізуйте узагальнену функцію формування звітів про бронювання готелю з параметрами: функція групування, функція фільтрації, функція форматування рядка. Сформуйте за її допомогою три різні звіти без дублювання коду.

2. За допомогою ланцюжка *Map* реалізуйте розрахунок вартості проживання з послідовних кроків: ціна за ніч, кількість ночей, знижка лояльності, туристичний збір, форматування рядка.

3. За допомогою комбінатора *Fork* обчисліть рентабельність страви: один зубець обчислює собівартість за техкартою, другий - ціну продажу, функція поєднання - відсоток націнки.

4. Реалізуйте пошук ціни інгредієнта за комбінатором *Alt* у трьох джерелах: договірний прайс постачальника, останній прихідний документ, середня ринкова ціна.

5. Побудуйте трансдуктор, що за журналом замовлень відбирає замовлення понад 1000 грн, обчислює для них чайові 5 % і повертає їхню загальну суму.

6. Додайте до ланцюжка розрахунку вартості проживання із завдання 2 журналювання проміжних значень за допомогою функції *Tap*.

7. За допомогою функцій *ReplaceWhen* і виразу *with* позначте в переліку номерів готелю всі номери, ремонт яких заплановано на поточний місяць, як недоступні для бронювання.

8. Виразіть через згортку функції пошуку мінімальної ціни, побудови словника «категорія - кількість страв» і перевірки, чи всі

страви меню мають ціну понад 50 грн. Для кожної функції вкажіть акумулятор і початкове значення.

9. Реалізуйте моноїд денного звіту ресторану (кількість чеків, виручка, середній чек, найпопулярніша страва) так, щоб результат паралельної згортки збігався з послідовною. Поясніть, які поля звіту доводиться зберігати в моноїді замість середнього чека.

10. Розширте функцію *ToSql* підтримкою методу *string.StartsWith* (умова *LIKE 'текст%'*) та операції *!* і побудуйте фільтр меню з умовою за назвою страви.

6. Дискриміновані об'єднання

Дискримінованe об'єднання (discriminated union, DU), відоме також як тип-сума (sum type) або тегове об'єднання (tagged union), - це тип, значення якого належить до одного з фіксованого набору варіантів, причому кожен варіант може мати власний склад даних. Перед використанням значення необхідно визначити, до якого саме варіанта воно належить.

У мові F# дискриміновані об'єднання підтримуються на рівні мови й широко використовуються. До версії C# 14 включно вбудованої підтримки в мові C# не було, і об'єднання моделювали абстрактними записами та зіставленням зі зразком. Версія C# 15 запровадила закриті ієрархії (*closed*) і типи-об'єднання (*union*). У розділі розглядаються обидва способи: класичне моделювання, що розкриває внутрішню будову об'єднань і залишається необхідним для коду на попередніх версіях мови, та вбудовані засоби C# 15.

На відміну від прийомів попередніх розділів, дискримінованe об'єднання не є окремим методом-розширенням чи локальним скороченням коду. За своєю природою воно ближче до шаблону проєктування (design pattern): має певну структуру і логіку, яку необхідно реалізувати навколо неї.

Мета розділу - сформувати розуміння дискримінованих об'єднань як засобу моделювання предметної області та обробки результатів операцій, а також навички їх реалізації в мові C#.

Після опрацювання розділу студент повинен:

- знати поняття алгебраїчного типу даних, типу-добутку й типу-суми та вміти обчислювати потужність типу;
- розуміти відмінність між ієрархією успадкування та дискримінованим об'єднанням;
- уміти реалізовувати дискриміновані об'єднання в мові C# за допомогою абстрактних записів, закритих ієрархій і типів-об'єднань C# 15 та обробляти їх виразами *switch*;
- уміти використовувати дискриміновані об'єднання для подання результатів пошуку, операцій без результату та введення користувача;
- знати узагальнені об'єднання *Maybe*, *Result* і *Either* та вміти обирати між ними.

Приклади розділу містяться в проєкті *Ch06.DiscriminatedUnions*.

6.1. Обмеження ієрархії успадкування

Нехай розробляється система резервування для мережі закладів гостинності. Спочатку вона обслуговує лише проживання в готелях;

передбачено два види бронювання - без харчування та з харчуванням (лістинг 6.1).

Лістинг 6.1 - Класи бронювання проживання

```
1. public class RoomStay
2. {
3.     public int Id { get; set; }
4.     public string Hotel { get; set; } = "";
5.     public string RoomType { get; set; } = "";
6.     public DateOnly CheckIn { get; set; }
7.     public int Nights { get; set; }
8. }
9.
10.  public class RoomStayWithMeals : RoomStay
11.  {
12.      public int MealsPerDay { get; set; }
13.  }
```

Для сторінки особистого кабінету гостя потрібно сформувати текстовий опис бронювань. Завдяки оператору *is* це нескладно (лістинг 6.2).

Лістинг 6.2 - Формування опису бронювання

```
1. public static string FormatStay(RoomStay s) =>
2.     $"Готель: {s.Hotel}" + Environment.NewLine +
3.     $"Номер: {s.RoomType}, ночей: {s.Nights}" +
4.     (s is RoomStayWithMeals m
5.         ? Environment.NewLine + $"Харчування: {m.MealsPerDay} раз/день"
6.         : string.Empty);
```

Згодом мережа починає приймати бронювання столиків у ресторанах. Бронювання столика не має ні готелю, ні категорії номера, ні кількості ночей, натомість має час, кількість гостей і потребу в дитячому стільці (лістинг 6.3).

Лістинг 6.3 - Клас бронювання столика

```
1. public class TableBooking
2. {
3.     public int Id { get; set; }
4.     public string Restaurant { get; set; } = "";
5.     public DateTime Time { get; set; }
6.     public int Guests { get; set; }
7.     public bool NeedsHighChair { get; set; }
8. }
```

Подати бронювання столика як нащадка класу *RoomStay* неможливо без спотворення предметної області. Іноді для розв'язання такої проблеми всі поля об'єднують в одному класі й додають логічну ознаку, що вказує, які з полів слід брати до уваги (лістинг 6.4).

Лістинг 6.4 - Об'єднаний клас з ознакою типу

```
1. public class CustomerReservation
2. {
3.     public int Id { get; set; }
4.     public string Venue { get; set; } = "";       // готель або ресторан
5.     public string RoomType { get; set; } = "";    // лише для готелю
6.     public DateTime Start { get; set; }
7.     public int Nights { get; set; }               // лише для готелю
8.     public int Guests { get; set; }               // лише для столика
9.     public bool NeedsHighChair { get; set; }      // лише для столика
10.      public bool IsTableBooking { get; set; }
11.  }
```

Такий підхід є невдалим з кількох причин. По-перше, він порушує **принцип розділення інтерфейсів**: кожен об'єкт змушений містити поля, які до нього не стосуються. По-друге, щоб уникнути дублювання, різні за змістом поняття об'єднуються в одне поле (*Venue* означає то готель, то ресторан; *Start* - то дату заїзду, то час візиту), і код втрачає терміни, що відображають предметну область. По-третє, ніщо не заважає створити об'єкт із суперечливим поєднанням значень.

Альтернатива - зберігати типи цілком незалежними. Тоді втрачається можливість обробляти всі бронювання гостя однією колекцією, наприклад відобразити їх в одній таблиці в хронологічному порядку.

6.2. Дискримінований об'єднання в мові C#

6.2.1. Алгебраїчні типи даних

Щоб зрозуміти природу дискримінованого об'єднання, доцільно розглянути тип даних як **множину допустимих значень**. Кількість елементів цієї множини називають **потужністю типу** (cardinality) і позначають $|T|$. Так, тип *bool* має два значення, $|bool| = 2$; перелічення днів тижня має сім значень; тип *byte* - 256.

Складені типи у функціональному програмуванні будуються з простіших двома основними операціями, через що їх називають **алгебраїчними типами даних** (algebraic data types, ADT).

Тип-добуток (product type) містить одночасно значення кожного зі своїх компонентів: «A **і** B». Його множина значень - декартів добуток множин компонентів, тому $|A \times B| = |A| \cdot |B|$. У мові C# типами-добутками є кортежі, записи, структури й класи з полями. Наприклад, запис *(bool Vip, DayOfWeek Day)* має $2 \cdot 7 = 14$ можливих значень.

Тип-сума (sum type) містить значення **рівно одного** з варіантів: «A **або** B». Його множина значень - розмічене об'єднання множин варіантів, тому $|A + B| = |A| + |B|$. Слово «розмічене» (дискримінований) означає, що кожне значення несе позначку свого варіанта: навіть якщо два варіанти

мають однаковий склад даних, вони розрізняються. Тип-сума *bool* + *DayOfWeek* має 2 + 7 = 9 значень (рис. 6.1).

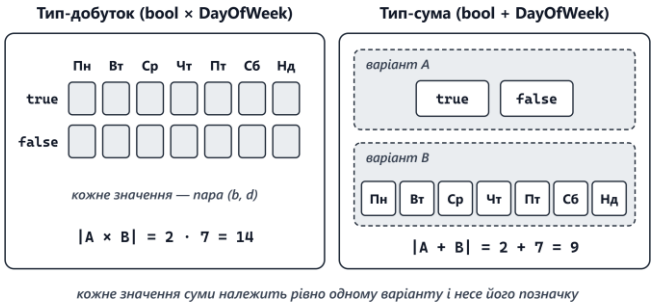


Рисунок 6.1 - Тип-добуток і тип-сума: потужність множин значень

Алгебраїчні типи можна вкладати: варіанти суми самі є добутками, а компоненти добутку можуть бути сумами. Саме так будується модель бронювань: «бронювання - це **або** проживання (готель **і** категорія **і** дата **і** кількість ночей), **або** столик (ресторан **і** час **і** кількість гостей)».

Практичне значення цієї «алгебри» полягає у відповідності між потужністю типу та кількістю станів, які мусить коректно обробити програма. Об'єднаний клас із логічною ознакою типу (лістинг 6.4) є добутком усіх полів: його потужність значно більша за кількість змістовно допустимих комбінацій, і зайві комбінації відповідають некоректним станам, від яких код змушений захищатися перевірками. Тип-сума має рівно стільки значень, скільки існує коректних випадків. Цей принцип у функціональному проєктуванні формулюють як вимогу **робити некоректні стани непредставлюваними** (make illegal states unrepresentable).

Звичний тип із «порожнім» значенням теж є сумою: *int?* = *int* + {*null*}, тобто його потужність на одиницю більша за потужність *int*. Об'єднання *Maybe<T>* (підрозділ 6.7) є явним записом тієї самої суми T + 1, проте з обов'язковою обробкою обох варіантів.

6.2.2. Моделювання об'єднання ієрархією записів

Мовою F# об'єднання для цієї задачі оголошується так:

type Reservation = | RoomStay | RoomStayWithMeals | TableBooking

Значення типу *Reservation* належить до одного з трьох варіантів, кожен з яких має власний набір властивостей. Найближчим аналогом у мові C# є абстрактний базовий тип, від якого успадковуються варіанти. У лістингу 6.5 для цього використано абстрактний запис із вкладеними закритими (*sealed*) записами-варіантами. Закритий конструктор базового типу

гарантує, що інших варіантів, окрім оголошених усередині нього, створити неможливо, а записи забезпечують незмінність і рівність за значенням.

Лістинг 6.5 - Дискримінованe об'єднання Reservation

```
1. public abstract record Reservation(int Id)
2. {
3.     // Закритий конструктор: нові варіанти можна оголосити
4.     // лише всередині типу Reservation
5.     private Reservation() : this(0) { }
6.
7.     public sealed record RoomStay(int Id, string Hotel,
8.         string RoomType, DateOnly CheckIn, int Nights)
9.         : Reservation(Id);
10.
11.      public sealed record RoomStayWithMeals(int Id, string Hotel,
12.          string RoomType, DateOnly CheckIn, int Nights,
13.          int MealsPerDay) : Reservation(Id);
14.
15.      public sealed record TableBooking(int Id, string Restaurant,
16.          DateTime Time, int Guests, bool NeedsHighChair)
17.          : Reservation(Id);
18.  }
```

На перший погляд код подібний до вихідної ієрархії, проте відмінність принципова. Базовий тип абстрактний - створити «просто резервування» неможливо. Замість генеалогічного дерева, у якому всі класи мусять відповідати поняттям батьківського класу, утворюється набір рівноправних альтернатив (рис. 6.2). Варіант *TableBooking* не зобов'язаний узгоджуватися з поняттями проживання і використовує терміни власної предметної області: він є не розширенням бронювання номера, а альтернативою йому.

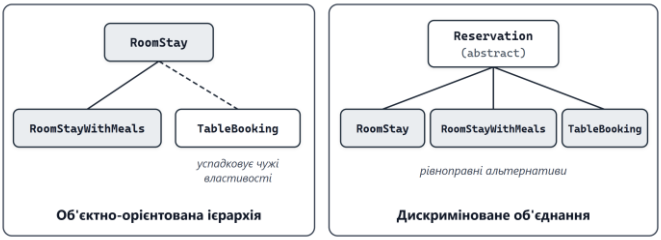


Рисунок 6.2 - Ієрархія успадкування та дискримінованe об'єднання

Дискримінованe об'єднання дає змогу зберігати різнорідні бронювання в одному масиві. Обробка виконується виразом *switch*, що розглядає кожен варіант (лістинг 6.6).

Лістинг 6.6 - Обробка варіантів об'єднання виразом switch

```
1. public static string Format(Reservation r) => r switch
2. {
3.     Reservation.RoomStay s =>
4.         $"{s.CheckIn:dd.MM} {s.Hotel}: {s.RoomType}, " +
```

```
5.          $"{s.Nights} ноч.",
6.      Reservation.RoomStayWithMeals m =>
7.          $"{m.CheckIn:dd.MM} {m.Hotel}: {m.RoomType}, " +
8.          $"{m.Nights} ноч., харчування {m.MealsPerDay}×",
9.      Reservation.TableBooking t =>
10.           $"{t.Time:dd.MM HH:mm} {t.Restaurant}: столик " +
11.           $"на {t.Guests}" + (t.NeedsHighChair ? ", дит. стілець" : ""),
12.       _ => throw new ArgumentOutOfRangeException(nameof(r))
13.   };
```

Для такої ієрархії компілятор не може перевірити, що вираз *switch* охоплює всі варіанти (як це робить компілятор F#): він не враховує, що закритий конструктор унеможливлює інших нащадків. Тому без гілки _ видається попередження про неповноту виразу. Гілка _ з генеруванням винятку явно позначає ситуацію, яка за правильної роботи програми виникнути не може. Недолік такого рішення полягає в тому, що після додавання нового варіанта компілятор не вкаже місць, де його не оброблено: помилка виявиться лише під час виконання.

Перелік бронювань гостя формується одним запитом LINQ (лістинг 6.7).

Лістинг 6.7 - Опрацювання колекції різнорідних бронювань

```
1. Reservation[] reservations =
2. [
3.     new Reservation.RoomStay(1, "«Буковина»", "Стандарт",
4.         new DateOnly(2026, 12, 29), 3),
5.     new Reservation.TableBooking(2, "«Під ратушею»",
6.         new DateTime(2026, 12, 31, 20, 0, 0), 4, true),
7.     new Reservation.RoomStayWithMeals(3, "«Карпати»", "Люкс",
8.         new DateOnly(2027, 1, 2), 5, 3)
9. ];
10.
11.  var overview = reservations
12.      .Select(Format)
13.      .Order();
```

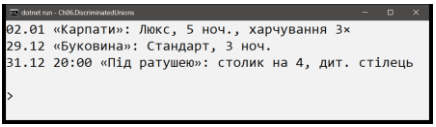


Рисунок 6.3 - Результат виконання прикладу «Резервування»

Дискримінований об'єднання можна порівняти з запечатаним конвертом, у якому міститься один із кількох можливих документів: до відкриття конверта невідомо, який саме. Функція, якій вміст не важливий, може передати конверт далі, не відкриваючи його; функція, що обробляє вміст, зобов'язана розглянути всі можливі варіанти. Цю властивість часто ілюструють мисленнєвим експериментом Е. Шредінгера: до перевірки система перебуває в одному з кількох станів, і лише перевірка визначає, в якому саме.

6.2.3. Закриті ієрархії (C# 15)

Версія C# 15 розв'язує проблему перевірки повноти модифікатором **`closed`**. Від закритого класу чи запису можна успадковуватися лише в межах збірки, де його оголошено, тож множина безпосередніх нащадків відома компілятору. Закритий тип неявно є абстрактним, а закритий конструктор для обмеження нащадків більше не потрібен. Нащадків можна оголошувати як окремі типи верхнього рівня (лістинг 6.8).

Лістинг 6.8 - Закрита ієрархія бронювань

```
// Закрита ієрархія (C# 15): нащадків можна оголосити лише
// в тій самій збірці; базовий тип неявно абстрактний
public closed record Booking(int Id);

public sealed record RoomStay(int Id, string Hotel, string RoomType,
    DateOnly CheckIn, int Nights) : Booking(Id);

public sealed record TableBooking(int Id, string Restaurant,
    DateTime Time, int Guests) : Booking(Id);
```

Вираз *switch*, що розглядає всіх безпосередніх нащадків, компілятор вважає повним, і гілка _ не потрібна (лістинг 6.9). Якщо згодом до ієрархії додати новий варіант, компілятор видасть попередження CS8509 для кожного виразу *switch*, де його не оброблено, і зазначить, який саме шаблон пропущено.

Лістинг 6.9 - Повний вираз switch без гілки за замовчуванням

```
// Гілка _ не потрібна: компілятор знає всіх нащадків Booking
public static string Format(Booking b) => b switch
{
    RoomStay s => $"{s.CheckIn:dd.MM} {s.Hotel}: {s.Nights} ноч.",
    TableBooking t =>
        $"{t.Time:dd.MM HH:mm} {t.Restaurant}: на {t.Guests}"
};
```

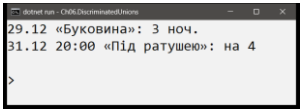


Рисунок 6.4 - Результат виконання прикладу «Закрита ієрархія»

Перевірка повноти поширюється лише на безпосередніх нащадків і лише на вирази *switch*; інструкція *switch* з пропущеним варіантом компілюється без попередження.

6.2.4. Типи-об'єднання (C# 15)

Закрита ієрархія вимагає спільного базового типу. На практиці ж варіанти часто є незалежними типами, оголошеними в різних модулях системи: у підрозділі 6.1 саме тому довелося відмовитися від спільного опрацювання незалежних типів бронювань. Нехай готель пропонує

гостям додаткові послуги - відвідування СПА, трансфер та екскурсію, - описані незалежними записами без спільного предка (лістинг 6.10).

Лістинг 6.10 - Незалежні типи послуг

```
1. // Незалежні типи різних модулів: спільного базового типу немає
2. public sealed record SpaVisit(string Procedure, DateTime Time);
3. public sealed record Transfer(string From, string To, DateTime Time);
4. public sealed record Excursion(string Route, int People);
```

Тип-об'єднання (union type) C# 15 поєднує наявні типи в тип-суму, не вимагаючи від них спільного предка. Оголошення займає один рядок (лістинг 6.11).

Лістинг 6.11 - Оголошення типу-об'єднання

```
1. // Тип-об'єднання (C# 15): значення є рівно одним із перелічених типів
2. public union GuestService(SpaVisit, Transfer, Excursion);
```

Компілятор створює для об'єднання структуру з конструктором для кожного типу-варіанта та властивістю *Value* типу *object?*, що містить поточний варіант. Кожен тип-варіант неявно перетворюється на тип-об'єднання, а присвоєння значення іншого типу є помилкою компіляції. Шаблони у виразі *switch* застосовуються до вмісту *Value*, а повнота перевіряється за переліком типів-варіантів (лістинги 6.12 і 6.13).

Лістинг 6.12 - Обробка типу-об'єднання

```
1. public static string Describe(GuestService service) => service switch
2. {
3.     SpaVisit s => $"СПА: {s.Procedure} о {s.Time:HH:mm}",
4.     Transfer t => $"Трансфер {t.From} -> {t.To} о {t.Time:HH:mm}",
5.     Excursion e => $"Екскурсія «{e.Route}», осіб: {e.People}"
6. };
```

Лістинг 6.13 - Формування плану послуг гостя

```
1. var day = new DateTime(2026, 12, 30);
2. // Неявне перетворення кожного типу-варіанта на об'єднання
3. GuestService[] plan =
4. [
5.     new Transfer("Вокзал", "Готель", day.AddHours(9)),
6.     new SpaVisit("Масаж", day.AddHours(15)),
7.     new Excursion("Хотинська фортеця", 6)
8. ];
9.
10.  foreach (var s in plan)
11.      Console.WriteLine(Describe(s));
12.
13.  // Властивість Value містить поточний варіант
14.  Console.WriteLine(plan[2].Value?.GetType().Name);
```

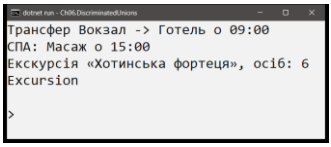


Рисунок 6.5 - Результат виконання прикладу «Тип-об'єднання»

Порівняння способів подання типу-суми в мові C# наведено в таблиці 6.1.

Таблиця 6.1 - Способи подання дискримінованого об'єднання в мові C#

Ознака	**Абстрактний запис із закритим конструктором**	**Закрита ієрархія (closed)**	**Тип-об'єднання (union)**
Версія мови	C# 9 і новіші	C# 15	C# 15
Спільний базовий тип	Обов'язковий	Обов'язковий	Не потрібен
Перевірка повноти *switch*	Ні, потрібна гілка _	Так	Так
Варіанти	Вкладені записи	Нащадки в тій самій збірці	Будь-які наявні типи
Подання в пам'яті	Посилання на об'єкт варіанта	Посилання на об'єкт варіанта	Структура з полем *object?*; типи-значення упаковуються

Закриту ієрархію доцільно використовувати, коли варіанти є поняттями однієї предметної області зі спільними властивостями (як ідентифікатор бронювання); тип-об'єднання - коли поєднуються незалежні типи, зокрема з різних бібліотек.

6.3. Моделювання форм імені гостя

Готель, що приймає іноземних гостей, мусить коректно відображати імена, оформлені за різними національними традиціями. Для імені, побудованого за англійською традицією (звертання, ім'я, додаткові імена, прізвище), достатньо простого класу (лістинг 6.14).

Лістинг 6.14 - Клас для імені англійського зразка

```
1. public class EnglishStyleName
2. {
3.     public string Honorific { get; init; } = "";
4.     public string FirstName { get; init; } = "";
5.     public IEnumerable<string> MiddleNames { get; init; } = [];
6.     public string LastName { get; init; } = "";
7. }
```

Проте українське ім'я складається з прізвища, імені та імені по батькові й у документах записується саме в такому порядку. Китайське ім'я записується в порядку «родове ім'я - особисте ім'я», шанобливе звертання (наприклад, «xiansheng», аналог «пан») ставиться після імені, а багато людей мають також «західне» ім'я для професійного спілкування. Жодна з цих форм не вписується в структуру англійського імені без перекручення змісту полів, а додавання інструкцій *if* для кожної національної традиції швидко зробить код некерованим.

Дискримінований об'єднання дає змогу подати кожну форму імені структурою, що точно відповідає реальності (лістинг 6.15).

Лістинг 6.15 - Дискримінований об'єднання форм імені

```
1. public abstract record GuestName
2. {
3.     private GuestName() { }
4.
5.     // Ім'я, побудоване за англійською традицією
6.     public sealed record English(string Honorific, string FirstName,
7.         string[] MiddleNames, string LastName) : GuestName;
8.
9.     // Українське ім'я: прізвище, ім'я, по батькові
10.      public sealed record Ukrainian(string LastName, string FirstName,
11.          string Patronymic) : GuestName;
12.
13.      // Китайське ім'я: родове ім'я, особисте ім'я, шанобливе
14.      // звертання після імені та необов'язкове «західне» ім'я
15.      public sealed record Chinese(string FamilyName, string GivenName,
16.          string Honorific, string? CourtesyName) : GuestName;
17.  }
```

Функція форматування розглядає кожен варіант окремо (лістинг 6.16), а реєстр гостей може містити імена всіх форм (лістинг 6.17).

Лістинг 6.16 - Форматування імені залежно від форми

```
1. public static string FormatName(GuestName name) => name switch
2. {
3.     GuestName.English e =>
4.         string.Join(" ", [e.Honorific, e.FirstName,
5.                           .. e.MiddleNames, e.LastName]),
6.     GuestName.Ukrainian u =>
7.         $"{u.LastName} {u.FirstName} {u.Patronymic}",
8.     GuestName.Chinese c =>
9.         $"{c.FamilyName} {c.GivenName} {c.Honorific}" +
10.          (c.CourtesyName is null ? "" : $" ({c.CourtesyName})"),
11.      _ => throw new ArgumentOutOfRangeException(nameof(name))
12.  };
```

Лістинг 6.17 - Реєстр гостей з іменами різних форм

```
1. GuestName[] registry =
2. [
3.     new GuestName.Ukrainian("Ткач", "Олена", "Петрівна"),
4.     new GuestName.English("Mr", "John", ["Robert"], "Smith"),
5.     new GuestName.Chinese("Wang", "Lei", "xiansheng", "Leo")
6. ];
7. var output = string.Join(Environment.NewLine,
8.                          registry.Select(FormatName));
```



Рисунок 6.6 - Результат виконання прикладу «Імена гостей»

Той самий принцип можна застосувати до будь-якої національної традиції: назви полів завжди відповідатимуть її поняттям, а форматування буде коректним без перепризначення наявних полів.

6.4. Результат пошуку в базі даних

Дискриміновані об'єднання особливо доцільно використовувати як типи результатів функцій, що звертаються до зовнішніх джерел даних. Функцію пошуку гостя за ідентифікатором часто оголошують так, ніби вона завжди повертає об'єкт гостя. Насправді це лише один із можливих результатів: гостя з таким ідентифікатором може не існувати, або під час звернення до бази даних може виникнути помилка, не пов'язана з кодом програми (мережевий збій тощо). Повернення *null* не пояснює, що саме сталося, а генерування винятку переносить проблему в інше місце програми.

Усі три можливі результати можна явно описати дискримінованим об'єднанням (лістинг 6.18).

Лістинг 6.18 - Дискриміноване об'єднання результатів пошуку гостя

```
public abstract record GuestLookupResult(int Id)
{
    private GuestLookupResult() : this(0) { }

    public sealed record Found(int Id, Guest Guest)
        : GuestLookupResult(Id);

    public sealed record NotFound(int Id) : GuestLookupResult(Id);

      public sealed record LookupError(int Id, Exception Error)
          : GuestLookupResult(Id);
  }
```

Функція пошуку повертає один із трьох варіантів, і стан результату визначено вже в момент повернення (лістинг 6.19). Код, що використовує функцію, не мусить аналізувати результат, щоб з'ясувати, чи була операція успішною: варіанти повністю описують кожен випадок.

Лістинг 6.19 - Функція пошуку гостя

```
public GuestLookupResult GetGuest(int id)
{
    try
    {
        var guest = db.Find(id);
        return guest is null
            ? new GuestLookupResult.NotFound(id)
            : new GuestLookupResult.Found(id, guest);
    }
      catch (Exception e)
      {
          return new GuestLookupResult.LookupError(id, e);
      }
  }
```

Обробка результату зводиться до виразу *switch* (лістинг 6.20).

Лістинг 6.20 - Обробка результату пошуку

```
public static string DescribeGuest(GuestRepository repo, int id) =>
    repo.GetGuest(id) switch
    {
        GuestLookupResult.Found f => $"Гість №{id}: {f.Guest.Name}",
        GuestLookupResult.NotFound => $"Гостя №{id} не знайдено",
        GuestLookupResult.LookupError e =>
            $"Помилка пошуку №{id}: {e.Error.Message}",
        _ => throw new InvalidOperationException()
    };
```

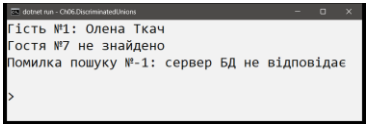


Рисунок 6.7 - Результат виконання прикладу «Пошук гостя»

6.5. Операції без результату

Попередній приклад стосувався функції, що повертає значення. Дискримінованe об'єднання корисне і для операцій, які значення не повертають, але можуть завершитися помилкою, наприклад надсилання гостеві електронного листа з підтвердженням бронювання. У цьому випадку важливі лише два стани: успіх і невдача з відомостями про помилку (лістинги 6.21 і 6.22).

Лістинг 6.21 - Об'єднання результатів надсилання повідомлення

```
1. public abstract record SendResult
2. {
3.     private SendResult() { }
4.
5.     public sealed record Success : SendResult;
6.     public sealed record Failure(Exception Error) : SendResult;
7. }
```

Лістинг 6.22 - Надсилання підтвердження з поверненням об'єднання

```
1. public SendResult SendConfirmation(string recipient, string text)
2. {
3.     try
4.     {
5.         mail.Send(recipient, text);
6.         return new SendResult.Success();
7.     }
8.     catch (Exception e)
9.     {
10.          return new SendResult.Failure(e);
11.      }
12.  }
```

Використання функції наведено в лістингу 6.23.

Лістинг 6.23 - Обробка результату надсилання

```
1. var result = notifier.SendConfirmation(address,
2.     "Бронювання №1042 підтверджено");
3.
4. var message = result switch
5. {
6.     SendResult.Success => "Підтвердження надіслано",
7.     SendResult.Failure f =>
8.         "Помилка надсилання: " + f.Error.Message,
9.     _ => "Невідомий результат"
10.  };
```

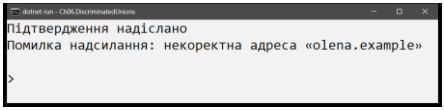


Рисунок 6.8 - Результат виконання прикладу «Підтвердження»

Функція повідомляє про помилку, не генеруючи винятку, і при цьому жоден варіант результату не містить властивостей, які йому не потрібні (на відміну від класу *ExecutionResult* з розділу 5).

6.6. Обробка введення користувача

Інтерактивна програма - наприклад, кіоск самообслуговування в ресторані - постійно отримує введення від користувача. Найчастіше потрібне ціле число (номер пункту меню, кількість гостей чи порцій), іноді - текст. Якби модуль введення повертав необроблений рядок, у кожному місці програми, що очікує число, довелося б перевіряти рядок на порожність і розбирати його. Доцільніше описати можливі стани введення дискримінованим об'єднанням і зосередити перевірки в одному місці (лістинг 6.24).

Лістинг 6.24 - Дискримінований об'єднання станів введення

```
1. public abstract record UserInput
2. {
3.     private UserInput() { }
4.
5.     public sealed record Text(string Value) : UserInput;
6.     public sealed record Integer(int Value) : UserInput;
7.     public sealed record Empty : UserInput;
8.     public sealed record ConsoleError(Exception Error) : UserInput;
9. }
```

Можливість помилок консолі не слід виключати, оскільки консоль перебуває поза контролем програми. Обробка введення будується як поступовий перехід від зовнішнього «нечистого» світу до контрольованого чистого коду, подібно до багатоступеневого шлюзу (рис. 6.9).

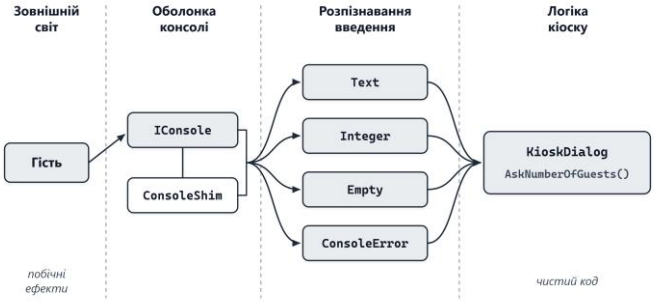


Рисунок 6.9 - Етапи обробки введення користувача

Перший рівень - тонка оболонка над системною консоллю, прихована за інтерфейсом (лістинг 6.25). Це єдина частина системи з побічними

ефектами, тож її обсяг мінімальний. Інтерфейс дає змогу під час тестування підставляти замість консолі імітацію із заздалегідь заданими відповідями.

Лістинг 6.25 - Оболонка над системною консоллю

```
1. public interface IConsole
2. {
3.     UserInput ReadInput(string prompt);
4.     void WriteMessage(string message);
5. }
6.
7. // Тонка оболонка над системною консоллю: єдине місце
8. // з побічними ефектами введення-виведення
9. public sealed class ConsoleShim : IConsole
10.  {
11.      public UserInput ReadInput(string prompt)
12.      {
13.          try
14.          {
15.              Console.WriteLine(prompt);
16.              var input = Console.ReadLine();
17.              return new UserInput.Text(input ?? "");
18.          }
19.          catch (Exception e)
20.          {
21.              return new UserInput.ConsoleError(e);
22.          }
23.      }
24.
25.      public void WriteMessage(string message) =>
26.          Console.WriteLine(message);
27.  }
```

Другий рівень аналізує отриманий текст і перетворює його на відповідний варіант об'єднання (лістинг 6.26).

Лістинг 6.26 - Розпізнавання введення

```
1. public sealed class UserInteraction(IConsole console)
2. {
3.     public UserInput GetInput(string prompt) =>
4.         console.ReadInput(prompt) switch
5.         {
6.             UserInput.Text t when string.IsNullOrWhiteSpace(t.Value) =>
7.                 new UserInput.Empty(),
8.             UserInput.Text t when int.TryParse(t.Value, out var n) =>
9.                 new UserInput.Integer(n),
10.            var other => other
11.         };
12.
13.     public void WriteMessage(string message) =>
14.         console.WriteMessage(message);
15. }
```

Тепер запит цілого числа в межах допустимого діапазону записується стисло (лістинг 6.27): якщо введено коректне число, воно повертається; інакше користувачеві виводиться пояснення, і функція рекурсивно

повторює запит. Блок *try/catch* тут не потрібен, оскільки помилки консолі вже оброблено на нижчому рівні.

Лістинг 6.27 - Запит кількості гостей з повторенням

```
1. public int AskNumberOfGuests() =>
2.     ui.GetInput("Скільки гостей буде за столиком?") switch
3.     {
4.         UserInput.Integer { Value: >= 1 and <= 12 } i => i.Value,
5.         UserInput.Integer => Retry("Допустимо від 1 до 12 гостей"),
6.         UserInput.Empty => Retry("Відповідь не може бути порожньою"),
7.         _ => Retry("Введіть ціле число")
8.     };
9.
10.  private int Retry(string message)
11.  {
12.      ui.WriteMessage(message);
13.      return AskNumberOfGuests();
14.  }
```

Результат виконання з імітованою консоллю, на яку послідовно подаються відповіді «» (порожній рядок), «четверо», «20» та «4»:

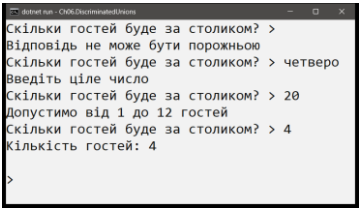


Рисунок 6.10 - Результат виконання прикладу «Введення з консолі»

Якщо в програмі багато місць, де потрібне числове введення, винесення перевірки в структуру результату позбавляє від значного обсягу повторюваного коду.

6.7. Узагальнені об'єднання

Усі розглянуті об'єднання були специфічними для конкретних задач. Оголошення нового об'єднання для кожної функції є надмірним, тож доцільно мати набір узагальнених об'єднань для типових ситуацій. Підрозділи 6.7.1-6.7.4 розглядають їх класичну реалізацію ієрархіями записів, що використовується й у наступних розділах посібника, а підрозділ 6.7.5 - реалізацію засобами C# 15. Розширені можливості цих структур розглядаються в розділі 7.

6.7.1. Maybe

Якщо результат функції може бути відсутнім, використовується об'єднання **`Maybe`** (у різних мовах і бібліотеках - *Option*, *Optional*) з двома

варіантами: значення є (*Something*, також *Some*, *Just*) або його немає (*Nothing*, також *None*) (лістинг 6.28).

Лістинг 6.28 - Об'єднання Maybe з двома варіантами

```
1. public abstract record Maybe<T>
2. {
3.     private Maybe() { }
4.
5.     public sealed record Something(T Value) : Maybe<T>;
6.     public sealed record Nothing : Maybe<T>;
7. }
```

Maybe є «обгорткою» навколо фактичного типу результату, що сигналізує зовнішньому коду: значення може бути відсутнім. Функцію пошуку страви в меню наведено в лістингу 6.29, а її використання - у лістингу 6.30.

Лістинг 6.29 - Пошук страви з поверненням Maybe

```
1. public Maybe<MenuItem> FindDish(string code)
2. {
3.     try
4.     {
5.         var data = Query(code);
6.         return data is null
7.             ? new Maybe<MenuItem>.Nothing()
8.             : new Maybe<MenuItem>.Something(data);
9.     }
10.      catch (Exception e)
11.      {
12.          Console.WriteLine($"[журнал] {e.Message}");
13.          return new Maybe<MenuItem>.Nothing();
14.      }
15.  }
```

Лістинг 6.30 - Обробка результату типу Maybe

```
1. var message = repo.FindDish(code) switch
2. {
3.     Maybe<MenuItem>.Something s =>
4.         $"{s.Value.Name}: {s.Value.Price} грн",
5.     Maybe<MenuItem>.Nothing => "Страву не знайдено",
6.     _ => throw new InvalidOperationException()
7. };
```

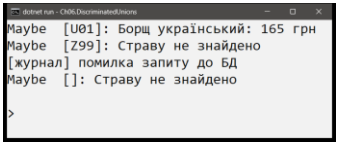


Рисунок 6.11 - Результат виконання прикладу «Maybe і Result» (фрагмент)

Помилки в такому варіанті обробляються недостатньо добре: стан *Nothing* запобігає необробленому винятку, а помилка записується до

журналу, проте кінцевий користувач не отримує жодних відомостей про її причину.

6.7.2. Result

Альтернативою є об'єднання `**Result**`, яке відображає можливість помилки замість результату (лістинг 6.31). Функцію пошуку наведено в лістингу 6.32, а обробку результату - у лістингу 6.33.

Лістинг 6.31 - Об'єднання Result

```
1. public abstract record Result<T>
2. {
3.     private Result() { }
4.
5.     public sealed record Success(T Value) : Result<T>;
6.     public sealed record Failure(Exception Error) : Result<T>;
7. }
```

Лістинг 6.32 - Пошук страви з поверненням Result

```
1. public Result<MenuItem?> FindDishResult(string code)
2. {
3.     try
4.     {
5.         return new Result<MenuItem?>.Success(Query(code));
6.     }
7.     catch (Exception e)
8.     {
9.         return new Result<MenuItem?>.Failure(e);
10.      }
11.  }
```

Лістинг 6.33 - Обробка результату типу Result

```
1. var message = repo.FindDishResult(code) switch
2. {
3.     Result<MenuItem?>.Success { Value: null } =>
4.         "Страву не знайдено",
5.     Result<MenuItem?>.Success s =>
6.         $"{s.Value!.Name}: {s.Value.Price} грн",
7.     Result<MenuItem?>.Failure f =>
8.         "Помилка: " + f.Error.Message,
9.     _ => throw new InvalidOperationException()
10.  };
```

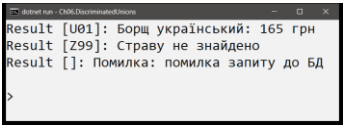


Рисунок 6.12 - Результат виконання прикладу «Maybe і Result» (фрагмент)

Тепер помилка відображається окремим варіантом, але перевірку на *null* у разі успіху доводиться виконувати коду, що отримує результат.

6.7.3. Вибір між Maybe і Result

Maybe повідомляє про відсутність даних і позбавляє від перевірок на *null*, але фактично приховує помилки. *Result* коректно відображає помилки, проте перекладає перевірку на *null* на код-споживач. Практичним компромісом є поєднання обох структур у *Maybe* з трьома варіантами: *Something*, *Nothing* і *Error*. Такий варіант відходить від класичного означення цих структур, однак охоплює практично всі ситуації, що виникають під час взаємодії з зовнішніми джерелами даних (лістинги 6.34 і 6.35).

Лістинг 6.34 - Maybe з трьома варіантами

```
1. public abstract record Maybe<T>
2. {
3.     private Maybe() { }
4.
5.     public sealed record Something(T Value) : Maybe<T>;
6.     public sealed record Nothing : Maybe<T>;
7.     public sealed record Error(Exception CapturedError) : Maybe<T>;
8. }
```

Лістинг 6.35 - Пошук страви з поверненням Maybe з трьома варіантами

```
1. public Maybe<MenuItem> FindDish(string code)
2. {
3.     try
4.     {
5.         var data = Query(code);
6.         return data is null
7.             ? new Maybe<MenuItem>.Nothing()
8.             : new Maybe<MenuItem>.Something(data);
9.     }
10.      catch (Exception e)
11.      {
12.          return new Maybe<MenuItem>.Error(e);
13.      }
14.  }
```

Код, що отримує результат, обробляє всі три стани одним виразом (лістинг 6.36) і може надати кінцевому користувачеві змістовну відповідь.

Лістинг 6.36 - Обробка Maybe з трьома варіантами

```
1. var message = repo.FindDish(code) switch
2. {
3.     Maybe<MenuItem>.Something s =>
4.         $"{s.Value.Name}: {s.Value.Price} грн",
5.     Maybe<MenuItem>.Nothing => "Страву не знайдено",
6.     Maybe<MenuItem>.Error e =>
7.         "Помилка: " + e.CapturedError.Message,
8.     _ => throw new InvalidOperationException()
9. };
```

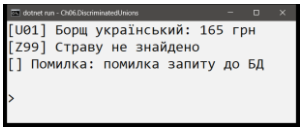


Рисунок 6.13 - Результат виконання прикладу «Maybe з трьома станами» (фрагмент)

Та сама структура придатна і для функцій, що повертають колекції: порожню колекцію доцільно подавати варіантом *Nothing* (лістинги 6.37 і 6.38).

Лістинг 6.37 - Пошук страв категорії з поверненням Maybe

```
1. public Maybe<IEnumerable<MenuItem>> FindByCategory(Category c)
2. {
3.     try
4.     {
5.         var data = items.Values.Where(m => m.Category == c)
6.                                .ToArray();
7.         return data.Length == 0
8.             ? new Maybe<IEnumerable<MenuItem>>.Nothing()
9.             : new Maybe<IEnumerable<MenuItem>>.Something(data);
10.     }
11.     catch (Exception e)
12.     {
13.         return new Maybe<IEnumerable<MenuItem>>.Error(e);
14.     }
15. }
```

Лістинг 6.38 - Обробка колекції в Maybe

```
1. var desserts = repo.FindByCategory(Category.Dessert) switch
2. {
3.     Maybe<IEnumerable<MenuItem>>.Something s =>
4.         "Десерти: " + string.Join(", ", s.Value.Select(x => x.Name)),
5.     Maybe<IEnumerable<MenuItem>>.Nothing => "Десертів немає",
6.     Maybe<IEnumerable<MenuItem>>.Error e =>
7.         "Помилка: " + e.CapturedError.Message,
8.     _ => throw new InvalidOperationException()
9. };
```

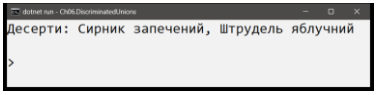


Рисунок 6.14 - Результат виконання прикладу «Maybe з трьома станами» (фрагмент)

6.7.4. Either

Maybe і *Result* описують невизначеність результату однієї функції. Якщо ж функція може повертати значення двох принципово різних типів, використовується об'єднання **`Either`** з варіантами *Left* і *Right* (лістинг 6.39).

За усталеною угодою в разі обробки помилок варіант *Right* позначає «правильний» результат, а *Left* - альтернативний.

Лістинг 6.39 - Об'єднання Either

```
1. public abstract record Either<TLeft, TRight>
2. {
3.     private Either() { }
4.
5.     public sealed record Left(TLeft Value) : Either<TLeft, TRight>;
6.     public sealed record Right(TRight Value) : Either<TLeft, TRight>;
7. }
```

У лістингу 6.40 гість ідентифікується введеним рядком, який може бути або адресою електронної пошти, або номером картки лояльності.

Лістинг 6.40 - Ідентифікація гостя за допомогою Either

```
1. // Гість ідентифікується або адресою e-mail,
2. // або номером картки лояльності
3. public static Either<string, int> ParseGuestKey(string input) =>
4.     int.TryParse(input, out var card)
5.         ? new Either<string, int>.Right(card)
6.         : new Either<string, int>.Left(input.Trim().ToLower());
7.
8. public static string Describe(Either<string, int> key) => key switch
9. {
10.      Either<string, int>.Left l => "пошук за e-mail: " + l.Value,
11.      Either<string, int>.Right r => "пошук за карткою №" + r.Value,
12.      _ => throw new InvalidOperationException()
13.  };
```

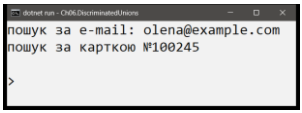


Рисунок 6.15 - Результат виконання прикладу «Either»

За потреби структуру можна розширити до трьох і більше варіантів, хоча обсяг шаблонного коду з узагальненими параметрами при цьому зростає.

6.7.5. Узагальнені об'єднання засобами C# 15

Тип-об'єднання може бути узагальненим і мати тіло з власними членами. У лістингу 6.41 об'єднання *Outcome<T>* поєднує успішний результат *Ok<T>* і невдачу *Fail* з поясненням причини та містить функції *Map* і *Bind*, що перетворюють успішне значення й пропускають невдачу без змін (детально ці функції розглядаються в розділі 7). Гілка *null* обробляє значення за замовчуванням: неініціалізоване об'єднання (*default*) має порожню властивість *Value*.

Лістинг 6.41 - Узагальнене об'єднання Outcome

```
1. public sealed record Ok<T>(T Value);
2. public sealed record Fail(string Reason);
```

```
3.
4. // Узагальнене об'єднання з власними членами
5. public union Outcome<T>(Ok<T>, Fail)
6. {
7.     public Outcome<TOut> Map<TOut>(Func<T, TOut> f) => this switch
8.     {
9.         Ok<T> ok => new Ok<TOut>(f(ok.Value)),
10.          Fail fail => fail,
11.          null => new Fail("Значення не ініціалізовано")
12.     };
13.
14.     public Outcome<TOut> Bind<TOut>(Func<T, Outcome<TOut>> f) =>
15.         this switch
16.         {
17.             Ok<T> ok => f(ok.Value),
18.             Fail fail => fail,
19.             null => new Fail("Значення не ініціалізовано")
20.         };
21. }
```

Функції пошуку ціни страви та перевірки кількості порцій повертають *Outcome* (лістинг 6.42). Значення варіантів неявно перетворюються на тип результату, тож конструктори об'єднання явно не викликаються.

Лістинг 6.42 - Функції з результатом Outcome

```
1. private static readonly Dictionary<string, decimal> Prices =
2.     new() { ["Борщ"] = 145m, ["Вареники"] = 120m };
3.
4. public static Outcome<decimal> PriceOf(string dish) =>
5.     Prices.TryGetValue(dish, out var p)
6.         ? new Ok<decimal>(p)
7.         : new Fail($"Страви «{dish}» немає в меню");
8.
9. public static Outcome<int> Portions(int n) =>
10.     n is > 0 and <= 20
11.         ? new Ok<int>(n)
12.         : new Fail($"Неприпустима кількість порцій: {n}");
```

Якщо для об'єднання визначити методи *Select* і *SelectMany* (лістинг 6.43), послідовність залежних кроків записується в синтаксисі запитів LINQ: обчислення продовжується, доки кожен крок успішний, і зупиняється на першій невдачі (лістинг 6.44). Механізм синтаксису запитів пояснюється в підрозділі 7.7.5.

Лістинг 6.43 - Підтримка синтаксису запитів LINQ для Outcome

```
1. public static class OutcomeLinq
2. {
3.     public static Outcome<TOut> Select<T, TOut>(
4.         this Outcome<T> o, Func<T, TOut> f) => o.Map(f);
5.
6.     public static Outcome<TOut> SelectMany<T, TMid, TOut>(
7.         this Outcome<T> o, Func<T, Outcome<TMid>> bind,
8.         Func<T, TMid, TOut> project) =>
9.         o.Bind(x => bind(x).Map(y => project(x, y)));
10. }
```

Лістинг 6.44 - Оформлення замовлення як ланцюжок залежних кроків

```
1. Outcome<decimal> Order(string dish, int n) =>
2.     from price in PriceOf(dish)
3.     from count in Portions(n)
4.     select price * count;
5.
6. Console.WriteLine(Show(Order("Вареники", 3)));
7. Console.WriteLine(Show(Order("Суші", 2)));
8. Console.WriteLine(Show(Order("Борщ", 40)));
```

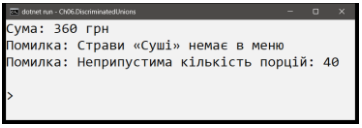


Рисунок 6.16 - Результат виконання прикладу «Узагальнене об'єднання»

Бібліотека класів .NET 11 містить лише інфраструктуру типів-об'єднань (атрибут *UnionAttribute* та інтерфейс *IUnion*), але не готові типи *Option<T>* чи *Result<T, E>*. Тому наступні розділи посібника використовують класичну реалізацію *Maybe* з бібліотеки предметної області, а читач може за бажанням переписати її засобами C# 15 (див. практичні завдання).

6.8. Висновки

1. Дискриміноване об'єднання - це тип, значення якого належить до одного з фіксованого набору варіантів із власним складом даних; перед використанням значення визначається його варіант.

2. На відміну від ієрархії успадкування, варіанти об'єднання є рівноправними альтернативами і не зобов'язані поділяти поняття базового класу.

3. Дискриміноване об'єднання є типом-сумою ($|A + B| = |A| + |B|$), тоді як записи й кортежі є типами-добутками ($|A \times B| = |A| \cdot |B|$). Узгодження потужності типу з кількістю коректних станів робить некоректні стани непредставлюваними.

4. До версії C# 14 включно дискриміновані об'єднання моделюються абстрактним записом із закритим конструктором і вкладеними записами-варіантами; повноту розгляду варіантів компілятор не перевіряє. Версія C# 15 запровадила закриті ієрархії (*closed*) і типи-об'єднання (*union*), для яких повнота виразів *switch* перевіряється.

5. Дискриміновані об'єднання доцільно використовувати як типи результатів функцій, що взаємодіють із зовнішнім середовищем: вони явно описують усі можливі наслідки операції й усувають потребу в *null* та винятках.

6. Обробку введення доцільно будувати як послідовне перетворення від неперевірених даних зовнішнього світу до варіантів об'єднання, з якими працює чистий код.

7. Узагальнені об'єднання *Maybe*, *Result* і *Either* описують типові ситуації: можливу відсутність значення, можливу помилку і два альтернативні типи результату. Практичним компромісом є *Maybe* з варіантами *Something*, *Nothing* і *Error*.

6.9. Контрольні запитання

1. Що таке дискриміноване об'єднання? Які ще назви має це поняття?

2. Чому бронювання столика не можна коректно подати нащадком класу бронювання номера?

3. Які недоліки має об'єднання всіх полів в одному класі з ознакою типу?

4. Як змоделювати дискриміноване об'єднання в мові C#? Яку роль відіграє закритий конструктор базового типу?

5. Чому у виразі *switch* над об'єднанням, змодельованим абстрактним записом, доводиться додавати гілку _? Як цю проблему розв'язує модифікатор *closed*?

6. Які переваги дає дискриміноване об'єднання для подання форм імені гостей з різних країн?

7. Які варіанти результату має функція пошуку в базі даних? Чому повернення *null* є недостатньо інформативним?

8. Як побудувати обробку введення користувача у вигляді «шлюзу» від нечистого коду до чистого?

9. Чим відрізняються *Maybe* і *Result*? Які переваги має *Maybe* з трьома варіантами?

10. Для чого призначене об'єднання *Either*? Яка угода стосується варіантів *Left* і *Right*?

11. Що таке потужність типу? Обчисліть потужність запису *(bool, bool, DayOfWeek)* і типу-суми *bool* + *DayOfWeek*.

12. Що означає принцип «робити некоректні стани непредставлюваними»? Наведіть приклад із предметної області HoReCa.

13. Чим тип-об'єднання C# 15 відрізняється від закритої ієрархії? Як компілятор подає тип-об'єднання в пам'яті?

6.10. Практичні завдання

1. Опишіть дискримінованим об'єднанням способи оплати рахунку в ресторані (готівка з решткою, банківська картка з кодом авторизації,

безготівковий рахунок компанії, сертифікат). Реалізуйте функцію формування рядка фіскального чека для кожного варіанта.

2. Змоделюйте статус замовлення на кухні (прийнято, готується з указаним кухарем, готово з часом приготування, скасовано з причиною) і реалізуйте функцію, що за колекцією замовлень формує зведення для екрана кухні.

3. Реалізуйте функцію пошуку номера готелю за номером кімнати з поверненням об'єднання з варіантами «вільний», «зайнятий до дати», «на ремонті», «не існує», «помилка доступу до даних».

4. Реалізуйте діалог кіоску самообслуговування, що запитує код страви (рядок із трьох символів) і кількість порцій (від 1 до 10) з використанням об'єднання *UserInput* та імітованої консолі.

5. Перепишіть функції пошуку гостя та бронювання так, щоб вони повертали *Maybe* з трьома варіантами, і реалізуйте функцію формування повідомлення адміністраторові, що коректно обробляє всі комбінації станів двох результатів.

6. Реалізуйте розбір рядка знижки, що може містити або відсоток («15 %»), або фіксовану суму («200 грн»), з поверненням *Either<int, decimal>*.

7. Перепишіть об'єднання способів оплати із завдання 1 як типоб'єднання C# 15, а статус замовлення із завдання 2 - як закриту ієрархію. Додайте новий варіант до кожного з типів і простежте, які попередження видасть компілятор.

8. Реалізуйте узагальнене об'єднання *Option<T>* засобами C# 15 з функціями *Map*, *Bind*, *Select* і *SelectMany* та перепишіть з його використанням пошук страви з лістингу 6.29.

7. Функціональний потік

Звернення до зовнішніх систем - баз даних, вебсервісів, файлових сховищ - супроводжується значним обсягом захисного коду. Перш ніж використати отримані дані, необхідно:

1) перехопити й обробити можливі винятки (збій мережі, недоступність сервера бази даних тощо);

2) перевірити, що отриманий результат не дорівнює *null*;

3) перевірити, що отримані дані є коректними за змістом.

Такий код відволікає від бізнес-логіки, заради якої функцію написано. Дискримінований об'єднання *Maybe* з розділу 6 частково розв'язує проблему, однак і воно потребує шаблонного коду для аналізу результату на кожному кроці. У цьому розділі *Maybe* доповнюється функціями *Map* і *Bind*, що дають змогу будувати ланцюжки обчислень без блоків *try/catch* і перевірок на *null*. Така структура є прикладом **монади** - одного з центральних понять функціонального програмування. Далі розглядаються закони монад, а також монади *Reader* і *State*.

Мета розділу - сформувати розуміння монади як засобу побудови безпечних ланцюжків обчислень і навички реалізації та застосування монад *Maybe*, *Reader* і *State* у мові C#.

Після опрацювання розділу студент повинен:

- уміти реалізовувати функції *Map* і *Bind* для *Maybe* та розуміти відмінність між ними;
- розуміти принцип «залізничного» програмування (railway oriented programming);
- уміти обробляти в ланцюжках *Maybe* асинхронні операції, журналювання та вкладені значення;
- знати закони монад і вміти перевіряти їх виконання;
- уміти застосовувати монаду *Reader* для впровадження залежностей і монаду *State* для обчислень зі станом;
- розуміти, чому *IEnumerable<T>* і *Task<T>* можна розглядати як монади;
- знати поняття функтора, аплікативного функтора й монади, їх закони та співвідношення;
- уміти використовувати аплікативну перевірку даних і синтаксис запитів LINQ для власних монад.

Приклади розділу містяться в проєкті *Ch07.FunctionalFlow*; реалізації монад - у просторі імен *HoReCa.Domain.Functional* спільної бібліотеки.

7.1. Maybe з функціями Map і Bind

Подальший виклад спирається на *Maybe* з трьома варіантами (див. підрозділ 6.7.3). У реалізації, наведеній у лістингу 7.1, варіанти *Nothing* і *Error*

додатково містять ознаку *Handled*, призначення якої пояснюється в підрозділі 7.4.

Лістинг 7.1 - Тип Maybe з трьома варіантами

```
/// <summary>
/// Maybe з трьома варіантами: значення є, значення немає, помилка.
/// Ознака Handled дає змогу обробляти стан лише один раз.
/// </summary>
public abstract record Maybe<T>
{
    private Maybe() { }

    public sealed record Something(T Value) : Maybe<T>;
      public sealed record Nothing(bool Handled = false) : Maybe<T>;
      public sealed record Error(Exception CapturedError,
                                 bool Handled = false) : Maybe<T>;
}
```

Перетворення довільного значення на *Maybe* виконує метод-розширення *ToMaybe* (лістинг 7.2): значення *null* перетворюється на *Nothing*, будь-яке інше - на *Something*.

Лістинг 7.2 - Метод ToMaybe

```
public static Maybe<T> ToMaybe<T>(this T? value) =>
    value is null
        ? new Maybe<T>.Nothing()
        : new Maybe<T>.Something(value);
```

7.1.1. Функція Map

Функція *Map* для *Maybe* аналогічна функції *Map* з розділу 5, але містить логіку, що визначає варіант результату (лістинг 7.3).

Лістинг 7.3 - Функція Map для Maybe

```
// Map: функція перетворює значення на значення
public static Maybe<TOut> Map<TIn, TOut>(this Maybe<TIn> @this,
                                         Func<TIn, TOut?> f)
{
    try
    {
        return @this switch
        {
            Maybe<TIn>.Something s => f(s.Value).ToMaybe(),
              Maybe<TIn>.Nothing n => new
    Maybe<TOut>.Nothing(n.Handled),
              Maybe<TIn>.Error e =>
                  new Maybe<TOut>.Error(e.CapturedError, e.Handled),
              _ => new Maybe<TOut>.Error(
                  new InvalidOperationException("невідомий стан Maybe"))
          };
      }
      catch (Exception ex)
      {
          return new Maybe<TOut>.Error(ex);
      }
}
```

Можливі такі випадки:

- поточне значення - *Something*: функція застосовується до значення, а результат загортається в новий *Maybe*; якщо функція повернула *null*, результатом буде *Nothing*;
- поточне значення - *Nothing*: функція не виконується, повертається *Nothing*;
- поточне значення - *Error*: функція не виконується, помилка передається далі;
- під час виконання функції виник виняток: він перехоплюється і перетворюється на *Error*.

Практичне значення цієї логіки видно на прикладі. У лістингу 7.4 наведено процедурну функцію формування привітання для працівника, що входить до касової системи ресторану.

Лістинг 7.4 - Процедурне формування привітання

```
1. public string MakeGreetingProcedural(int staffId)
2. {
3.     try
4.     {
5.         var s = repo.GetById(staffId);
6.         if (s != null)
7.         {
8.             return "Вітаємо, " + s.Position + " " + s.Name;
9.         }
10.          return "Працівника не знайдено";
11.     }
12.     catch (Exception e)
13.     {
14.         return "Сталася помилка: " + e.Message;
15.     }
16. }
```

Мета функції проста: отримати дані працівника і привітати його. Проте через можливість *null* і винятків більша частина коду є захисною. До того ж проблему інтерпретації результату перекладено на код, що викликає функцію: про помилку чи відсутність працівника він дізнається лише з тексту повідомлення.

З використанням *Maybe* і *Map* функція набуває вигляду, наведеного в лістингу 7.5.

Лістинг 7.5 - Формування привітання за допомогою Maybe

```
1. public Maybe<string> MakeGreeting(int staffId) =>
2.     staffId.ToMaybe()
3.         .Map(x => repo.GetById(x))
4.         .Map(x => "Вітаємо, " + x.Position + " " + x.Name);
```

Якщо сховище повертає *null*, наступний виклик *Map* отримує *Nothing* і не виконує функцію формування привітання. Якщо в сховищі виникає виняток, він перетворюється на *Error* і передається до кінця ланцюжка. Лямбда-вираз, що формує привітання, виконується лише тоді, коли

попередній крок повернув реальне значення і не згенерував винятку. Отже, функція з лістингу 7.5 функціонально еквівалентна функції з лістингу 7.4, але не містить захисного коду.

Важливо також, що функція повертає не рядок, а *Maybe<string>*. Тип результату повідомляє коду, що викликає функцію, про можливу невдачу і зобов'язує його розглянути всі варіанти (лістинг 7.6). Рішення щодо реакції на кожен варіант повністю належить коду-споживачеві.

Лістинг 7.6 - Обробка результату Maybe

```
1. var message = greeter.MakeGreeting(id) switch
2. {
3.     Maybe<string>.Something s => s.Value,
4.     Maybe<string>.Nothing => "Працівника не знайдено",
5.     Maybe<string>.Error e =>
6.         "Сталася помилка: " + e.CapturedError.Message,
7.     _ => throw new InvalidOperationException()
8. };
```

Для кожного ідентифікатора (наявного, відсутнього та такого, що спричиняє збій) процедурна й функціональна версії дають однакові результати:

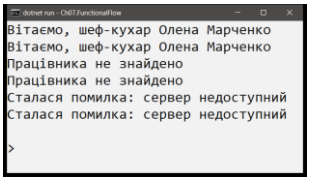


Рисунок 7.1 - Результат виконання прикладу «Maybe і привітання»

7.1.2. Залізничне програмування

Найвдаліший опис такого стилю запропонував С. Влащин (S. Wlaschin) під назвою **залізничне програмування** (railway oriented programming). Обчислення порівнюється із залізничною колією зі стрілками: потяг рухається «успішною» колією *Something*, і на кожній стрілці (виклику *Map* чи *Bind*) або продовжує рух нею, або переводиться на колію *Nothing* чи *Error*, якою без виконання будь-якої подальшої роботи доїжджає до кінцевої станції (рис. 7.2).

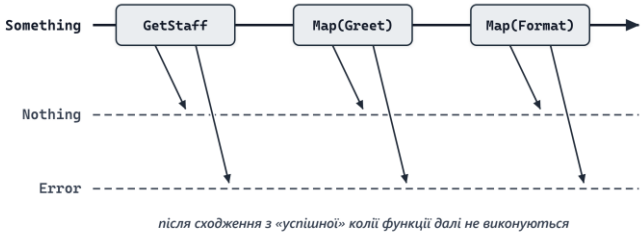


Рисунок 7.2 - Залізнична модель обробки результатів

Структура, що загортає значення та надає функцію зв'язування для безпечного виконання послідовних операцій, називається **монадою** (monad). Монаду можна уявити як контейнер, що містить значення, але не розкриває, яке саме, і дає змогу «підвішувати» до нього функції, виконання яких відбувається в безпечному середовищі - без ризику винятків *NullReferenceException* чи необроблених помилок. Функція зв'язування подібна до естафети: кожен виклик виконує операцію і передає результат наступному учасникові.

7.1.3. Налагодження ланцюжків

Ланцюжки викликів ускладнюють покрокове налагодження: у середовищі розробки доводиться послідовно заходити у вкладені лямбда-вирази. Тому під час налагодження зручно зберігати результат кожного кроку в окремій змінній (лістинг 7.7).

Лістинг 7.7 - Покроковий запис ланцюжка Maybe

```
1. public Maybe<string> MakeGreetingSteps(int staffId)
2. {
3.     var idMaybe = staffId.ToMaybe();
4.     var staffMaybe = idMaybe.Map(x => repo.GetById(x));
5.     var greeting = staffMaybe.Map(
6.         x => "Вітаємо, " + x.Position + " " + x.Name);
7.     return greeting;
8. }
```

Обидва варіанти функціонально тотожні. Покроковий варіант дає змогу переглянути кожне проміжне значення; завдяки незмінності змінних усі проміжні результати зберігаються в незмінному вигляді. Проте такі змінні залишаються в області видимості до кінця функції, тож для великих проміжних об'єктів доцільніше використовувати ланцюжок. Вибір між варіантами є переважно питанням стилю.

7.2. Відмінність між Map і Bind

Строго кажучи, для монади визначаються дві функції з різними сигнатурами:

- **`Map`** приймає функцію, що перетворює значення типу *T1* на значення типу *T2* (з даних у дані);
- **`Bind`** (також *flatMap*, *SelectMany*, >>=) приймає функцію, що перетворює значення типу *T1* на *Maybe<T2>* (з даних у монаду), і повертає результат цієї функції без додаткового загортання.

Відмінність добре видно, якщо уявити *Maybe* контейнером, а функції - перетвореннями вмісту (рис. 7.3). *Map* відкриває контейнер, застосовує функцію до вмісту й загортає результат у новий контейнер. Якщо функція сама повертає контейнер, *Map* загорне його вдруге, утворивши *Maybe<Maybe<T2>>*; *Bind* натомість повертає контейнер, створений функцією, без додаткового загортання.

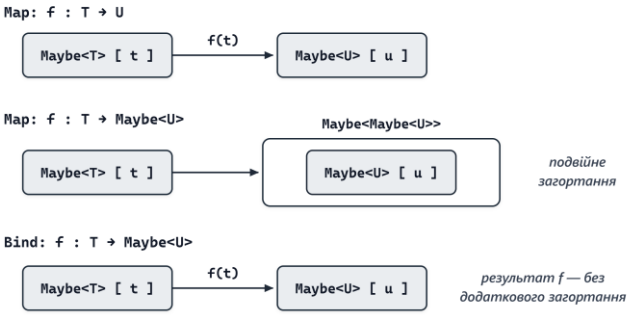


Рисунок 7.3 - Функції Map і Bind для контейнера Maybe

Реалізацію *Bind* наведено в лістингу 7.8.

Лістинг 7.8 - Функція Bind для Maybe

```
1. // Bind: функція перетворює значення на нове Maybe
2. public static Maybe<TOut> Bind<TIn, TOut>(this Maybe<TIn> @this,
3.                                           Func<TIn, Maybe<TOut>> f)
4. {
5.     try
6.     {
7.         return @this switch
8.         {
9.             Maybe<TIn>.Something s => f(s.Value),
10.              Maybe<TIn>.Nothing n => new
     Maybe<TOut>.Nothing(n.Handled),
11.              Maybe<TIn>.Error e =>
12.                  new Maybe<TOut>.Error(e.CapturedError, e.Handled),
13.              _ => new Maybe<TOut>.Error(
14.                  new InvalidOperationException("невідомий стан Maybe"))
15.          };
```

```
16.      }
17.      catch (Exception ex)
18.      {
19.          return new Maybe<TOut>.Error(ex);
20.      }
21.  }
```

Якщо сховище даних саме повертає *Maybe* (наприклад, *Maybe<Staff>*), його виклик виконується через *Bind*, а подальші перетворення даних - через *Map* (лістинг 7.9).

Лістинг 7.9 - Поєднання Bind і Map

```
1. public Maybe<string> DescribeStaff(int staffId) =>
2.     staffId.ToMaybe()
3.         .Bind(x => repo.FindById(x))       // int -> Maybe<Staff>
4.         .Map(x => $"{x.Name} ({x.Position})"); // Staff -> string
```

Для перетворення *Maybe* на конкретне значення в кінці ланцюжка зручно використовувати функцію *Match* (лістинги 7.10 і 7.11), яка приймає по одній функції для кожного варіанта.

Лістинг 7.10 - Функція Match

```
1. public static TOut Match<T, TOut>(this Maybe<T> @this,
2.                                    Func<T, TOut> something,
3.                                    Func<TOut> nothing,
4.                                    Func<Exception, TOut> error) =>
5.     @this switch
6.     {
7.         Maybe<T>.Something s => something(s.Value),
8.         Maybe<T>.Nothing => nothing(),
9.         Maybe<T>.Error e => error(e.CapturedError),
10.          _ => throw new InvalidOperationException()
11.      };
```

Лістинг 7.11 - Використання функції Match

```
1. var text = greeter.DescribeStaff(id).Match(
2.     something: x => x,
3.     nothing: () => "не знайдено",
4.     error: e => "помилка: " + e.Message);
```

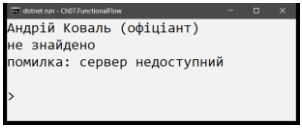


Рисунок 7.4 - Результат виконання прикладу «Map і Bind» (фрагмент)

Слід зазначити, що назва *Map* у мові C# може викликати асоціації з методом *Select* для колекцій чи з бібліотекою AutoMapper. Тому в деяких кодових базах для обох операцій використовують одну назву *Bind*. Проте розрізнення *Map* і *Bind* відповідає усталеній термінології функціонального програмування, і в посібнику воно зберігається.

7.3. Maybe і примітивні типи

Поширеною помилкою реалізації *Maybe* є визначення відсутності значення через порівняння зі значенням за замовчуванням для типу (*default*). Така «наївна» реалізація наведена в лістингу 7.12.

Лістинг 7.12 - Наївна реалізація Map з порівнянням з default

```
1. // Наївна реалізація: «значення відсутнє», якщо воно
2. // дорівнює значенню за замовчуванням для типу
3. public static Maybe<TOut> NaiveMap<TIn, TOut>(this Maybe<TIn> @this,
4.                                                Func<TIn, TOut> f) =>
5.     @this switch
6.     {
7.         Maybe<TIn>.Something s when !EqualityComparer<TIn>
8.                 .Default.Equals(s.Value, default) =>
9.             new Maybe<TOut>.Something(f(s.Value)),
10.          Maybe<TIn>.Something => new Maybe<TOut>.Nothing(),
11.          Maybe<TIn>.Nothing => new Maybe<TOut>.Nothing(),
12.          Maybe<TIn>.Error e => new Maybe<TOut>.Error(e.CapturedError),
13.          _ => throw new InvalidOperationException()
14.      };
```

Для посилальних типів значенням за замовчуванням є *null*, і реалізація працює коректно. Проте для типів-значень (*int*, *decimal*, *bool* тощо) значенням за замовчуванням є 0 або *false*, які є цілком коректними значеннями (лістинг 7.13).

Лістинг 7.13 - Обробка нульового значення

```
1. // Порожній кошик гостя: 0 позицій - коректне значення
2. var itemsInCart = new Maybe<int>.Something(0);
3.
4. var naive = itemsInCart.NaiveMap(x => x + 1);
5. var correct = itemsInCart.Map(x => x + 1);
```

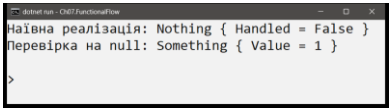


Рисунок 7.5 - Результат виконання прикладу «Примітивні типи»

Наївна реалізація помилково вважає нуль позицій у кошику відсутністю значення. Реалізація з лістингу 7.3 перевіряє саме *null* (оператор *is null*), а для типів-значень така перевірка завжди хибна, тож проблема не виникає. Якщо для певного типу-значення відсутність потрібно позначати явно, доцільно використовувати тип із підтримкою *null* (*int?*, *decimal?*).

7.4. Maybe і журналювання

У професійній розробці важливо журналювати не лише помилки, а й інформаційні події та попередження. Аналізувати варіант *Maybe* інструкціями *if* після кожного кроку незручно: код швидко розростається. Доцільніше доповнити *Maybe* методами-обробниками подій (лістинг 7.14).

Лістинг 7.14 - Методи-обробники OnSomething, OnNothing, OnError

```
1. public static Maybe<T> OnSomething<T>(this Maybe<T> @this,
2.                                       Action<T> action)
3. {
4.     if (@this is Maybe<T>.Something s)
5.         action(s.Value);
6.     return @this;
7. }
8.
9. public static Maybe<T> OnNothing<T>(this Maybe<T> @this, Action action)
10.  {
11.      if (@this is Maybe<T>.Nothing { Handled: false })
12.      {
13.          action();
14.          return new Maybe<T>.Nothing(Handled: true);
15.      }
16.      return @this;
17.  }
18.
19.  public static Maybe<T> OnError<T>(this Maybe<T> @this,
20.                                     Action<Exception> action)
21.  {
22.      if (@this is Maybe<T>.Error { Handled: false } e)
23.      {
24.          action(e.CapturedError);
25.          return e with { Handled = true };
26.      }
27.      return @this;
28.  }
```

Варіанти *Nothing* і *Error* передаються вздовж ланцюжка без змін, тож без додаткових заходів кожен обробник *OnNothing* у довгому ланцюжку спрацьовував би повторно. Для запобігання цьому використовується ознака *Handled*: обробник виконує дію лише для необробленого стану і повертає той самий стан із позначкою «оброблено». У лістингу 7.15 у ланцюжку три обробники *OnNothing*, але спрацьовує лише перший.

Лістинг 7.15 - Журналювання в ланцюжку Maybe

```
1. var report = 9.ToMaybe()
2.     .Bind(x => repo.FindById(x))
3.     .OnSomething(x => Console.WriteLine($"[інфо] знайдено {x.Name}"))
4.     .OnNothing(() => Console.WriteLine("[увага] працівника 9 немає"))
5.     .OnError(e => Console.WriteLine($"[помилка] {e.Message}"))
6.     .Map(x => x.Name)
7.     .OnNothing(() => Console.WriteLine("[увага] крок 2: немає"))
8.     .Map(x => x.ToUpper())
9.     .OnNothing(() => Console.WriteLine("[увага] крок 3: немає"));
```

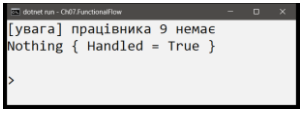


Рисунок 7.6 - Результат виконання прикладу «Журналювання»

7.5. Maybe та асинхронні операції

Звернення до зовнішніх систем у сучасних застосунках, як правило, асинхронні. Для них доцільно реалізувати асинхронні варіанти функцій зв'язування (лістинг 7.16). Значення при цьому має два рівні обгортання: *Task* позначає, що операцію ще потрібно дочекатися, а *Maybe* - що вона може завершитися невдачею.

Лістинг 7.16 - Асинхронні функції BindAsync і MapAsync

```
1. public static async Task<Maybe<TOut>> BindAsync<TIn, TOut>(
2.     this Maybe<TIn> @this, Func<TIn, Task<Maybe<TOut>>> f)
3. {
4.     try
5.     {
6.         return @this switch
7.         {
8.             Maybe<TIn>.Something s => await f(s.Value),
9.             Maybe<TIn>.Nothing n => new Maybe<TOut>.Nothing(n.Handled),
10.              Maybe<TIn>.Error e =>
11.                  new Maybe<TOut>.Error(e.CapturedError, e.Handled),
12.              _ => new Maybe<TOut>.Error(
13.                  new InvalidOperationException("невідомий стан Maybe"))
14.          };
15.     }
16.     catch (Exception ex)
17.     {
18.         return new Maybe<TOut>.Error(ex);
19.     }
20. }
21.
22. public static async Task<Maybe<TOut>> MapAsync<TIn, TOut>(
23.     this Maybe<TIn> @this, Func<TIn, Task<TOut?>> f) =>
24.     await @this.BindAsync(async x => (await f(x)).ToMaybe());
```

Асинхронні та синхронні виклики в одному ланцюжку зручно розділяти: кожен асинхронний крок завершується оператором *await*, після чого отримане *Maybe* використовується в наступних кроках (лістинг 7.17).

Лістинг 7.17 - Використання MapAsync

```
1. public async Task<Maybe<string>> MakeGreetingAsync(int staffId)
2. {
3.     var staff = await staffId.ToMaybe()
4.         .MapAsync(x => repo.GetByIdAsync(x));
5.     var greeting = staff.Map(x => "Вітаємо, " + x.Name);
6.     return greeting;
7. }
```

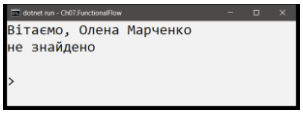


Рисунок 7.7 - Результат виконання прикладу «Async»

7.6. Вкладені Maybe

Якщо функції доступу до даних повертають *Maybe*, послідовне застосування до них *Map* призводить до вкладених типів *Maybe<Maybe<T>>*. Нехай три сервіси послідовно повертають код закладу, керівника закладу та номер телефону керівника, і кожен результат загорнуто в *Maybe* (лістинг 7.18).

Лістинг 7.18 - Вкладені Maybe і їх «розгортання»

```
1. // Map із функцією, що повертає Maybe, дає вкладений тип
2. Maybe<Maybe<string>> nested = one.GetVenueCode()
3.     .Map(x => two.GetManager(x));
4.
5. // Bind «розгортає» вкладеність
6. Maybe<string> phone = one.GetVenueCode()
7.     .Bind(x => two.GetManager(x))
8.     .Bind(x => three.GetPhone(x));
9.
10.  Maybe<string> flattened = nested.Flatten();
```

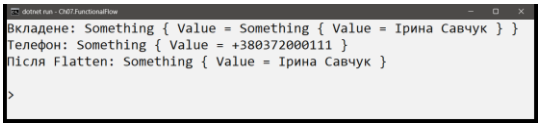


Рисунок 7.8 - Результат виконання прикладу «Вкладені Maybe»

Функція *Bind* саме й розв'язує цю проблему: вона приймає функцію, що повертає *Maybe*, і не загортає результат удруге. Для вже отриманого вкладеного значення можна застосувати функцію *Flatten* (лістинг 7.19), реалізовану через *Bind* з тотожною функцією.

Лістинг 7.19 - Функція Flatten

```
1. public static Maybe<T> Flatten<T>(this Maybe<Maybe<T>> @this) =>
2.     @this.Bind(inner => inner);
```

Аналогом цієї операції для колекцій є метод *SelectMany*, що перетворює послідовність послідовностей на одну послідовність. Можливість «вирівнювання» вкладених структур пов'язана із законами монад, які розглядаються далі.

7.7. Функтори, аплікативні функтори та монади

Поняття функтора, аплікативного функтора й монади походять з теорії категорій - розділу математики, що вивчає структури та відображення між ними. Для практичного програмування достатньо їх операційного розуміння: кожне з понять описує **контекст обчислення** (можливу відсутність значення, помилку, множинність, асинхронність, стан) і набір операцій, що дають змогу працювати зі значенням у цьому контексті, не виймаючи його. Кожне поняття визначається операціями та **законами** - рівностями, які ці операції мусять задовольняти. Закони гарантують, що

структура поводиться передбачувано й що ланцюжки операцій можна перегруповувати та спрощувати, як алгебраїчні вирази.

7.7.1. Закони монад

Структура вважається монадою, якщо вона має операцію загортання значення (*return*, або *unit*; для *Maybe* - *ToMaybe*) та операцію зв'язування *Bind*, які задовольняють три закони. Нехай *f* і *g* - функції вигляду *T -> Maybe<U>*, *a* - значення, *m* - монадичне значення.

1) **Ліва тотожність** (left identity): загортання значення з подальшим зв'язуванням з функцією *f* еквівалентне безпосередньому виклику *f*: *return(a).Bind(f) == f(a)*.

2) **Права тотожність** (right identity): зв'язування з операцією загортання не змінює монадичного значення: *m.Bind(return) == m*.

3) **Асоціативність** (associativity): порядок групування послідовних зв'язувань не впливає на результат: *m.Bind(f).Bind(g) == m.Bind(x => f(x).Bind(g))*.

Перші два закони означають, що монада не втручається ні в дані, ні у виконання переданих функцій: вона лише «проводить» їх. Третій закон гарантує, що вкладені зв'язування завжди дають той самий результат, що й послідовні. Оскільки *Maybe* реалізовано на основі записів із рівністю за значенням, закони можна перевірити безпосередньо (лістинг 7.20).

Лістинг 7.20 - Перевірка законів монад для Maybe

```
1. // Функції вигляду T -> Maybe<U>
2. Func<int, Maybe<int>> doublePrice = x => (x * 2).ToMaybe();
3. Func<int, Maybe<string>> label = x => $"{x} грн".ToMaybe();
4. Func<int, Maybe<int>> unit = x => x.ToMaybe();   // return
5.
6. var price = 150;
7. var m = price.ToMaybe();
8.
9. // 1. Ліва тотожність: return(a).Bind(f) == f(a)
10.  var leftIdentity = m.Bind(doublePrice) == doublePrice(price);
11.
12.  // 2. Права тотожність: m.Bind(return) == m
13.  var rightIdentity = m.Bind(unit) == m;
14.
15.  // 3. Асоціативність:
16.  //    m.Bind(f).Bind(g) == m.Bind(x => f(x).Bind(g))
17.  var associativity =
18.      m.Bind(doublePrice).Bind(label) ==
19.      m.Bind(x => doublePrice(x).Bind(label));
```

```
dotnet run - Ch07.FunctionalFlow
Ліва тотожність: True
Права тотожність: True
Асоціативність: True

>
```

Рисунок 7.9 - Результат виконання прикладу «Закони монад»

7.7.2. Функтор

Функтор (functor) - найпростіша з трьох структур. Це узагальнений тип *F<T>* з операцією *Map*: (T -> U) -> (F<T> -> F<U>), тобто операцією, що «піднімає» звичайну функцію до функції над контекстом. Функтор мусить задовольняти два закони:

1) **тотожність**: *m.Map(x => x) == m* - відображення тотожною функцією не змінює значення;

2) **композиція**: *m.Map(f).Map(g) == m.Map(x => g(f(x)))* - два послідовні відображення рівносильні одному відображенню композицією функцій.

Закони означають, що *Map* змінює лише вміст контейнера, але не його «форму»: *Nothing* залишається *Nothing*, послідовність зберігає кількість і порядок елементів. Функторами є *Maybe* з функцією *Map*, *IEnumerable<T>* з методом *Select*, *Task<T>* з продовженням *ContinueWith*. Перевірку законів наведено в лістингу 7.21.

Лістинг 7.21 - Перевірка законів функтора для Maybe і IEnumerable

```
1. Func<int, int> withVat = x => x * 6 / 5;
2. Func<int, string> label = x => $"{x} грн";
3. var m = 150.ToMaybe();
4. int[] menu = [150, 120, 45];
5.
6. // 1. Тотожність: m.Map(x => x) == m
7. Console.WriteLine(m.Map(x => x) == m);
8. Console.WriteLine(menu.Select(x => x).SequenceEqual(menu));
9.
10.  // 2. Композиція: m.Map(f).Map(g) == m.Map(x => g(f(x)))
11.  Console.WriteLine(m.Map(withVat).Map(label) ==
12.                    m.Map(x => label(withVat(x))));
13.  Console.WriteLine(menu.Select(withVat).Select(label)
14.       .SequenceEqual(menu.Select(x => label(withVat(x)))));
```



Рисунок 7.10 - Результат виконання прикладу «Закони функтора»

Закон композиції має практичне значення: ланцюжок *Select(f).Select(g)* можна замінити одним *Select* з композицією функцій без зміни результату, що використовують оптимізатори запитів.

7.7.3. Аплікативний функтор

Функтор дає змогу застосувати до значення в контексті функцію **одного** аргументу. Якщо ж потрібно поєднати **кілька** значень, кожне з яких перебуває у власному контексті, - наприклад, ім'я, телефон і кількість гостей, кожне з яких може бути некоректним, - *Map* недостатньо.

Аплікативний функтор (applicative functor) доповнює функтор операціями загортання значення (*Pure*, аналог *return*) та застосування *Apply*, що поєднує незалежні значення в контексті за допомогою функції кількох аргументів.

Відмінність аплікативного стилю від монадичного найкраще видно на прикладі перевірки введених даних. Монадичне зв'язування *Bind* є **послідовним**: наступний крок залежить від результату попереднього й виконується лише після його успіху, тому обчислення зупиняється на першій помилці. Аплікативне поєднання застосовується до **незалежних** обчислень: усі вони виконуються, а результати (зокрема помилки) поєднуються. Для перевірки форми бронювання столика це означає, що відвідувач отримує одразу повний перелік помилок, а не виправляє їх по одній.

Тип *Validation<T>* (лістинг 7.22) подібний до *Result*, але у варіанті *Invalid* містить перелік помилок. Операція *Apply* (лістинг 7.23) у разі невдачі обох аргументів поєднує їхні переліки помилок; оскільки поєднання переліків є моноїдом (підрозділ 5.12.2), результат не залежить від того, як згруповано перевірки.

Лістинг 7.22 - Тип Validation

```
// Результат перевірки: значення або непорожній перелік помилок
public abstract record Validation<T>
{
    private Validation() { }
    public sealed record Valid(T Value) : Validation<T>;
    public sealed record Invalid(ImmutableList<string> Errors)
        : Validation<T>;
}
```

Лістинг 7.23 - Аплікативне поєднання перевірок

```
// Аплікативне поєднання двох незалежних перевірок:
// помилки обох накопичуються, а не обривають обчислення
public static Validation<TOut> Apply<A, B, TOut>(
    this Validation<A> va, Validation<B> vb, Func<A, B, TOut> f) =>
    (va, vb) switch
    {
        (Validation<A>.Valid a, Validation<B>.Valid b) =>
            Ok(f(a.Value, b.Value)),
        (Validation<A>.Invalid a, Validation<B>.Invalid b) =>
              new Validation<TOut>.Invalid(a.Errors.AddRange(b.Errors)),
          (Validation<A>.Invalid a, _) =>
              new Validation<TOut>.Invalid(a.Errors),
          (_, Validation<B>.Invalid b) =>
              new Validation<TOut>.Invalid(b.Errors),
          _ => throw new InvalidOperationException()
      };
```

Правила перевірки окремих полів є незалежними чистими функціями (лістинг 7.24). У лістингу 7.25 з них побудовано дві функції перевірки форми - монадичну та аплікативну.

Лістинг 7.24 - Правила перевірки полів форми бронювання

```
1. public static Validation<string> CheckName(string s) =>
2.     s.Trim().Length >= 2
3.         ? ValidationExtensions.Ok(s.Trim())
4.         : ValidationExtensions.Fail<string>("ім'я закоротке");
5.
6. public static Validation<string> CheckPhone(string s) =>
7.     s.StartsWith("+380") && s.Length == 13
8.         ? ValidationExtensions.Ok(s)
9.         : ValidationExtensions.Fail<string>("невірний телефон");
10.
11.  public static Validation<int> CheckGuests(int n) =>
12.      n is >= 1 and <= 12
13.          ? ValidationExtensions.Ok(n)
14.          : ValidationExtensions.Fail<int>("гостей має бути 1-12");
```

Лістинг 7.25 - Монадична та аплікативна перевірка форми

```
1. // Монадичний стиль: зупинка на першій помилці
2. public static Validation<TableRequest> Sequential(string name,
3.     string phone, int guests) =>
4.     CheckName(name).Bind(n =>
5.     CheckPhone(phone).Bind(p =>
6.     CheckGuests(guests).Bind(g =>
7.         ValidationExtensions.Ok(new TableRequest(n, p, g)))));
8.
9. // Аплікативний стиль: усі перевірки незалежні, помилки разом
10.  public static Validation<TableRequest> Parallel(string name,
11.      string phone, int guests) =>
12.      CheckName(name)
13.          .Apply(CheckPhone(phone), (n, p) => (n, p))
14.          .Apply(CheckGuests(guests),
15.                 (np, g) => new TableRequest(np.n, np.p, g));
```

Лістинг 7.26 - Перевірка коректних і некоректних даних

```
1. Console.WriteLine(Parallel("Олена", "+380501234567", 4).Show());
2. Console.WriteLine(Sequential("О", "050", 20).Show());
3. Console.WriteLine(Parallel("О", "050", 20).Show());
```

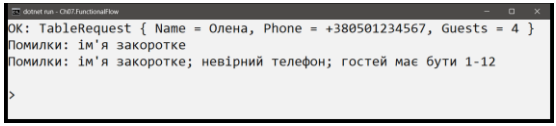


Рисунок 7.11 - Результат виконання прикладу «Аплікативна перевірка»

Для некоректних даних монадична перевірка повідомила лише про першу помилку, а аплікативна - про всі три.

7.7.4. Співвідношення понять

Три структури утворюють ієрархію (рис. 7.12): кожен аплікативний функтор є функтором, а кожна монада - аплікативним функтором, оскільки *Map* і *Apply* можна виразити через *Bind* і *return*. Зворотне не

виконується: *Validation* є аплікативним функтором, але її монадичне зв'язування втрачає властивість накопичення помилок. Узагальнену характеристику понять наведено в таблиці 7.1.



Рисунок 7.12 - Ієрархія функтор - аплікативний функтор - монада

Таблиця 7.1 - Функтор, аплікативний функтор і монада

Поняття	**Основна операція**	**Що дає змогу виразити**	**Приклади в .NET і посібнику**
Функтор	*Map*: (T -> U) -> F<T> -> F<U>	Перетворення значення в контексті	*Select* для *IEnumerable<T>*, *Map* для *Maybe*
Аплікативний функтор	*Apply*: F<T>, F<U>, (T, U -> V) -> F<V>	Поєднання незалежних значень у контексті	*Zip*, *Task.WhenAll*, *Validation*
Монада	*Bind*: F<T>, (T -> F<U>) -> F<U>	Послідовність залежних кроків	*SelectMany*, *await*, *Bind* для *Maybe*, *Reader*, *State*

Вибір найслабшої достатньої структури є корисним принципом проєктування: якщо кроки незалежні, аплікативний стиль дає змогу виконувати їх паралельно (як *Task.WhenAll*) і збирати всі помилки; монадичне зв'язування потрібне лише тоді, коли наступний крок залежить від результату попереднього.

7.7.5. Синтаксис запитів LINQ як монадична нотація

Вкладені виклики *Bind* з лямбда-виразами, кожен з яких використовує результати попередніх кроків, погано читаються. Мова Haskell для цього має спеціальну **do-нотацію**. У мові C# її роль відіграє **синтаксис запитів**

LINQ (query syntax). Компілятор перетворює запит на виклики методів за фіксованими правилами, не перевіряючи, що джерело є колекцією: достатньо, щоб для типу існували методи з відповідними назвами та сигнатурами. Перше речення *from* з подальшим *select* перетворюється на виклик *Select* (тобто *Map*), а кожне наступне *from* - на виклик *SelectMany* (тобто *Bind*) з функцією проекції, що зберігає раніше отримані змінні.

Отже, якщо для монади визначити методи *Select* і *SelectMany* (лістинг 7.27), ланцюжок залежних кроків можна записати як запит (лістинг 7.28). У лістингу наведено й еквівалентний ланцюжок викликів, на який (у спрощеному вигляді) перетворює запит компілятор.

Лістинг 7.27 - Підтримка синтаксису запитів LINQ для Maybe

```
1. // Підтримка синтаксису запитів LINQ (from ... select)
2. public static Maybe<TOut> Select<TIn, TOut>(this Maybe<TIn> @this,
3.                                             Func<TIn, TOut?> f) =>
4.     @this.Map(f);
5.
6. public static Maybe<TOut> SelectMany<TIn, TMid, TOut>(
7.     this Maybe<TIn> @this,
8.     Func<TIn, Maybe<TMid>> bind,
9.     Func<TIn, TMid, TOut?> project) =>
10.      @this.Bind(x => bind(x).Map(y => project(x, y)));
```

Лістинг 7.28 - Ланцюжок пошуку телефону керівника як запит LINQ

```
1. // Синтаксис запитів: кожне from - крок зв'язування (Bind)
2. var call =
3.     from venue in one.GetVenueCode()
4.     from manager in two.GetManager(venue)
5.     from phone in three.GetPhone(manager)
6.     select $"{manager} ({venue}): {phone}";
7.
8. // Те саме після перетворення компілятором (спрощено)
9. var sameCall = one.GetVenueCode()
10.      .SelectMany(venue => two.GetManager(venue),
11.                  (venue, manager) => (venue, manager))
12.      .SelectMany(t => three.GetPhone(t.manager),
13.          (t, phone) => $"{t.manager} ({t.venue}): {phone}");
14.
15.  Console.WriteLine(call);
16.  Console.WriteLine(call == sameCall);
```

Якщо будь-який крок повертає *Nothing* чи *Error*, решта кроків не виконується (лістинг 7.29).

Лістинг 7.29 - Запит із відсутнім значенням

```
1. var unknown =
2.     from manager in two.GetManager("LV-07")
3.     from phone in three.GetPhone(manager)
4.     select phone;
5. Console.WriteLine(unknown);
```

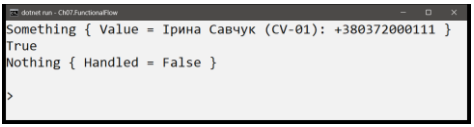


Рисунок 7.13 - Результат виконання прикладу «Запит LINQ для Maybe»

Порівняно з ланцюжком *Bind* запит має дві переваги: змінні всіх попередніх кроків (*venue*, *manager*) доступні в наступних без вкладення лямбда-виразів, а сам запис читається як послідовність кроків. Синтаксис запитів однаково застосовний до колекцій, *Maybe*, *State*, а також до типів-об'єднань C# 15 (підрозділ 6.7.5).

7.8. Монада Reader

Нехай потрібно розрахувати підсумкову суму рахунку: додати ПДВ, сервісний збір і округлити результат. Параметри розрахунку (ставки, правило округлення) залежать від закладу і задаються його конфігурацією (лістинг 7.30).

Лістинг 7.30 - Конфігурація закладу

```
// Конфігурація закладу - «середовище» обчислень
public sealed record VenueConfig(string Venue, decimal VatRate,
                                 decimal ServiceRate, bool RoundUp);
```

Щоб скористатися ланцюжком *Map*, конфігурацію доводиться передавати разом зі значенням через кожен крок у вигляді кортежу (лістинг 7.31). Такий код працює, але повторюване передавання конфігурації погіршує його читабельність.

Лістинг 7.31 - Передавання конфігурації в кортежах

```
// Конфігурацію доводиться передавати через кожен крок
public static string FinalBill(decimal net, VenueConfig config) =>
    (Config: config, Amount: net)
        .Map(x => (x.Config, Amount: x.Amount * (1 + x.Config.VatRate)))
        .Map(x => (x.Config,
                   Amount: x.Amount * (1 + x.Config.ServiceRate)))
        .Map(x => (x.Config, Amount: x.Config.RoundUp
                                   ? Math.Ceiling(x.Amount)
                                   : Math.Round(x.Amount, 2)))
         .Map(x => $"{x.Config.Venue}: {x.Amount} грн");
```

Ці проблеми розв'язує **монада `Reader`** - функціональний аналог впровадження залежностей (dependency injection) на рівні функції, а не класу. *Reader* зберігає функцію від «середовища» (environment), але не виконує її доти, доки середовище не буде надано викликом *Run* (лістинг 7.32). Функція *Ask* повертає саме середовище, що дає змогу звертатися до нього в ланцюжку.

Лістинг 7.32 - Реалізація монади Reader

```
/// <summary>
```

```
/// Reader: обчислення, що потребує «середовища» типу TEnv.
/// Середовище надається лише під час виклику Run.
/// </summary>
public sealed class Reader<TEnv, T>(Func<TEnv, T> run)
{
    public T Run(TEnv environment) => run(environment);

    public Reader<TEnv, TOut> Map<TOut>(Func<T, TOut> f) =>
        new(env => f(run(env)));

    public Reader<TEnv, TOut> Bind<TOut>(
        Func<T, Reader<TEnv, TOut>> f) =>
        new(env => f(run(env)).Run(env));
}

public static class Reader
{
    // Обчислення, що просто повертає середовище
    public static Reader<TEnv, TEnv> Ask<TEnv>() => new(env => env);
}
```

Найпростіші приклади використання наведено в лістингу 7.33.

Лістинг 7.33 - Найпростіше використання Reader

```
var reader = new Reader<int, string>(e => $"гостей: {e}");
var result = reader.Run(4);   // "гостей: 4"

var chained = new Reader<int, int>(e => e * 450)
    .Map(x => x / 2)
    .Map(x => $"{x} грн");
var result2 = chained.Run(4); // "900 грн"
```



Рисунок 7.14 - Результат виконання прикладу «Reader: прості приклади»

Кожен виклик *Map* або *Bind* «огортає» попередню функцію, не виконуючи її; обчислення відбувається лише під час виклику *Run*. Окремі кроки розрахунку рахунку подаються функціями, що повертають *Reader* (лістинг 7.34), а підсумковий розрахунок - їх зв'язуванням (лістинг 7.35).

Лістинг 7.34 - Кроки розрахунку у вигляді Reader

```
public static Reader<VenueConfig, decimal> AddVat(decimal net) =>
    new(cfg => net * (1 + cfg.VatRate));

public static Reader<VenueConfig, decimal> AddService(decimal amount) =>
    new(cfg => amount * (1 + cfg.ServiceRate));

public static Reader<VenueConfig, decimal> Round(decimal amount) =>
    new(cfg => cfg.RoundUp ? Math.Ceiling(amount)
                           : Math.Round(amount, 2));
```

Лістинг 7.35 - Зв'язування кроків розрахунку

```
// Функція не залежить від конкретної конфігурації:
// вона повертає опис обчислення
public static Reader<VenueConfig, string> FinalBill(decimal net) =>
    AddVat(net)
        .Bind(AddService)
        .Bind(Round)
        .Bind(total => Reader.Ask<VenueConfig>()
            .Map(cfg => $"{cfg.Venue}: {total} грн"));
```

Функція *FinalBill* є чистою: вона не залежить від конкретної конфігурації і повертає лише опис обчислення. Конфігурація надається в момент виконання, тож один і той самий опис можна виконати для різних закладів (лістинг 7.36), а в модульних тестах - для тестової конфігурації.

Лістинг 7.36 - Виконання розрахунку для різних закладів

```
var restaurant = new VenueConfig("Ресторан", 0.20m, 0.10m, false);
var cafe = new VenueConfig("Кав'ярня", 0.20m, 0m, true);

var bill = FinalBill(1234.50m);   // обчислення ще не виконано

Console.WriteLine(bill.Run(restaurant));
Console.WriteLine(bill.Run(cafe));
```

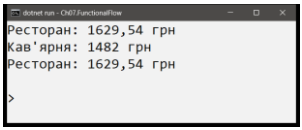


Рисунок 7.15 - Результат виконання прикладу «Reader»

Монаду *Reader* можна уявити як контейнер, який відкривається лише ключем відповідного типу. За потреби функції, що повертають *Reader*, можна виносити в інтерфейси сервісів; середовищем може бути не лише конфігурація, а й, наприклад, з'єднання з базою даних.

7.9. Монада State

Монада **`State`** подібна до *Reader*, проте середовище (стан) не лише читається, а й змінюється: кожен крок обчислення отримує поточний стан і повертає результат разом із новим станом. Формально обчислення зі станом - це функція *S -> (T, S)*. Реалізацію наведено в лістингу 7.37: функція *Bind* передає новий стан, отриманий на попередньому кроці, наступному кроку, а допоміжні функції *Return*, *Get*, *Put* і *Modify* відповідно повертають значення без зміни стану, читають стан, замінюють його та змінюють функцією.

Лістинг 7.37 - Реалізація монади State

```
/// <summary>
/// State: обчислення, що отримує стан типу TState і повертає
/// результат разом із новим станом.
```

```
4. /// </summary>
5. public sealed class State<TState, T>(
6.     Func<TState, (T Value, TState State)> run)
7. {
8.     public (T Value, TState State) Run(TState initial) => run(initial);
9.
10.      public State<TState, TOut> Map<TOut>(Func<T, TOut> f) =>
11.          new(s =>
12.          {
13.              var (value, next) = run(s);
14.              return (f(value), next);
15.          });
16.
17.      public State<TState, TOut> Bind<TOut>(
18.          Func<T, State<TState, TOut>> f) =>
19.          new(s =>
20.          {
21.              var (value, next) = run(s);
22.              return f(value).Run(next);
23.          });
24.  }
25.
26.  public static class State
27.  {
28.      public static State<TS, T> Return<TS, T>(T value) =>
29.          new(s => (value, s));
30.
31.      public static State<TS, TS> Get<TS>() => new(s => (s, s));
32.
33.      public static State<TS, bool> Put<TS>(TS newState) =>
34.          new(_ => (true, newState));
35.
36.      public static State<TS, bool> Modify<TS>(Func<TS, TS> f) =>
37.          new(s => (true, f(s)));
38.  }
```

Нехай моделюється облік складу кухні. Стан - незмінний словник *ImmutableDictionary* «інгредієнт - залишок у кілограмах». Операція списання повертає ознаку успішності та новий стан; якщо залишку недостатньо, стан не змінюється. Операція поставки збільшує залишок (лістинг 7.38).

Лістинг 7.38 - Операції складу кухні у вигляді State

```
1. // Стан складу кухні: незмінний словник «інгредієнт - кг»
2. public static class Kitchen
3. {
4.     public static State<ImmutableDictionary<string, decimal>, bool>
5.         Consume(string item, decimal kg) =>
6.         new(stock => stock.GetValueOrDefault(item) >= kg
7.             ? (true, stock.SetItem(item, stock[item] - kg))
8.             : (false, stock));
9.
10.      public static State<ImmutableDictionary<string, decimal>, bool>
11.          Deliver(string item, decimal kg) =>
12.          new(stock => (true,
13.              stock.SetItem(item, stock.GetValueOrDefault(item) + kg)));
14.  }
```

Послідовність операцій зміни зв'язується функцією *Bind* (лістинг 7.39). Виконання починається лише під час виклику *Run* з початковим станом.

Лістинг 7.39 - Зв'язування операцій складу

```
var shift = Kitchen.Consume("борошно", 2m)
    .Bind(a => Kitchen.Consume("сир", 1.5m)
    .Bind(b => Kitchen.Deliver("сир", 5m)
    .Bind(c => Kitchen.Consume("сир", 3m)
    .Map(d => new[] { a, b, c, d })));

var (results, finalStock) = shift.Run(Initial);
```

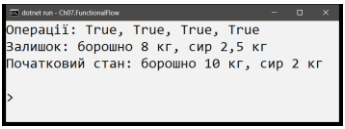


Рисунок 7.16 - Результат виконання прикладу «State: Bind»

Початковий словник залишається незмінним, а кожна операція створює новий стан. Вкладені виклики *Bind* погано читаються. Як і для *Maybe* (підрозділ 7.7.5), після реалізації методів *Select* і *SelectMany* (лістинг 7.40) операції складу можна записати в синтаксисі запитів LINQ (лістинг 7.41).

Лістинг 7.40 - Підтримка синтаксису запитів LINQ для State

```
public static class StateLinqExtensions
{
    public static State<TS, TOut> Select<TS, TIn, TOut>(
        this State<TS, TIn> @this, Func<TIn, TOut> f) => @this.Map(f);

    public static State<TS, TOut> SelectMany<TS, TIn, TMid, TOut>(
        this State<TS, TIn> @this,
        Func<TIn, State<TS, TMid>> bind,
        Func<TIn, TMid, TOut> project) =>
          @this.Bind(x => bind(x).Map(y => project(x, y)));
}
```

Лістинг 7.41 - Операції складу в синтаксисі запитів LINQ

```
var shift =
    from a in Kitchen.Consume("борошно", 2m)
    from b in Kitchen.Consume("сир", 3m)      // не вистачає
    from c in Kitchen.Deliver("сир", 5m)
    from stock in State.Get<ImmutableDictionary<string, decimal>>()
    select $"списання сиру: {b}, сиру на складі: {stock["сир"]} кг";

var (report, _) = shift.Run(Initial);
```

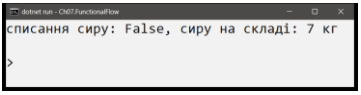


Рисунок 7.17 - Результат виконання прикладу «State: запит LINQ»

Монади можна поєднувати. Якщо значенням обчислення зі станом є *Maybe*, допоміжна функція *BindMaybe* дає змогу припиняти подальші операції після першої невдачі, зберігаючи стан на момент невдачі (лістинги 7.42 і 7.43).

Лістинг 7.42 - Поєднання State і Maybe

```
1. // Поєднання State і Maybe: після першої невдачі
2. // наступні операції не виконуються
3. public static State<ImmutableDictionary<string, decimal>,
4.                     Maybe<decimal>>
5.     Take(string item, decimal kg) =>
6.     new(stock => stock.GetValueOrDefault(item) >= kg
7.         ? (new Maybe<decimal>.Something(stock[item] - kg),
8.            stock.SetItem(item, stock[item] - kg))
9.         : (new Maybe<decimal>.Nothing(), stock));
10.
11.  public static State<TS, Maybe<TOut>> BindMaybe<TS, TIn, TOut>(
12.      this State<TS, Maybe<TIn>> @this,
13.      Func<TIn, State<TS, Maybe<TOut>>> f) =>
14.      @this.Bind(m => m switch
15.      {
16.          Maybe<TIn>.Something s => f(s.Value),
17.          Maybe<TIn>.Error e => State.Return<TS, Maybe<TOut>>(
18.              new Maybe<TOut>.Error(e.CapturedError)),
19.          _ => State.Return<TS, Maybe<TOut>>(new Maybe<TOut>.Nothing())
20.      });
```

Лістинг 7.43 - Списання з припиненням після першої невдачі

```
1. var order = Take("борошно", 0.5m)
2.     .BindMaybe(_ => Take("сир", 3m))       // невдача
3.     .BindMaybe(_ => Take("борошно", 0.5m)); // не виконується
4.
5. var (result, stock) = order.Run(Initial);
```

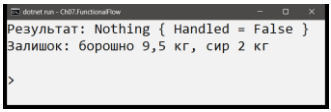


Рисунок 7.18 - Результат виконання прикладу «State і Maybe»

Перше списання борошна виконано, списання сиру не вдалося, а третю операцію не виконано зовсім.

7.10. Монади, що вже використовуються в C#

Деякі стандартні типи платформи .NET також задовольняють закони монад.

Перелічувані послідовності. Для *IEnumerable<T>* операцією загортання є створення послідовності з одного елемента, а операцією зв'язування - метод *SelectMany*. Виконання законів перевіряється в лістингу 7.44.

Лістинг 7.44 - Закони монад для IEnumerable

```
Func<int, IEnumerable<int>> portions = x => [x, x * 2];
Func<int, IEnumerable<string>> describe = x => [$"{x} г"];
Func<int, IEnumerable<int>> unit = x => [x];

int[] m = [100, 150];

var left = new[] { 100 }.SelectMany(portions)
                        .SequenceEqual(portions(100));
var right = m.SelectMany(unit).SequenceEqual(m);
var assoc = m.SelectMany(portions).SelectMany(describe)
     .SequenceEqual(m.SelectMany(x =>
    portions(x).SelectMany(describe)));
```

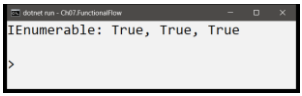


Рисунок 7.19 - Результат виконання прикладу «Закони для IEnumerable»

Задачі. Для *Task<T>* операцією загортання є *Task.FromResult*, а роль зв'язування виконує оператор *await* у поєднанні з подальшим викликом асинхронної функції. Вкладені очікування «вирівнюються» до значення результату (лістинг 7.45).

Лістинг 7.45 - Закони монад для Task

```
Func<int, Task<int>> addTips = x => Task.FromResult(x + x / 10);
Func<int, Task<string>> format = x => Task.FromResult($"{x} грн");

// Ліва тотожність: await Task.FromResult(a) і подальший виклик f
var left = await addTips(await Task.FromResult(500))
           == await addTips(500);

// Права тотожність: обгортання результату не змінює значення
var right = await Task.FromResult(await addTips(500))
             == await addTips(500);

 // Асоціативність: вкладені await «вирівнюються»
 var assoc = await format(await addTips(500))
             == await ((Func<Task<string>>)(async () =>
                 await format(await addTips(500))))();
```

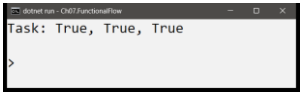


Рисунок 7.20 - Результат виконання прикладу «Закони для Task»

Отже, розробник, який працює з LINQ і *async/await*, фактично вже використовує монади.

7.11. Інші монадичні структури

Реалізація *Maybe* з трьома варіантами, що використовується в посібнику, строго кажучи, поєднує дві класичні монади. Класична монада *Maybe* (*Option*) має лише два варіанти - *Some* і *None*. Обробку помилок у функціональних мовах зазвичай виконує монада **`Either`** (або *Result*) з варіантами *Right* («успішний» шлях) і *Left* (помилка). Класичні структури можна отримати з наведеної реалізації, вилучивши відповідно варіант *Error* або *Nothing*. Поєднання обох структур не є канонічним, але є практичним для взаємодії з зовнішніми ресурсами.

У чисто функціональних мовах, зокрема Haskell, широко використовуються й інші монади:

- **Identity** - повертає передане значення без змін; корисна в теоретичних побудовах, але в C# практичного застосування майже не має;
- **IO** - дає змогу описувати взаємодію з зовнішнім світом без порушення чистоти функцій; у C# аналогічну роль відіграють інверсія керування та впровадження залежностей;
- **Writer** - накопичує додаткові дані (наприклад, записи журналу) під час кожного зв'язування.

Мова C# є гібридною, і прагнення реалізувати в ній усі монади чисто функціональних мов здебільшого не виправдане. Найбільшу практичну цінність мають *Maybe*/*Either*; решту структур доцільно вивчати з теоретичною метою.

7.12. Комплексний приклад: розрахунок вартості трансферу

Розглянуті засоби доцільно поєднати в завершеному прикладі. Готель пропонує гостям трансфер з аеропорту чи вокзалу. Вартість трансферу визначається так:

1) за назвами початкового і кінцевого пунктів отримуються адреси із системи картографії;

2) асинхронний сервіс маршрутизації визначає маршрут між адресами;

3) сервіс моніторингу дорожнього руху повідомляє про ремонтні роботи на маршруті;

4) сервіс тарифікації обчислює вартість маршруту;

5) за наявності ремонтних робіт вартість збільшується на 10 %.

Інтерфейси зовнішніх сервісів наведено в лістингу 7.46.

Лістинг 7.46 - Інтерфейси зовнішніх сервісів

```
1. public interface IMappingSystem
2. {
3.     Maybe<Address> GetAddress(Location location);
```

```
4. }
5.
6. public interface IRoutePlanner
7. {
8.     Task<Maybe<Route>> DetermineRoute(Address from, Address to);
9. }
10.
11.  public interface ITrafficMonitor
12.  {
13.      Maybe<TrafficAdvice> GetAdvice(Route route);
14.  }
15.
16.  public interface IPricingCalculator
17.  {
18.      decimal PriceRoute(Route route);
19.  }
```

Функцію розрахунку наведено в лістингу 7.47.

Лістинг 7.47 - Розрахунок вартості трансферу

```
1. public async Task<Maybe<decimal>> DeterminePrice(Location from,
2.                                                  Location to)
3. {
4.     var addresses = mapping.GetAddress(from)
5.         .Bind(a => mapping.GetAddress(to)
6.             .Map(b => (From: a, To: b)));
7.
8.     var route = await addresses.BindAsync(
9.         x => router.DetermineRoute(x.From, x.To));
10.
11.      var hasRoadworks = route
12.          .Bind(r => traffic.GetAdvice(r))
13.          .Match(a => a.RoadworksOnRoute, () => false, _ => false);
14.
15.      var price = route.Map(r => pricing.PriceRoute(r));
16.
17.      var finalPrice = price.Map(p => hasRoadworks
18.          ? Math.Round(p * 1.1m, 2)
19.          : p);
20.
21.      return finalPrice;
22.  }
```

Результати для коректного запиту, для невідомого пункту відправлення та для збою сервісу маршрутизації:

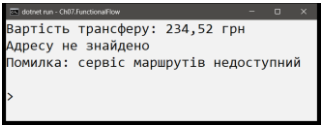


Рисунок 7.21 - Результат виконання прикладу «Трансфер»

Функція не містить жодної явної обробки помилок, хоча будь-який із зовнішніх сервісів може згенерувати виняток або не знайти даних. Якщо, наприклад, сервіс маршрутизації недоступний, *Maybe* переходить у стан

Error, і жодна з наступних операцій не виконується; кінцевий результат містить вихідний виняток. Код-споживач самостійно визначає, як повідомити про це користувача. Кортеж адрес будується так, що за відсутності першої адреси друга не запитується, а обидві адреси стають доступними наступному кроку. Об'єктно-орієнтована реалізація тієї самої логіки з блоками *try/catch* і перевірками кожного об'єкта була б у два-три рази довшою.

7.13. Висновки

1. Монада - це структура, що загортає значення й надає операцію зв'язування *Bind* для побудови ланцюжків обчислень; монада *Maybe* дає змогу будувати такі ланцюжки без блоків *try/catch* і перевірок на *null*.

2. Функція *Map* застосовує до значення функцію, що перетворює дані на дані, функція *Bind* - функцію, що перетворює дані на монаду; *Bind* запобігає появі вкладених монад.

3. Відсутність значення слід визначати перевіркою на *null*, а не порівнянням зі значенням за замовчуванням, інакше коректні нульові значення типів-значень помилково трактуються як відсутні.

4. Методи-обробники *OnSomething*, *OnNothing*, *OnError* з ознакою обробленого стану забезпечують журналювання в ланцюжках без дублювання повідомлень; асинхронні операції обробляються функціями *BindAsync* і *MapAsync*.

5. Монада задовольняє три закони: ліву тотожність, праву тотожність і асоціативність; їх виконання для *Maybe*, *IEnumerable<T>* і *Task<T>* можна перевірити безпосередньо.

6. Монада *Reader* описує обчислення, що залежить від середовища, яке надається під час виконання; вона є функціональним аналогом впровадження залежностей.

7. Монада *State* описує обчислення зі станом як функцію *S -> (T, S)*.

8. Функтор - узагальнений тип з операцією *Map*, що задовольняє закони тотожності й композиції; аплікативний функтор додає операцію *Apply* для поєднання незалежних значень у контексті; монада додає операцію *Bind* для послідовності залежних кроків. Кожна монада є аплікативним функтором, а кожен аплікативний функтор - функтором.

9. Аплікативна перевірка даних (*Validation*) накопичує всі помилки незалежних перевірок, тоді як монадичне зв'язування зупиняється на першій помилці.

10. Синтаксис запитів LINQ є монадичною нотацією мови C#: компілятор перетворює речення *from* на виклики *Select* і *SelectMany*, тож запити можна писати для будь-якого типу з цими методами, зокрема *Maybe*, *State* і власних типів-об'єднань.

7.14. Контрольні запитання

1. Які типові дії доводиться виконувати перед використанням даних, отриманих від зовнішньої системи? Як *Maybe* спрощує цей код?

2. Опишіть поведінку функції *Map* для кожного варіанта *Maybe*.

3. У чому полягає принцип залізничного програмування?

4. Чим функція *Bind* відрізняється від *Map*? У яких випадках виникають вкладені *Maybe*?

5. Чому порівняння зі значенням за замовчуванням є некоректним способом визначення відсутності значення?

6. Для чого призначена ознака *Handled* в обробниках *OnNothing* і *OnError*?

7. Сформулюйте три закони монад. Що означає кожен з них?

8. Яку задачу розв'язує монада *Reader*? Коли виконується обчислення, описане *Reader*?

9. Чим монада *State* відрізняється від *Reader*? Який тип має обчислення зі станом?

10. Які методи потрібно реалізувати, щоб для власної монади можна було використовувати синтаксис запитів LINQ?

11. Чому *IEnumerable<T>* і *Task<T>* можна вважати монадами? Які операції відіграють роль загортання і зв'язування?

12. Які класичні монади поєднує реалізація *Maybe* з трьома варіантами?

13. Що таке функтор? Сформулюйте його закони та поясніть їх практичне значення.

14. Чим аплікативне поєднання відрізняється від монадичного зв'язування? Чому аплікативна перевірка форми повідомляє про всі помилки?

15. Поясніть ієрархію «функтор - аплікативний функтор - монада». Наведіть приклади кожної структури в платформі .NET.

16. На які виклики методів компілятор перетворює запит LINQ з трьома реченнями *from*? Чому синтаксис запитів застосовний не лише до колекцій?

7.15. Практичні завдання

1. Реалізуйте функцію отримання номера телефону гостя за номером кімнати: за кімнатою знаходиться бронювання, за бронюванням - гість, за гостем - телефон. Кожне сховище повертає *Maybe*. Використайте *Bind*, *Map* і *Match*; перевірте всі варіанти результату.

2. Перепишіть функцію із завдання 1 у синтаксисі запитів LINQ.

3. Додайте до ланцюжка із завдання 1 журналювання за допомогою *OnSomething*, *OnNothing* і *OnError*. Переконайтеся, що кожне попередження записується один раз.

4. Реалізуйте асинхронну версію функції із завдання 1, у якій сховища повертають *Task*<*Maybe*<*T*>>.

5. Реалізуйте за допомогою монади *Reader* розрахунок вартості проживання з параметрами готелю (ставка туристичного збору, знижка для постійних гостей, валюта) і виконайте його для двох готелів мережі.

6. Змоделюйте за допомогою монади *State* касову зміну: стан містить суму готівки в касі та кількість чеків; операції - продаж, повернення, інкасація. Запишіть послідовність операцій у синтаксисі запитів LINQ.

7. Перевірте виконання законів монад для власної реалізації *Either*<*TLeft, TRight*> з функцією *Bind*.

8. Реалізуйте аплікативну перевірку форми реєстрації гостя готелю (прізвище, ім'я, дата народження, номер документа, дата заїзду не раніше поточної) за допомогою *Validation* і *Apply*. Порівняйте результат з монадичною перевіркою тих самих полів.

9. Перевірте виконання законів функтора для *Validation* з функцією *Map*, яку потрібно реалізувати самостійно.

8. Власна бібліотека монад

У розділі 7 поняття монади розглянуто на прикладі *Maybe*, а також монад *Reader* і *State*. Для практичної розробки цього недостатньо: прикладні задачі потребують кількох взаємопов'язаних монадичних типів з однаковим набором операцій, що утворюють цілісну бібліотеку. У цьому розділі проєктується й реалізується така бібліотека: монада *Either* для обчислень з інформативною помилкою, структура *Exceptional* для результатів, що можуть містити виняток, монада *Try* для відкладених обчислень, а також монадичні операції для стандартного типу *Task<T>*. Наприкінці розділу бібліотеку застосовано в комплексному прикладі, що поєднує асинхронність і обробку помилок, та перевірено виконання законів монад для її типів.

Мета розділу - сформувати навички проєктування та реалізації власної бібліотеки монадичних типів у мові C# і їх спільного використання в прикладних задачах.

Після опрацювання розділу студент повинен:

- знати принципи проєктування бібліотеки монад: єдиний набір операцій, незмінність, закритість ієрархії варіантів, підтримку синтаксису запитів LINQ;
- уміти реалізовувати монаду *Either* з функціями *Match*, *Map*, *Bind* і *MapLeft*;
- розуміти відмінність між негайним (*Exceptional*) і відкладеним (*Try*) поданням обчислень, що можуть згенерувати виняток;
- уміти доповнювати стандартний тип *Task<T>* монадичними операціями та обґрунтовувати вибір реалізації через *async/await*;
- уміти поєднувати монади *Task* і *Either* у ланцюжках асинхронних операцій з обробкою помилок;
- уміти перевіряти закони монад для власних типів і розпізнавати типові помилки їх реалізації.

Під час підготовки розділу як додаткове джерело використано книгу C. Bueno «C# Functional: Monads from Zero to Hero» (2024); реалізації першоджерела переосмислено й виправлено (див. підрозділ 8.7.2). Типи бібліотеки містяться в просторі імен *HoReCa.Domain.Functional* спільної бібліотеки, приклади - у проєкті *Ch08.MonadLibrary*.

8.1. Проєктування бібліотеки монад

Під **бібліотекою монад** надалі розуміється набір узагальнених типів, кожен з яких описує окремий **обчислювальний контекст** (можливу відсутність значення, помилку, виняток, відкладеність, асинхронність), і набір однойменних операцій над ними. Однаковість операцій є ключовою вимогою: розробник, який засвоїв роботу з одним типом бібліотеки, без

додаткового навчання використовує решту, а ланцюжки з різних типів читаються однаково.

Склад бібліотеки, що розробляється в розділі, з урахуванням типів, реалізованих раніше, наведено в таблиці 8.1.

Таблиця 8.1 - Склад бібліотеки монад

Тип	**Контекст обчислення**	**Варіанти**	**Розділ**
Maybe<T>	Значення може бути відсутнім або отриманим з помилкою	*Something*, *Nothing*, *Error*	6, 7
Either<TLeft, TRight>	Результат або інформативна помилка	*Right*, *Left*	8.2
Exceptional<T>	Результат або перехоплений виняток	успіх, виняток	8.3
Try<T>	Відкладене обчислення, що може згенерувати виняток	- (функція)	8.4
Task<T>	Асинхронне обчислення	- (стандартний тип)	8.5
Reader<TEnv, T>	Обчислення, що залежить від середовища	- (функція)	7.8
State<TS, T>	Обчислення зі станом	- (функція)	7.9

Для всіх типів бібліотеки встановлюються такі **принципи проєктування**:

1) **єдиний словник операцій**: загортання значення (*return*; у бібліотеці - неявне перетворення, *Of* або *Return*), перетворення *Map*, зв'язування *Bind*, вихід з контексту *Match*, значення за замовчуванням *GetOrElse*; методи *Select* і *SelectMany* забезпечують підтримку синтаксису запитів LINQ (підрозділ 7.7.5);

2) **незмінність**: значення монадичних типів після створення не змінюються, тому вони реалізуються записами (*record*) або структурами лише для читання (*readonly struct*);

3) **закритість ієрархії варіантів**: базовий тип має приватний конструктор, тож за межами його оголошення не можна створити варіант, не передбачений функцією *Match*;

4) **явна обробка всіх варіантів**: значення з контексту отримується лише через *Match* або *GetOrElse*, а не через властивість, що генерує виняток для «неуспішного» варіанта;

5) **виконання законів монад** (підрозділ 7.7.1), що перевіряється для кожного типу (підрозділ 8.7.1).

Окремим проєктним рішенням є поведінка *Map* і *Bind* у разі винятку у функції-аргументі. Для *Maybe* (розділ 7), *Exceptional* і *Try* такий виняток перехоплюється й стає варіантом результату, оскільки саме перехоплення винятків є призначенням цих типів. Для *Either* виняток не перехоплюється: варіант *Left* призначений для очікуваних, передбачених бізнес-логікою помилок, а виняток сигналізує про дефект програми і не повинен «маскуватися» під бізнес-помилку.

У промисловій розробці бібліотеку монад доцільно виокремити в окремий проєкт (пакет NuGet), спільний для всіх застосунків команди. Готові бібліотеки такого призначення (LanguageExt, CSharpFunctionalExtensions тощо) розглядаються в розділі 13; власна реалізація дає змогу зрозуміти їх будову й обґрунтовано обирати між ними.

8.2. Монада Either

8.2.1. Структура типу

Об'єднання *Either* з варіантами *Left* і *Right* розглянуто в підрозділі 6.7.4 як засіб подання результату одного з двох типів. Доповнене функціями зв'язування, воно стає **монадою `Either`** - стандартним у функціональних мовах засобом обробки помилок: варіант *Right* («правильний») містить успішний результат, варіант *Left* - опис помилки довільного типу (рядок, код помилки, запис з подробицями). На відміну від *Maybe*, *Either* не лише повідомляє про невдачу, а й пояснює її причину.

Початок оголошення типу наведено в лістингу 8.1; члени типу розглядаються в наступних лістингах. Неявні перетворення дають змогу повертати з функції безпосередньо значення або опис помилки без явного виклику конструктора варіанта. Якщо типи *TLeft* і *TRight* збігаються, перетворення стають неоднозначними, і варіанти слід створювати конструкторами *Left* і *Right*.

Лістинг 8.1 - Тип Either з двома варіантами

```
/// <summary>
/// Either: Right - успішний результат, Left - помилка
/// (альтернативний результат). Ієрархія закрита приватним
/// конструктором, тож інших варіантів бути не може.
/// </summary>
```

```
public abstract record Either<TLeft, TRight>
{
    private Either() { }

    public sealed record Left(TLeft Value) : Either<TLeft, TRight>;
    public sealed record Right(TRight Value) : Either<TLeft, TRight>;

    // Неявні перетворення: return "помилка"; або return value;
    public static implicit operator Either<TLeft, TRight>(
        TLeft value) => new Left(value);
    public static implicit operator Either<TLeft, TRight>(
        TRight value) => new Right(value);
```

Єдиним способом отримати значення з *Either* є функція *Match* (лістинг 8.2), що приймає по функції для кожного варіанта. Гілка відкидання _ у виразі *switch* потрібна компілятору, проте завдяки приватному конструктору базового типу вона недосяжна.

Лістинг 8.2 - Функція Match для Either

```
// Єдиний «вихід» з контексту: по функції на кожен варіант
public T Match<T>(Func<TLeft, T> onLeft, Func<TRight, T> onRight) =>
    this switch
    {
        Left l => onLeft(l.Value),
        Right r => onRight(r.Value),
        _ => throw new InvalidOperationException("невідомий варіант")
    };
```

8.2.2. Функції Map і Bind

Функції *Map* і *Bind* (лістинг 8.3) реалізовано через *Match*. Для варіанта *Right* функція-аргумент застосовується до значення, для варіанта *Left* помилка передається далі без змін, тобто ланцюжок «перемикається» на колію помилки (підрозділ 7.1.2) і решта кроків не виконується. Таку поведінку називають **коротким замиканням** (short-circuiting).

Лістинг 8.3 - Функції Map і Bind для Either

```
// Map: перетворює успішне значення, помилку передає далі
public Either<TLeft, TOut> Map<TOut>(Func<TRight, TOut> f) =>
    Match<Either<TLeft, TOut>>(
        l => new Either<TLeft, TOut>.Left(l),
        r => new Either<TLeft, TOut>.Right(f(r)));

// Bind: функція сама повертає Either; перша помилка
// зупиняє ланцюжок (коротке замикання)
public Either<TLeft, TOut> Bind<TOut>(
    Func<TRight, Either<TLeft, TOut>> f) =>
    Match(l => new Either<TLeft, TOut>.Left(l), f);
```

Допоміжні функції наведено в лістингу 8.4: *MapLeft* перетворює помилку (наприклад, код помилки на текст для користувача), *GetOrElse* повертає значення або запасне значення в разі помилки, а перевизначений метод *ToString* подає значення у стислій формі. Модифікатор *sealed* у

перевизначенні *ToString* забороняє похідним записам генерувати власну реалізацію цього методу.

Лістинг 8.4 - Допоміжні функції MapLeft, GetOrElse і ToString

```
// Перетворення помилки (наприклад, коду на текст повідомлення)
public Either<TOut, TRight> MapLeft<TOut>(Func<TLeft, TOut> f) =>
    Match<Either<TOut, TRight>>(
        l => new Either<TOut, TRight>.Left(f(l)),
        r => new Either<TOut, TRight>.Right(r));

// Значення за замовчуванням у разі помилки
public TRight GetOrElse(TRight fallback) =>
    Match(_ => fallback, r => r);

public bool IsRight => this is Right;

public sealed override string ToString() =>
     Match(l => $"Left({l})", r => $"Right({r})");
```

Нехай під час оформлення замовлення столика введена кількість гостей розбирається, перевіряється на відповідність місткості зали й перетворюється на кількість столиків. Перші два кроки можуть завершитися помилкою, третій - ні (лістинг 8.5). Умовні вирази у функціях мають різні типи гілок (*int* і *string*), тому їх тип визначається цільовим типом *Either<string, int>*, до якого обидві гілки перетворюються неявно.

Лістинг 8.5 - Кроки оформлення замовлення столика

```
// Кроки оформлення замовлення столика; кожен може не вдатися
public static Either<string, int> ParseGuests(string input) =>
    int.TryParse(input, out var n)
        ? n
        : $"«{input}» не є кількістю гостей";

public static Either<string, int> CheckCapacity(int guests) =>
    guests is > 0 and <= 24
        ? guests
          : $"зала вміщує від 1 до 24 гостей, а не {guests}";

// Чиста функція без можливості помилки: для неї - Map
public static int TablesNeeded(int guests) => (guests + 3) / 4;
```

Кроки, що повертають *Either*, з'єднуються функцією *Bind*, а крок, що повертає звичайне значення, - функцією *Map* (лістинг 8.6).

Лістинг 8.6 - Ланцюжок Bind - Map - Match

```
foreach (var input in new[] { "10", "десять", "40" })
{
    var message = ParseGuests(input)
        .Bind(CheckCapacity)        // T -> Either<L, U>
        .Map(TablesNeeded)          // T -> U
        .Match(
            onLeft: error => $"Відмова: {error}",
            onRight: tables => $"Підготувати столиків: {tables}");
    Console.WriteLine($"{input,-7} -> {message}");
}
```

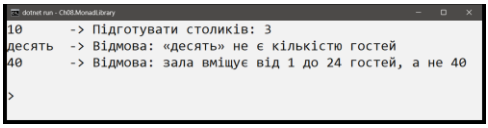


Рисунок 8.1 - Результат виконання прикладу «Either: Map і Bind»

На відміну від *Maybe*, результат містить причину відмови, сформульовану тим кроком, на якому відбулася невдача.

8.2.3. Синтаксис запитів LINQ

Методи *Select* і *SelectMany* (лістинг 8.7) реалізуються однаково для всіх типів бібліотеки: *Select* збігається з *Map*, а *SelectMany* поєднує *Bind* з функцією проєкції, що має доступ до результатів обох кроків.

Лістинг 8.7 - Підтримка синтаксису запитів LINQ для Either

```
1. // Підтримка синтаксису запитів LINQ (підрозділ 7.7.5)
2. public static Either<TL, TOut> Select<TL, TIn, TOut>(
3.     this Either<TL, TIn> @this, Func<TIn, TOut> f) => @this.Map(f);
4.
5. public static Either<TL, TOut> SelectMany<TL, TIn, TMid, TOut>(
6.     this Either<TL, TIn> @this,
7.     Func<TIn, Either<TL, TMid>> bind,
8.     Func<TIn, TMid, TOut> project) =>
9.     @this.Bind(a => bind(a).Map(b => project(a, b)));
```

У лістингу 8.8 розрахунок передоплати за банкет (30 % вартості) записано як запит: кожне речення *from* є кроком, що може завершитися помилкою, а змінні всіх попередніх кроків доступні в наступних.

Лістинг 8.8 - Розрахунок передоплати в синтаксисі запитів LINQ

```
1. // Залежні кроки в синтаксисі запитів LINQ
2. Either<string, decimal> Deposit(string guests, string price) =>
3.     from g in ParseGuests(guests)
4.     from ok in CheckCapacity(g)
5.     from p in decimal.TryParse(price, out var v) && v > 0
6.         ? (Either<string, decimal>)v
7.         : "некоректна ціна банкетного меню"
8.     select ok * p * 0.3m;
9.
10.  Console.WriteLine(Deposit("12", "850"));
11.  Console.WriteLine(Deposit("12", "-1"));
12.  Console.WriteLine(Deposit("сто", "850").GetOrElse(0m));
```

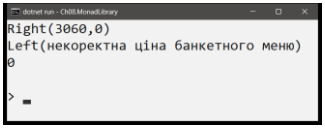


Рисунок 8.2 - Результат виконання прикладу «Either: запит LINQ»

8.3. Exceptional: результат з винятком

8.3.1. Структура типу

Тип *Either<Exception, T>* придатний для подання результату, що може містити виняток, проте для такого поширеного випадку доцільно мати спеціалізований тип. Тип **`Exceptional<T>`** містить або значення типу *T*, або перехоплений виняток. На відміну від *Either*, його реалізовано **структурою лише для читання** (лістинг 8.9, початок оголошення): значення не потребує виділення пам'яті в купі, що суттєво для типу, який повертається з великої кількості функцій. Неявні перетворення дають змогу повертати з функції як значення, так і виняток.

Лістинг 8.9 - Структура Exceptional

```
1. /// <summary>
2. /// Exceptional: результат обчислення - значення або виняток.
3. /// Незмінна структура без виділення пам'яті в купі.
4. /// </summary>
5. public readonly struct Exceptional<T>
6. {
7.     private readonly T? value;
8.     private readonly Exception? exception;
9.
10.      public bool IsSuccess { get; }
11.
12.      private Exceptional(T value)
13.      {
14.          this.value = value;
15.          exception = null;
16.          IsSuccess = true;
17.      }
18.
19.      private Exceptional(Exception ex)
20.      {
21.          value = default;
22.          exception = ex ?? throw new ArgumentNullException(nameof(ex));
23.          IsSuccess = false;
24.      }
25.
26.      public static implicit operator Exceptional<T>(T value) =>
27.          new(value);
28.      public static implicit operator Exceptional<T>(Exception ex) =>
29.          new(ex);
```

Подання структурою має недолік: для кожної структури існує **значення за замовчуванням** (*default*), створене без виклику конструктора, у якому немає ні значення, ні винятку. Функція *Match* (лістинг 8.10) трактує такий стан як помилку з поясненням, а не передає у функцію обробки *null*. Аналогічну проблему для типів-об'єднань C# 15 розглянуто в підрозділі 6.7.5.

Лістинг 8.10 - Функція Match для Exceptional

```
// default(Exceptional<T>) не має ні значення, ні винятку:
// такий стан трактується як помилка, а не як null
public TOut Match<TOut>(Func<Exception, TOut> onException,
                        Func<T, TOut> onSuccess) =>
    IsSuccess
        ? onSuccess(value!)
        : onException(exception ?? new InvalidOperationException(
            "неініціалізоване значення Exceptional"));
```

8.3.2. Операції над Exceptional

Загортання обчислення, що може згенерувати виняток, виконує функція *Exceptional.Of* (лістинг 8.11). Функції *Map* і *Bind* (лістинг 8.12) перехоплюють винятки функції-аргументу: *Map* виражено через *Bind* і *Of*, а *Bind* для неуспішного значення передає далі той самий виняток.

Лістинг 8.11 - Функція Exceptional.Of

```
// Загортання обчислення, що може згенерувати виняток
public static Exceptional<T> Of<T>(Func<T> computation)
{
    try
    {
        return computation();
    }
    catch (Exception ex)
    {
        return ex;
    }
}
```

Лістинг 8.12 - Функції Map і Bind для Exceptional

```
// Map і Bind перехоплюють винятки функції-аргументу:
// виняток стає варіантом результату, а не «вилітає» назовні
public Exceptional<TOut> Map<TOut>(Func<T, TOut> f) =>
    Bind(x => Exceptional.Of(() => f(x)));

public Exceptional<TOut> Bind<TOut>(
    Func<T, Exceptional<TOut>> f)
{
    if (!IsSuccess)
        return Match<Exceptional<TOut>>(ex => ex, _ => default);
    try
    {
        return f(value!);
    }
    catch (Exception ex)
    {
        return ex;
    }
}
```

Лістинг 8.13 - Функції GetOrElse і ToString для Exceptional

```
1. public T GetOrElse(T fallback) => Match(_ => fallback, x => x);
2.
3. public override string ToString() =>
4.     Match(ex => $"Exception({ex.GetType().Name}: {ex.Message})",
5.           x => $"Success({x})");
```

У лістингу 8.14 функція обчислення середнього чека на гостя повертає *Exceptional<decimal>*: тип результату повідомляє коду-споживачеві про можливу невдачу, а функція *Display* обробляє обидва варіанти явно.

Лістинг 8.14 - Функція з результатом Exceptional

```
1. // Середній чек на гостя; неявні перетворення дають змогу
2. // повертати як значення, так і виняток
3. static Exceptional<decimal> AverageCheck(decimal total, int guests)
4. {
5.     try
6.     {
7.         return Math.Round(total / guests, 2);
8.     }
9.     catch (DivideByZeroException ex)
10.      {
11.          return ex;
12.      }
13.  }
14.
15.  static void Display(Exceptional<decimal> result) =>
16.      Console.WriteLine(result.Match(
17.          onException: ex => $"Помилка: {ex.Message}",
18.          onSuccess: x => $"Середній чек: {x} грн"));
```

Лістинг 8.15 - Використання Exceptional

```
1. Display(AverageCheck(4380m, 6));
2. Display(AverageCheck(4380m, 0));
3.
4. // Map: подальше перетворення лише для успішного значення
5. var withTips = AverageCheck(4380m, 6).Map(x => x * 1.1m);
6. // Bind: наступний крок теж повертає Exceptional
7. var perHalf = AverageCheck(4380m, 6)
8.     .Bind(x => AverageCheck(x, 2));
9. Console.WriteLine($"{withTips}; {perHalf}");
10.
11.  // Значення за замовчуванням для структури
12.  Exceptional<decimal> empty = default;
13.  Console.WriteLine(empty);
```

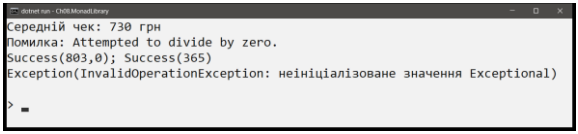


Рисунок 8.3 - Результат виконання прикладу «Exceptional»

Останній рядок виводу ілюструє обробку значення за замовчуванням структури.

8.3.3. Аплікативне застосування Apply

Для *Exceptional* реалізовано також аплікативну операцію *Apply* (підрозділ 7.7.3): функцію, що сама перебуває в контексті *Exceptional*, застосовують до значення в контексті (лістинг 8.16). У поєднанні з каррінгом (розділ 9) це дає змогу застосувати функцію кількох аргументів до кількох незалежно отриманих значень: *Map* частково застосовує карровану функцію до першого аргументу, а *Apply* - до другого (лістинг 8.17).

Лістинг 8.16 - Функція Apply для Exceptional

```
1. // Apply: функція в контексті застосовується до значення
2. // в контексті (аплікативний функтор, підрозділ 7.7.3)
3. public static Exceptional<TOut> Apply<T, TOut>(
4.     this Exceptional<Func<T, TOut>> f, Exceptional<T> arg) =>
5.     f.Bind(func => arg.Map(func));
```

Лістинг 8.17 - Аплікативне обчислення середнього чека

```
1. // Каррована функція двох аргументів (розділ 9)
2. Func<decimal, Func<int, decimal>> average =
3.     total => guests => Math.Round(total / guests, 2);
4.
5. Exceptional<decimal> Average(string total, string guests) =>
6.     Exceptional.Of(() => decimal.Parse(total))
7.         .Map(average)                    // Exceptional<Func>
8.         .Apply(Exceptional.Of(() => int.Parse(guests)));
9.
10.  Console.WriteLine(Average("4380", "6"));
11.  Console.WriteLine(Average("4380", "шість"));
12.  Console.WriteLine(Average("4380", "0"));
```

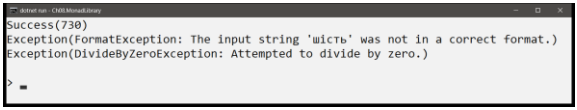


Рисунок 8.4 - Результат виконання прикладу «Exceptional: Apply»

Помилки розбору та ділення на нуль не потребують жодного блоку *try/catch* у прикладному коді. Підтримку синтаксису запитів LINQ для *Exceptional* реалізовано так само, як для *Either* (лістинг 8.18).

Лістинг 8.18 - Підтримка синтаксису запитів LINQ для Exceptional

```
1. public static Exceptional<TOut> Select<T, TOut>(
2.     this Exceptional<T> @this, Func<T, TOut> f) => @this.Map(f);
3.
4. public static Exceptional<TOut> SelectMany<T, TMid, TOut>(
5.     this Exceptional<T> @this,
6.     Func<T, Exceptional<TMid>> bind,
7.     Func<T, TMid, TOut> project) =>
8.     @this.Bind(a => bind(a).Map(b => project(a, b)));
```

8.4. Монада Try: відкладене обчислення

8.4.1. Негайне і відкладене подання обчислень

Exceptional подає **результат** уже виконаного обчислення: функція *Exceptional.Of* виконує передане обчислення негайно. Іноді ж потрібно подати саме **обчислення** - описати послідовність кроків, які можуть згенерувати виняток, а виконати її пізніше, повторно або не виконувати зовсім (наприклад, лише за потреби, як у thunk з підрозділу 5.2). Для цього призначена **монада `Try`**: значення *Try<T>* є функцією, що повертає *Exceptional<T>*, а функції *Map* і *Bind* будують нову функцію, не виконуючи попередньої. Обчислення відбувається лише під час виклику *Run* (лістинг 8.19, початок оголошення). За будовою *Try* подібна до монади *Reader* (підрозділ 7.8), але функція не має параметра середовища.

Лістинг 8.19 - Монада Try

```
1. /// <summary>
2. /// Try: ВІДКЛАДЕНЕ обчислення, що може згенерувати виняток.
3. /// Створення Try і виклики Map/Bind нічого не виконують;
4. /// обчислення відбувається лише під час Run.
5. /// </summary>
6. public sealed class Try<T>
7. {
8.     private readonly Func<Exceptional<T>> run;
9.
10.      internal Try(Func<Exceptional<T>> run) => this.run = run;
11.
12.      // Виконання: результат подається значенням Exceptional
13.      public Exceptional<T> Run() => run();
```

Лістинг 8.20 - Функції Map і Bind для Try

```
1. public Try<TOut> Map<TOut>(Func<T, TOut> f) =>
2.     new(() => Run().Map(f));
3.
4. public Try<TOut> Bind<TOut>(Func<T, Try<TOut>> f) =>
5.     new(() => Run().Bind(x => f(x).Run()));
```

Лістинг 8.21 - Функції загортання Try.Of і Try.Return

```
1. public static Try<T> Of<T>(Func<T> computation) =>
2.     new(() => Exceptional.Of(computation));
3.
4. public static Try<T> Return<T>(T value) =>
5.     new(() => value);
```

Функції *Map* і *Bind* монади *Try* делегують роботу відповідним функціям *Exceptional*, тож винятки перехоплюються однаково. Важливо, що лямбда-вираз *() => Run().Map(f)* лише **описує** виклик *Run* попереднього кроку, а не виконує його.

8.4.2. Використання Try

Нехай собівартість порції обчислюється діленням вартості закладки продуктів на кількість порцій; функція *PortionCost* (лістинг 8.22) виводить повідомлення під час кожного виконання, що дає змогу простежити момент обчислення.

Лістинг 8.22 - Обчислення собівартості порції

```
1. // Собівартість порції: вартість закладки / кількість порцій.
2. // Ділення decimal на нуль генерує DivideByZeroException
3. static decimal PortionCost(decimal batchCost, int portions)
4. {
5.     Console.WriteLine($"  обчислення: {batchCost} / {portions}");
6.     return batchCost / portions;
7. }
```

Лістинг 8.23 - Відкладене виконання Try

```
1. var cost = Try.Of(() => PortionCost(1260m, 0))
2.     .Map(c => Math.Round(c * 1.2m, 2));   // + 20 % націнки
3. Console.WriteLine("Опис обчислення побудовано");
4.
5. var result = cost.Run();                  // лише тепер
6. Console.WriteLine(result);
7. Console.WriteLine(result.Match(
8.     onException: _ => "ціну не визначено: порцій немає",
9.     onSuccess: p => $"ціна порції {p} грн"));
10.
11.  var fixedCost = Try.Of(() => PortionCost(1260m, 18))
12.      .Map(c => Math.Round(c * 1.2m, 2));
13.  Console.WriteLine(fixedCost.Run());
14.  Console.WriteLine(fixedCost.Run());       // повторне виконання
```

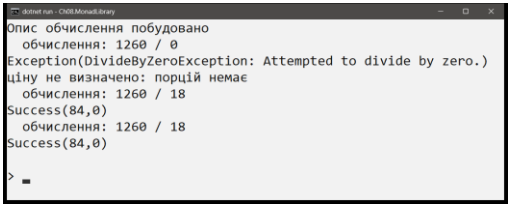


Рисунок 8.5 - Результат виконання прикладу «Try: відкладене виконання»

Повідомлення про обчислення з'являється лише після рядка «Опис обчислення побудовано», тобто під час виклику *Run*. Кожен виклик *Run* виконує обчислення заново; якщо результат потрібно обчислити один раз, функцію слід мемоізувати (розділ 11).

Синтаксис запитів LINQ (лістинг 8.24) дає змогу описати послідовність кроків, кожен з яких може згенерувати виняток (лістинг 8.25). Перший виняток зупиняє ланцюжок: у другому випадку помилка розбору кількості порцій унеможливлює ділення, і функція *PortionCost* не викликається.

Лістинг 8.24 - Підтримка синтаксису запитів LINQ для Try

```
1. public static Try<TOut> Select<T, TOut>(
2.     this Try<T> @this, Func<T, TOut> f) => @this.Map(f);
3.
4. public static Try<TOut> SelectMany<T, TMid, TOut>(
5.     this Try<T> @this, Func<T, Try<TMid>> bind,
6.     Func<T, TMid, TOut> project) =>
7.     @this.Bind(a => bind(a).Map(b => project(a, b)));
```

Лістинг 8.25 - Ланцюжок кроків у вигляді Try

```
1. // Кожен крок може згенерувати виняток; перший виняток
2. // зупиняє ланцюжок, решта кроків не виконуються
3. Try<decimal> MenuPrice(string batch, string portions) =>
4.     from cost in Try.Of(() => decimal.Parse(batch))
5.     from count in Try.Of(() => int.Parse(portions))
6.     from portion in Try.Of(() => PortionCost(cost, count))
7.     select Math.Round(portion * 1.2m, 2);
8.
9. Console.WriteLine(MenuPrice("1260", "18").Run());
10.  Console.WriteLine(MenuPrice("1260", "вісімнадцять").Run());
```

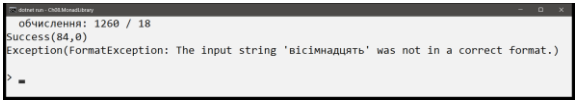


Рисунок 8.6 - Результат виконання прикладу «Try: ланцюжок»

Вибір між *Exceptional* і *Try* визначається тим, що передається між частинами програми: готовий результат (*Exceptional*) чи опис дії, момент виконання якої визначає код-споживач (*Try*).

8.5. Task<T> як монада

8.5.1. Функції Map і Bind для Task

У підрозділі 7.10 показано, що тип *Task<T>* задовольняє закони монад: операцією загортання є *Task.FromResult*, а роль зв'язування виконує оператор *await*. Проте явних функцій *Map* і *Bind* стандартна бібліотека не надає, тож ланцюжки асинхронних операцій записуються послідовністю інструкцій з *await*. Бібліотеку доповнено методами-розширеннями, що надають *Task<T>* той самий словник операцій, що й іншим типам (лістинг 8.26).

Лістинг 8.26 - Функції Map і Bind для Task

```
// Map: перетворення результату асинхронної операції
public static async Task<TOut> Map<T, TOut>(
    this Task<T> task, Func<T, TOut> f) => f(await task);

// Bind: наступна асинхронна операція залежить від результату
// попередньої; await «вирівнює» Task<Task<TOut>> до Task<TOut>
public static async Task<TOut> Bind<T, TOut>(
    this Task<T> task, Func<T, Task<TOut>> f) =>
    await f(await task);
```

Функції реалізовано асинхронними методами: компілятор будує скінченний автомат, що продовжує обчислення після завершення задачі, а оператор *await* у *Bind* «вирівнює» вкладену задачу *Task<Task<TOut>>* до *Task<TOut>*. Альтернативну реалізацію через метод *ContinueWith* і причини відмови від неї розглянуто в підрозділі 8.7.2.

У лістингу 8.27 наведено імітацію асинхронних сервісів постачальника: залишку позиції на складі та ціни за кілограм. Для невідомої позиції сервіс генерує виняток.

Лістинг 8.27 - Асинхронні сервіси постачальника

```
// Імітація асинхронних сервісів постачальника
static async Task<int> StockKg(string ingredient)
{
    await Task.Delay(10);
    return ingredient switch
    {
        "борошно" => 40,
        "сир" => 6,
        _ => throw new KeyNotFoundException(
             $"позиції «{ingredient}» немає в каталозі")
    };
}

static async Task<decimal> PricePerKg(string ingredient)
{
    await Task.Delay(10);
    return ingredient == "сир" ? 310m : 28m;
}
```

Лістинг 8.28 - Функції Select, SelectMany і Where для Task

```
public static Task<TOut> Select<T, TOut>(
    this Task<T> task, Func<T, TOut> f) => task.Map(f);

public static async Task<TOut> SelectMany<T, TMid, TOut>(
    this Task<T> task, Func<T, Task<TMid>> bind,
    Func<T, TMid, TOut> project)
{
    var a = await task;
    return project(a, await bind(a));
}

// Where: невиконана умова перетворює задачу на помилкову
public static async Task<T> Where<T>(
```

```
14.      this Task<T> task, Func<T, bool> predicate)
15.  {
16.      var x = await task;
17.      return predicate(x)
18.          ? x
19.          : throw new InvalidOperationException(
20.              "умову фільтра не виконано");
21.  }
```

Методи *Select* і *SelectMany* дають змогу записувати ланцюжки асинхронних операцій у синтаксисі запитів LINQ (лістинг 8.29).

Лістинг 8.29 - Map, Bind і запит LINQ для Task

```
1. // Map: перетворення результату
2. string text = await StockKg("сир").Map(kg => $"{kg} кг");
3. Console.WriteLine(text);
4.
5. // Bind: наступна асинхронна операція
6. decimal value = await StockKg("сир")
7.     .Bind(kg => PricePerKg("сир").Map(p => kg * p));
8. Console.WriteLine($"вартість залишку: {value} грн");
9.
10.  // Те саме в синтаксисі запитів LINQ
11.  var query =
12.      from kg in StockKg("борошно")
13.      from price in PricePerKg("борошно")
14.      select kg * price;
15.  Console.WriteLine($"вартість залишку: {await query} грн");
```

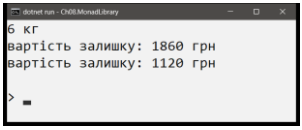


Рисунок 8.7 - Результат виконання прикладу «Task: Map і Bind»

8.5.2. Відновлення після помилок

Помилкою асинхронної операції є виняток, збережений у задачі й повторно згенерований оператором *await*. Функції відновлення (лістинг 8.30) перетворюють помилкову задачу на успішну: *Recover* обчислює значення за винятком, *GetOrElse* підставляє фіксоване значення, *OrElse* звертається до запасного джерела даних. Функція *Where* (лістинг 8.28) діє навпаки: невиконання умови перетворює успішну задачу на помилкову.

Лістинг 8.30 - Функції Recover, OrElse і GetOrElse для Task

```
1. // Recover: обчислення значення за винятком
2. public static async Task<T> Recover<T>(
3.     this Task<T> task, Func<Exception, T> recover)
4. {
5.     try
6.     {
7.         return await task;
8.     }
9.     catch (Exception ex)
```

```
    {
        return recover(ex);
    }
}

// OrElse: запасна асинхронна операція
public static async Task<T> OrElse<T>(
    this Task<T> task, Func<Task<T>> fallback)
{
    try
    {
        return await task;
    }
    catch
    {
        return await fallback();
    }
}

public static Task<T> GetOrElse<T>(this Task<T> task, T value) =>
    task.Recover(_ => value);
```

Для переходу від винятків до явного результату призначена функція *ToExceptional* (лістинг 8.31): після неї помилка асинхронної операції стає значенням, яке можна обробляти функціями підрозділу 8.3.

Лістинг 8.31 - Функція ToExceptional

```
// Перехід від «винятків у задачі» до явного результату
public static async Task<Exceptional<T>> ToExceptional<T>(
    this Task<T> task)
{
    try
    {
        return await task;
    }
    catch (Exception ex)
    {
        return ex;
    }
}
```

Лістинг 8.32 - Відновлення після помилок асинхронних операцій

```
// Recover / GetOrElse: значення замість винятку
int a = await StockKg("шафран").Recover(ex =>
{
    Console.WriteLine($"журнал: {ex.Message}");
    return 0;
});
int b = await StockKg("шафран").GetOrElse(-1);
// OrElse: запасне джерело даних
int c = await StockKg("шафран").OrElse(() => StockKg("сир"));
Console.WriteLine($"{a}, {b}, {c}");

// Where: умова, невиконання якої є помилкою
var enough = await StockKg("сир")
    .Where(kg => kg >= 10)
    .ToExceptional();
```

```
16.  Console.WriteLine(enough);
```

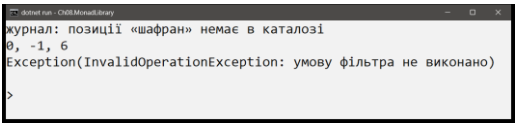


Рисунок 8.8 - Результат виконання прикладу «Task: відновлення»

8.6. Поєднання монад: Task і Either

8.6.1. Постановка задачі

Реальні операції часто перебувають одночасно в кількох контекстах. Звернення до зовнішнього сервісу є асинхронним (*Task*) і може завершитися бізнес-помилкою (*Either*), тож його результат має тип *Task<Either<TL, TR>>*. Функції *Bind* кожного з типів окремо для такого значення незручні: *Bind* для *Task* надає функції значення *Either*, яке доводиться аналізувати вручну. Тому для поєднаного контексту реалізуються власні функції *BindAsync* і *MapAsync*, що «проходять» одразу крізь обидва рівні загортання. У функціональних мовах такі поєднання узагальнюються поняттям **перетворювача монад** (monad transformer); у мові C# їх доцільно реалізовувати для конкретних пар типів.

Розглядається система схвалення заявок на банкет у ресторані. Вимоги до неї такі:

1) заявка перевіряється: відомий гість, від 10 до 120 осіб, вартість меню від 600 грн на особу, дата не раніше ніж за три дні; гість має отримати перелік усіх порушень одразу;

2) зовнішній сервіс повертає рейтинг гостя (0-100); у разі збою сервісу заявка не схвалюється;

3) інший зовнішній сервіс повертає історію бронювань гостя, зокрема кількість неявок;

4) за трьох і більше неявок у заявці відмовляють; інакше розраховується вартість банкету й передоплата: 20 % для гостей з рейтингом від 80, 50 % для решти.

Моделі даних і інтерфейси сервісів наведено в лістингах 8.33 і 8.34. Обидва сервіси повертають *Task<Either<string, T>>*.

Лістинг 8.33 - Моделі даних системи схвалення банкетів

```
1. public sealed record BanquetRequest(int GuestId, int Persons,
2.                                      decimal PricePerPerson,
3.                                      DateOnly Date);
4. public sealed record GuestRating(int GuestId, int Score);  // 0-100
5. public sealed record BookingHistory(int GuestId, int Visits,
6.                                      int NoShows);
7. public sealed record BanquetApproval(int GuestId, decimal Total,
8.                                       decimal Deposit, string Terms);
```

Лістинг 8.34 - Інтерфейси зовнішніх сервісів

```
// Зовнішні сервіси: асинхронні й такі, що можуть не відповісти
public interface IRatingService
{
    Task<Either<string, GuestRating>> GetRating(int guestId);
}

public interface IHistoryService
{
    Task<Either<string, BookingHistory>> GetHistory(int guestId);
}
```

8.6.2. Функції для поєднаного контексту

Функції для типу *Task<Either<TL, TR>>* наведено в лістингу 8.35. *BindAsync* очікує завершення задачі, а потім для варіанта *Right* запускає наступну асинхронну операцію, а для варіанта *Left* повертає вже завершену задачу з тією самою помилкою. Функція *AsTask* загортає синхронне значення *Either* у завершену задачу, щоб синхронні кроки можна було включати в асинхронний ланцюжок.

Лістинг 8.35 - Функції BindAsync, MapAsync і AsTask для Task<Either>

```
// Поєднання з Task: асинхронний крок, що може завершитися
// помилкою, має тип Task<Either<TL, TR>>
public static async Task<Either<TL, TOut>> BindAsync<TL, TIn, TOut>(
    this Task<Either<TL, TIn>> @this,
    Func<TIn, Task<Either<TL, TOut>>> f)
{
    var current = await @this;
    return await current.Match(
        l => Task.FromResult<Either<TL, TOut>>(
             new Either<TL, TOut>.Left(l)),
         f);
}

public static async Task<Either<TL, TOut>> MapAsync<TL, TIn, TOut>(
    this Task<Either<TL, TIn>> @this, Func<TIn, TOut> f) =>
    (await @this).Map(f);

public static Task<Either<TL, TR>> AsTask<TL, TR>(
    this Either<TL, TR> @this) => Task.FromResult(@this);
```

8.6.3. Реалізація бізнес-логіки

Перевірка заявки (лістинг 8.36) використовує масив правил (підрозділ 4.1.2) і збирає всі порушення в одне повідомлення, тож гість дізнається про всі помилки одразу, як в аплікативній перевірці з підрозділу 7.7.3. Рішення щодо заявки (лістинг 8.37) є чистою функцією, що не звертається до сервісів і тому легко тестується.

Лістинг 8.36 - Перевірка заявки на банкет

```
// Перевірка заявки: усі порушені правила збираються разом
static Either<string, BanquetRequest> Validate(BanquetRequest r)
```

```
3. {
4.     (bool Ok, string Error)[] rules =
5.     [
6.         (r.GuestId > 0, "невідомий гість"),
7.         (r.Persons is >= 10 and <= 120, "від 10 до 120 осіб"),
8.         (r.PricePerPerson >= 600m, "меню від 600 грн на особу"),
9.         (r.Date > Today.AddDays(2), "не раніше ніж за 3 дні"),
10.      ];
11.      var errors = rules.Where(x => !x.Ok).Select(x => x.Error);
12.      return errors.Any() ? string.Join("; ", errors) : r;
13.  }
```

Лістинг 8.37 - Рішення щодо заявки

```
1. // Бізнес-правило - чиста функція без звернень до сервісів
2. static Either<string, BanquetApproval> Decide(BanquetRequest r,
3.     GuestRating rating, BookingHistory h)
4. {
5.     if (h.NoShows >= 3)
6.         return $"відмова: {h.NoShows} неявки в історії";
7.     var total = r.Persons * r.PricePerPerson;
8.     var share = rating.Score >= 80 ? 0.2m : 0.5m;
9.     return new BanquetApproval(r.GuestId, total, total * share,
10.          $"передоплата {share:P0}");
11.  }
```

Схвалення заявки є ланцюжком кроків, з'єднаних функцією *BindAsync* (лістинг 8.38): перевірка, отримання рейтингу, отримання історії, рішення. Функція не містить жодної інструкції *if* чи блоку *try/catch*, хоча кожен з чотирьох кроків може завершитися невдачею; перша невдача зупиняє ланцюжок.

Лістинг 8.38 - Послідовне схвалення заявки

```
1. // Послідовний ланцюжок: Either + Task, жодних if і try/catch
2. public Task<Either<string, BanquetApproval>> Approve(
3.     BanquetRequest request) =>
4.     Validate(request).AsTask()
5.         .BindAsync(r => ratings.GetRating(r.GuestId)
6.             .BindAsync(rating => history.GetHistory(r.GuestId)
7.                 .MapAsync(h => (rating, h))))
8.         .BindAsync(x => Decide(request, x.rating, x.h).AsTask());
```

Запити рейтингу та історії в лістингу 8.38 виконуються послідовно, хоча вони незалежні: запит історії не використовує рейтингу. Як зазначено в підрозділі 7.7.4, для незалежних кроків достатньо аплікативного поєднання, яке дає змогу виконувати їх одночасно. Функція *ZipAsync* (лістинг 8.39) приймає дві вже запущені задачі й поєднує їхні результати в кортеж; у лістингу 8.40 її використано для одночасного звернення до обох сервісів.

Лістинг 8.39 - Аплікативне поєднання ZipAsync

```
// Аплікативне поєднання двох НЕЗАЛЕЖНИХ асинхронних кроків:
// обидві задачі вже запущено, тож вони виконуються одночасно
public static async Task<Either<TL, (A, B)>> ZipAsync<TL, A, B>(
    this Task<Either<TL, A>> first, Task<Either<TL, B>> second)
{
    var a = await first;
    var b = await second;
    return a.Bind(x => b.Map(y => (x, y)));
}
```

Лістинг 8.40 - Схвалення заявки з одночасними запитами

```
// Рейтинг та історія незалежні: запити виконуються одночасно
public Task<Either<string, BanquetApproval>> ApproveParallel(
    BanquetRequest request) =>
    Validate(request).AsTask()
        .BindAsync(r => ratings.GetRating(r.GuestId)
            .ZipAsync(history.GetHistory(r.GuestId)))
        .BindAsync(x => Decide(request, x.Item1, x.Item2).AsTask());
```

8.6.4. Обробка результату

Код-споживач перетворює результат на відповідь функцією *Match* (лістинг 8.41). У прикладі формується пара «код стану HTTP - тіло відповіді»: 200 (OK) для схваленої заявки та 400 (Bad Request) для відмови. У вебзастосунку ASP.NET Core з мінімальним API обробник кінцевої точки виконує те саме перетворення, повертаючи з гілок *Match* результати *Results.Ok* і *Results.BadRequest*; бізнес-логіка при цьому не залежить від вебфреймворку.

Лістинг 8.41 - Обробка заявок на банкет

```
var service = new BanquetService(new FakeRatings(),
                                 new FakeHistory());
var date = new DateOnly(2026, 12, 20);
BanquetRequest[] requests =
[
    new(1, 40, 950m, date),                  // постійний гість
    new(2, 25, 700m, date),                  // новий гість
    new(3, 30, 800m, date),                  // неявки
    new(4, 30, 800m, date),                  // збій сервісу
    new(5, 4, 300m, new DateOnly(2026, 11, 18)),
];
foreach (var r in requests)
{
    var result = await service.ApproveParallel(r);
    // Відповідь HTTP: 200 OK або 400 Bad Request
    var (status, body) = result.Match(
        onLeft: e => (400, e),
        onRight: a => (200,
            $"{a.Total:N0} грн, {a.Terms}: {a.Deposit:N0} грн"));
    Console.WriteLine($"гість {r.GuestId}: {status} {body}");
}
```

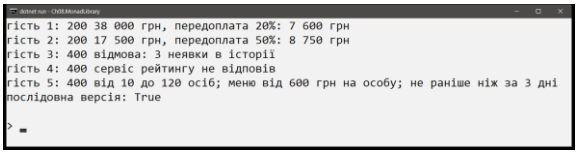


Рисунок 8.9 - Результат виконання прикладу «Банкет»

Для кожної заявки отримано очікуваний результат: схвалення з різною передоплатою, відмову за бізнес-правилом, відмову через збій зовнішнього сервісу та повний перелік порушень для некоректної заявки. Останній рядок підтверджує, що послідовна версія дає той самий результат, що й версія з одночасними запитами.

8.7. Перевірка бібліотеки та типові помилки реалізації

8.7.1. Закони монад

Кожен тип бібліотеки має задовольняти три закони монад (підрозділ 7.7.1). Для *Either* рівність значень забезпечують записи (лістинг 8.42), для структури *Exceptional* - стандартна реалізація методу *Equals* структур, що порівнює поля (лістинг 8.43). Значення *Try* є функціями, рівність яких не можна перевірити безпосередньо, тому порівнюються результати їх виконання (лістинг 8.44). У перевірках використано як успішні, так і помилкові значення.

Лістинг 8.42 - Закони монад для Either

```
1. Func<int, Either<string, int>> f = EitherDemo.CheckCapacity;
2. Func<int, Either<string, int>> g =
3.     n => n % 2 == 0 ? n / 2 : "непарна кількість";
4. Func<int, Either<string, int>> unit = x => x;   // return
5. Either<string, int> m = 12;
6.
7. Console.WriteLine(string.Join(" ",
8.     unit(12).Bind(f) == f(12),                  // ліва
9.     m.Bind(unit) == m,                          // права
10.      m.Bind(f).Bind(g) == m.Bind(x => f(x).Bind(g))));
```

Лістинг 8.43 - Закони монад для Exceptional

```
1. Func<int, Exceptional<int>> h = x => 120 / x;     // 0 -> виняток
2. Func<int, Exceptional<int>> k = x => x + 1;
3. Exceptional<int> e = 5;
4. Exceptional<int> Ret(int x) => x;
5.
6. Console.WriteLine(string.Join(" ",
7.     Ret(5).Bind(h).Equals(h(5)),
8.     e.Bind(Ret).Equals(e),
9.     e.Bind(h).Bind(k).Equals(e.Bind(x => h(x).Bind(k)))));
```

Лістинг 8.44 - Закони монад для Try

```
1. // Try - це функція, тож порівнюються результати виконання
2. Func<int, Try<int>> t1 = x => Try.Of(() => 120 / x);
3. Func<int, Try<int>> t2 = x => Try.Return(x * 2);
4. var tm = Try.Return(0);
5. bool Same<T>(Try<T> a, Try<T> b) =>
6.     a.Run().ToString() == b.Run().ToString();
7.
8. Console.WriteLine(string.Join(" ",
9.     Same(Try.Return(4).Bind(t1), t1(4)),
10.      Same(tm.Bind(Try.Return), tm),
11.      Same(tm.Bind(t1).Bind(t2),
12.           tm.Bind(x => t1(x).Bind(t2)))));
```

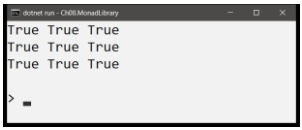


Рисунок 8.10 - Результат виконання прикладу «Закони монад бібліотеки»

Кожен рядок виводу відповідає одному типу й містить результати перевірки лівої тотожності, правої тотожності та асоціативності. У промисловій бібліотеці такі перевірки оформлюють модульними тестами, зокрема тестами на основі властивостей (property-based testing), що перевіряють закони на великій кількості випадково згенерованих значень і функцій.

8.7.2. Типові помилки реалізації

Аналіз поширених реалізацій монад у мові C#, зокрема наведених у книзі C. Bueno, дає змогу виокремити типові помилки.

1) **Відсутність значення, створена конструктором значення.** Якщо варіант «немає значення» *Maybe* створюється викликом конструктора зі значенням *default(T)*, ознака наявності значення встановлюється в *true*, і «порожній» *Maybe* поводиться як такий, що містить значення за замовчуванням. Варіанти мають створюватися окремими конструкторами або бути окремими типами (як у лістингу 7.1), а порівняння зі значенням *default* не повинне означати відсутності значення (підрозділ 7.3).

2) **Втрата накопичення помилок.** Функція перевірки, що повертає результат одразу після першого невдалого правила, зводить *Validation* до звичайного *Either*: користувач дізнається лише про першу помилку. Накопичення забезпечує аплікативне поєднання (підрозділ 7.7.3) або збирання помилок усіх правил (лістинг 8.36).

3) **«Відкладене» обчислення, що виконується негайно.** Якщо функція *Of* виконує обчислення й зберігає результат (лістинг 8.45), тип фактично є *Exceptional*, а не *Try*, і метод *Run* не має сенсу.

4) **Реалізація Map для Task через ContinueWith.** Метод *ContinueWith* (лістинг 8.46) виконує продовження незалежно від результату задачі, зокрема для помилкової задачі; звернення до *t.Result* у продовженні загортає виняток у *AggregateException*, тож код-споживач отримує не той тип винятку, що згенерувала операція. Крім того, продовження виконується в планувальнику, заданому за замовчуванням, а не в контексті синхронізації виклику, і не підтримує скасування. Реалізація через *async/await* (лістинг 8.26) позбавлена цих недоліків.

5) **Властивість Value, що генерує виняток.** Відкрита властивість зі значенням, яка генерує виняток для «неуспішного» варіанта, повертає в код ті самі перевірки і ті самі винятки, від яких монада мала звільнити. Значення слід отримувати лише через *Match* або *GetOrElse*.

Помилки 3 і 4 продемонстровано в лістингу 8.47.

Лістинг 8.45 - Помилкова реалізація «відкладеного» обчислення

```
1. // ПОМИЛКОВА реалізація: обчислення виконується в момент
2. // створення, тож «відкладеного» виконання немає
3. public abstract record EagerTry<T>
4. {
5.     public sealed record Success(T Value) : EagerTry<T>;
6.     public sealed record Failure(Exception Error) : EagerTry<T>;
7.
8.     public static EagerTry<T> Of(Func<T> f)
9.     {
10.          try { return new Success(f()); }
11.          catch (Exception ex) { return new Failure(ex); }
12.      }
13.  }
```

Лістинг 8.46 - Помилкова реалізація Map через ContinueWith

```
1. // ПОМИЛКОВА реалізація Map через ContinueWith: продовження
2. // виконується й для помилкової задачі, а t.Result загортає
3. // виняток в AggregateException
4. static Task<TOut> MapViaContinue<T, TOut>(this Task<T> task,
5.                                           Func<T, TOut> f) =>
6.     task.ContinueWith(t => f(t.Result));
```

Лістинг 8.47 - Демонстрація типових помилок

```
1. var lazy = Try.Of(() =>
2. {
3.     Console.WriteLine("  Try: обчислення");
4.     return 1;
5. });
6. var eager = EagerTry<int>.Of(() =>
7. {
8.     Console.WriteLine("  EagerTry: обчислення");
9.     return 1;
10.  });
11.  Console.WriteLine("Обидва описи створено");
12.
```

```
13.  foreach (var map in new Func<Task<int>, Task<int>>[]
14.           {
15.               t => t.MapViaContinue(x => x + 1),
16.               t => t.Map(x => x + 1),
17.           })
18.  {
19.      try { await map(FailingStock()); }
20.      catch (Exception ex)
21.      {
22.          Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
23.      }
24.  }
```

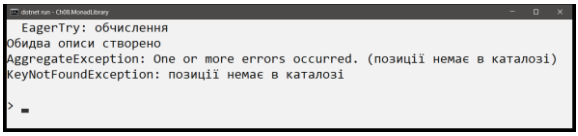


Рисунок 8.11 - Результат виконання прикладу «Типові помилки»

Обчислення *EagerTry* виконано ще до рядка «Обидва описи створено», тоді як обчислення *Try* не виконано зовсім, оскільки *Run* не викликався. Реалізація *Map* через *ContinueWith* замінила вихідний виняток *KeyNotFoundException* на *AggregateException*, а реалізація через *async/await* зберегла його.

8.8. Висновки

1. Бібліотека монад - це набір узагальнених типів, кожен з яких описує окремий обчислювальний контекст, з однаковим набором операцій: загортання, *Map*, *Bind*, *Match*, *GetOrElse*, *Select* і *SelectMany*.

2. Типи бібліотеки мають бути незмінними, мати закриту ієрархію варіантів, надавати доступ до значення лише через явну обробку всіх варіантів і задовольняти закони монад.

3. Монада *Either* подає результат або інформативну помилку; її функція *Bind* реалізує коротке замикання: перша помилка зупиняє ланцюжок і передається до його кінця.

4. Структура *Exceptional<T>* подає результат уже виконаного обчислення - значення або виняток; функції *Map* і *Bind* перехоплюють винятки, а значення за замовчуванням структури потребує окремої обробки.

5. Монада *Try* подає відкладене обчислення: *Map* і *Bind* лише будують опис, а виконання відбувається під час кожного виклику *Run*.

6. Стандартний тип *Task<T>* доповнюється монадичними операціями методами-розширеннями; їх слід реалізовувати через *async/await*, а не через *ContinueWith*.

7. Для поєднаних контекстів (*Task<Either<TL, TR>>*) реалізуються власні функції зв'язування *BindAsync* і *MapAsync*; незалежні асинхронні

кроки доцільно поєднувати аплікативно (*ZipAsync*) для одночасного виконання.

8. Закони монад для власних типів перевіряються безпосередньо порівнянням значень, а для типів-функцій - порівнянням результатів їх виконання.

8.9. Контрольні запитання

1. Які типи входять до бібліотеки монад, розробленої в розділах 7-8? Який обчислювальний контекст описує кожен з них?

2. Сформулюйте принципи проєктування бібліотеки монад. Для чого базовий тип варіантів має приватний конструктор?

3. Чим монада *Either* відрізняється від *Maybe*? Що зберігають варіанти *Left* і *Right*?

4. Що означає коротке замикання в ланцюжку *Bind*?

5. Чому функції *Map* і *Bind* для *Either* не перехоплюють винятків, а для *Exceptional* перехоплюють?

6. Які переваги та недоліки має реалізація *Exceptional<T>* структурою? Як обробляється значення *default*?

7. Чим *Try<T>* відрізняється від *Exceptional<T>*? Коли виконується обчислення, описане *Try*?

8. Як за допомогою *Map* і *Apply* застосувати функцію двох аргументів до двох значень у контексті *Exceptional*?

9. Яке призначення функцій *Recover*, *OrElse* і *Where* для *Task<T>*?

10. Чому реалізація *Map* для *Task<T>* через *ContinueWith* є помилковою?

11. Для чого потрібні функції *BindAsync* і *MapAsync* для типу *Task<Either<TL, TR>>*? Що таке перетворювач монад?

12. У яких випадках асинхронні кроки слід поєднувати аплікативно, а не монадично?

8.10. Практичні завдання

1. Доповніть монаду *Either* функціями *Tap* (виконання дії для успішного значення без зміни результату, підрозділ 5.8) і *Ensure* (перетворення успішного значення на помилку, якщо не виконано умову). Застосуйте їх у ланцюжку оформлення замовлення столика.

2. Замініть рядковий тип помилки в системі схвалення банкетів записом *BanquetError* з кодом і повідомленням. Використайте *MapLeft* для формування тексту відповіді та різних кодів стану HTTP залежно від коду помилки.

3. Реалізуйте функцію *ToEither* для перетворення *Exceptional<T>* на *Either<string, T>* і функцію *ToMaybe* для перетворення *Either<TL, TR>* на *Maybe<TR>*. Перевірте закони функтора для *Map* кожного з типів.

4. Реалізуйте функцію *Retry(int attempts)* для монади *Try*, що повторно виконує обчислення в разі винятку не більше заданої кількості разів. Перевірте її на імітації сервісу постачальника, що відповідає лише з третьої спроби.

5. Реалізуйте монаду *Writer<T>*, що разом зі значенням накопичує записи журналу (перелік рядків), з функціями *Map*, *Bind*, *Select* і *SelectMany*. Опишіть з її допомогою розрахунок рахунку, кожен крок якого додає запис до журналу. Перевірте закони монад.

9. Каррінг і часткове застосування

Каррінг (currying) і **часткове застосування** (partial application) - дві споріднені функціональні техніки, що походять з математичної логіки. Каррінг названо на честь американського математика Х. Б. Каррі (див. підрозділ 1.4), у працях якого з комбінаторної логіки ця техніка дістала систематичний розвиток.

Суть каррінгу зручно пояснити на прикладі. Нехай функція *Add(a, b)* обчислює суму двох чисел. Виклик *Add(100, 200)* дає 300. Якщо ж функцію каррировано, то виклик із одним аргументом *Add(100)* повертає не число, а **нову функцію**, у якій перший параметр зафіксовано зі значенням 100; ця функція додає 100 до будь-якого переданого їй числа. Отже, з однієї базової функції можна отримати багато спеціалізованих. Цей механізм зовнішньо нагадує успадкування в ООП, проте за суттю відрізняється від нього: логіку містить лише одна базова функція, а похідні функції є лише посиланнями на неї із зафіксованими значеннями частини параметрів.

Мета розділу - сформувати розуміння каррінгу та часткового застосування і навички їх реалізації в мові C#.

Після опрацювання розділу студент повинен:

- знати означення каррінгу та часткового застосування й розуміти відмінність між ними;
- уміти отримувати спеціалізовані функції з універсальної функції з багатьма параметрами;
- уміти реалізовувати каррінг у мові C# трьома способами: «вбудовано», статичним класом і методами-розширеннями;
- уміти реалізовувати методи часткового застосування;
- оцінювати доцільність застосування цих технік у мові C#.

Приклади розділу містяться в проєкті *Ch09.Currying*; методи *Curry* і *Partial* - у файлі *Functional/Currying.cs* спільної бібліотеки.

9.1. Поняття каррінгу

У лістингу 9.1 функцію додавання збору до ціни записано одразу в каррованій формі: функція приймає розмір збору і повертає функцію, що додає цей збір до ціни.

Лістинг 9.1 - Каррована функція додавання збору

```
1. // Каррована функція «вбудовано»: дві стрілки,
2. // тип - Func<decimal, Func<decimal, decimal>>
3. var addFee = (decimal fee) => (decimal price) => price + fee;
4.
5. var withDelivery = addFee(60m);  // доставка 60 грн
6. var a = withDelivery(475m);      // 535
7. var b = withDelivery(0m);        // 60
8. var c = withDelivery(1240m);     // 1300
```



Рисунок 9.1 - Результат виконання прикладу «Вбудований каррінг»

Функція *withDelivery* є спеціалізованою версією *addFee* із зафіксованим розміром збору 60 грн.

Каррінг - це перетворення функції від *n* аргументів на послідовність із *n* функцій, кожна з яких приймає один аргумент (такі функції називають **унарними**, unary) і повертає наступну функцію послідовності; остання функція повертає кінцевий результат. Формально функція типу *(A, B, C) -> R* перетворюється на функцію типу *A -> (B -> (C -> R))*.

9.2. Каррінг функцій з багатьма параметрами

Для функції з двома параметрами можливі лише два варіанти: передати перший параметр і отримати функцію або передати обидва параметри і отримати значення. Переваги каррінгу повніше виявляються для функцій з більшою кількістю параметрів.

Нехай ресторан отримує прайс-листи від постачальників у текстовому форматі CSV. Функція розбору прайс-листа одного конкретного формату наведена в лістингу 9.2.

Лістинг 9.2 - Розбір прайс-листа фіксованого формату

```
1. // Розбір прайс-листа одного конкретного формату
2. public static IEnumerable<PriceItem> ParseFarm(string text) =>
3.     text.Split("\r\n")
4.         .Skip(1) // пропустити заголовок
5.         .Select(x => x.Split(','))
6.         .Select(x => new PriceItem(x[0], x[1], decimal.Parse(x[2],
7.             System.Globalization.CultureInfo.InvariantCulture)));
```

Проте формати постачальників відрізняються: «Агроферма» надсилає файли із заголовком, розділювачем рядків Windows (*\r\n*) і комами між полями; «Молочар» використовує вертикальну риску як розділювач полів (оскільки назви товарів можуть містити коми); «Гриби Карпат» надсилає файли без заголовка, з розділювачем рядків Linux (*\n*). Створення трьох майже однакових функцій призвело б до дублювання коду. Доцільніше винести всі змінні аспекти формату в параметри (лістинг 9.3).

Лістинг 9.3 - Узагальнена функція розбору прайс-листа

```
1. public static IEnumerable<PriceItem> ParsePriceList(
2.     bool skipHeader,
3.     string lineBreak,
4.     string delimiter,
5.     string text) =>
6.     text.Split(lineBreak)
7.         .Skip(skipHeader ? 1 : 0)
8.         .Select(x => x.Split(delimiter))
9.         .Select(x => new PriceItem(x[0], x[1], decimal.Parse(x[2],
10.              System.Globalization.CultureInfo.InvariantCulture)));
```

У нефункціональному стилі для кожного формату доводиться передавати всі параметри (лістинг 9.4).

Лістинг 9.4 - Виклик функції з повним набором параметрів

```
1. var farm = ParsePriceList(true, "\r\n", ",", SupplierData.Farm);
2. var dairy = ParsePriceList(true, "\r\n", "|", SupplierData.Dairy);
3. var mushrooms = ParsePriceList(false, "\n", ",",
4.                                SupplierData.Mushrooms);
```

Каррінг дає змогу передавати параметри по одному. Кожен виклик каррованої функції повертає або нову функцію з на один параметр меншою кількістю параметрів, або кінцеве значення, якщо передано всі параметри (лістинг 9.5).

Лістинг 9.5 - Поступова спеціалізація каррованої функції

```
1. var curriedParse = Parse.Curry();
2.
3. // три параметри, що залишилися: string, string, string
4. var withHeader = curriedParse(true);
5. var noHeader = curriedParse(false);
6.
7. // два параметри
8. var withHeaderWindows = withHeader("\r\n");
9. var noHeaderLinux = noHeader("\n");
10.
11.  // один параметр
12.  var withHeaderWindowsComma = withHeaderWindows(",");
13.  var withHeaderWindowsPipe = withHeaderWindows("|");
14.  var noHeaderLinuxComma = noHeaderLinux(",");
15.
16.  // дані
17.  var farm = withHeaderWindowsComma(SupplierData.Farm);
18.  var dairy = withHeaderWindowsPipe(SupplierData.Dairy);
19.  var mushrooms = noHeaderLinuxComma(SupplierData.Mushrooms);
```

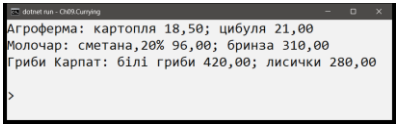


Рисунок 9.2 - Результат виконання прикладу «Карровані виклики» (фрагмент)

Поступово будується вузькоспеціалізована функція, що приймає лише текст прайс-листа, а всі проміжні функції зберігаються для повторного використання. Функції утворюють своєрідне «генеалогічне дерево», у якому кожен вибір значення параметра є розгалуженням (рис. 9.3).

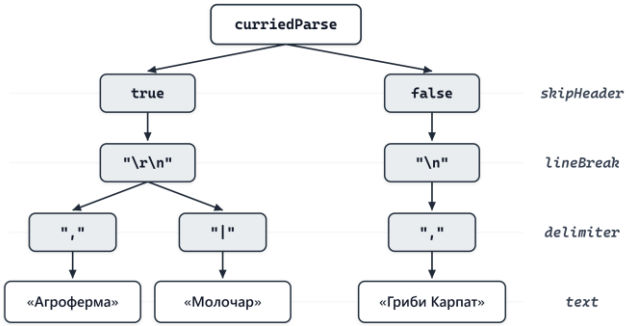


Рисунок 9.3 - Дерево функцій, отриманих каррінгом функції розбору прайс-листа

За потреби всі аргументи можна передати одним виразом послідовних викликів (лістинг 9.6).

Лістинг 9.6 - Послідовна передача всіх аргументів

```
1. var dairy2 = curriedParse(true)("\r\n")("|")(SupplierData.Dairy);
```

Ще один практичний приклад - отримання спеціалізованих функцій журналювання з однієї загальної функції (лістинги 9.7 і 9.8). Для каррінгу дії без результату використовується перевантаження методу *Curry* для делегата *Action*.

Лістинг 9.7 - Загальна функція журналювання

```
1. private static void Log(LogLevel level, string message) =>
2.     Console.WriteLine($"[{level}] {message}");
```

Лістинг 9.8 - Спеціалізовані функції журналювання

```
1. Action<LogLevel, string> logger = Log;
2.
3. var curriedLogger = logger.Curry();
4. var logInfo = curriedLogger(LogLevel.Info);
5. var logWarning = curriedLogger(LogLevel.Warning);
6. var logError = curriedLogger(LogLevel.Error);
7.
8. logInfo("зміну відкрито");
9. logWarning("закінчується борошно");
```

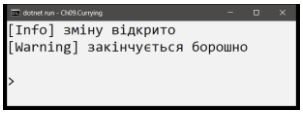


Рисунок 9.4 - Результат виконання прикладу «Журнал»

Отже, каррінг має такі корисні властивості:

- з однієї функції отримується кілька готових до використання варіантів, які можна передавати як значення;
- усі проміжні функції доступні для безпосереднього використання або для побудови нових функцій.

9.3. Каррінг і функції вищого порядку

Каррінг корисний і в поєднанні з функціями вищого порядку. Якщо мати карровані версії арифметичних операцій (лістинг 9.9), то частково застосовані операції стають готовими «будівельними блоками» для ланцюжка *Map* (лістинг 9.10).

Лістинг 9.9 - Карровані арифметичні операції

```
1. var add = F.Curry((decimal x, decimal y) => x + y);
2. var subtract = F.Curry((decimal x, decimal y) => y - x);
3. var multiply = F.Curry((decimal x, decimal y) => x * y);
```

Лістинг 9.10 - Розрахунок ціни страви ланцюжком каррованих операцій

```
1. // Ціна страви: + ПДВ 20 % -> + сервісний збір 10 %
2. // -> - знижка 50 грн -> округлення
3. var finalPrice = (decimal net) =>
4.     net.Map(multiply(1.20m))
5.        .Map(multiply(1.10m))
6.        .Map(subtract(50m))
7.        .Map(x => Math.Round(x, 2));
```

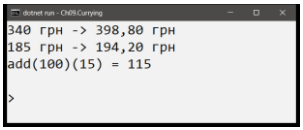


Рисунок 9.5 - Результат виконання прикладу «Арифметика»

Слід звернути увагу на порядок параметрів у функції *subtract*: зменшуване передається другим, щоб частково застосована функція *subtract(50m)* віднімала 50 від переданого їй значення. Вибір порядку параметрів, за якого найчастіше фіксовані параметри стоять першими, є важливим принципом проєктування функцій, призначених для каррінгу.

9.4. Реалізація каррінгу в .NET

У функціональних мовах, зокрема в F# і Haskell, усі функції є каррованими автоматично. У мові C# така можливість відсутня, проте каррінг можна реалізувати трьома способами.

Перший спосіб - одразу оголошувати функцію в каррованій формі, як у лістингу 9.1. Такий делегат має дві стрілки і тип *Func<decimal, Func<decimal, decimal>>*. Починаючи з версії C# 10, тип лямбда-виразу виводиться автоматично (natural type), тож можна використовувати ключове слово *var*; у ранніх версіях тип делегата потрібно вказувати явно. Недолік способу - функцію завжди доводиться викликати в каррованій формі.

Другий спосіб - статичний клас із методами *Curry* для функцій із різною кількістю параметрів (лістинг 9.11).

Лістинг 9.11 - Статичний клас для каррінгу

```
1. public static class F
2. {
3.     public static Func<T1, Func<T2, TOut>> Curry<T1, T2, TOut>(
4.         Func<T1, T2, TOut> f) =>
5.         x => y => f(x, y);
6.
7.     public static Func<T1, Func<T2, Func<T3, TOut>>>
8.         Curry<T1, T2, T3, TOut>(Func<T1, T2, T3, TOut> f) =>
9.         x => y => z => f(x, y, z);
10.  }
```

Метод фактично розміщує шари делегатів між викликом кінцевої функції та кодом, що її використовує. Недоліками є необхідність окремого методу для кожної кількості параметрів і необхідність явно вказувати типи параметрів лямбда-виразу, оскільки компілятор не може вивести їх із контексту (див. лістинг 9.9).

Третій спосіб - методи-розширення для делегатів (лістинг 9.12). Це найзручніший варіант: громіздкі оголошення зосереджуються в одному файлі бібліотеки, а використання є стислим (лістинг 9.13).

Лістинг 9.12 - Методи-розширення для каррінгу

```
1. public static class CurryExtensions
2. {
3.     public static Func<T1, Func<T2, TOut>> Curry<T1, T2, TOut>(
4.         this Func<T1, T2, TOut> @this) =>
5.         x => y => @this(x, y);
6.
7.     public static Func<T1, Func<T2, Func<T3, TOut>>>
8.         Curry<T1, T2, T3, TOut>(this Func<T1, T2, T3, TOut> @this) =>
9.         x => y => z => @this(x, y, z);
10.
11.      public static Func<T1, Func<T2, Func<T3, Func<T4, TOut>>>>
12.          Curry<T1, T2, T3, T4, TOut>(
13.              this Func<T1, T2, T3, T4, TOut> @this) =>
14.          x => y => z => w => @this(x, y, z, w);
```

```
15.
16.      // Каррінг дії (Action) з двома параметрами
17.      public static Func<T1, Action<T2>> Curry<T1, T2>(
18.          this Action<T1, T2> @this) =>
19.          x => y => @this(x, y);
20.  }
```

Лістинг 9.13 - Використання методу-розширення Curry

```
1. // Лямбда-вираз спочатку зберігається в змінній-делегаті
2. var add = (decimal x, decimal y) => x + y;
3. var curriedAdd = add.Curry();
4. var add10 = curriedAdd(10m);
5. var answer = add10(100m);
6. // answer = 110
```



Рисунок 9.6 - Результат виконання прикладу «Метод-розширення Curry»

Метод-розширення можна застосувати лише до змінної типу делегата, тому лямбда-вираз спочатку присвоюється змінній (див. також підрозділ 5.6).

9.5. Часткове застосування

Часткове застосування подібне до каррінгу, і ці терміни нерідко (некоректно) вживають як синоніми. Відмінність полягає в такому: каррінг завжди перетворює функцію на послідовність **унарних** функцій, тоді як **часткове застосування** - це фіксація довільної кількості аргументів функції за один крок з отриманням функції від решти аргументів. Значення результату обчислюється, коли передано всі аргументи.

У прикладі з прайс-листами формат «Гриби Карпат» використовується лише одним постачальником, тож створювати проміжні функції для кожного його параметра немає потреби. Водночас перші два параметри для «Агроферми» та «Молочара» спільні. Часткове застосування дає змогу зафіксувати потрібну кількість параметрів за один виклик (лістинг 9.14).

Лістинг 9.14 - Часткове застосування функції розбору

```
1. var noHeaderLinuxComma = Parse.Partial(false, "\n", ",");
2. var withHeaderWindows = Parse.Partial(true, "\r\n");
3. var withHeaderWindowsComma = withHeaderWindows.Partial(",");
4. var withHeaderWindowsPipe = withHeaderWindows.Partial("|");
5.
6. var farm = withHeaderWindowsComma(SupplierData.Farm);
7. var dairy = withHeaderWindowsPipe(SupplierData.Dairy);
8. var mushrooms = noHeaderLinuxComma(SupplierData.Mushrooms);
```

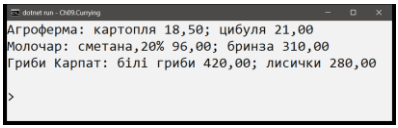


Рисунок 9.7 - Результат виконання прикладу «Часткове застосування»

Такий код стисліший за каррований варіант і водночас зберігає можливість повторного використання проміжних функцій.

9.6. Реалізація часткового застосування в .NET

Елегантної загальної реалізації часткового застосування в мові C# не існує: для кожного поєднання кількості вхідних і вихідних параметрів потрібен окремий метод-розширення. Для прикладу з лістингу 9.14 необхідні три методи: «з чотирьох параметрів до одного», «з чотирьох параметрів до двох», «з двох параметрів до одного» (лістинг 9.15).

Лістинг 9.15 - Методи-розширення для часткового застосування

```
1. public static class PartialExtensions
2. {
3.     // 4 параметри -> 1
4.     public static Func<T4, TOut> Partial<T1, T2, T3, T4, TOut>(
5.         this Func<T1, T2, T3, T4, TOut> f, T1 one, T2 two, T3 three) =>
6.         four => f(one, two, three, four);
7.
8.     // 4 параметри -> 2
9.     public static Func<T3, T4, TOut> Partial<T1, T2, T3, T4, TOut>(
10.         this Func<T1, T2, T3, T4, TOut> f, T1 one, T2 two) =>
11.         (three, four) => f(one, two, three, four);
12.
13.     // 2 параметри -> 1
14.     public static Func<T2, TOut> Partial<T1, T2, TOut>(
15.         this Func<T1, T2, TOut> f, T1 one) =>
16.         two => f(one, two);
17. }
```

Такі методи можна додавати до бібліотеки поступово, за потребою, або одноразово створити повний набір для типових комбінацій.

9.7. Висновки

1. Каррінг - це перетворення функції від *n* аргументів на послідовність *n* унарних функцій; кожен виклик каррованої функції повертає нову функцію або, після передачі всіх аргументів, кінцеве значення.

2. Каррінг дає змогу отримувати з однієї універсальної функції множину спеціалізованих функцій без дублювання логіки; проміжні функції доступні для повторного використання.

3. Часткове застосування фіксує довільну кількість аргументів за один крок; на відміну від каррінгу, результат не обов'язково є унарною функцією.

4. У мові C# каррінг і часткове застосування не підтримуються на рівні мови і реалізуються «вбудованими» каррованими лямбда-виразами, статичними класами або методами-розширеннями для делегатів; останній спосіб є найзручнішим.

5. Реалізація цих технік потребує певного обсягу шаблонного коду, а компілятор не завжди здатен вивести типи параметрів лямбда-виразів. Тому в мові C# каррінг і часткове застосування є корисними, але не обов'язковими засобами функціонального стилю.

9.8. Контрольні запитання

1. Що таке каррінг? На честь кого названо цю техніку?

2. Що повертає каррована функція, якщо їй передано не всі аргументи?

3. Чим каррінг відрізняється від часткового застосування?

4. Який тип має каррована функція двох аргументів типу *decimal*, що повертає *decimal*?

5. Як каррінг допомагає уникнути дублювання коду під час розбору даних різних форматів?

6. Чому порядок параметрів має значення для функцій, призначених для каррінгу?

7. Які способи реалізації каррінгу в мові C# існують? Які переваги та недоліки має кожен із них?

8. Чому метод-розширення *Curry* не можна застосувати безпосередньо до лямбда-виразу?

9. Чому реалізація часткового застосування в C# потребує багатьох методів-розширень?

9.9. Практичні завдання

1. Реалізуйте функцію розрахунку вартості проживання з параметрами: ставка туристичного збору, знижка постійного гостя, ціна за ніч, кількість ночей. За допомогою каррінгу отримайте спеціалізовані функції для двох готелів мережі та для постійних і нових гостей.

2. Реалізуйте каррований форматувальник рядка меню (валюта, кількість знаків після коми, ширина назви, позиція меню) і отримайте з нього функції для друкованого меню та для екрана кіоску.

3. Доповніть бібліотеку методами *Curry* для делегатів *Func* з п'ятьма параметрами та *Action* з трьома параметрами.

4. Реалізуйте методи часткового застосування «з трьох параметрів до одного» і «з трьох параметрів до двох» та застосуйте їх до функції розрахунку чайових (ставка, мінімальна сума, сума рахунку).

5. За допомогою каррованих арифметичних операцій і функції *Map* побудуйте розрахунок вартості банкету: ціна за персону × кількість гостей + оренда залу - знижка.

10. Нескінченні цикли

У попередніх розділах показано, як функціональне програмування замінює цикли *for* і *foreach* методами LINQ (*Select*, *Aggregate* тощо). Такий підхід ефективний, якщо перебирається колекція відомої довжини або послідовність, яка сама визначає момент завершення. Проте в багатьох задачах кількість ітерацій наперед невідома: обчислення має тривати, доки не буде виконано певну умову. У цьому розділі розглядаються функціональні способи організації таких **невизначених циклів** (indefinite loops) у мові C# - рекурсія, трамплінінг і власні ітератори - та компроміси, пов'язані з кожним із них.

Мета розділу - сформувати навички реалізації невизначених циклів у функціональному стилі та вміння обирати спосіб реалізації з урахуванням обмежень мови C#.

Після опрацювання розділу студент повинен:

- розуміти обмеження рекурсії в мові C# і поняття хвостової рекурсії;
- уміти реалізовувати невизначений цикл за допомогою методу *IterateUntil* і трампліна;
- розуміти будову перелічувача *IEnumerator<T>* і вміти реалізовувати власні перелічувачі та блоки ітераторів;
- уміти обробляти послідовності станів, отримані власним ітератором, методами LINQ.

Приклади розділу містяться в проєкті *Ch10.IndefiniteLoops*.

10.1. Постановка задачі

Як навчальний приклад розглядається модель мотиваційної програми для офіціантів ресторану. Правила моделі такі:

- шкала програми містить 100 позицій; усі учасники починають з позиції 1, мета - досягти позиції 100;
- учасники ходять по черзі; за хід учасник просувається на випадкову кількість позицій від 1 до 6 (умовна кількість продажів за зміну);
- на шкалі є особливі пороги: досягнення порогу-бонусу (наприклад, обслуговування VIP-гостя) переносить учасника вперед, а порогу-штрафу (наприклад, скарга гостя) - назад;
- учасник, що отримав максимальний результат 6, ходить ще раз;
- переможцем є учасник, який першим досягне позиції 100.

Механіка моделі збігається з механікою класичних настільних ігор «на доріжці», у яких перехід на клітинку переносить фішку вперед або назад. Бонуси та штрафи не обов'язково розрізняти в коді: обидва переносять учасника з позиції x на позицію y і різняться лише напрямком. Тож їх можна описати одним словником (лістинг 10.1, рис. 10.1).

Лістинг 10.1 - Особливі пороги мотиваційної програми

```
// Особливі пороги мотиваційної програми:
// бонус переносить учасника вперед, штраф - назад
public static readonly IReadOnlyDictionary<int, int> Jumps =
    new Dictionary<int, int>
    {
        { 5, 15 },   // відгук «відмінно» від гостя
        { 8, 34 },   // обслуговування банкету
        { 17, 4 },   // скарга гостя
        { 19, 60 },  // VIP-гість
          { 27, 15 },  // помилка в рахунку
          { 36, 16 },  // запізнення на зміну
          { 48, 70 },  // продаж дегустаційного сету
          { 53, 30 },  // розбитий посуд
          { 63, 92 },  // відзнака керівника
          { 64, 22 },  // порушення стандартів
          { 80, 44 },  // повернення страви
          { 91, 69 }   // негативний відгук онлайн
     };
```

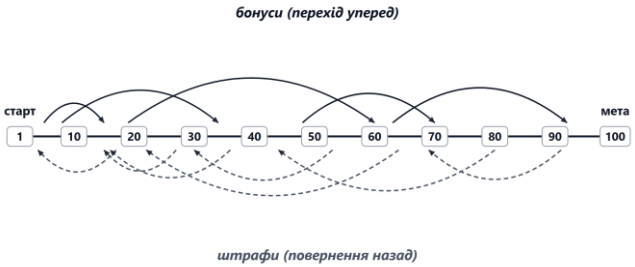


Рисунок 10.1 - Пороги мотиваційної програми: бонуси та штрафи

Джерело випадкових чисел приховано за інтерфейсом (лістинг 10.2). Це дає змогу підставляти в тестах детерміновану реалізацію; у прикладах розділу використовується генератор із фіксованим початковим значенням, тож усі реалізації дають відтворюваний результат.

Лістинг 10.2 - Інтерфейс джерела випадкових значень

```
// Джерело випадкової кількості продажів за хід (1-6)
public interface ISalesRoll { int Roll(); }

// Відтворювана послідовність для навчальних прикладів
public sealed class SeededRoll(int seed) : ISalesRoll
{
    private readonly Random random = new(seed);
    public int Roll() => random.Next(1, 7);
}
```

Імперативна реалізація наведена в лістингу 10.3.

Лістинг 10.3 - Імперативна реалізація моделі

```
public static class ImperativeContest
```

```
2. {
3.     public static int Play(int noPlayers, ISalesRoll die)
4.     {
5.         var current = 1;
6.         var points = new Dictionary<int, int>();
7.         for (var i = 1; i <= noPlayers; i++)
8.             points.Add(i, 1);
9.
10.          while (!points.Any(x => x.Value >= 100))
11.          {
12.              var roll = die.Roll();
13.              points[current] += roll;
14.              if (ContestRules.Jumps.ContainsKey(points[current]))
15.                  points[current] = ContestRules.Jumps[points[current]];
16.              // за «6» учасник ходить ще раз
17.              if (roll == 6) continue;
18.              current += 1;
19.              if (current > noPlayers)
20.                  current = 1;
21.          }
22.          return points.First(x => x.Value >= 100).Key;
23.      }
24.  }
```

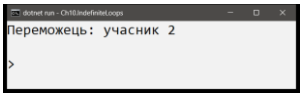


Рисунок 10.2 - Результат виконання прикладу «Імперативний варіант»

Методом *Select* таку логіку реалізувати неможливо: невідомо, коли буде виконано умову завершення, і цикл *while* повторюється, доки вона не виконається. Цикл *while* є інструкцією керування потоком і не відповідає функціональному стилю. Існує кілька способів його заміни, кожен з яких є певним компромісом.

10.2. Рекурсія

Класичним функціональним способом організації невизначеного циклу є **рекурсія**: функція викликає саму себе, а умова визначає, чи потрібна наступна ітерація, чи слід повернути результат. Якщо рекурсивний виклик є останньою операцією функції (його результат повертається без подальших обчислень), рекурсію називають **хвостовою** (tail recursion).

Стан моделі описується незмінними записами (лістинг 10.4), а логіку одного ходу - чистими функціями (лістинг 10.5).

Лістинг 10.4 - Незмінний стан моделі

```
1. public sealed record Player(int Number, int Points);
2.
3. public sealed record GameState(IReadOnlyList<Player> Players,
4.                                int CurrentPlayer, int Turn);
```

Лістинг 10.5 - Функції обчислення наступного стану

```
1. public static Player UpdatePlayer(Player p, int roll)
2. {
3.     var afterRoll = p with { Points = p.Points + roll };
4.     return afterRoll with
5.     {
6.         Points = ContestRules.Jumps.TryGetValue(afterRoll.Points,
7.                                                 out var target)
8.             ? target
9.             : afterRoll.Points
10.     };
11. }
12.
13. public static GameState NextTurn(GameState g, int roll) =>
14.     g with
15.     {
16.         Turn = g.Turn + 1,
17.         CurrentPlayer = roll == 6
18.             ? g.CurrentPlayer
19.             : g.CurrentPlayer % g.Players.Count + 1,
20.         Players = g.Players
21.             .Select(x => x.Number == g.CurrentPlayer
22.                 ? UpdatePlayer(x, roll)
23.                 : x)
24.             .ToArray()
25.     };
26.
27. public static bool IsFinished(GameState g) =>
28.     g.Players.Any(x => x.Points >= 100);
29.
30. public static GameState Start(int noPlayers) =>
31.     new(Enumerable.Range(1, noPlayers)
32.                   .Select(n => new Player(n, 1)).ToArray(), 1, 0);
33.
34. public static int Winner(GameState g) =>
35.     g.Players.First(x => x.Points >= 100).Number;
```

Рекурсивна реалізація циклу наведена в лістингу 10.6; рекурсивний виклик у ній є хвостовим.

Лістинг 10.6 - Рекурсивна реалізація моделі

```
1. public static GameState PlayRecursive(GameState g, ISalesRoll die)
2. {
3.     var next = NextTurn(g, die.Roll());
4.     return IsFinished(next)
5.         ? next
6.         : PlayRecursive(next, die);
7. }
```

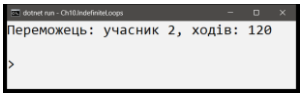


Рисунок 10.3 - Результат виконання прикладу «Рекурсія»

Проблема полягає в тому, що кожен вкладений виклик займає новий кадр стека середовища виконання .NET. За великої кількості ітерацій це знижує продуктивність або призводить до аварійного завершення програми через виняток *StackOverflowException*, який неможливо перехопити. Якщо гарантовано, що ітерацій буде небагато, рекурсивний підхід прийнятний. Проте ця гарантія має переглядатися за кожної зміни сценаріїв використання: функція, що сьогодні виконує кілька ітерацій, згодом може виконувати їх сотні тисяч.

Примітка. У мові F# та багатьох інших функціональних мовах реалізовано **оптимізацію хвостових викликів** (tail call optimization): хвостова рекурсія компілюється в цикл або в інструкцію переходу проміжного коду, тож стек не зростає. Середовище CLR підтримує хвостові виклики (префікс *tail.* у проміжній мові), однак компілятор C# їх не генерує, і планів змінити це немає.

10.3. Трамплінінг

10.3.1. Прихований цикл: IterateUntil

Найпрагматичніший спосіб - інкапсулювати цикл *while* в одному узагальненому методі-розширенні, приховавши його в бібліотеці (лістинг 10.7). Метод приймає функцію оновлення стану та умову завершення.

Лістинг 10.7 - Метод IterateUntil на основі циклу while

```
1. public static class LoopExtensions
2. {
3.     public static T IterateUntil<T>(this T @this,
4.                                     Func<T, T> update,
5.                                     Func<T, bool> endCondition)
6.     {
7.         var current = @this;
8.         while (!endCondition(current))
9.         {
10.              current = update(current);
11.          }
12.          return current;
13.      }
14.  }
```

Оскільки використовується звичайний цикл, проблеми переповнення стека немає. Код не є чисто функціональним, проте імперативна конструкція присутня лише в одному місці бібліотеки. Якщо в майбутньому мова отримає оптимізацію хвостових викликів, реалізацію методу можна буде змінити без змін у коді, що його використовує (лістинг 10.8).

Лістинг 10.8 - Моделювання за допомогою IterateUntil

```
1. var final = Start(3).IterateUntil(
2.     g => NextTurn(g, die.Roll()),
3.     IsFinished);
```

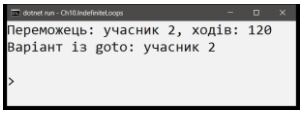


Рисунок 10.4 - Результат виконання прикладу «IterateUntil»

Той самий метод можна реалізувати без циклу *while*, за допомогою інструкції *goto* з мітками (лістинг 10.9). За ефективністю варіанти рівноцінні; вибір між ними є питанням стилю. Слід наголосити, що поза такими ретельно ізольованими випадками інструкцію *goto* використовувати не рекомендується: довільні переходи порушують передбачуваність порядку виконання і ускладнюють налагодження.

Лістинг 10.9 - Метод IterateUntil на основі goto

```
1. public static class GotoLoopExtensions
2. {
3.     public static T IterateUntilGoto<T>(this T @this,
4.                                         Func<T, T> update,
5.                                         Func<T, bool> endCondition)
6.     {
7.         var current = @this;
8.     LoopBeginning:
9.         if (endCondition(current))
10.              goto LoopEnding;
11.          current = update(current);
12.          goto LoopBeginning;
13.      LoopEnding:
14.          return current;
15.      }
16.  }
```

10.3.2. Трамплін

Термін **трамплін** (trampoline) у функціональному програмуванні має точніший зміст. Це техніка виконання рекурсивних алгоритмів без зростання стека: замість рекурсивного виклику функція повертає опис наступного кроку - thunk, тобто відкладене обчислення (див. підрозділ 5.2). Зовнішній цикл-«трамплін» по черзі виконує такі кроки, доки не буде отримано кінцевий результат. Техніка відома з 1990-х років і застосовувалася, зокрема, під час реалізації мов сімейства Lisp мовою C. Відмінність між звичайною рекурсією та трампліном за використанням стека показано на рис. 10.5: рекурсивні виклики накопичують кадри стека, доки не буде досягнуто умови завершення, тоді як трамплін виконує кожен крок в одному й тому самому кадрі методу *Run*.

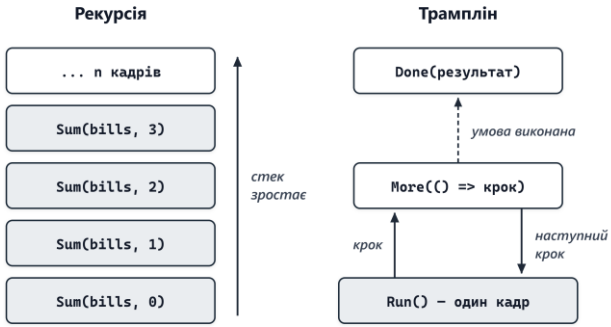


Рисунок 10.5 - Використання стека рекурсією та трампліном

Реалізація наведена в лістингу 10.10: крок обчислення є дискримінованим об'єднанням з варіантами *Done* (результат) і *More* (відкладений наступний крок), а метод *Run* виконує кроки в циклі.

Лістинг 10.10 - Реалізація трампліна

```
// Трамплін: крок обчислення або завершений (Done),
// або повертає відкладений наступний крок (More)
public abstract record Trampoline<T>
{
    private Trampoline() { }

    public sealed record Done(T Result) : Trampoline<T>;
    public sealed record More(Func<Trampoline<T>> Next) : Trampoline<T>;
}

public static class TrampolineExtensions
{
    public static T Run<T>(this Trampoline<T> start)
    {
        var step = start;
        while (step is Trampoline<T>.More more)
            step = more.Next();
        return ((Trampoline<T>.Done)step).Result;
    }
}
```

Для порівняння в лістингу 10.11 наведено звичайну рекурсивну функцію підсумовування рахунків, а в лістингу 10.12 - ту саму функцію з трампліном. Логіка ідентична, але рекурсивний виклик загорнуто в лямбда-вираз і повернуто як варіант *More*.

Лістинг 10.11 - Рекурсивне підсумовування рахунків

```
1. // Звичайна рекурсія: кожен виклик займає кадр стека
2. public static decimal SumRecursive(decimal[] checks, int i,
3.                                    decimal acc) =>
4.     i == checks.Length
5.         ? acc
6.         : SumRecursive(checks, i + 1, acc + checks[i]);
```

Лістинг 10.12 - Підсумовування рахунків із трампліном

```
1. // Та сама логіка з трампліном: рекурсивний виклик
2. // повертається як відкладений крок і виконується циклом Run
3. public static Trampoline<decimal> SumTrampoline(decimal[] checks,
4.                                                  int i, decimal acc) =>
5.     i == checks.Length
6.         ? new Trampoline<decimal>.Done(acc)
7.         : new Trampoline<decimal>.More(
8.             () => SumTrampoline(checks, i + 1, acc + checks[i]));
```

Звичайна рекурсія для мільйона рахунків спричинила б переповнення стека, тоді як трамплін виконує мільйон кроків без зростання стека (лістинг 10.13).

Лістинг 10.13 - Виконання мільйона кроків трампліном

```
1. // Мільйон закритих рахунків по 100 грн
2. var checks = Enumerable.Repeat(100m, 1_000_000).ToArray();
3.
4. var total = SumTrampoline(checks, 0, 0m).Run();
```

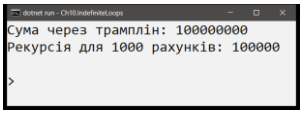


Рисунок 10.6 - Результат виконання прикладу «Трамплін»

Ціною такої надійності є створення об'єкта на кожному кроці, що знижує продуктивність порівняно з простим циклом (див. розділ 12).

10.4. Власні ітератори

Третій спосіб полягає у створенні власної реалізації інтерфейсу *IEnumerable<T>*. Перелічувана послідовність не є масивом: це лише посилання на поточний елемент та інструкція отримання наступного. Отже, можна реалізувати послідовність із будь-якою поведінкою, зокрема таку, що перебирає стани моделі, доки не буде досягнуто кінцевого стану.

10.4.1. Будова перелічувача

Інтерфейс *IEnumerable<T>* вимагає реалізації єдиного методу *GetEnumerator()*, що повертає перелічувач. Основну роботу виконує перелічувач - реалізація інтерфейсу *IEnumerator<T>*, члени якого наведено в таблиці 10.1.

Таблиця 10.1 - Члени інтерфейсу IEnumerator<T>

Член	**Призначення**	**Результат**
Current	Повертає поточний елемент	Поточний елемент (до першого виклику *MoveNext* значення не визначене)
IEnumerator.Current	Те саме для неузагальненого інтерфейсу *IEnumerator*	Поточний елемент як *object*
MoveNext()	Перехід до наступного елемента	*true*, якщо елемент існує; *false*, якщо перебір завершено
Reset()	Повернення до початку послідовності	-
Dispose()	Звільнення ресурсів (з інтерфейсу *IDisposable*)	-

Спрощена реалізація перелічувача масиву наведена в лістингу 10.14. Реальна реалізація в бібліотеці .NET містить додаткові перевірки, проте принцип роботи той самий.

Лістинг 10.14 - Спрощений перелічувач масиву

```
1. public class ArrayEnumerator<T>(T[] data) : IEnumerator<T>
2. {
3.     protected readonly T[] Data = data;
4.     protected int Pos = -1;
5.
6.     public T Current => Data[Pos];
7.     object? IEnumerator.Current => Current;
8.
9.     public virtual bool MoveNext()
10.      {
11.          Pos++;
12.          return Pos < Data.Length;
13.      }
14.
15.      public void Reset() => Pos = -1;
16.      public void Dispose() { }
17.  }
```

10.4.2. Власні перелічувачі

Змінюючи логіку методу *MoveNext*, можна отримати довільну поведінку перебору. Перелічувач у лістингу 10.15 перебирає кожен другий елемент, а в лістингу 10.16 - повертає кожен елемент двічі.

Лістинг 10.15 - Перелічувач кожного другого елемента

```
// Перебір кожного другого елемента: { 1, 2, 3, 4 } -> { 2, 4 }
public sealed class EveryOtherEnumerator<T>(T[] data)
    : ArrayEnumerator<T>(data)
{
    public override bool MoveNext()
    {
        Pos += 2;
        return Pos < Data.Length;
    }
}
```

Лістинг 10.16 - Перелічувач із дублюванням елементів

```
// Кожен елемент двічі: { 1, 2, 3 } -> { 1, 1, 2, 2, 3, 3 }
public sealed class DuplicateEnumerator<T>(T[] data)
    : ArrayEnumerator<T>(data)
{
    private bool isCopy = true;

    public override bool MoveNext()
    {
        if (isCopy)
            Pos++;
        isCopy = !isCopy;
        return Pos < Data.Length;
    }
}
```

У лістингу 10.17 реалізовано послідовність, що перебирає елементи у зворотному порядку, разом із її перелічувачем. Використання такої послідовності не відрізняється від використання стандартних колекцій (лістинг 10.18).

Лістинг 10.17 - Послідовність зі зворотним перебором

```
public sealed class BackwardsEnumerable<T>(IEnumerable<T> data)
    : IEnumerable<T>
{
    private readonly T[] items = data.ToArray();

    public IEnumerator<T> GetEnumerator() =>
        new BackwardsEnumerator<T>(items);

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}

public sealed class BackwardsEnumerator<T>(T[] data) : IEnumerator<T>
{
    private int pos = data.Length;

    public T Current => data[pos];
    object? IEnumerator.Current => Current;

    public bool MoveNext()
    {
        pos--;
```

```
22.          return pos >= 0;
23.      }
24.
25.      public void Reset() => pos = data.Length;
26.      public void Dispose() { }
27.  }
```

Лістинг 10.18 - Зворотний перебір номерів столиків

```
1. var tables = new[] { 1, 2, 3, 4, 5, 6, 7, 8 };
2. var backwards = new BackwardsEnumerable<int>(tables);
3. var order = string.Join(", ", backwards);
4. // order = "8, 7, 6, 5, 4, 3, 2, 1"
```

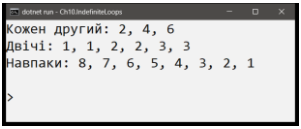


Рисунок 10.7 - Результат виконання прикладу «Власні перелічувачі»

10.4.3. Ітератор послідовних станів

Замість масиву власна послідовність може отримувати початковий стан, функцію обчислення наступного стану та умову завершення. Перелічувач у лістингу 10.19 повертає початковий стан, а потім кожен наступний, включно з кінцевим. Реалізація узагальнена і не залежить від конкретної моделі.

Лістинг 10.19 - Ітератор послідовних станів

```
1. // Нескінченний (до умови) перебір станів: кожен наступний
2. // стан обчислюється з попереднього; кінцевий стан включається
3. public sealed class StateEnumerator<T>(T start, Func<T, T> next,
4.                                        Func<T, bool> isFinal)
5.     : IEnumerator<T>
6. {
7.     private readonly T initial = start;
8.     private bool started;
9.     private bool finished;
10.
11.      public T Current { get; private set; } = start;
12.      object? IEnumerator.Current => Current;
13.
14.      public bool MoveNext()
15.      {
16.          if (finished)
17.              return false;
18.          Current = started ? next(Current) : Current;
19.          started = true;
20.          finished = isFinal(Current);
21.          return true;
22.      }
23.
24.      public void Reset()
25.      {
26.          Current = initial;
```

```
27.          started = false;
28.          finished = false;
29.      }
30.
31.      public void Dispose() { }
32.  }
33.
34.  public sealed class StateIterator<T>(T start, Func<T, T> next,
35.                                        Func<T, bool> isFinal)
36.      : IEnumerable<T>
37.  {
38.      public IEnumerator<T> GetEnumerator() =>
39.          new StateEnumerator<T>(start, next, isFinal);
40.
41.      IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
42.  }
```

Властивість *Current* перелічувача змінюється, тож реалізація не є строго функціональною; це компроміс, локалізований усередині класу.

Мова C# має засіб, що значно спрощує створення ітераторів, - **блоки ітераторів** з інструкцією *yield return* (лістинг 10.20). Компілятор автоматично генерує клас-перелічувач з поведінкою, аналогічною лістингу 10.19.

Лістинг 10.20 - Ітератор станів на основі yield return

```
1. public static class Unfold
2. {
3.     // Той самий перебір станів, реалізований блоком ітератора:
4.     // компілятор сам генерує клас-перелічувач
5.     public static IEnumerable<T> States<T>(T start, Func<T, T> next,
6.                                            Func<T, bool> isFinal)
7.     {
8.         var current = start;
9.         yield return current;
10.          while (!isFinal(current))
11.          {
12.              current = next(current);
13.              yield return current;
14.          }
15.      }
16.  }
```

10.4.4. Використання ітераторів станів

Ітератор станів є повноцінною перелічуваною послідовністю, тож до нього можна застосовувати як цикл *foreach*, так і будь-які методи LINQ. Використання *foreach* є найпростішим, але повертає в код імперативну конструкцію. Якщо потрібен лише кінцевий стан, достатньо методу *Last* (лістинг 10.21).

Лістинг 10.21 - Отримання кінцевого стану методом Last

```
1. var contest = new StateIterator<GameState>(
2.     Start(3),
3.     g => NextTurn(g, die.Roll()),
4.     IsFinished);
5.
6. var endState = contest.Last();
```

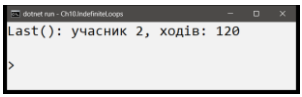


Рисунок 10.8 - Результат виконання прикладу «Ітератор станів» (фрагмент)

Методи *Where* і *Select* дають змогу сформувати повідомлення про проміжні стани, наприклад про лідера програми кожні 30 ходів (лістинг 10.22).

Лістинг 10.22 - Формування повідомлень про проміжні стани

```
1. var leaders = Unfold.States(Start(3),
2.         g => NextTurn(g, die2.Roll()), IsFinished)
3.     .Where(g => g.Turn % 30 == 0)
4.     .Select(g => $"хід {g.Turn}: лідирує учасник " +
5.         g.Players.MaxBy(p => p.Points)!.Number);
```

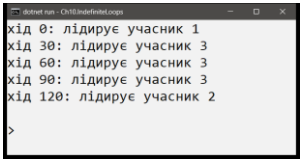


Рисунок 10.9 - Результат виконання прикладу «Ітератор станів» (фрагмент)

Метод *Select* замінює стан повідомленням і тим самим втрачає сам стан. Якщо потрібні і повідомлення, і кінцевий стан, доцільно використати метод *Aggregate* з кортежем як акумулятором (лістинг 10.23).

Лістинг 10.23 - Накопичення повідомлень і кінцевого стану методом Aggregate

```
1. var (messages, final) = new StateIterator<GameState>(
2.         Start(3), g => NextTurn(g, die3.Roll()), IsFinished)
3.     .Aggregate(
4.         (Messages: new List<string>(), State: Start(3)),
5.         (acc, g) => (g.Players.Any(p => p.Points >= 100)
6.             ? [.. acc.Messages, $"хід {g.Turn}: перемога!"]
7.             : acc.Messages,
8.             g));
```



Рисунок 10.10 - Результат виконання прикладу «Ітератор станів» (фрагмент)

Слід пам'ятати, що методи LINQ, які достроково припиняють перебір (*First*, *Take*, *TakeWhile* тощо), припинять і моделювання. Це може бути як помилкою, так і бажаною поведінкою - наприклад, для обмеження кількості кроків.

10.5. Висновки

1. Невизначені цикли, кількість ітерацій яких наперед невідома, неможливо реалізувати стандартними методами LINQ над колекціями фіксованої довжини.

2. Рекурсія є чисто функціональним способом організації таких циклів, проте в мові C# вона обмежена глибиною стека: компілятор C# не виконує оптимізацію хвостових викликів.

3. Метод *IterateUntil* інкапсулює цикл *while* в одному місці бібліотеки; це прагматичний компроміс без ризику переповнення стека.

4. Трамплін виконує рекурсивні алгоритми без зростання стека: функція повертає відкладений наступний крок, а цикл виконує кроки до отримання результату.

5. Власні реалізації *IEnumerator<T>* і блоки ітераторів *yield return* дають змогу будувати послідовності з довільною поведінкою, зокрема послідовності станів до виконання умови, які обробляються методами LINQ.

6. Кожен із розглянутих способів є компромісом між функціональною чистотою, продуктивністю та обсягом коду; вибір залежить від вимог конкретної задачі.

10.6. Контрольні запитання

1. Чим невизначений цикл відрізняється від перебору колекції? Чому його не можна реалізувати методом *Select*?

2. Що таке хвостова рекурсія? Що таке оптимізація хвостових викликів і чому вона недоступна в мові C#?

3. Які ризики пов'язані з використанням рекурсії в мові C#?

4. Як працює метод *IterateUntil*? Чому його вважають компромісом?

5. Що таке трамплін? Чим він відрізняється від методу *IterateUntil*?

6. Які члени має інтерфейс *IEnumerator<T>* і яке їх призначення?

7. Як змінити логіку методу *MoveNext*, щоб отримати перебір у зворотному порядку?

8. Що таке блок ітератора? Як компілятор C# реалізує інструкцію *yield return*?

9. Як отримати кінцевий стан і повідомлення про проміжні стани з ітератора станів засобами LINQ?

10. Як вплине на моделювання застосування до ітератора станів методу *Take(10)*?

10.7. Практичні завдання

1. Змоделюйте заповнення залу ресторану: групи гостей випадкового розміру (від 1 до 6 осіб) прибувають по черзі, доки не буде вичерпано місткість залу. Реалізуйте модель рекурсивно і за допомогою *IterateUntil*; порівняйте результати при однаковому початковому значенні генератора.

2. Реалізуйте за допомогою трампліна обчислення кількості днів, за які буде вичерпано запас вина в погребі ресторану, якщо щоденна витрата задається функцією від номера дня. Перевірте роботу для запасу, що вичерпується за 500 000 днів.

3. Реалізуйте власний перелічувач, що перебирає номери готелю «змійкою»: перший поверх - зростаюче, другий - спадаюче тощо.

4. Реалізуйте блок ітератора, що генерує розклад прибирання номерів: від заданого часу з кроком, що залежить від категорії номера, до кінця робочої зміни.

5. Застосуйте ітератор станів до моделі із завдання 1 і сформуйте методом *Aggregate* звіт про кожну групу гостей та кінцевий стан залу.

11. Мемоізація

Чисті функції мають не лише ту перевагу, що їхні результати передбачувані. Оскільки для однакових аргументів чиста функція завжди повертає однаковий результат, повторне обчислення можна замінити збереженим раніше значенням. На цьому ґрунтується **мемоізація** (memoization) - техніка оптимізації, за якої результати викликів функції запам'ятовуються за значеннями аргументів і за повторного виклику з тими самими аргументами повертаються без повторного обчислення.

Мемоізація подібна до кешування, зокрема до методу *GetOrAdd* класу *MemoryCache*: за ключем перевіряється наявність значення, і якщо його немає, значення обчислюється переданою функцією та зберігається. Проте, на відміну від кешування загального призначення, мемоізація, як правило, не потребує механізмів інвалідації чи оновлення збережених значень: результати зберігаються лише протягом виконання певної одиниці роботи (наприклад, одного складного обчислення), після чого відкидаються. Отже, мемоізація не замінює кешування, а доповнює його. Вона особливо корисна в багатокрокових, зокрема рекурсивних, обчисленнях, у яких ті самі підобчислення з різних причин виконуються неодноразово.

Мета розділу - сформувати розуміння мемоізації та навички її реалізації й застосування в мові C#.

Після опрацювання розділу студент повинен:

- знати означення мемоізації та її відмінність від кешування;
- розуміти, чому мемоізація коректна лише для чистих функцій;
- уміти реалізовувати мемоізацію функцій з одним і кількома параметрами та з довільним ключем;
- уміти застосовувати мемоізацію для скорочення кількості звернень до дорогих зовнішніх ресурсів.

Приклади розділу містяться в проєкті *Ch11.Memoization*; методи мемоізації - у файлі *Functional/Memoization.cs* спільної бібліотеки.

11.1. Постановка задачі: граф професійних зв'язків

Нехай у мережі закладів аналізуються професійні зв'язки шеф-кухарів: двоє кухарів пов'язані, якщо вони працювали на одній кухні. **Число професійного зв'язку** кухаря - це найменша кількість таких «кроків» від центрального кухаря мережі до заданого кухаря. Для самого центрального кухаря число дорівнює 0, для кухарів, які працювали з ним на одній кухні, - 1, для їхніх колег - 2 і т. д. Якщо зв'язку немає, число не визначене (рис. 11.1). Аналогічну ідею в популярній культурі називають «числом Бекона», а в науковому середовищі - «числом Ердеша», що визначається через спільні публікації.



Рисунок 11.1 - Граф професійних зв'язків шеф-кухарів (на ребрах - назви кухонь)

Дані про кухні, на яких працював кухар, надає віддалений сервіс професійних профілів. Кожне звернення до нього є дорогою операцією, тому лічильник фіксує кількість звернень (лістинг 11.1).

Лістинг 11.1 - Віддалений сервіс професійних профілів

```
1. // Віддалений сервіс професійних профілів кухарів:
2. // кожне звернення є дорогим, тому лічильник фіксує їх кількість
3. public sealed class KitchenApi
4. {
5.     private static readonly Kitchen[] Kitchens =
6.     [
7.         new("«Під ратушею»", ["Олійник", "Марченко", "Ткачук"]),
8.         new("«Карпати»", ["Марченко", "Бойко", "Литвин"]),
9.         new("«Буковина»", ["Ткачук", "Савчук"]),
10.          new("«Дністер»", ["Литвин", "Кравець"]),
11.          new("«Прут»", ["Кравець", "Гнатюк"]),
12.          new("«Черемош»", ["Савчук", "Мельник"]),
13.          new("«Хотин»", ["Шевчук"])
14.     ];
15.
16.     public int Calls { get; private set; }
17.
18.     public IEnumerable<Kitchen> GetChefKitchens(string chef)
19.     {
20.         Calls++;
21.         return Kitchens.Where(k => k.Chefs.Contains(chef)).ToArray();
22.     }
23. }
```

Число зв'язку обчислюється **пошуком у ширину** (breadth-first search):

1) центральний кухар отримує число 0 і утворює початковий «фронт» пошуку;

2) для кожного кухаря фронту отримуються кухні, на яких він працював, а з них - усі кухарі; ті з них, кому число ще не призначено, отримують число, на одиницю більше за поточний рівень, і утворюють новий фронт;

3) кроки повторюються, доки шуканому кухареві не буде призначено число або доки фронт не стане порожнім.

Стан пошуку описується незмінним записом (лістинг 11.2), а сам пошук - невизначеним циклом на основі методу *IterateUntil* з розділу 10 (лістинг 11.3).

Лістинг 11.2 - Стан пошуку в ширину

```
1. public sealed record SearchState(
2.     ImmutableDictionary<string, int> Distances,
3.     string[] Frontier,
4.     int Level);
```

Лістинг 11.3 - Обчислення числа професійного зв'язку

```
1. // Число професійного зв'язку: кількість «кроків» спільної
2. // роботи на кухнях від центрального шеф-кухаря до заданого
3. public static int Distance(string target,
4.     Func<string, IEnumerable<Kitchen>> getKitchens)
5. {
6.     var start = new SearchState(
7.         ImmutableDictionary<string, int>.Empty.Add("Олійник", 0),
8.         ["Олійник"],
9.         0);
10.
11.      var final = start.IterateUntil(
12.          s => Step(s, getKitchens),
13.          s => s.Distances.ContainsKey(target) || s.Frontier.Length ==
    0);
14.
15.      return final.Distances.GetValueOrDefault(target, -1);
16. }
17.
18. // Один крок пошуку в ширину: наступний «рівень» кухарів
19. private static SearchState Step(SearchState s,
20.     Func<string, IEnumerable<Kitchen>> getKitchens)
21. {
22.     var next = NextChefs(s, getKitchens);
23.     return new SearchState(
24.         s.Distances.AddRange(
25.             next.Select(c => KeyValuePair.Create(c, s.Level + 1))),
26.         next,
27.         s.Level + 1);
28. }
29.
30. private static string[] NextChefs(SearchState s,
31.     Func<string, IEnumerable<Kitchen>> getKitchens) =>
32.     s.Frontier
33.         .SelectMany(getKitchens)
34.         .SelectMany(k => k.Chefs)
35.         .Distinct()
36.         .Where(c => !s.Distances.ContainsKey(c))
37.         .ToArray();
```

Функція отримання кухонь передається в *Distance* як параметр, тож метод не залежить від конкретного джерела даних. Обчислення чисел зв'язку для кількох кухарів наведено в лістингу 11.4.

Лістинг 11.4 - Обчислення чисел зв'язку без мемоізації

```
1. var report = Chefs.Select(c =>
2.     (Chef: c, Distance: ChefConnections.Distance(
3.         c, api.GetChefKitchens)));
```

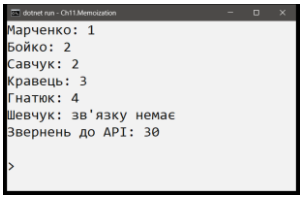


Рисунок 11.2 - Результат виконання прикладу «Без мемоізації»

Результати коректні, проте кількість звернень до віддаленого сервісу значна: кожне обчислення починає пошук від центрального кухаря і повторно запитує профілі тих самих кухарів. Графи професійних зв'язків щільно перетинаються, тож за реальних обсягів даних більшість звернень є повторними.

11.2. Застосування мемоізації

Мемоізована версія функції отримання кухонь звертається до сервісу лише один раз для кожного кухаря, а за повторних викликів повертає збережений результат (лістинг 11.5).

Лістинг 11.5 - Повторні виклики мемоізованої функції

```
1. Func<string, IEnumerable<Kitchen>> getKitchens = api.GetChefKitchens;
2. var getKitchensMemo = getKitchens.Memoize();
3.
4. var k1 = getKitchensMemo("Марченко");
5. var k2 = getKitchensMemo("Марченко");
6. var k3 = getKitchensMemo("Марченко");
```

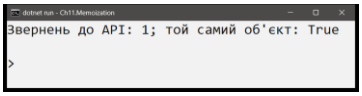


Рисунок 11.3 - Результат виконання прикладу «Повторні виклики»

Функцію викликано тричі, але звернення до сервісу відбулося лише один раз, а всі виклики повернули той самий об'єкт. Мемоізована версія створюється окремим рядком: метод-розширення можна застосувати лише до змінної типу делегата, а групу методів *api.GetChefKitchens* спочатку потрібно присвоїти такій змінній (див. також підрозділ 5.6).

Обчислення чисел зв'язку з мемоізованою функцією відрізняється від лістингу 11.4 лише тим, що в метод *Distance* передається мемоізована версія (лістинг 11.6).

Лістинг 11.6 - Обчислення чисел зв'язку з мемоізацією

```
1. Func<string, IEnumerable<Kitchen>> getKitchens = api.GetChefKitchens;
2. var getKitchensMemo = getKitchens.Memoize();
3.
4. var report = Chefs.Select(c =>
5.     (Chef: c, Distance: ChefConnections.Distance(
6.         c, getKitchensMemo)));
```

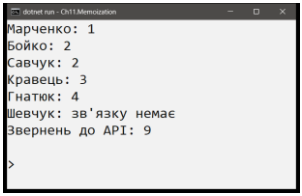


Рисунок 11.4 - Результат виконання прикладу «З мемоізацією»

Результати не змінилися, а кількість звернень до сервісу скоротилася з 30 до 9 - по одному на кожного кухаря, профіль якого знадобився. Мемоізована функція створюється для однієї одиниці роботи (формування звіту) і після її завершення разом зі збереженими значеннями стає недосяжною для збирача сміття. Якщо збережені значення мають існувати довше, доцільніше використовувати кешування загального призначення, наприклад *MemoryCache*.

Слід зазначити, що деякі функціональні мови надають вбудовану підтримку мемоізації; у мові F# вона, як і в C#, реалізується бібліотечними засобами.

11.3. Реалізація мемоізації в мові C#

Мемоізація функції з одним параметром реалізується методом-розширенням, що повертає замикання над словником (лістинг 11.7).

Лістинг 11.7 - Мемоізація функції з одним параметром

```
1. public static partial class MemoizeExtensions
2. {
3.     public static Func<T1, TOut> Memoize<T1, TOut>(
4.         this Func<T1, TOut> @this) where T1 : notnull
5.     {
6.         var cache = new Dictionary<T1, TOut>();
7.         return x =>
8.         {
9.             if (!cache.TryGetValue(x, out var value))
10.              {
11.                  value = @this(x);
12.                  cache.Add(x, value);
13.              }
14.              return value;
15.          };
16.     }
17. }
```

Для функцій з кількома параметрами потрібні окремі перевантаження. Як ключ словника доцільно використовувати кортеж параметрів (лістинг 11.8): рівність кортежів визначається рівністю їхніх елементів, тож перетворювати аргументи на рядок за допомогою *ToString* не потрібно. Підхід із рядковим ключем вимагав би, щоб метод *ToString* повертав змістовне подання аргументу, що виконується для простих типів, але не для більшості класів (за замовчуванням *ToString* повертає лише ім'я типу).

Лістинг 11.8 - Мемоізація функції з двома параметрами

```
1. public static partial class MemoizeExtensions
2. {
3.     // Ключ - кортеж параметрів: рівність кортежів визначається
4.     // рівністю елементів, тож ToString() не потрібен
5.     public static Func<T1, T2, TOut> Memoize<T1, T2, TOut>(
6.         this Func<T1, T2, TOut> @this)
7.     {
8.         var cache = new Dictionary<(T1, T2), TOut>();
9.         return (x, y) =>
10.          {
11.              if (!cache.TryGetValue((x, y), out var value))
12.              {
13.                  value = @this(x, y);
14.                  cache.Add((x, y), value);
15.              }
16.              return value;
17.          };
18.      }
19.  }
```

У лістингу 11.9 мемоізовано функцію отримання ціни інгредієнта на дату під час розрахунку собівартості 40 порцій страви: без мемоізації ціна обчислювалася б 120 разів (по три інгредієнти на кожну порцію).

Лістинг 11.9 - Мемоізація під час розрахунку собівартості

```
1. Func<string, DateOnly, decimal> price = PriceOn;
2. var priceMemo = price.Memoize();
3.
4. var today = new DateOnly(2026, 11, 13);
5. (string Item, decimal Kg)[] deruny =
6.     [("картопля", 0.25m), ("гриби", 0.08m), ("сметана", 0.05m)];
7.
8. // 40 порцій деруни: собівартість кожної порції
9. var cost = Enumerable.Range(1, 40)
10.      .Select(_ => deruny.Sum(x => priceMemo(x.Item, today) * x.Kg))
11.      .Sum();
```

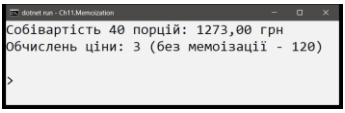


Рисунок 11.5 - Результат виконання прикладу «Мемоізація з двома параметрами» (фрагмент)

Якщо аргументом є складний об'єкт, рівність якого не визначена за значенням або визначена не тими властивостями, ключ можна формувати функцією, яку задає розробник (лістинги 11.10 і 11.11).

Лістинг 11.10 - Мемоізація з довільною функцією ключа

```
1. public static partial class MemoizeExtensions
2. {
3.     // Для складних об'єктів ключ формує функція, задана розробником
4.     public static Func<T1, TOut> Memoize<T1, TKey, TOut>(
5.         this Func<T1, TOut> @this,
6.         Func<T1, TKey> keySelector) where TKey : notnull
7.     {
8.         var cache = new Dictionary<TKey, TOut>();
9.         return x =>
10.          {
11.              var key = keySelector(x);
12.              if (!cache.TryGetValue(key, out var value))
13.              {
14.                  value = @this(x);
15.                  cache.Add(key, value);
16.              }
17.              return value;
18.          };
19.      }
20.  }
```

Лістинг 11.11 - Використання функції ключа

```
1. Func<PricedIngredient, string> describe =
2.     i => $"{i.Name}: {i.PricePerKg} грн/кг";
3. var describeMemo = describe.Memoize(i => i.Name);
```



Рисунок 11.6 - Результат виконання прикладу «Мемоізація з двома параметрами» (фрагмент)

Останній приклад наочно демонструє ризик неправильно обраного ключа: другий виклик з іншою ціною повернув збережений результат, оскільки ключ враховує лише назву інгредієнта. Ключ має однозначно визначати результат функції.

11.4. Обмеження мемоізації

Мемоізація коректна лише для **чистих функцій**. Якщо мемоізована функція має побічні ефекти, вони виконуються лише під час першого виклику. Наприклад, запис до журналу про звернення до сервісу за повторних викликів не відбуватиметься. Якщо ж функція має повертати різні значення для однакових аргументів (наприклад, створювати об'єкти з унікальними ідентифікаторами або залежати від поточного часу),

мемоізація призведе до помилок. У таких випадках доцільно розділити відповідальність: одна функція виконує лише отримання даних (і може бути мемоізована), інша - перетворення отриманих даних.

Слід також враховувати:

- **споживання пам'яті** - мемоізована функція зберігає всі результати до знищення замикання; для великих обсягів даних потрібні стратегії обмеження розміру сховища;

- **багатопотоковість** - словник *Dictionary* не є потокобезпечним; для паралельного використання доцільно застосовувати *ConcurrentDictionary*;

- **тривалість зберігання** - якщо результати мають зберігатися між викликами методу чи навіть між запусками програми (для обчислень, що тривають десятки хвилин), можна передавати в метод мемоізації зовнішнє сховище (*MemoryCache*, база даних). Проте в такому разі доцільніше розглядати повноцінне рішення кешування.

11.5. Висновки

1. Мемоізація - це запам'ятовування результатів викликів функції за значеннями аргументів з поверненням збереженого результату за повторного виклику.

2. На відміну від кешування загального призначення, мемоізація зазвичай діє в межах однієї одиниці роботи й не потребує інвалідації збережених значень.

3. Мемоізація коректна лише для чистих функцій: побічні ефекти мемоізованої функції виконуються тільки під час першого виклику.

4. У мові C# мемоізація реалізується методами-розширеннями, що повертають замикання над словником; для кількох параметрів ключем доцільно використовувати кортеж, для складних об'єктів - функцію формування ключа.

5. Мемоізація суттєво скорочує кількість звернень до дорогих ресурсів у багатокрокових обчисленнях із повторюваними підобчисленнями, зокрема в пошуку на графах.

11.6. Контрольні запитання

1. Що таке мемоізація? Чим вона відрізняється від кешування?

2. Чому мемоізацію можна застосовувати лише до чистих функцій?

3. Як працює пошук у ширину для обчислення числа професійного зв'язку?

4. Чому мемоізація скорочує кількість звернень до сервісу в прикладі з графом кухарів?

5. Чому мемоізовану функцію потрібно створювати окремою змінною?

6. Чому ключ у вигляді кортежу кращий за рядковий ключ, сформований методом *ToString*?

7. До яких помилок може призвести неправильно обраний ключ мемоізації?

8. Які обмеження має мемоізація щодо пам'яті та багатопотоковості?

11.7. Практичні завдання

1. Реалізуйте мемоізовану рекурсивну функцію розрахунку собівартості страви за технологічною картою, у якій напівфабрикати можуть складатися з інших напівфабрикатів. Порівняйте кількість звернень до функції отримання техкарти з мемоізацією та без неї.

2. Реалізуйте потокобезпечну версію методу *Memoize* на основі *ConcurrentDictionary* і перевірте її в паралельних обчисленнях (*Parallel.For*).

3. Реалізуйте метод *Memoize* для функцій з трьома параметрами і застосуйте його до функції розрахунку вартості доставки (адреса, вага, час доставки).

4. Доповніть пошук у графі професійних зв'язків поверненням не лише числа зв'язку, а й ланцюжка кухарів і кухонь, через які встановлено зв'язок.

5. Реалізуйте варіант мемоізації з обмеженням кількості збережених значень (зберігаються лише N останніх використаних результатів).

Частина III. Практичне застосування

12. Практичне функціональне програмування в C#

Теоретичний матеріал посібника завершено. Подальше поглиблення теорії функціонального програмування потребує вивчення теорії категорій і чисто функціональних мов, зокрема Haskell. Третя частина посібника присвячена практичним аспектам: продуктивності функціонального коду, організації промислових проєктів, стороннім бібліотекам і побудові завершеного застосунку.

Під час обговорення функціонального програмування мовою C# найчастіше виникають три запитання: чому не використовувати одразу мову F# (відповідь наведено в підрозділі 1.5.3), що таке монади (розділ 7) і чи не є функціональний код на C# менш продуктивним, ніж об'єктно-орієнтований. Цей розділ присвячено передусім третьому запитанню, а також типовим практичним питанням упровадження функціонального стилю.

Мета розділу - сформувати обґрунтоване уявлення про продуктивність функціонального коду в мові C# і про практичні аспекти застосування функціонального стилю в промисловій розробці.

Після опрацювання розділу студент повинен:

- уміти проводити порівняльні вимірювання продуктивності за допомогою BenchmarkDotNet;
- знати відносну продуктивність розглянутих у посібнику функціональних прийомів порівняно з імперативними;
- розуміти вплив операцій введення-виведення на загальну продуктивність;
- уміти обґрунтовувати рішення щодо ступеня використання функціонального стилю в проєкті.

Приклади розділу містяться в проєктах *Ch12.Performance* і *Ch12.FSharpLib* (мова F#); повні результати вимірювань - у каталозі *docs/benchmarks* репозиторію.

12.1. Постановка експерименту

Для порівняння продуктивності потрібна задача, що допускає як імперативне, так і різні функціональні розв'язання та містить і визначений, і невизначений цикл. Як таку задачу обрано аналіз журналу руху товару на складі ресторану. Журнал подано рядком символів: символ «+» означає надходження одиниці товару, «-» - витрату одиниці. Початковий залишок дорівнює нулю. Потрібно розв'язати дві задачі:

1) визначити підсумковий залишок після всіх операцій (визначений цикл: необхідно обробити весь рядок);

2) визначити номер операції, після якої залишок уперше став від'ємним, тобто виник дефіцит (невизначений цикл: обробка припиняється після досягнення умови).

Для рядка +++-----++++ залишок послідовно змінюється так, як показано в таблиці 12.1.

Таблиця 12.1 - Зміна залишку для рядка «+++-----++++»

№ операції	Операція	Залишок	Примітка
1	+	1	
2	+	2	
3	+	3	
4	-	2	
5	-	1	
6	-	0	
7	-	-1	перший дефіцит
8	-	-2	
9	+	-1	повторний дефіцит не враховується
10	+	0	
11	+	1	
12	+	2	

Отже, підсумковий залишок дорівнює 2, а перший дефіцит виникає після сьомої операції:

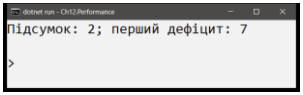


Рисунок 12.1 - Результат виконання прикладу «Приклад» (фрагмент)

Для вимірювань використано журнал із 7000 операцій (файл *Data/stock-moves.txt*). Перший дефіцит у ньому виникає після операції № 1593, підсумковий залишок становить 270. Усі розглянуті реалізації дають однакові результати:

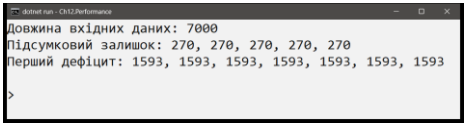


Рисунок 12.2 - Результат виконання прикладу «Перевірка всіх реалізацій»

Вимірювання виконано за допомогою бібліотеки **BenchmarkDotNet** (версія 0.15.2) - стандартного засобу мікробенчмаркінгу для .NET, який

багаторазово виконує код, відкидає «розігрівні» запуски й обчислює середній час виконання, стандартне відхилення та обсяг виділеної пам'яті. Конфігурація: процесор AMD Ryzen 7 5800H, Windows 11, .NET 10.0.15 (x64, RyuJIT AVX2), 3 розігрівні та 10 вимірювальних ітерацій, режим збирання Release. Абсолютні значення залежать від обладнання та версії платформи, тому аналізувати доцільно передусім співвідношення між варіантами. Слід також ураховувати, що від версії до версії .NET продуктивність LINQ та інших засобів суттєво зростає: результати на ранніх версіях платформи можуть значно відрізнятися від наведених.

12.2. Імперативне рішення як база порівняння

Як і в будь-якому експерименті, потрібна контрольна точка, з якою порівнюються всі інші результати. Нею є імперативне розв'язання обох задач (лістинги 12.1 і 12.2).

Лістинг 12.1 - Імперативне обчислення підсумкового залишку

```
1. public static int FinalLevel(string input)
2. {
3.     var level = 0;
4.     foreach (var c in input)
5.     {
6.         if (c == '+')
7.             level++;
8.         else
9.             level--;
10.      }
11.      return level;
12.  }
```

Лістинг 12.2 - Імперативний пошук першого дефіциту

```
1. public static int FirstDeficit(string input)
2. {
3.     var level = 0;
4.     var operation = 0;
5.     foreach (var c in input)
6.     {
7.         operation++;
8.         if (c == '+')
9.             level++;
10.          else
11.              level--;
12.          if (level == -1)
13.              return operation;
14.      }
15.      return -1;
16.  }
```

Невизначений цикл виконується швидше за визначений, оскільки обробляє лише частину рядка (1593 з 7000 символів). Результати базового рішення наведено в перших рядках таблиць 12.2 і 12.3.

12.3. Визначений цикл: функціональні рішення

Задачу про підсумковий залишок можна розв'язати одним виразом - агрегуванням методом *Sum*, що додає +1 або -1 залежно від символу (лістинг 12.3). Рядок у мові C# є послідовністю символів, тож до нього застосовні методи LINQ.

Лістинг 12.3 - Обчислення залишку методом Sum

```
1. public static int FinalLevelSum(string input) =>
2.     input.Sum(c => c switch
3.     {
4.         '+' => 1,
5.         _ => -1
6.     });
```

Варіант із попереднім перетворенням символів на числа методом *Select* наведено в лістингу 12.4, а варіант із групуванням символів і словником - у лістингу 12.5.

Лістинг 12.4 - Обчислення залишку методами Select і Sum

```
1. public static int FinalLevelSelectSum(string input) =>
2.     input.Select(c => c == '+' ? 1 : -1).Sum();
```

Лістинг 12.5 - Обчислення залишку групуванням

```
1. public static int FinalLevelGroup(string input)
2. {
3.     var grouped = input.GroupBy(c => c)
4.                        .ToDictionary(g => g.Key, g => g.Count());
5.     return grouped.GetValueOrDefault('+')
6.          - grouped.GetValueOrDefault('-');
7. }
```

Для порівняння використано також реалізацію мовою F#, підключену до проєкту на C# (лістинги 12.6 і 12.7). Функція *Seq.fold* є аналогом *Aggregate*, а *Seq.scan* повертає послідовність усіх проміжних значень згортки, у якій методом *Seq.findIndex* шукається перше значення -1. Обидві мови компілюються в той самий проміжний код, тож функції F# доступні з C# як статичні методи.

Лістинг 12.6 - Розв'язання задач мовою F#

```
1. module Stock =
2.     // Підсумковий залишок: згортка послідовності операцій
3.     let finalLevel (start: int) (input: string) =
4.         input
5.         |> Seq.fold (fun acc c ->
6.             match c with
7.             | '+' -> acc + 1
8.             | '-' -> acc - 1
9.             | _ -> acc) start
10.
11.      // Позиція першого дефіциту: послідовність проміжних
12.      // залишків (scan) і пошук першого значення -1
13.      let firstDeficit (start: int) (input: string) =
```

```
        input
        |> Seq.scan (fun acc c ->
            match c with
            | '+' -> acc + 1
            | '-' -> acc - 1
            | _ -> acc) start
        |> Seq.findIndex (fun level -> level = -1)
```

Лістинг 12.7 - Виклик функцій F# з коду C#

```
public static int FinalLevel(string input) =>
    Ch12.FSharpLib.Stock.finalLevel(0, input);

public static int FirstDeficit(string input) =>
    Ch12.FSharpLib.Stock.firstDeficit(0, input);
```

Результати вимірювань наведено в таблиці 12.2 (відношення обчислено щодо імперативного рішення).

Таблиця 12.2 - Продуктивність рішень задачі про підсумковий залишок

Рішення	**Середній час, мкс**	**Відношення**	**Виділено пам'яті**
Імперативне (база)	5,36	1,00	0 Б
Sum з виразом *switch*	7,35	1,37	0 Б
Select + *Sum*	20,12	3,76	88 Б
GroupBy + словник	58,52	10,92	33 944 Б
F# (*Seq.fold*)	18,45	3,44	32 Б

Функціональні рішення поступаються імперативному. Найближчим до нього є рішення з одним викликом *Sum*: різниця становить лише 37 %, а додаткова пам'ять не виділяється. Кожен додатковий етап конвеєра (*Select*) збільшує витрати, а побудова проміжних структур даних (групування, словник) погіршує як час виконання, так і використання пам'яті.

12.4. Невизначений цикл: функціональні рішення

12.4.1. Рекурсія

У лістингу 12.8 наведено рекурсивне рішення, у якому кожен виклик отримує залишок рядка без першого символу (*s[1..]*).

Лістинг 12.8 - Рекурсивний пошук із копіюванням рядка

```
1. // Рекурсія з копіюванням залишку рядка на кожному кроці
2. public static int FirstDeficitRecursiveSlice(string input)
3. {
4.     int Find(string s, int level, int operation) =>
5.         level == -1
6.             ? operation
7.             : Find(s[1..], s[0] == '+' ? level + 1 : level - 1,
8.                    operation + 1);
9.     return Find(input, 0, 0);
10.  }
```

Таке рішення виявляється катастрофічно неефективним (таблиця 12.3): час виконання зростає майже в 900 разів, а обсяг виділеної пам'яті сягає 19,8 МБ. Проте причиною цього є не рекурсія як така, а створення на кожному кроці нового рядка - копії залишку вихідного рядка. Рекурсивне рішення, що передає індекс замість копії рядка (лістинг 12.9), поступається імперативному лише на 35 % і не виділяє пам'яті. Отже, рекурсія в C# небезпечна передусім через обмеженість стека (див. розділ 10), а не через низьку швидкодію.

Лістинг 12.9 - Рекурсивний пошук з індексом

```
1. // Рекурсія з індексом: рядок не копіюється
2. public static int FirstDeficitRecursiveIndex(string input)
3. {
4.     int Find(int i, int level) =>
5.         level == -1
6.             ? i
7.             : Find(i + 1, input[i] == '+' ? level + 1 : level - 1);
8.     return Find(0, 0);
9. }
```

12.4.2. Метод IterateUntil

Рішення з методом *IterateUntil* з розділу 10 потребує об'єкта стану. Варіант на основі запису наведено в лістингу 12.10, на основі кортежу - у лістингу 12.11.

Лістинг 12.10 - Пошук із IterateUntil і записом

```
1. public sealed record StockState(string Input, int Level, int Operation);
2.
3. public static int FirstDeficitRecord(string input) =>
4.     new StockState(input, 0, 0)
5.         .IterateUntil(
6.             x => x with
7.             {
8.                 Operation = x.Operation + 1,
9.                 Level = x.Input[x.Operation] == '+'
10.                      ? x.Level + 1
11.                      : x.Level - 1
12.             },
13.             x => x.Level == -1)
14.         .Operation;
```

Лістинг 12.11 - Пошук із IterateUntil і кортежем

```
1. public static int FirstDeficitTuple(string input)
2. {
3.     var (_, _, operation) = (Input: input, Level: 0, Operation: 0)
4.         .IterateUntil(
5.             x => (x.Input,
6.                   x.Input[x.Operation] == '+'
7.                       ? x.Level + 1
8.                       : x.Level - 1,
9.                   x.Operation + 1),
10.              x => x.Level == -1);
11.      return operation;
12.  }
```

Запис є посилальним типом, тож на кожному кроці створюється новий об'єкт у керованій купі, що відображається в обсязі виділеної пам'яті. Кортеж є типом-значенням і розміщується в стеку, тож додаткова пам'ять не виділяється, а час виконання втричі менший. Варто зазначити, що в цьому випадку можна використати й запис-структуру (*record struct*), який поєднує зручний синтаксис записів із розміщенням у стеку.

12.4.3. Власний перелічувач

Рішення на основі власного перелічувача (див. розділ 10) наведено в лістингу 12.12.

Лістинг 12.12 - Пошук за допомогою власного перелічувача

```
1. public sealed class StockEnumerable(string input) : IEnumerable<int>
2. {
3.     public IEnumerator<int> GetEnumerator() =>
4.         new StockEnumerator(input);
5.     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
6. }
7.
8. // Повертає номери операцій, доки залишок не стане -1
9. public sealed class StockEnumerator(string input) : IEnumerator<int>
10.  {
11.      private int level;
12.      private int operation;
13.
14.      public int Current => operation;
15.      object IEnumerator.Current => Current;
16.
17.      public bool MoveNext()
18.      {
19.          if (level == -1 || operation >= input.Length)
20.              return false;
21.          level += input[operation] == '+' ? 1 : -1;
22.          operation++;
23.          return true;
24.      }
25.
26.      public void Reset() => (level, operation) = (0, 0);
27.      public void Dispose() { }
28.  }
```

```
29.
30.  public static int FirstDeficitEnumerable(string input) =>
31.      new StockEnumerable(input).Last();
```

Результати всіх рішень задачі про перший дефіцит наведено в таблиці 12.3.

Таблиця 12.3 - Продуктивність рішень задачі про перший дефіцит

Рішення	**Середній час, мкс**	**Відношення**	**Виділено пам'яті**
Імперативне (база)	1,11	1,00	0 Б
Рекурсія з копіюванням рядка	987,81	888,4	19 802 584 Б
Рекурсія з індексом	1,50	1,35	0 Б
IterateUntil із записом	9,53	8,57	51 008 Б
IterateUntil з кортежем	2,90	2,61	0 Б
Власний перелічувач	1,15	1,03	0 Б
F# (*Seq.scan*)	16,89	15,19	216 Б

Власний перелічувач практично не поступається імперативному рішенню, оскільки по суті ним і є, інкапсульованим за інтерфейсом *IEnumerable<T>*. Рішення мовою F#, викликане з C#, у цьому експерименті повільніше за всі рішення на C#, крім рекурсії з копіюванням рядка: узагальнені функції *Seq* працюють через інтерфейс перелічувача рядка й не мають доступу до оптимізацій, які компілятор C# застосовує до циклу *foreach* над рядком.

12.5. Вплив операцій введення-виведення

Порівняння обчислень у пам'яті не відображає реальних умов роботи більшості застосунків, які отримують дані з файлів, баз даних чи мережевих сервісів. Для оцінки цього впливу ті самі рішення було модифіковано так, щоб вони зчитували журнал із файлу локальної файлової системи (лістинг 12.13).

Лістинг 12.13 - Варіанти рішень із читанням файлу

```
1. [Benchmark]
2. public int ImperativeFinalWithFile() =>
3.     Imperative.FinalLevel(File.ReadAllText(Input.FilePath));
4.
5. [Benchmark]
6. public int FunctionalFinalWithFile() =>
7.     Definite.FinalLevelSum(File.ReadAllText(Input.FilePath));
```

Результати наведено в таблиці 12.4.

Таблиця 12.4 - Продуктивність рішень із читанням файлу

Рішення	**Середній час, мкс**	**Відношення до імперативного з файлом**
Лише читання файлу	89,52	-
Залишок: імперативне + файл	97,64	1,00
Залишок: *Sum* + файл	113,56	1,16
Дефіцит: імперативне + файл	101,10	1,00
Дефіцит: *IterateUntil* з кортежем + файл	105,30	1,04

Навіть найпростіша операція введення - читання невеликого локального файлу - займає основну частину часу виконання (рис. 12.3). Різниця між імперативним і функціональним рішеннями, що в пам'яті становила 37 % і 161 %, з урахуванням читання файлу скорочується до 16 % і 4 % відповідно. Для звернень до мережевих сервісів чи бази даних, які тривають мілісекунди, частка обробки даних у пам'яті стає ще меншою.

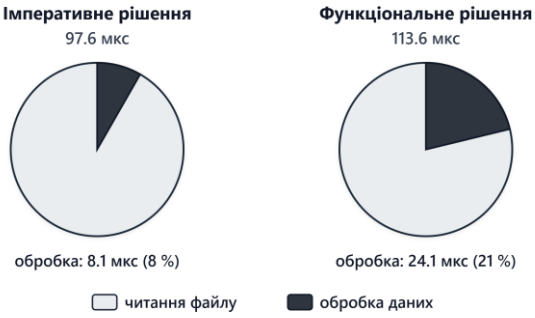


Рисунок 12.3 - Співвідношення часу читання файлу та обробки даних

12.6. Інтерпретація результатів

Результати експерименту дають змогу зробити такі висновки:

1) функціональний код на C# дещо поступається за продуктивністю ретельно написаному імперативному коду; водночас неоптимальний код у будь-якій парадигмі може бути вкрай неефективним (див. рекурсію з копіюванням рядка);

2) найефективнішими з функціональних засобів є стислі операції LINQ без проміжних структур даних; для об'єктів стану з міркувань продуктивності доцільно обирати типи-значення (кортежі, *record struct*);

3) в реальних застосунках, що взаємодіють із зовнішнім середовищем, різниця в продуктивності обробки даних у пам'яті здебільшого незначна порівняно з витратами на введення-виведення.

Отже, питання про доцільність функціонального стилю зводиться до того, чи є продуктивність визначальним критерієм для конкретного проєкту. Для систем реального часу, ігрових рушіїв, тривимірної графіки або програм для пристроїв з обмеженими ресурсами, де значення має кожна мікросекунда, функціональний стиль у критичних фрагментах може бути недоцільним. Проте значна частина промислових систем - корпоративні вебзастосунки, системи обробки бізнес-даних, інтеграційні сервіси - обмежена передусім швидкістю розробки, надійністю та вартістю супроводу. Код читають і супроводжують значно довше, ніж пишуть, тож чіткіший і менш схильний до помилок функціональний код може бути економічно вигіднішим, навіть якщо він дещо повільніший. Крім того, відсутність спільного змінного стану спрощує паралельну обробку та горизонтальне масштабування (див. підрозділ 1.6.5).

12.7. Практичні питання впровадження функціонального стилю

12.7.1. Ступінь функціональності кодової бази

Функціональні засоби мови C# не є окремим фреймворком, тож ступінь їх використання можна обирати довільно: від усієї кодової бази до окремого проєкту, класу, функції чи навіть рядка коду. Так само довільною є глибина застосування: від заміни циклів запитами LINQ до систематичного використання монад, дискримінованих об'єднань і часткового застосування. Визначальними чинниками є цілі проєкту, його обмеження та рівень підготовки команди: навіть найкращий функціональний код має супроводжувати вся команда, тож упровадження складніших засобів потребує спільного рішення та навчання.

12.7.2. Структура функціонального рішення

Функціональне рішення на C# доцільно структурувати так само, як звичайний об'єктно-орієнтований проєкт, - за класами та каталогами. У мові F# код групується в модулі, що за змістом ближчі до просторів імен C#, однак у C# класи обов'язкові. Можна зробити всі класи й методи статичними, проте це ускладнить взаємодію з бібліотеками .NET і сторонніми пакетами та модульне тестування. Практичним компромісом є архітектура «функціональне ядро, імперативна оболонка» (див. підрозділ 1.2.4): впровадження залежностей через контейнер інверсії керування (IoC) виконується у зовнішньому імперативному шарі, а функції ядра отримують усі залежності як параметри.

12.7.3. Спільне використання функціональних засобів

Власні реалізації монад, дискримінованих об'єднань і методів-розширень доцільно зосереджувати в окремому проєкті спільних засобів (у репозиторії посібника - *HoReCa.Domain.Functional*). Для використання в кількох рішеннях такий проєкт варто оформити як пакет NuGet у внутрішньому репозиторії пакетів організації: це спрощує поширення виправлень і нових можливостей.

12.7.4. Обґрунтування функціонального підходу для команди

Переконання колег у доцільності функціонального підходу залежить від характеру проєктів і культури організації. Для систем із граничними вимогами до продуктивності функціональний стиль, імовірно, буде складно обґрунтувати. В інших випадках доцільно наголошувати на таких аргументах (приблизно в порядку важливості):

- **надійність** - передбачувані результати без побічних ефектів і висока тестованість зменшують кількість помилок у промисловій експлуатації;
- **швидкість розробки** - після опанування стилю код пишеться і вдосконалюється швидше, а витрати на виправлення помилок зменшуються;
- **відповідність предметно-орієнтованому проєктуванню** (domain-driven design) - функціональний код добре узгоджується з моделюванням предметної області;
- **підтримка платформою** - розширення функціональних можливостей є задекларованим напрямом розвитку мови C#;
- **доступність для вивчення** - функціональна парадигма оперує меншою кількістю базових понять, ніж ООП;
- **поступовість** - функціональний стиль можна впроваджувати частинами, починаючи з окремих функцій;
- **зрілість** - парадигма існує з 1960-х років і перевірена в багатьох промислових системах.

Починати обговорення доцільно зі знайомих понять (LINQ, незмінність, чисті функції), а не з монад і каррінгу.

12.7.5. Доцільність проєктів на F#

Включення проєктів на F# до рішення не створює технічних проблем. Найдоцільніше використовувати F# для фрагментів, що реалізують правила перетворення даних, - там функціональна мова дає найчистіший і найнадійніший код. Водночас слід ураховувати, що виклик загальних функцій F# з C# може бути менш продуктивним, ніж спеціалізований код C# (див. таблиці 12.2 і 12.3). Головним чинником рішення є готовність команди супроводжувати код на F#; таке рішення має ухвалюватися спільно.

12.7.6. Межі можливостей функціонального стилю

Функціональний стиль забезпечує якісну кодову базу, простішу в супроводі та зрозумілішу для нових учасників команди. Проте він не усуває необхідності в автоматизованому тестуванні, рецензуванні коду й контролі якості і не розв'язує організаційних проблем, зокрема пов'язаних із нечіткими вимогами. Водночас стислість функціонального коду полегшує виявлення невідповідності між назвою функції та її фактичною поведінкою.

12.7.7. Методика функціонального розв'язання задачі

Універсальної методики не існує, проте можна запропонувати таку послідовність дій:

1) розбити задачу на логічні кроки так, як вони описувалися б усно: «спочатку X, потім Y, потім Z»;

2) реалізувати кожен крок окремою невеликою функцією;

3) за можливості подавати дані у вигляді перелічуваних послідовностей (примітивів, об'єктів чи навіть делегатів): функціональний стиль найпотужніший в операціях над списками, подібно до мови SQL;

4) уникати надто довгих ланцюжків викликів: проміжні змінні зі змістовними іменами полегшують налагодження;

5) ретельно покривати модульними тестами кожну невелику функцію.

12.7.8. Фрагменти з граничними вимогами до продуктивності

Якщо певний фрагмент коду, наприклад функцію, що виконується тисячі разів за секунду, неможливо зробити достатньо продуктивним у функціональному стилі, його доцільно реалізувати в імперативному стилі з використанням засобів високопродуктивного програмування платформи

.NET (*Span<T>*, пули масивів тощо). Прагматизм у таких випадках важливіший за послідовність стилю. Імперативний фрагмент варто ізолювати за функціональним інтерфейсом, як це зроблено з методом *IterateUntil* у розділі 10.

12.8. Висновки

1. Функціональний код на C# дещо поступається за продуктивністю оптимальному імперативному коду; найменший розрив мають стислі операції LINQ без проміжних структур і рішення з об'єктами стану типу-значення.

2. Неефективність окремих рішень зумовлена конкретними деталями реалізації (копіювання даних, створення об'єктів у купі), а не функціональним стилем як таким; зокрема, рекурсія з індексом поступається циклу лише на 35 %.

3. Операції введення-виведення домінують у загальному часі виконання реальних застосунків, тож різниця між підходами з урахуванням введення-виведення зазвичай незначна.

4. Ступінь і глибину використання функціонального стилю обирають з урахуванням цілей проєкту, вимог до продуктивності та підготовки команди; критичні за продуктивністю фрагменти доцільно реалізовувати імперативно, ізолюючи їх за функціональним інтерфейсом.

5. Практичною архітектурою функціонального рішення на C# є «функціональне ядро, імперативна оболонка» з окремим проєктом спільних функціональних засобів.

12.9. Контрольні запитання

1. Чому для порівняння продуктивності важливо мати контрольне (базове) рішення?

2. Які показники вимірює BenchmarkDotNet? Чому абсолютні значення часу менш інформативні, ніж співвідношення?

3. Чому рішення з одним викликом *Sum* ефективніше за рішення із *Select* і *Sum*?

4. Що є справжньою причиною низької продуктивності рекурсивного рішення з копіюванням рядка?

5. Чому кортеж як об'єкт стану в *IterateUntil* ефективніший за запис? Який тип даних поєднує переваги обох?

6. Як операції введення-виведення впливають на співвідношення продуктивності імперативного і функціонального коду?

7. У яких класах систем функціональний стиль у критичних фрагментах може бути недоцільним?

8. Як структурувати функціональне рішення на C#? Яку роль відіграє архітектура «функціональне ядро, імперативна оболонка»?

9. Які аргументи доцільно наводити на користь упровадження функціонального стилю в команді?

10. Як діяти, якщо фрагмент коду неможливо зробити достатньо продуктивним у функціональному стилі?

12.10. Практичні завдання

1. Відтворіть вимірювання розділу на власному комп'ютері та порівняйте співвідношення результатів із наведеними в таблицях 12.2-12.4.

2. Додайте до порівняння рішення з *IterateUntil* і записом-структурою (*record struct*) та рішення на основі методу *Aggregate* з достроковим виходом через виняток; поясніть результати.

3. Виміряйте продуктивність функціонального та імперативного рішень для журналу, що зчитується з бази даних SQLite, і порівняйте частку часу обробки даних із результатами для файлу.

4. Реалізуйте задачу про підсумковий залишок мовою F# без використання узагальнених функцій *Seq* (циклом по індексу) і порівняйте продуктивність виклику з C#.

5. Розробіть для своєї команди коротку записку (1-2 сторінки) з обґрунтуванням доцільності функціонального стилю для конкретного проєкту, спираючись на аргументи підрозділу 12.7.4 і результати власних вимірювань.

13. Функціональні бібліотеки NuGet

У попередніх розділах функціональні структури - дискриміновані об'єднання, монади *Maybe*, *Reader*, *State*, мемоізація - реалізовувалися самостійно. Проте в репозиторії пакетів NuGet доступна низка бібліотек, що надають такі структури в готовому вигляді. У цьому розділі розглядаються найпоширеніші з них: їхні можливості, способи використання та критерії вибору.

Більшість таких бібліотек є проєктами з відкритим вихідним кодом, які розвивають окремі розробники або невеликі спільноти, нерідко у вільний від основної роботи час. Це має як переваги, так і ризики. Перевагою є висока якість, зумовлена фаховою зацікавленістю авторів і відсутністю комерційного тиску. Ризик полягає в тому, що підтримку бібліотеки може бути припинено з причин, які неможливо передбачити, а з появою нових версій платформи .NET бібліотека може стати непридатною до використання. Тому перед впровадженням бібліотеки в промисловий проєкт доцільно оцінити:

- активність розвитку проєкту (частоту випусків, реакцію на звернення в системі відстеження завдань);
- наявність спільноти розробників або корпоративної підтримки;
- ліцензійні умови використання;
- обсяг залежності кодової бази від бібліотеки і вартість її можливої заміни.

Відкритість вихідного коду дає змогу в разі припинення офіційної підтримки самостійно супроводжувати власну версію бібліотеки. Рішення про використання сторонньої бібліотеки чи власної реалізації ухвалюється з урахуванням обмежень конкретного проєкту та організації; відомості про версії бібліотек, наведені в розділі, відповідають стану на момент підготовки посібника і потребують актуалізації.

Мета розділу - ознайомити з функціональними бібліотеками NuGet та сформувати навички їх застосування й обґрунтованого вибору.

Після опрацювання розділу студент повинен:

- знати призначення та основні можливості бібліотек OneOf, LanguageExt, Functional.Maybe і CSharpFunctionalExtensions;
- уміти застосовувати їхні структури для розв'язання задач, розглянутих у попередніх розділах;
- уміти порівнювати бібліотечні реалізації з власними та оцінювати ризики залежності від сторонніх бібліотек.

Приклади розділу містяться в проєкті *Ch13.Libraries*, що використовує пакети OneOf 3.0.271, LanguageExt.Core 4.4.9, Functional.Maybe 2.0.20 і CSharpFunctionalExtensions 3.7.0.

13.1. OneOf

Бібліотека **OneOf** (автор - Г. Макінтайр, H. McIntyre) є однією з найпопулярніших функціональних бібліотек для C#. Її мета - надати дискриміновані об'єднання, подібні до об'єднань мови F#. Об'єднання оголошується узагальненим типом *OneOf<T0, T1, ...>* (до 32 варіантів), без абстрактного базового класу. Це дає змогу оголошувати об'єднання «на місці», без допоміжної інфраструктури, яку потребує підхід із розділу 6.

Оскільки спільного базового типу немає, для обробки об'єднання використовується метод *Match*, що приймає по одній функції для кожного варіанта в порядку їх оголошення (лістинг 13.1). Тут об'єднання резервувань із розділу 6 переписано засобами OneOf.

Лістинг 13.1 - Обробка об'єднання методом Match

```
1. public static string Format(
2.     OneOf<RoomStay, RoomStayWithMeals, TableBooking> r) =>
3.     r.Match(
4.         s => $"{s.Hotel}: {s.Nights} ноч.",
5.         m => $"{m.Hotel}: {m.Nights} ноч., харчування {m.Meals}×",
6.         t => $"{t.Restaurant}: столик на {t.Guests}");
```

Порівняно з підходом розділу 6 слід ураховувати такі особливості:

- гілки за замовчуванням немає, тож функцію необхідно задати для кожного варіанта (що водночас гарантує повноту обробки - на відміну від виразу *switch* над ієрархією записів);
- уточнення умов ключовим словом *when* і шаблони властивостей недоступні - відповідну логіку доводиться розміщувати всередині функцій;
- усі функції мають повертати значення одного типу.

Для перевірки варіанта та отримання значення використовуються властивості *IsT0*, *AsT0* тощо, де номер відповідає позиції типу в оголошенні (лістинг 13.2).

Лістинг 13.2 - Перевірка варіанта властивостями IsT і AsT

```
1. public static string Describe(
2.     OneOf<RoomStay, RoomStayWithMeals, TableBooking> r)
3. {
4.     if (r.IsT0)
5.         return "проживання: " + r.AsT0.Hotel;
6.     if (r.IsT1)
7.         return "проживання з харчуванням: " + r.AsT1.Hotel;
8.     return "столик: " + r.AsT2.Restaurant;
9. }
```

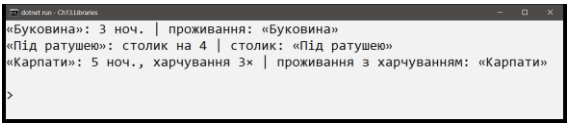


Рисунок 13.1 - Результат виконання прикладу «OneOf»

Бібліотека містить допоміжні типи для варіантів без даних (*None*, *Some*, *Yes*, *No* тощо), однак не реалізує монад: функцій *Map* чи *Bind* для *OneOf* немає. Їх можна реалізувати самостійно, побудувавши аналог *Maybe* з типів *Something<T>* і *None* (лістинги 13.3 і 13.4).

Лістинг 13.3 - Власна функція Map для об'єднання OneOf

```
1. public sealed record Something<T>(T Value);
2.
3. public static class OneOfMaybeExtensions
4. {
5.     // Власна функція Map перетворює об'єднання OneOf на монаду
6.     public static OneOf<Something<TOut>, None> Map<TIn, TOut>(
7.         this OneOf<Something<TIn>, None> @this, Func<TIn, TOut> f) =>
8.         @this.Match<OneOf<Something<TOut>, None>>(
9.             s => new Something<TOut>(f(s.Value)),
10.              none => none);
11.  }
```

Лістинг 13.4 - Використання Maybe на основі OneOf

```
1. var greeting = FindGuest(1)
2.     .Map(name => "Вітаємо, " + name)
3.     .Match(s => s.Value, _ => "Гостя не знайдено");
```

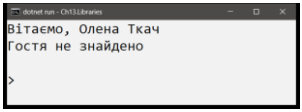


Рисунок 13.2 - Результат виконання прикладу «OneOf і Maybe»

OneOf - легка й проста у використанні бібліотека, доцільна в тих випадках, коли потрібні лише дискриміновані об'єднання.

13.2. LanguageExt

Бібліотека **LanguageExt** (автор - П. Лаут, Р. Louth; пакет *LanguageExt.Core*) - найповніша спроба реалізувати функціональну парадигму в мові C#. Вона містить велику кількість типів і функцій, тож у розділі розглядаються лише засоби, що відповідають матеріалу попередніх розділів. Більшість функцій зосереджено в статичному класі *Prelude*, тож його доцільно підключати директивою *using static LanguageExt.Prelude;* (за потреби - глобально). Слід зазначити, що станом на момент підготовки посібника стабільною є версія 4; нова версія 5, що перебуває в розробці, суттєво змінює програмний інтерфейс.

13.2.1. Option

Аналог *Maybe* у LanguageExt має назву **`Option<T>`** з варіантами *Some* і *None*. Значення створюються функціями *Some(...)* і *None* з класу *Prelude* (лістинг 13.5). Використання *null* у LanguageExt не передбачено: відсутність значення завжди виражається явно варіантом *None*.

Лістинг 13.5 - Функція, що повертає Option

```
1. public static Option<Staff> GetById(int id) =>
2.     Staff.TryGetValue(id, out var s)
3.         ? Some(s)
4.         : None;
```

Функція формування привітання з розділу 7 у LanguageExt має вигляд, наведений у лістингу 13.6. Назви та зміст функцій *Bind* і *Map* відповідають розділу 7: *Bind* приймає функцію, що повертає *Option*, а *Map* - функцію, що повертає звичайне значення.

Лістинг 13.6 - Формування привітання за допомогою Option

```
1. public static Option<string> MakeGreeting(int staffId) =>
2.     Some(staffId)
3.         .Bind(StaffRepository.GetById)
4.         .Map(x => "Вітаємо, " + x.Position + " " + x.Name);
```

Для перетворення *Option* на конкретне значення використовується метод *Match* з функціями для варіантів *Some* і *None* (лістинг 13.7).

Лістинг 13.7 - Обробка Option методом Match

```
1. var message = MakeGreeting(id).Match(
2.     Some: s => s,
3.     None: () => "Працівника не знайдено");
```

LanguageExt містить також багатий набір допоміжних функцій для «піднесених» (elevated) значень: *IfSome*/*IfNone* і *Iter* (аналоги *Tap* для відповідних варіантів), *Filter* (аналог *Where*), *Fold* (аналог *Aggregate*), *MatchUnsafe* (допускає повернення *null*) тощо (лістинг 13.8).

Лістинг 13.8 - Допоміжні функції Option

```
1. StaffRepository.GetById(2)
2.     .Filter(x => x.Position == "офіціант")
3.     .IfSome(x => Console.WriteLine($"Офіціант: {x.Name}"));
4.
5. var namesLength = StaffRepository.GetById(1)
6.     .Fold(0, (acc, x) => acc + x.Name.Length);
```

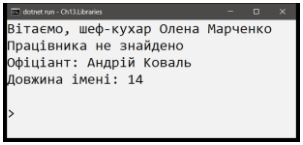


Рисунок 13.3 - Результат виконання прикладу «LanguageExt: Option»

13.2.2. Either

Монада **`Either<L, R>`** в LanguageExt зазвичай використовується для обробки помилок (варіант *Left* містить відомості про помилку), проте придатна й як загальне дискримінованe об'єднання двох типів. У лістингу

13.9 засобами *Either* переписано форматування імен гостей з розділу 6. Типи варіантів не обов'язково мають бути пов'язані успадкуванням.

Лістинг 13.9 - Форматування імені гостя за допомогою Either

```
1. public static string FormatName(Either<UkrainianName, EnglishName> n) =>
2.     n.Match(
3.         Right: e => $"{e.Honorific} {e.First} {e.Last}",
4.         Left: u => $"{u.Last} {u.First} {u.Patronymic}");
```

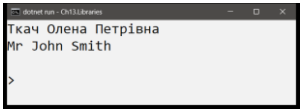


Рисунок 13.4 - Результат виконання прикладу «LanguageExt: Either»

13.2.3. Мемоізація

Для мемоізації LanguageExt надає метод-розширення *Memo* (лістинг 13.10). Він підтримує функції без параметрів і з одним параметром; для функцій з кількома параметрами потрібна власна реалізація (див. розділ 11).

Лістинг 13.10 - Мемоізація засобами LanguageExt

```
1. var calls = 0;
2. Func<string, decimal> price = item =>
3. {
4.     calls++;   // лише для демонстрації кількості викликів
5.     return item == "гриби" ? 280m : 18.5m;
6. };
7. var priceMemo = price.Memo();
8.
9. priceMemo("гриби");
10.  priceMemo("картопля");
11.  priceMemo("гриби");
12.  // calls = 2
```

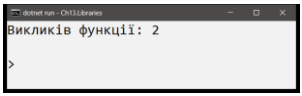


Рисунок 13.5 - Результат виконання прикладу «LanguageExt: Memo»

13.2.4. Reader

Реалізація монади *Reader* у LanguageExt подібна до розглянутої в розділі 7. Функція *asks* створює обчислення, що читає значення із середовища, а *Run* виконує обчислення для конкретного середовища і повертає результат, що враховує можливі помилки (лістинг 13.11).

Лістинг 13.11 - Монада Reader у LanguageExt

```
var bill = asks((VenueConfig cfg) => cfg.VatRate)
    .Map(vat => 1000m * (1 + vat))
    .Bind(gross => asks((VenueConfig cfg) =>
        Math.Round(gross * (1 + cfg.ServiceRate), 2)));

var result = bill.Run(new VenueConfig(0.20m, 0.10m));
```

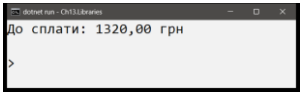


Рисунок 13.6 - Результат виконання прикладу «LanguageExt: Reader»

13.2.5. State

Монада *State* у LanguageExt реалізована як функція від стану, тож обчислення виконується безпосереднім викликом зі значенням початкового стану. Функції *get* і *put* читають і замінюють стан, а підтримка синтаксису запитів LINQ дає змогу записувати послідовність операцій наочно (лістинг 13.12).

Лістинг 13.12 - Монада State у LanguageExt

```
var computation =
    from a in State<int, int>(s => (10, s))
    from b in State<int, int>(s => (a * s, s))
    from c in State<int, int>(s => (b - s, s))
    from _ in put(5)
    from s in get<int>()
    select c / s;

var result = computation(10);
// result.Value = 18, result.State = 5
```

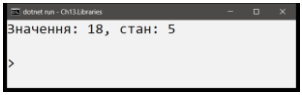


Рисунок 13.7 - Результат виконання прикладу «LanguageExt: State»

13.2.6. Оцінка бібліотеки LanguageExt

LanguageExt - глибока й ретельно спроєктована бібліотека, що містить практично всі засоби, необхідні для максимально можливої реалізації функціональної парадигми в мові C#; розглянуті засоби становлять лише незначну її частину. Водночас слід зважати на наслідки впровадження такої великої бібліотеки: вона визначає стиль усієї кодової бази, потребує від команди часу на опанування, а значні зміни програмного інтерфейсу між основними версіями збільшують вартість супроводу. Альтернативою є власна невелика бібліотека функціональних засобів, як-от *HoReCa.Domain.Functional* у репозиторії посібника: її реалізація потребує

невеликого обсягу коду і дає змогу дотримуватися зручного для команди стилю.

13.3. Functional.Maybe

Бібліотека **Functional.Maybe** (супроводжує A. Цвєтков, A. Tsvetkov; заснована на проєкті Data.Maybe B. Казаріна, W. Casarin) - легка реалізація монади *Maybe*. На відміну від підходу розділу 6, *Maybe<T>* реалізовано як структуру лише для читання з логічною властивістю *HasValue*. Функція зв'язування має назву *Select*, що узгоджується з термінологією LINQ (лістинг 13.13). Чи є таке перевантаження назви зручним, чи радше заплутує, - питання вподобань команди.

Лістинг 13.13 - Перерахунок температури за допомогою Functional.Maybe

```
1. public static string FahrenheitToCelsius(decimal tempF) =>
2.     tempF.ToMaybe()
3.         .Select(x => x - 32)
4.         .Select(x => x * 5)
5.         .Select(x => x / 9)
6.         .Select(x => Math.Round(x, 1))
7.         .Select(x => x + " °C")
8.         .Value;
```

Бібліотека містить метод-розширення *Lookup* для словників, що повертає *Maybe* замість генерування винятку за відсутнього ключа (лістинг 13.14; пор. з підрозділом 4.2).

Лістинг 13.14 - Пошук у словнику з поверненням Maybe

```
1. // Метод Lookup повертає Maybe замість генерування винятку
2. public static Maybe<Reservation> FromDb(string code) =>
3.     Db.Lookup(code);
4.
5. public static Maybe<Reservation> FromPlatform(string code) =>
6.     Platform.Lookup(code);
```

Функція *Do* виконує дію лише за наявності значення (аналог *OnSomething* з розділу 7), а функція *Or* реалізує комбінатор альтернативи (див. підрозділ 5.5) з урахуванням монадичної семантики: кожна альтернатива задається функцією без параметрів, що повертає *Maybe* (лістинг 13.15). Для отримання конкретного значення використовуються властивості *HasValue* і *Value*.

Лістинг 13.15 - Пошук бронювання в кількох джерелах за допомогою Or

```
1. var reservation = FromDb(code)
2.     .Or(() => FromPlatform(code))
3.     .Do(r => Console.WriteLine($"[журнал] знайдено {r.Code}"));
4.
5. var message = reservation.HasValue
6.     ? "Гість: " + reservation.Value.Guest
7.     : "Бронювання не знайдено";
```

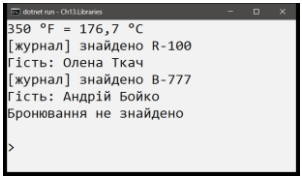


Рисунок 13.8 - Результат виконання прикладу «Functional.Maybe»

13.4. CSharpFunctionalExtensions

Бібліотеку **CSharpFunctionalExtensions** розвиває В. Хоріков (V. Khorikov). Вона містить дві основні структури - *Maybe* і *Result* - з виразним «плинним» (fluent) інтерфейсом.

13.4.1. Maybe

Функція формування привітання засобами CSharpFunctionalExtensions наведена в лістингу 13.16. Метод *AsMaybe* перетворює значення, що може бути *null* (зокрема типу *int?*), на *Maybe*.

Лістинг 13.16 - Формування привітання за допомогою Maybe

```
1. public static Maybe<string> MakeGreeting(int? guestId) =>
2.     guestId.AsMaybe()
3.         .Bind(id => GuestRepository.GetById(id).AsMaybe())
4.         .Map(g => "Вітаємо, " + g.Name);
```

Для отримання конкретного значення використовуються властивості *HasValue*, *Value* або метод *Match* (лістинг 13.17).

Лістинг 13.17 - Обробка Maybe методом Match

```
1. var text = MakeGreeting(id).Match(
2.     x => x,
3.     () => "Гостя не знайдено");
```

Бібліотека містить також методи логічної обробки вмісту: *Where* (перетворює значення, що не задовольняє умову, на відсутнє) та *Or* (задає альтернативне значення). У лістингу 13.18 гостям, молодшим за 18 років, бар пропонує безалкогольне меню.

Лістинг 13.18 - Методи Where і Or

```
1. public static Maybe<string> BarMessage(int? guestId) =>
2.     guestId.AsMaybe()
3.         .Bind(id => GuestRepository.GetById(id).AsMaybe())
4.         .Where(g => g.Age >= 18)
5.         .Map(g => "Коктейльна карта для " + g.Name)
6.         .Or("Безалкогольне меню");
```

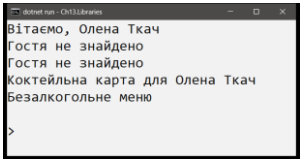


Рисунок 13.9 - Результат виконання прикладу «CSharpFunctionalExtensions» (фрагмент)

13.4.2. Result

Структура **`Result`** подібна до монади *Either* (див. підрозділ 7.11): результат є або успіхом зі значенням, або невдачею з описом помилки. Бібліотека містить набір методів плинного інтерфейсу для побудови ланцюжків перевірок (лістинг 13.19): *ToResult* перетворює *Maybe* на *Result* із заданим повідомленням про помилку; *Ensure* перевіряє умову і в разі невиконання переводить результат у стан невдачі (наступні перевірки після цього не виконуються); *Tap* виконує побічну дію для успішного результату; *Finally* виконується незалежно від стану і перетворює *Result* на кінцеве значення.

Лістинг 13.19 - Перевірка можливості поселення за допомогою Result

```
1. public static string CheckIn(int guestId) =>
2.     GuestRepository.GetById(guestId).AsMaybe()
3.         .ToResult("Гостя не знайдено")
4.         .Ensure(g => g.Age >= 18, "Лише з дорослими")
5.         .Ensure(g => !g.Blacklisted, "Відмовлено в поселенні")
6.         .Tap(g => Console.WriteLine($"[журнал] поселення {g.Name}"))
7.         .Finally(r => r.IsSuccess
8.             ? "Ласкаво просимо, " + r.Value.Name
9.             : "Помилка: " + r.Error);
```

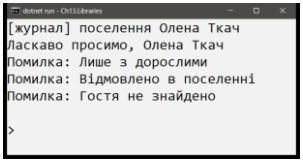


Рисунок 13.10 - Результат виконання прикладу «CSharpFunctionalExtensions» (фрагмент)

13.4.3. Бібліотеки тверджень для модульних тестів

Для бібліотеки CSharpFunctionalExtensions існує доповнення для бібліотеки тверджень Fluent Assertions, що містить твердження для перевірки *Maybe* і *Result* у модульних тестах. Слід зазначити, що з версії 8 бібліотека Fluent Assertions поширюється за комерційною ліцензією для комерційного використання; вільними альтернативами є, зокрема, бібліотеки Shouldly та AwesomeAssertions. Перед вибором бібліотеки тверджень необхідно перевірити актуальні ліцензійні умови.

13.4.4. Оцінка бібліотеки CSharpFunctionalExtensions

CSharpFunctionalExtensions не така всеосяжна, як LanguageExt, проте її структури ретельно спроєктовані, а виразні назви методів і плинний інтерфейс дають змогу стисло описувати складну логіку перевірок. Бібліотека заслуговує на розгляд, якщо проєкту потрібні саме *Maybe* і *Result*.

13.5. Мова F#

Мова F# не є пакетом NuGet, однак проєкти на F# можна безпосередньо підключати до проєктів на C# у межах одного рішення: обидві мови компілюються в той самий проміжний код. Приклади такої взаємодії наведено в розділах 3 і 12; детальний розгляд мови F# виходить за межі посібника.

13.6. Висновки

1. У NuGet доступні функціональні бібліотеки різного масштабу - від всеосяжної LanguageExt до спеціалізованих OneOf, Functional.Maybe і CSharpFunctionalExtensions.

2. OneOf надає дискриміновані об'єднання без базового класу з обов'язковою обробкою всіх варіантів, але без монадичних операцій.

3. LanguageExt реалізує *Option*, *Either*, *Reader*, *State*, мемоізацію та багато інших засобів; її впровадження визначає стиль усієї кодової бази.

4. Functional.Maybe і CSharpFunctionalExtensions надають легкі реалізації *Maybe* (і *Result*) з плинним інтерфейсом.

5. Використання бібліотеки зменшує обсяг власного коду, але створює залежність від стилю, ліцензії та підтримки сторонньої бібліотеки; власна бібліотека функціональних засобів потребує певних зусиль, проте повністю контрольована командою. Вибір ґрунтується на оцінці ризиків і переваг для конкретного проєкту.

13.7. Контрольні запитання

1. Які ризики пов'язані з використанням бібліотек з відкритим вихідним кодом у промислових проєктах? Які критерії доцільно оцінити перед їх впровадженням?

2. Чим дискриміновані об'єднання OneOf відрізняються від реалізації на основі абстрактних записів?

3. Чому *OneOf* не є монадою? Як можна додати до нього монадичні операції?

4. Як у LanguageExt називається аналог *Maybe*? Чим відрізняються функції *Map* і *Bind* для *Option*?

5. Для чого призначені функції *IfSome*, *Filter*, *Fold* бібліотеки LanguageExt?

6. Які обмеження має мемоізація в LanguageExt?

7. Як реалізовано *Maybe* у бібліотеці Functional.Maybe? Для чого призначені функції *Do* і *Or*?

8. Які методи плинного інтерфейсу надає структура *Result* бібліотеки CSharpFunctionalExtensions?

9. Які чинники впливають на вибір між сторонньою бібліотекою і власною реалізацією функціональних засобів?

13.8. Практичні завдання

1. Перепишіть засобами OneOf дискримінований об'єднання способів оплати з практичного завдання розділу 6 і реалізуйте для нього функцію формування рядка чека.

2. Реалізуйте засобами LanguageExt ланцюжок пошуку телефону гостя за номером кімнати (через бронювання та гостя) з розділу 7 з використанням *Option*, *Bind*, *Map* і *Match*.

3. Перепишіть розрахунок вартості проживання з монадою *Reader* (розділ 7) засобами LanguageExt.

4. За допомогою методів *ToResult*, *Ensure*, *Tap* і *Finally* бібліотеки CSharpFunctionalExtensions реалізуйте перевірку можливості бронювання банкетного залу: зал вільний на дату, кількість гостей не перевищує місткості, внесено передоплату.

5. Порівняйте для однієї задачі власну реалізацію *Maybe* з розділу 7 і реалізації бібліотек LanguageExt, Functional.Maybe та CSharpFunctionalExtensions за обсягом коду, читабельністю та можливостями; оформіть порівняння у вигляді таблиці.

14. Комплексний приклад: симуляція «Сезон закладу»

У цьому розділі розглянуті в посібнику засоби поєднуються в завершеному застосунку, побудованому у функціональному стилі. Як приклад обрано покрокову текстову симуляцію управління сезонним закладом гостинності. Такий жанр зручний для навчання: він містить невизначений цикл, взаємодію з користувачем, звернення до зовнішнього сервісу, випадкові події та стан, що змінюється з кожним кроком, - тобто весь спектр задач, у яких функціональний підхід потребує компромісів.

Мета розділу - продемонструвати проєктування та реалізацію завершеного застосунку у функціональному стилі мовою C#.

Після опрацювання розділу студент повинен:

- уміти проєктувати застосунок за архітектурою «функціональне ядро, імперативна оболонка»;
- уміти ізолювати недетерміновані залежності (консоль, час, генератор випадкових чисел, мережа) за інтерфейсами;
- уміти будувати ігровий цикл як невизначену ітерацію незмінного стану з послідовністю фаз;
- уміти поєднувати в одному застосунку дискриміновані об'єднання, монаду *Maybe*, функції вищого порядку та зіставлення зі зразком.

Код розділу міститься в проєкті *Ch14.VenueSeason*.

14.1. Опис симуляції

Користувач у ролі власника відкриває сезонну кав'ярню на гірськолижному курорті. Сезон триває 12 тижнів. На початку власник має 100 000 грн і закуповує все необхідне:

- **кавомашини** - визначають, скільки гостей заклад може обслужити за тиждень;
- **продукти** (партіями) - витрачаються пропорційно кількості гостей;
- **рекламні кампанії** - потрібні для проведення дегустаційних акцій;
- **страхові поліси** - покривають збитки від надзвичайних подій;
- **ремонтні комплекти** - дають змогу відновити обладнання в разі поломки.

Кожен тиждень складається з таких кроків:

1) виведення прогнозу погоди, отриманого від зовнішнього погодного сервісу;

2) вибір дії на тиждень із доступних за поточних умов (закупівля на ринку, дегустаційна акція, кейтеринг-замовлення або звичайна робота з повним завантаженням); акція та кейтеринг потребують виконання міні-гри - швидкого й точного введення коду замовлення;

3) підрахунок гостей, виручки, витрат і використаних продуктів;
4) випадкова подія (позитивна або негативна);
5) перевірка умов завершення.

Симуляція завершується після 12-го тижня або достроково - у разі банкрутства (від'ємного капіталу чи втрати всіх кавомашин). Сезон вважається успішним, якщо підсумковий капітал становить не менше 200 000 грн.

14.2. Архітектура застосунку

Застосунок побудовано за архітектурою «функціональне ядро, імперативна оболонка» (рис. 14.1). Ядро - ігровий цикл, що застосовує до незмінного стану послідовність фаз тижня, кожна з яких є функцією, що перетворює старий стан на новий. Усі недетерміновані залежності - консоль (взаємодія з власником), годинник, генератор випадкових чисел і мережевий погодний сервіс - винесено в тонку оболонку і приховано за інтерфейсами. Завдяки цьому всю логіку ядра можна протестувати з детермінованими імітаціями залежностей.



Рисунок 14.1 - Архітектура симуляції «Сезон закладу»

14.3. Реалізація

14.3.1. Взаємодія з користувачем

Насамперед потрібен засіб обміну даними між застосунком і користувачем. Можливі результати взаємодії описано двома дискримінованими об'єднаннями (див. розділ 6): *UserInput* - для введення (ціле число, текст, порожнє введення або помилка консолі) та *Operation* - для операцій без результату, як-от виведення повідомлення (лістинг 14.1).

Лістинг 14.1 - Об'єднання результатів взаємодії з користувачем

```
// Введення від гравця
public abstract record UserInput
{
    private UserInput() { }
    public sealed record Integer(int Value) : UserInput;
    public sealed record Text(string Value) : UserInput;
    public sealed record Empty : UserInput;
    public sealed record InputError(Exception Error) : UserInput;
}

// Результат операції без даних (виведення повідомлення)
public abstract record Operation
{
    private Operation() { }
    public sealed record Success : Operation;
    public sealed record Failure(Exception Error) : Operation;
}
```

Клас *PlayerInteraction* (лістинг 14.2) є єдиним місцем, де введений рядок аналізується й перетворюється на варіант об'єднання. Метод *WriteMessageConditional* поєднує виведення повідомлення з умовою, що дає змогу уникнути інструкцій *if* у коді ядра.

Лістинг 14.2 - Взаємодія з користувачем

```
public interface IPlayerInteraction
{
    UserInput GetInput(params string[] prompt);
    Operation WriteMessage(params string[] lines);
    Operation WriteMessageConditional(bool condition,
                                      params string[] lines);
}

public sealed class PlayerInteraction(IConsole console)
    : IPlayerInteraction
{
    public UserInput GetInput(params string[] prompt)
    {
        try
        {
            WriteMessage(prompt);
            var text = console.ReadLine() ?? "";
            return text.Trim() switch
            {
                "" => new UserInput.Empty(),
                var t when int.TryParse(t, out var n) =>
                    new UserInput.Integer(n),
                var t => new UserInput.Text(t)
            };
        }
        catch (Exception e)
        {
            return new UserInput.InputError(e);
        }
    }

```

```
32.      public Operation WriteMessage(params string[] lines)
33.      {
34.          try
35.          {
36.              foreach (var line in lines)
37.                  console.WriteLine(line);
38.              return new Operation.Success();
39.          }
40.          catch (Exception e)
41.          {
42.              return new Operation.Failure(e);
43.          }
44.      }
45.
46.      public Operation WriteMessageConditional(bool condition,
47.                                               params string[] lines) =>
48.          condition
49.              ? WriteMessage(lines)
50.              : new Operation.Success();
51.  }
```

14.3.2. Правила симуляції

На початку роботи користувачеві пропонується ознайомитися з правилами. Позитивні відповіді задано колекцією, а виведення правил виконується методом *WriteMessageConditional* (лістинг 14.3).

Лістинг 14.3 - Пропозиція ознайомитися з правилами

```
1. public interface IDisplayInstructions
2. {
3.     void DisplayInstructions();
4. }
5.
6. public sealed class DisplayInstructions(IPlayerInteraction ui)
7.     : IDisplayInstructions
8. {
9.     private static readonly string[] PositiveResponses =
10.         ["ТАК", "Т", "ЗВІСНО", "ДАВАЙТЕ", "YES", "Y"];
11.
12.     void IDisplayInstructions.DisplayInstructions()
13.     {
14.         var answer = ui.GetInput("Показати правила симуляції?");
15.
16.         var display = answer switch
17.         {
18.             UserInput.Text t when PositiveResponses
19.                 .Contains(t.Value.ToUpper()) => true,
20.             _ => false
21.         };
22.
23.         ui.WriteMessageConditional(display,
24.             "«Сезон закладу» - правила",
25.             "Ви відкриваєте сезонну кав'ярню",
26.             "на гірськолижному курорті.",
27.             "Сезон триває 12 тижнів. Мета - завершити сезон",
28.             "з капіталом не менше 200 000 грн і не збанкрутувати.");
29.     }
```

```
30.  }
```

14.3.3. Стартові закупівлі

Модуль стартових закупівель містить «цикли в циклах»: перебір позицій закупівель, а всередині - повторення запиту доти, доки користувач не введе коректне значення; наприкінці - підтвердження всього вибору з можливістю почати закупівлі спочатку. Модуль має власний стан із метаданими (ознака згоди власника з вибором), а решті застосунку передається спільний стан запасів без цих метаданих (лістинг 14.4).

Лістинг 14.4 - Стан модуля закупівель і спільний стан запасів

```
1. // Стан модуля стартових закупівель (з метаданими модуля)
2. public sealed record InventorySelectionState(
3.     int CoffeeMachines, int FoodBatches, int AdCampaigns,
4.     int Insurances, int RepairKits, int Cash,
5.     bool OwnerIsHappyWithSelection);
6.
7. // Стан запасів, спільний для всіх модулів симуляції
8. public sealed record InventoryState(
9.     int CoffeeMachines, int FoodBatches, int AdCampaigns,
10.      int Insurances, int RepairKits, int Cash);
```

Кожна позиція закупівлі описується записом, що містить назву, ціну і функцію оновлення відповідної властивості стану (лістинг 14.5). Набір позицій оголошується масивом таких записів (лістинг 14.6): додавання нової позиції не потребує змін логіки модуля.

Лістинг 14.5 - Опис позиції закупівлі

```
1. public sealed record InventoryItem(
2.     string Name,
3.     int CostPerItem,
4.     Func<int, InventorySelectionState, InventorySelectionState> Update);
```

Лістинг 14.6 - Набір позицій закупівлі

```
1. private static readonly InventoryItem[] Items =
2. [
3.     new("Кавомашини", 15_000,
4.         (q, s) => s with { CoffeeMachines = q }),
5.     new("Продукти (партія)", 1_000,
6.         (q, s) => s with { FoodBatches = q }),
7.     new("Рекламні кампанії", 3_000,
8.         (q, s) => s with { AdCampaigns = q }),
9.     new("Страхові поліси", 2_000,
10.          (q, s) => s with { Insurances = q }),
11.      new("Ремонтні комплекти", 1_500,
12.          (q, s) => s with { RepairKits = q })
13.  ];
```

Узагальнена функція закупівлі однієї позиції (лістинг 14.7) обчислює максимальну доступну кількість, виконує перший запит і за допомогою методу *IterateUntil* (див. розділ 10) повторює запит, доки введене значення не стане коректним. Повідомлення про причину невдачі формується

виразом *switch* над варіантами *UserInput*. Схему процесу наведено на рис. 14.2.

Лістинг 14.7 - Закупівля однієї позиції з перевіркою введення

```
1. private InventorySelectionState MakeSelection(
2.     InventorySelectionState oldState, InventoryItem item)
3. {
4.     var affordable = oldState.Cash / item.CostPerItem;
5.     bool IsValid(int x) => x >= 0 && x <= affordable;
6.
7.     var firstAttempt = ui.GetInput(
8.         $"{item.Name}: {item.CostPerItem} грн за одиницю. " +
9.         $"Скільки придбати? (не більше {affordable})");
10.
11.      var validInput = firstAttempt.IterateUntil(
12.          x =>
13.          {
14.              ui.WriteMessage(x switch
15.              {
16.                  UserInput.Integer { Value: < 0 } =>
17.                      "Кількість не може бути від'ємною",
18.                  UserInput.Integer =>
19.                      "Недостатньо коштів для такої кількості",
20.                  UserInput.Empty => "Потрібно ввести значення",
21.                  UserInput.Text => "Потрібно ввести ціле число",
22.                  UserInput.InputError e =>
23.                      "Помилка введення: " + e.Error.Message,
24.                  _ => "Невідома відповідь"
25.              });
26.              return ui.GetInput("Спробуйте ще раз:");
27.          },
28.          x => x is UserInput.Integer i && IsValid(i.Value));
29.
30.      var quantity = ((UserInput.Integer)validInput).Value;
31.      return item.Update(quantity, oldState) with
32.      {
33.          Cash = oldState.Cash - quantity * item.CostPerItem
34.      };
35.  }
```

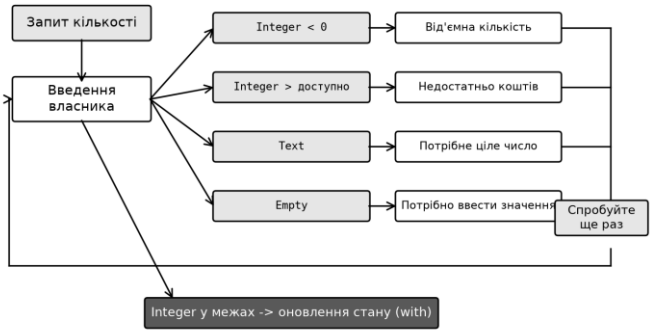


Рисунок 14.2 - Процес вибору кількості під час стартових закупівель

Після завершення умови *IterateUntil* значення гарантовано є варіантом *UserInput.Integer*, тому приведення типу безпечне. Запит підтвердження вибору реалізовано в лістингу 14.8. Тут використано рекурсивну локальну функцію: кількість некоректних відповідей на запитання «так/ні» практично обмежена, тож ризику переповнення стека немає.

Лістинг 14.8 - Підтвердження вибору закупівель

```
1. private InventorySelectionState UpdateOwnerIsHappy(
2.     InventorySelectionState oldState)
3. {
4.     string[] yes = ["ТАК", "Т", "YES", "Y"];
5.     string[] no = ["НІ", "Н", "NO", "N"];
6.
7.     DisplayInventory(oldState);
8.
9.     bool GetResponse(string message) =>
10.          ui.GetInput(message) switch
11.          {
12.              UserInput.Text t
13.                  when yes.Contains(t.Value.ToUpper()) => true,
14.              UserInput.Text t
15.                  when no.Contains(t.Value.ToUpper()) => false,
16.              _ => GetResponse("Відповідайте «так» або «ні»:")
17.          };
18.
19.      var happy = GetResponse("Чи влаштовують вас закупівлі?");
20.      return oldState with
21.      {
22.          OwnerIsHappyWithSelection = happy,
23.          // у разі відмови закупівлі починаються спочатку
24.          Cash = happy ? oldState.Cash : StartCapital
25.      };
26.  }
```

Загальна функція модуля (лістинг 14.9) поєднує описані елементи: метод *Aggregate* послідовно застосовує закупівлю кожної позиції до стану,

функція *Map* (див. розділ 5) - запит підтвердження, а *IterateUntil* повторює весь процес, доки власник не погодиться з вибором.

Лістинг 14.9 - Функція стартових закупівель

```
1. public InventoryState SelectInitialInventory()
2. {
3.     var initial = new InventorySelectionState(0, 0, 0, 0, 0,
4.                                                 StartCapital, false);
5.     var final = initial.IterateUntil(
6.         x => Items.Aggregate(x with { Cash = StartCapital },
7.                              MakeSelection)
8.                  .Map(UpdateOwnerIsHappy),
9.         x => x.OwnerIsHappyWithSelection);
10.
11.      return new InventoryState(final.CoffeeMachines, final.FoodBatches,
12.          final.AdCampaigns, final.Insurances, final.RepairKits,
13.          final.Cash);
14.  }
```

Фрагмент роботи модуля зі скриптованим користувачем (його відповіді позначено символом «>») наведено нижче. Некоректні відповіді «багато» та «-1» відхиляються з відповідними поясненнями.

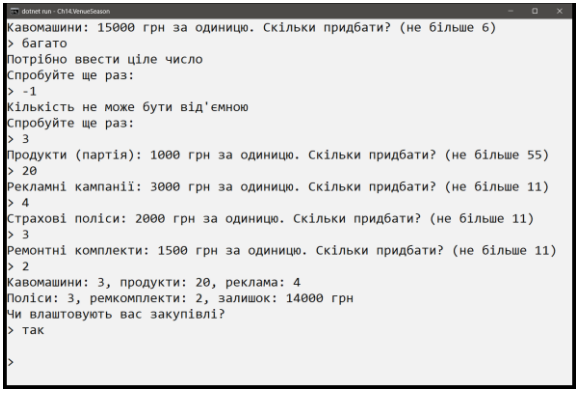


Рисунок 14.3 - Результат виконання прикладу «Підготовка» (фрагмент)

14.3.4. Ігровий цикл

Стан симуляції описується незмінним записом (лістинг 14.10), а можливі дії власника - переліченням (лістинг 14.11).

Лістинг 14.10 - Стан симуляції

```
1. public sealed record GameState(
2.     InventoryState Inventory,
3.     int Week,
4.     int GuestsTotal,
5.     int BonusGuests,
6.     OwnerActions ActionThisWeek,
7.     bool IsBankrupt,
8.     bool SeasonFinished);
```

Лістинг 14.11 - Дії власника

```
1. public enum OwnerActions
2. {
3.     Unavailable,
4.     BuyAtMarket,      // закупівля на оптовому ринку
5.     TastingPromo,     // дегустаційна акція
6.     CateringOrder,    // виїзне обслуговування (кейтеринг)
7.     RegularWork       // звичайна робота з повним завантаженням
8. }
```

Метод-розширення *ContinueTurn* (лістинг 14.12) виконує наступну функцію лише тоді, коли симуляцію не завершено. За змістом він подібний до функції *Bind* монади *Maybe* з двома станами: після завершення симуляції наступні фази тижня автоматично пропускаються, і перевіряти умови завершення після кожної фази не потрібно.

Лістинг 14.12 - Метод ContinueTurn

```
1. public static class GameStateExtensions
2. {
3.     // Подібно до Bind для Maybe: після завершення симуляції
4.     // наступні фази не виконуються
5.     public static GameState ContinueTurn(
6.         this GameState @this, Func<GameState, GameState> f) =>
7.         @this.IsBankrupt || @this.SeasonFinished
8.             ? @this
9.             : f(@this);
10.  }
```

Кожна фаза тижня реалізує узагальнений інтерфейс (лістинг 14.13), а весь ігровий цикл записується кількома рядками (лістинг 14.14): метод *Aggregate* послідовно застосовує фази тижня до стану, а *IterateUntil* повторює тижні до завершення симуляції.

Лістинг 14.13 - Інтерфейс фази тижня

```
1. public interface IGamePhase
2. {
3.     GameState DoPhase(IPlayerInteraction ui, GameState oldState);
4. }
```

Лістинг 14.14 - Ігровий цикл

```
1. public sealed class Game
2. {
3.     public GameState Play(GameState initialState,
4.                           IPlayerInteraction ui,
5.                           params IGamePhase[] phases) =>
6.         initialState.IterateUntil(
7.             x => phases.Aggregate(x,
8.                 (acc, phase) => acc.ContinueTurn(
9.                     s => phase.DoPhase(ui, s))),
10.             x => x.IsBankrupt || x.SeasonFinished);
11.  }
```

14.3.5. Фаза «Прогноз погоди»

Прогноз погоди отримується від зовнішнього вебсервісу. Звернення до мережі може завершитися невдачею з багатьох причин, тож результат загортається в монаду *Maybe* (див. розділ 7). Оскільки системний клас *HttpClient* не має інтерфейсу, придатного для підстановки імітації, для нього створюється оболонка (лістинг 14.15). Аналогічні оболонки створено для генератора випадкових чисел і поточного часу (лістинг 14.16): використання класу *Random* чи властивості *DateTime.Now* безпосередньо в коді ядра зробило б функції нечистими.

Лістинг 14.15 - Оболонка над HttpClient

```
1. public interface IHttpClient
2. {
3.     Task<HttpResponseMessage> GetAsync(string url);
4. }
5.
6. public sealed class HttpClientShim(HttpClient httpClient) : IHttpClient
7. {
8.     public Task<HttpResponseMessage> GetAsync(string url) =>
9.         httpClient.GetAsync(url);
10.  }
```

Лістинг 14.16 - Оболонки над генератором випадкових чисел і годинником

```
1. // Оболонки над недетермінованими засобами платформи
2. public interface IRandomNumberGenerator
3. {
4.     int BetweenZeroAnd(int maxExclusive);
5. }
6.
7. public sealed class RandomNumberGenerator(int? seed = null)
8.     : IRandomNumberGenerator
9. {
10.      private readonly Random random =
11.          seed is null ? new Random() : new Random(seed.Value);
12.
13.      public int BetweenZeroAnd(int maxExclusive) =>
14.          random.Next(0, maxExclusive);
15.  }
```

```
16.
17.  public interface ITimeService
18.  {
19.      DateTime Now();
20.  }
21.
22.  public sealed class TimeService : ITimeService
23.  {
24.      public DateTime Now() => DateTime.Now;
25.  }
```

Клас отримання даних із вебсервісу (лістинг 14.17) повертає *Maybe<T>*: у разі неуспішного коду відповіді - *Nothing*, у разі винятку - *Error*. Десеріалізація JSON виконується функцією *Map*, тож вона відбувається лише за наявності вмісту.

Лістинг 14.17 - Отримання даних вебсервісу з поверненням Maybe

```
1. public interface IFetchWebApiData
2. {
3.     Task<Maybe<T>> FetchData<T>(string url);
4. }
5.
6. public sealed class FetchWebApiData(IHttpClient httpClient)
7.     : IFetchWebApiData
8. {
9.     public async Task<Maybe<T>> FetchData<T>(string url)
10.      {
11.          try
12.          {
13.              var response = await httpClient.GetAsync(url);
14.              var content = response.IsSuccessStatusCode
15.                  ? (await
     response.Content.ReadAsStringAsync()).ToMaybe()
16.                  : new Maybe<string>.Nothing();
17.              return content.Map(
18.                  json => JsonSerializer.Deserialize<T>(json));
19.          }
20.          catch (Exception e)
21.          {
22.              return new Maybe<T>.Error(e);
23.          }
24.      }
25.  }
```

Структура даних погодного сервісу та сама фаза наведені в лістингах 14.18 і 14.19. Тиждень вибирається з отриманих даних функцією *Map*, а повідомлення для користувача формується функцією *Match* з урахуванням усіх трьох варіантів *Maybe*. У навчальному прикладі обробник HTTP повертає заздалегідь підготовлені дані, що забезпечує відтворюваність результатів без доступу до мережі; для роботи з реальним сервісом достатньо передати в оболонку звичайний *HttpClient*.

Лістинг 14.18 - Структура даних погодного сервісу

```
1. public sealed record WeatherData(WeekWeather[] weeks);
2. public sealed record WeekWeather(int week, int snow_cm, int temp_c);
```

Лістинг 14.19 - Фаза «Прогноз погоди»

```
1. public sealed class WeatherReport(IFetchWebApiData api) : IGamePhase
2. {
3.     private const string Url = "https://weather.example/api/season";
4.
5.     private static string Format(WeekWeather w) =>
6.         $"Тиждень {w.week}: сніг {w.snow_cm} см, " +
7.         $"температура {w.temp_c} °C";
8.
9.     public GameState DoPhase(IPlayerInteraction ui, GameState oldState)
10.      {
11.          // Синхронне очікування прийнятне: застосунок
     однокористувацький
12.          var data = api.FetchData<WeatherData>(Url).Result;
13.
14.          var thisWeek = data.Map(x =>
15.              x.weeks.SingleOrDefault(w => w.week == oldState.Week));
16.
17.          var message = thisWeek.Map(Format).Match(
18.              something: s => s,
19.              nothing: () => "Прогноз погоди недоступний",
20.              error: e => "Помилка погодного сервісу: " + e.Message);
21.
22.          ui.WriteMessage("", message);
23.          return oldState;
24.      }
25.  }
```

14.3.6. Фаза «Вибір дії»

Набір доступних дій залежить від випадкових умов тижня: наплив гостей на вихідних, робота оптового ринку, наявність рекламних кампаній, запит на кейтеринг. Відповідно до функціонального стилю ці умови обчислюються як послідовність логічних значень без інструкцій *if* і змінних, що змінюються. Кожна потенційна дія подається тернарним виразом, що повертає або дію, або спеціальне значення *Unavailable*, яке потім відфільтровується методом *Where*; доступні дії нумеруються перевантаженою формою *Select* з індексом (лістинг 14.20).

Лістинг 14.20 - Формування переліку доступних дій

```
1. var isWeekendRush = rnd.BetweenZeroAnd(100) > 66;
2. var marketOpen = rnd.BetweenZeroAnd(100) > (isWeekendRush ? 80 : 30);
3. var promoPossible = oldState.Inventory.AdCampaigns > 0
4.                     && rnd.BetweenZeroAnd(100) > 40;
5. var cateringRequest = rnd.BetweenZeroAnd(100) > 50;
6.
7. var options = new[]
8.     {
9.         marketOpen ? OwnerActions.BuyAtMarket
10.                    : OwnerActions.Unavailable,
11.          promoPossible ? OwnerActions.TastingPromo
12.                        : OwnerActions.Unavailable,
13.          cateringRequest ? OwnerActions.CateringOrder
14.                          : OwnerActions.Unavailable,
```

```
15.          OwnerActions.RegularWork
16.      }
17.      .Where(x => x != OwnerActions.Unavailable)
18.      .Select((x, i) => (Action: x, ChoiceNumber: i + 1))
19.      .ToArray();
```

Повідомлення з переліком дій формується поєднанням заголовка та пронумерованих варіантів методом *Concat* (лістинг 14.21).

Лістинг 14.21 - Формування повідомлення про доступні дії

```
1. ui.WriteMessage(new[]
2.     {
3.         isWeekendRush ? "Наплив гостей на вихідних!"
4.                       : "Звичайний тиждень.",
5.         "Доступні дії:"
6.     }
7.     .Concat(options.Select(x => $"  {x.ChoiceNumber} - " +
8.         x.Action switch
9.         {
10.              OwnerActions.BuyAtMarket => "закупівля на ринку",
11.              OwnerActions.TastingPromo => "дегустаційна акція",
12.              OwnerActions.CateringOrder => "кейтеринг-замовлення",
13.              _ => "звичайна робота"
14.          }))
15.      .ToArray());
```

Вибір користувача перевіряється в невизначеному циклі, після чого вираз *switch* обирає функцію-обробник дії, і вона застосовується до стану (лістинг 14.22).

Лістинг 14.22 - Вибір і виконання дії

```
1. var choice = ui.GetInput("Що робитимете цього тижня?")
2.     .IterateUntil(
3.         _ => ui.GetInput("Такого варіанта немає. Ще раз:"),
4.         x => x is UserInput.Integer i
5.              && options.Any(o => o.ChoiceNumber == i.Value));
6.
7. var action = options.Single(o =>
8.     o.ChoiceNumber == ((UserInput.Integer)choice).Value).Action;
9.
10.  Func<GameState, GameState> actionFunc = action switch
11.  {
12.      OwnerActions.BuyAtMarket => BuyAtMarket,
13.      OwnerActions.TastingPromo => s => TastingPromo(ui, s),
14.      OwnerActions.CateringOrder => s => CateringOrder(ui, s),
15.      _ => RegularWork
16.  };
17.
18.  return actionFunc(oldState) with { ActionThisWeek = action };
```

Дегустаційна акція та кейтеринг-замовлення потребують виконання міні-гри: користувач має швидко й точно ввести випадковий чотирилітерний код замовлення. Результатом є коефіцієнт успішності від 0 до 1, що дорівнює добутку точності введення (по 25 % за кожен правильний символ) і коефіцієнта часу (зменшується на 0,1 за кожну

секунду). Міні-гру реалізовано окремим модулем, залежним від генератора випадкових чисел, засобу взаємодії та годинника (лістинг 14.23).

Лістинг 14.23 - Міні-гра введення коду замовлення

```
1. public interface IPlayMiniGame
2. {
3.     decimal PlayForSuccessFactor();
4. }
5.
6. public sealed class OrderMiniGame(IRandomNumberGenerator rnd,
7.     IPlayerInteraction ui, ITimeService time) : IPlayMiniGame
8. {
9.     private static decimal RateAccuracy(string expected,
10.                                        string actual) =>
11.          expected.Zip(actual,
12.                       (x, y) => char.ToUpper(x) == char.ToUpper(y))
13.                  .Sum(ok => ok ? 1 : 0) / 4m;
14.
15.      public decimal PlayForSuccessFactor()
16.      {
17.          var code = new string(Enumerable.Range(0, 4)
18.              .Select(_ => (char)('A' + rnd.BetweenZeroAnd(26)))
19.              .ToArray());
20.
21.          var start = time.Now();
22.          var attempt =
23.              ui.GetInput($"Швидко введіть код замовлення: {code}");
24.          var input = attempt.IterateUntil(
25.              _ => ui.GetInput($"Повторіть код: {code}"),
26.              x => x is not UserInput.InputError);
27.          var end = time.Now();
28.
29.          var textAccuracy = input is UserInput.Text { Value.Length: 4 }
    t
30.              ? RateAccuracy(code, t.Value)
31.              : 0m;
32.          var seconds = (decimal)(end - start).TotalSeconds;
33.          var timeAccuracy = Math.Max(0m, 1m - 0.1m * seconds);
34.          return textAccuracy * timeAccuracy;
35.      }
36.  }
```

Обробник дегустаційної акції (лістинг 14.24) використовує коефіцієнт успішності для визначення кількості додаткових гостей і повідомлення користувачеві, сформованого виразом *switch* з реляційними шаблонами.

Лістинг 14.24 - Обробник дегустаційної акції

```
1. private GameState TastingPromo(IPlayerInteraction ui, GameState s)
2. {
3.     ui.WriteMessage("Дегустація: гості чекають на замовлення!");
4.     var accuracy = miniGame.PlayForSuccessFactor();
5.
6.     ui.WriteMessage(accuracy switch
7.     {
8.         >= 0.75m => "Блискуче! Про кав'ярню заговорив увесь курорт.",
9.         0m => "Замовлення переплутано, гості розійшлися.",
```

```
            _ => "Непогано: акція привабила нових гостей."
        });

        return s with
        {
            BonusGuests = s.BonusGuests + (int)(300 * accuracy),
            Inventory = s.Inventory with
            {
                AdCampaigns = s.Inventory.AdCampaigns - 1
            }
        };
    }
```

14.3.7. Фази «Підсумки тижня», «Випадкова подія» і «Кінець тижня»

Фаза підсумків (лістинг 14.25) не потребує участі користувача: вона обчислює кількість гостей з урахуванням місткості (більшої за звичайної роботи), бонусних гостей і запасу продуктів, виручку, витрати та новий стан.

Лістинг 14.25 - Фаза «Підсумки тижня»

```
public sealed class UpdateProgress(IRandomNumberGenerator rnd)
    : IGamePhase
{
    private const int AverageCheck = 150;
    private const int WeeklyCosts = 26_000;

    public GameState DoPhase(IPlayerInteraction ui, GameState oldState)
    {
        var inv = oldState.Inventory;
          var fullLoad =
              oldState.ActionThisWeek == OwnerActions.RegularWork;
          var capacity = inv.CoffeeMachines * (fullLoad ? 140 : 70);
          var guests = Math.Min(capacity + oldState.BonusGuests,
                                inv.FoodBatches * 100);
          var foodUsed = (guests + 99) / 100 + rnd.BetweenZeroAnd(2);
          var revenue = guests * AverageCheck;

          var newState = oldState with
          {
              GuestsTotal = oldState.GuestsTotal + guests,
              BonusGuests = 0,
              Inventory = inv with
              {
                  FoodBatches = Math.Max(0, inv.FoodBatches - foodUsed),
                  Cash = inv.Cash + revenue - WeeklyCosts
              }
          };

          ui.WriteMessage(
              $"Гостей за тиждень: {guests}, виручка: {revenue} грн",
              $"Капітал: {newState.Inventory.Cash} грн, " +
              $"партій продуктів: {newState.Inventory.FoodBatches}");
          return newState;
      }
```

```
35. }
```

Випадкові події (лістинг 14.26) задано масивом функцій, що за взаємодією та поточним станом обчислюють новий стан; генератор випадкових чисел обирає подію або її відсутність. Наслідки негативних подій залежать від наявності страхових полісів і ремонтних комплектів, що відображено тернарними виразами. Функція *Map* дає змогу виконати виведення повідомлення і повернути новий стан в одному виразі.

Лістинг 14.26 - Фаза «Випадкова подія»

```
1. public sealed class RandomEvents(IRandomNumberGenerator rnd)
2.     : IGamePhase
3. {
4.     private static readonly Func<IPlayerInteraction, GameState,
5.                                  GameState>[] Events =
6.     [
7.         // позитивні події
8.         (ui, s) => s.Map(_ => ui.WriteMessage(
9.                 "Подія: відомий блогер похвалив вашу каву!"))
10.             .Map(_ => s with { BonusGuests = s.BonusGuests + 150 }),
11.         (ui, s) => s.Map(_ => ui.WriteMessage(
12.                 "Подія: постачальник подарував 3 партії продуктів."))
13.             .Map(_ => s with { Inventory = s.Inventory with
14.                 { FoodBatches = s.Inventory.FoodBatches + 3 } }),
15.         // негативні події
16.         (ui, s) => s.Inventory.RepairKits > 0
17.             ? s.Map(_ => ui.WriteMessage(
18.                     "Подія: поломка кавомашини, " +
19.                     "використано ремкомплект."))
20.                .Map(_ => s with { Inventory = s.Inventory with
21.                     { RepairKits = s.Inventory.RepairKits - 1 } })
22.             : s.Map(_ => ui.WriteMessage(
23.                     "Подія: поломка кавомашини, ремонт неможливий."))
24.                .Map(_ => s with { Inventory = s.Inventory with
25.                     { CoffeeMachines = s.Inventory.CoffeeMachines - 1
    }
26.                 }),
27.         (ui, s) => s.Inventory.Insurances > 0
28.             ? s.Map(_ => ui.WriteMessage(
29.                     "Подія: затоплення підсобки, " +
30.                     "збиток покрито полісом."))
31.                .Map(_ => s with { Inventory = s.Inventory with
32.                     { Insurances = s.Inventory.Insurances - 1 } })
33.             : s.Map(_ => ui.WriteMessage(
34.                     "Подія: затоплення підсобки, збиток 15 000 грн."))
35.                .Map(_ => s with { Inventory = s.Inventory with
36.                     { Cash = s.Inventory.Cash - 15_000 } })
37.     ];
38.
39.     public GameState DoPhase(IPlayerInteraction ui, GameState
    oldState)
40.     {
41.         var roll = rnd.BetweenZeroAnd(10);
42.         return roll < Events.Length
43.             ? Events[roll](ui, oldState)
44.             : oldState;
```

```
45.     }
46. }
```

Фаза завершення тижня (лістинг 14.27) перевіряє умови банкрутства та завершення сезону й виводить підсумкове повідомлення.

Лістинг 14.27 - Фаза «Кінець тижня»

```
1. public sealed class EndOfWeek : IGamePhase
2. {
3.     private const int Target = 200_000;
4.
5.     public GameState DoPhase(IPlayerInteraction ui, GameState oldState)
6.     {
7.         var bankrupt = oldState.Inventory.Cash < 0
8.                        || oldState.Inventory.CoffeeMachines == 0;
9.         var finished = oldState.Week >= 12;
10.
11.          ui.WriteMessageConditional(bankrupt,
12.              "Кав'ярню закрито: продовжувати роботу неможливо.");
13.          ui.WriteMessageConditional(finished && !bankrupt,
14.              $"Сезон завершено. Гостей: {oldState.GuestsTotal}, " +
15.              $"капітал: {oldState.Inventory.Cash} грн.",
16.              oldState.Inventory.Cash >= Target
17.                  ? "Мету досягнуто - сезон успішний!"
18.                  : "Мету не досягнуто - сезон збитковий.");
19.
20.          return oldState with
21.          {
22.              IsBankrupt = bankrupt,
23.              SeasonFinished = finished,
24.              Week = oldState.Week + 1
25.          };
26.      }
27. }
```

14.3.8. Композиція застосунку

Усі компоненти поєднуються в головній програмі (лістинг 14.28). Саме тут, у межах імперативної оболонки, обираються конкретні реалізації залежностей: для інтерактивного режиму - системна консоль, справжній генератор випадкових чисел і годинник; для навчального прикладу - скриптований користувач, генератор із фіксованим початковим значенням та імітований годинник. Для невеликого застосунку контейнер інверсії керування не потрібен.

Лістинг 14.28 - Композиція застосунку

```
1. IConsole console = interactive
2.     ? new ConsoleShim()
3.     : new BotConsole(BotAnswer);
4. var ui = new PlayerInteraction(console);
5. IRandomNumberGenerator rnd = new RandomNumberGenerator(
6.     interactive ? null : 13);
7. ITimeService time =
8.     interactive ? new TimeService() : new FakeTimeService();
9. // Для реального погодного сервісу: new HttpClient()
```

```
var http = new HttpClientShim(new HttpClient(new
    FakeWeatherHandler()));

// Частини симуляції; у навчальному режимі виводяться окремо
void Part(string name, Action action)
{
    if (interactive) action();
    else BookDemo.Run(name, action);
}

InventoryState inventory = null!;
Part("Підготовка", () =>
{
    IDisplayInstructions instructions = new DisplayInstructions(ui);
    instructions.DisplayInstructions();
    inventory = new InventorySelection(ui).SelectInitialInventory();
});

Part("Сезон", () =>
{
    var initialState = new GameState(inventory, Week: 1,
        GuestsTotal: 0, BonusGuests: 0,
        ActionThisWeek: OwnerActions.RegularWork,
        IsBankrupt: false, SeasonFinished: false);

    var miniGame = new OrderMiniGame(rnd, ui, time);

    new Game().Play(initialState, ui,
        new WeatherReport(new FetchWebApiData(http)),
        new SelectAction(rnd, miniGame),
        new UpdateProgress(rnd),
        new RandomEvents(rnd),
        new EndOfWeek());
});
```

Перший тиждень симуляції:

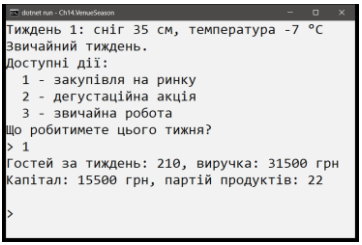


Рисунок 14.4 - Результат виконання прикладу «Сезон» (фрагмент)

Тиждень, у якому власник проводить дегустаційну акцію з міні-грою, і подія, наслідки якої нейтралізовано ремонтним комплектом:

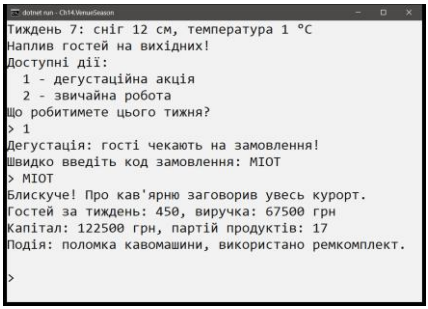


Рисунок 14.5 - Результат виконання прикладу «Сезон» (фрагмент)

Завершення сезону:

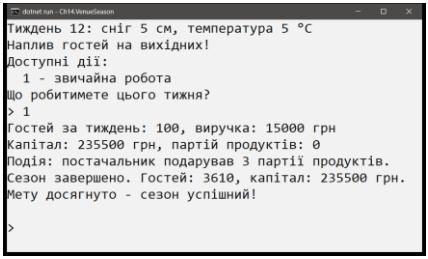


Рисунок 14.6 - Результат виконання прикладу «Сезон» (фрагмент)

14.4. Напрями розвитку симуляції

Симуляцію можна розвивати в багатьох напрямах, що дають змогу закріпити матеріал посібника:

- урахування погоди в кількості гостей (наприклад, коефіцієнт залежно від висоти снігового покриву);
- повноцінна фаза закупівлі на ринку з невизначеним циклом вибору товарів до команди «залишити ринок»;
- розширення набору випадкових подій (перевірка санітарної служби, звільнення працівника, вигідне корпоративне замовлення) з участю міні-гри;
- кешування даних погодного сервісу (мемоізація, див. розділ 11);
- збереження та відновлення стану симуляції у файлі з використанням *Maybe* для обробки помилок;
- модульні тести фаз з імітаціями залежностей.

14.5. Висновки

1. Застосунок у функціональному стилі мовою C# доцільно будувати за архітектурою «функціональне ядро, імперативна

оболонка»: недетерміновані залежності ізолюються за інтерфейсами, а логіка ядра складається з чистих функцій над незмінним станом.

2. Ігровий (або будь-який покроковий) цикл природно подається як невизначена ітерація стану, на кожному кроці якої метод *Aggregate* застосовує послідовність фаз; метод на кшталт *ContinueTurn* реалізує монадоподібне «коротке замикання» після завершення.

3. Дискриміновані об'єднання зосереджують аналіз введення користувача в одному місці, а вирази *switch* над ними замінюють розгалужені перевірки.

4. Звернення до зовнішніх сервісів доцільно загортати в монаду *Maybe*, що усуває потребу в захисному коді в логіці застосунку.

5. Ізоляція залежностей забезпечує відтворюваність виконання і тестованість: той самий код працює і в інтерактивному режимі, і з детермінованими імітаціями.

14.6. Контрольні запитання

1. Які компоненти симуляції належать до функціонального ядра, а які - до імперативної оболонки?

2. Чому генератор випадкових чисел, годинник і *HttpClient* приховано за інтерфейсами?

3. Яку роль відіграють дискриміновані об'єднання *UserInput* і *Operation*?

4. Як влаштовано повторення запиту до отримання коректного значення в модулі закупівель?

5. Чому модуль закупівель має власний стан, відмінний від спільного стану запасів?

6. Як працює метод *ContinueTurn*? З якою монадою його можна порівняти?

7. Як реалізовано ігровий цикл? Яку роль відіграють методи *Aggregate* і *IterateUntil*?

8. Як формується перелік доступних дій без інструкцій *if*?

9. Чому результат звернення до погодного сервісу загорнуто в *Maybe*? Як обробляються всі його варіанти?

10. Як забезпечено відтворюваність результатів у навчальному прикладі?

14.7. Практичні завдання

1. Реалізуйте вплив погоди на кількість гостей: за висоти снігового покриву понад 30 см кількість гостей збільшується на 20 %, менше 15 см - зменшується на 30 %. Значення погоди мають зберігатися в стані симуляції.

2. Реалізуйте фазу закупівлі на оптовому ринку з невизначеним циклом: власник обирає товари та кількість, доки не обере команду «залишити ринок»; перевірте коректність кожного введення.

3. Додайте до симуляції три нові випадкові події, одна з яких використовує міні-гру.

4. Реалізуйте мемоізацію звернень до погодного сервісу так, щоб дані завантажувалися лише один раз за сезон.

5. Реалізуйте збереження стану симуляції у файл JSON наприкінці кожного тижня та відновлення стану під час запуску; помилки файлових операцій обробляйте за допомогою *Maybe*.

6. Напишіть модульні тести для фаз «Підсумки тижня» і «Кінець тижня» з імітаціями генератора випадкових чисел і засобу взаємодії.

15. Висновки та напрями подальшого вивчення

У посібнику послідовно розглянуто шлях від функціональних засобів, доступних у стандартній мові C#, до концепцій, звичних для розробників на функціональних мовах. У першій частині показано, як LINQ, незмінні типи, кортежі, записи та зіставлення зі зразком дають змогу писати функціональний код без сторонніх бібліотек. Друга частина присвячена функціям вищого порядку, дискримінованим об'єднанням, монадам, каррінгу, невизначеним циклам і мемоізації. Третя частина розглядає практичні аспекти: продуктивність, організацію проєктів, сторонні бібліотеки та побудову завершеного застосунку.

Мова C# є і залишатиметься гібридною: повністю чисто функціональний код нею написати неможливо. Це не недолік, а властивість мови, що дає змогу поєднувати парадигми й переходити до функціонального стилю поступово, у темпі, прийнятному для команди та проєкту.

Мета розділу - узагальнити матеріал посібника, ознайомити з теоретичними основами, на які спирається подальше вивчення функціонального програмування (стратегії обчислення, персистентні структури даних, параметричність, теорія категорій, обчислювальні ефекти), та окреслити напрями подальшого вивчення.

15.1. Узагальнення матеріалу

Основні положення посібника можна звести до такого:

1) функціональне програмування - декларативна парадигма, що спирається на чисті функції, незмінні дані та композицію функцій; її математичні засади (лямбда-числення, комбінаторна логіка) передують появі обчислювальної техніки;

2) значну частину функціонального стилю в мові C# можна реалізувати стандартними засобами: LINQ, лямбда-виразами, кортежами, записами, виразами *switch*;

3) функції вищого порядку і комбінатори (*Map*, *Fork*, *Alt*, *Compose*, *Tap* тощо) усувають дублювання коду та шаблонні конструкції;

4) дискриміновані об'єднання явно описують усі можливі стани даних і результати операцій;

5) монади, зокрема *Maybe*, дають змогу будувати ланцюжки обчислень без захисного коду, а *Reader* і *State* - функціонально описувати залежності та стан;

6) каррінг, часткове застосування і мемоізація розширюють можливості повторного використання функцій та оптимізації обчислень;

7) невизначені цикли в C# потребують компромісу між функціональною чистотою та обмеженнями платформи (глибина стека, продуктивність);

8) функціональний код дещо поступається імперативному за продуктивністю обчислень у пам'яті, однак у реальних застосунках цю різницю здебільшого нівелюють операції введення-виведення;

9) практичною архітектурою функціонального застосунку на C# є «функціональне ядро, імперативна оболонка».

15.2. Теоретичні основи для подальшого вивчення

Посібник свідомо обмежувався поняттями, необхідними для практичного застосування функціонального стилю в мові C#. Подальше вивчення функціонального програмування неминуче спирається на ширшу теоретичну базу. У цьому підрозділі коротко розглянуто п'ять напрямів, з якими читач зустрінеться в спеціалізованій літературі, і показано їх зв'язок із матеріалом посібника. Приклади підрозділу містяться в проєкті *Ch15.Theory*.

15.2.1. Стратегії обчислення

Стратегія обчислення (evaluation strategy) визначає, коли обчислюються аргументи функцій і значення виразів. За **енергійного обчислення** (eager, strict evaluation) аргумент обчислюється до виклику функції, незалежно від того, чи буде він використаний. За **лінивого обчислення** (lazy evaluation, call-by-need) вираз обчислюється лише тоді, коли його значення вперше знадобилося, а результат запам'ятовується для повторних звернень.

Мова C#, як і більшість мов, використовує енергійне обчислення. Мова Haskell є лінивою за замовчуванням, що дає змогу, зокрема, оперувати нескінченними структурами даних. У мові C# ліниве обчислення доступне вибірково: через ітератори й відкладене виконання LINQ (підрозділи 2.3 і 3.2), функції-thunk (підрозділ 5.2), мемоізацію (розділ 11) і тип *Lazy<T>* (лістинг 15.1).

Лістинг 15.1 - Енергійне та ліниве обчислення

```
1. var calls = 0;
2. decimal CostOfSeasonMenu()
3. {
4.     calls++;                          // лічильник обчислень
5.     return 48_250m;                   // «дороге» обчислення
6. }
7.
8. // Енергійне обчислення: значення обчислюється одразу
9. var eager = CostOfSeasonMenu();
10.
11.  // Ліниве обчислення: лише за першої потреби, далі - з пам'яті
12.  var lazy = new Lazy<decimal>(CostOfSeasonMenu);
```

```
13.  Console.WriteLine($"Після оголошення: обчислень {calls}");
14.  Console.WriteLine($"{lazy.Value} / {lazy.Value}");
15.  Console.WriteLine($"Після двох звернень: обчислень {calls}");
```

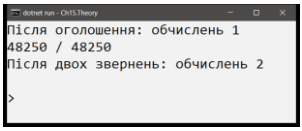


Рисунок 15.1 - Результат виконання прикладу «Енергійне і ліниве обчислення»

Перше обчислення виконано під час оголошення змінної *eager*; об'єкт *Lazy<decimal>* обчислив значення лише під час першого звернення до властивості *Value*, а друге звернення використало збережений результат. Ліниве обчислення коректне лише для чистих функцій: якщо обчислення має побічні ефекти, їх момент і кількість стають непередбачуваними. Це ще один аргумент на користь прозорості посилань (підрозділ 1.2.4).

15.2.2. Персистентні структури даних

Функціональний стиль вимагає незмінності, а наївна реалізація незмінної колекції копіювала б усі елементи під час кожної «зміни». **Персистентні структури даних** (persistent data structures) розв'язують цю проблему: нова версія структури створюється без зміни попередньої, а незмінені частини **спільно використовуються** (structural sharing) обома версіями. Класичним викладом цієї теорії є монографія К. Окасакі «Purely Functional Data Structures».

Найпростіший приклад - незмінний стек: додавання елемента створює один новий вузол, що посилається на попередню версію стека (лістинг 15.2, рис. 15.2). Колекції *ImmutableList<T>* та *ImmutableDictionary<TKey, TValue>* платформи .NET побудовано на збалансованих деревах, тож «зміна» потребує створення лише O(log n) нових вузлів.

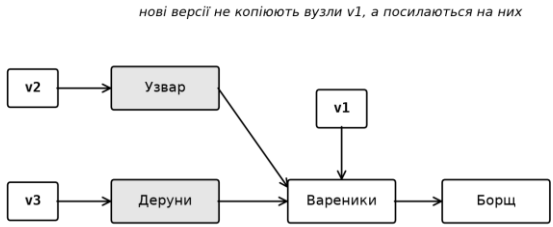


Рисунок 15.2 - Спільне використання вузлів версіями незмінного стека

Лістинг 15.2 - Версії незмінного стека зі спільними вузлами

```
1. // Історія замовлень столика як незмінний стек
```

```
2. var empty = ImmutableStack<string>.Empty;
3. var v1 = empty.Push("Борщ").Push("Вареники");
4. var v2 = v1.Push("Узвар");          // нова версія
5. var v3 = v1.Push("Деруни");         // інша гілка від v1
6.
7. Console.WriteLine(string.Join(", ", v2));
8. Console.WriteLine(string.Join(", ", v3));
9. Console.WriteLine(string.Join(", ", v1));
10.
11.  // Обидві нові версії спільно використовують вузли v1
12.  Console.WriteLine(ReferenceEquals(v2.Pop(), v1));
13.  Console.WriteLine(ReferenceEquals(v3.Pop(), v1));
```

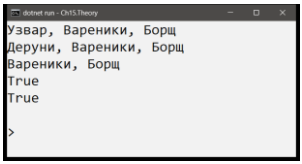


Рисунок 15.3 - Результат виконання прикладу «Персистентні структури»

Обидві нові версії, *v2* і *v3*, після вилучення верхнього елемента збігаються з *v1* за посиланням, тобто не містять копій її вузлів. Персистентність має і практичну цінність: усі попередні версії стану залишаються доступними, що спрощує реалізацію скасування дій, журналювання змін і паралельної обробки.

15.2.3. Системи типів і параметричність

Функціональні мови сімейства ML (F#, OCaml) і мова Haskell використовують систему типів **Гіндлі - Мілнера** (Hindley-Milner), що дає змогу компілятору вивести типи всіх виразів програми без жодної явної анотації. Виведення типів у мові C# значно обмеженіше: воно діє для локальних змінних (*var*), параметрів лямбда-виразів і аргументів узагальнених методів, проте не для сигнатур методів.

Важливою властивістю узагальнених функцій є **параметричність** (parametricity): функція, узагальнена за типом *T*, нічого не знає про значення цього типу і тому мусить обробляти їх однаково. Ф. Вадлер показав, що з одного лише типу такої функції випливають певні її властивості - так звані **«безкоштовні теореми»** (theorems for free). Так, повна функція типу *T -> T* може лише повернути аргумент, а функція типу *IEnumerable<T> -> IEnumerable<T>* може лише переставляти, відкидати чи повторювати елементи; тому її застосування перестановочне з будь-яким відображенням *Select* (лістинги 15.3 і 15.4).

Лістинг 15.3 - Узагальнені функції та їх властивості, що випливають із типу

```
// Тип T -> T: функція нічого не знає про T, тож для будь-якого
// T єдина повна (без винятків і побічних ефектів) реалізація -
// повернути аргумент
public static T Same<T>(T x) => x;

// Тип IEnumerable<T> -> IEnumerable<T>: функція може лише
// переставляти, відкидати чи повторювати елементи, але не
// створювати нові значення типу T
public static IEnumerable<T> EverySecond<T>(IEnumerable<T> xs) =>
    xs.Where((_, i) => i % 2 == 1);
```

Лістинг 15.4 - Перевірка «безкоштовної теореми»

```
string[] dishes = ["Борщ", "Вареники", "Узвар", "Деруни"];
Func<string, int> length = s => s.Length;

// «Безкоштовна теорема»: для будь-якої f
// EverySecond(xs).Select(f) == EverySecond(xs.Select(f))
var left = EverySecond(dishes).Select(length);
var right = EverySecond(dishes.Select(length));
Console.WriteLine(left.SequenceEqual(right));
Console.WriteLine(Same(145m));
```



Рисунок 15.4 - Результат виконання прикладу «Параметричність»

Практичний висновок полягає в тому, що узагальнені сигнатури є формою документації й гарантій: що загальнішим є тип функції, то менше можливостей для помилок він залишає.

15.2.4. Теорія категорій

Поняття функтора й монади (підрозділ 7.7) запозичено з **теорії категорій** - розділу математики, що вивчає структури через відображення між ними. **Категорія** складається з об'єктів і морфізмів (стрілок) між ними, причому морфізми можна компонувати, композиція асоціативна, а для кожного об'єкта існує тотожний морфізм. Ці аксіоми збігаються з властивостями композиції функцій (підрозділ 1.3.4): типи мови програмування разом із чистими функціями між ними утворюють категорію. Відповідність основних понять наведено в таблиці 15.1.

Таблиця 15.1 - Поняття теорії категорій і їх відповідники в мові C#

Поняття теорії категорій	**Відповідник у мові C#**	**Підрозділ посібника**
Об'єкт	Тип (*int*, *string*, *Dish*)	1.3.1

Поняття теорії категорій	**Відповідник у мові C#**	**Підрозділ посібника**
Морфізм	Чиста функція *Func<A, B>*	1.3.1
Композиція морфізмів	Композиція функцій *Compose*	1.3.4, 5.6
Тотожний морфізм	Тотожна функція *x => x*	1.3.4
Функтор	Узагальнений тип з *Map* (*Select*)	7.7.2
Моноїд	Тип з асоціативною операцією і нейтральним елементом	5.12.2
Монада	Узагальнений тип з *Bind* (*SelectMany*) і загортанням	7.7.1

Відоме визначення «монада - це моноїд у категорії ендофункторів» (С. Маклейн) стає зрозумілим саме в цьому контексті: зв'язування монад асоціативне, а операція загортання відіграє роль нейтрального елемента, тобто закони монад (підрозділ 7.7.1) є законами моноїду. Для систематичного вивчення доцільно звернутися до книги Б. Мілевського «Category Theory for Programmers».

15.2.5. Обчислювальні ефекти

Будь-яка корисна програма взаємодіє із зовнішнім світом, тобто має **ефекти** (effects): введення-виведення, зміну стану, генерування винятків, недетермінованість. Теорія функціонального програмування розглядає ефекти не як порушення чистоти, а як явно описані обчислення. У мові Haskell ефекти введення-виведення відображено в типі *IO<T>*: функція, що повертає *IO<T>*, не виконує дію, а описує її, і лише середовище виконання запускає опис. Сучасні дослідження розвивають цей підхід у вигляді **алгебраїчних ефектів** (algebraic effects) і систем ефектів, у яких тип функції перелічує ефекти, які вона може спричинити.

У мові C# ефекти в типах не відстежуються, проте розглянуті в посібнику прийоми реалізують ту саму ідею іншими засобами. *Maybe* і *Validation* роблять явним ефект невдачі, *State* - ефект зміни стану, *Reader* - залежність від середовища, *Task<T>* - асинхронність, а архітектура «функціональне ядро, імперативна оболонка» (підрозділ 1.2.4 і розділ 14) відокремлює опис рішень від їх виконання. Отже, вивчення теорії ефектів

дає змогу краще зрозуміти, чому розглянуті прийоми працюють і як їх поєднувати.

15.3. Напрями подальшого вивчення

15.3.1. Поглиблене вивчення функціонального C#

Найдоцільнішим наступним кроком є поглиблене вивчення функціонального програмування мовою C# за спеціалізованою літературою, зокрема монографією Е. Буонанно «Functional Programming in C#», що детальніше розглядає теоретичні основи та містить додатковий матеріал.

Проте головним засобом опанування функціонального стилю є практика: регулярне розв'язування задач із дотриманням обмежень, наведених у підрозділі 2.7, і поступове застосування функціональних прийомів у навчальних та промислових проєктах. Упроваджувати функціональний стиль в командних проєктах доцільно після обговорення з командою: прийоми розділів 2-4 зазвичай не викликають заперечень, тоді як несподіване використання монад може ускладнити супровід коду для колег.

15.3.2. Вивчення мови F#

Мова F# є природним наступним кроком для розробника, що опанував функціональний стиль у C#. Вона належить до платформи .NET і взаємодіє з C#, тож обидві мови можна поєднувати в межах одного рішення. Для вивчення F# доцільно звернутися до ресурсу С. Влащина «F# for Fun and Profit», книги І. Абрахама «Get Programming with F#» та інших видань.

15.3.3. Чисто функціональні мови

Наступним етапом може бути вивчення чисто функціональних мов, зокрема Haskell або Elm. Ці мови не взаємодіють безпосередньо з платформою .NET, тому їх вивчення має інші цілі:

- опанування функціональної парадигми в «чистому» вигляді з подальшим перенесенням ідей у практику розробки на .NET;
- зміна професійної спеціалізації;
- задоволення пізнавального інтересу та розширення професійного світогляду.

Для початку вивчення мови Haskell часто рекомендують книгу М. Липовачі «Learn You a Haskell for Great Good!». Для поглибленого вивчення теоретичних основ функціонального програмування доцільно звернутися до теорії категорій, типізованого лямбда-числення та теорії систем типів (див. підрозділ 15.2).

15.4. Контрольні запитання

1. Які основні групи функціональних засобів мови C# розглянуто в посібнику?

2. Чому мову C# називають гібридною і які наслідки це має для функціонального програмування?

3. Які прийоми функціонального стилю доцільно впроваджувати в команді насамперед, а які - після попереднього обговорення?

4. Які переваги дає вивчення мови F# розробникові на C#?

5. З якою метою доцільно вивчати чисто функціональні мови?

6. Чим енергійне обчислення відрізняється від лінивого? Які засоби лінивого обчислення має мова C#? Чому лініве обчислення коректне лише для чистих функцій?

7. Що таке персистентна структура даних? Як спільне використання вузлів зменшує витрати на «зміну» незмінної колекції?

8. Що таке параметричність? Що можна стверджувати про повну функцію типу *T -> T*, не знаючи її коду?

9. З яких елементів складається категорія? Наведіть відповідники понять «об'єкт», «морфізм», «функтор» у мові C#.

10. Що таке обчислювальні ефекти? Які прийоми посібника роблять ефекти явними в мові C#?

15.5. Практичні завдання

1. Оберіть невеликий фрагмент власного навчального або промислового проєкту і перепишіть його у функціональному стилі, використовуючи щонайменше три прийоми з різних розділів посібника. Порівняйте вихідний і перероблений код за обсягом, читабельністю та тестованістю.

2. Реалізуйте одну з задач розділу 14 мовою F# і підключіть отриманий проєкт до рішення на C#.

3. Підготуйте для команди розробників коротку доповідь (10-15 хвилин) про функціональне програмування в C# з прикладами з предметної області обраного проєкту.

4. Реалізуйте незмінну чергу замовлень кухні на основі двох незмінних стеків (подання Окасакі) з операціями додавання й вилучення. Переконайтеся, що попередні версії черги залишаються незмінними, і оцініть амортизовану складність операцій.

5. Запишіть сигнатури трьох узагальнених функцій обробки меню (наприклад, *IEnumerable<T> -> T?*, *(T -> bool) -> IEnumerable<T> -> IEnumerable<T>*) і сформулюйте для кожної властивості, що випливають із її типу. Перевірте їх тестами.

Глосарій

Глосарій містить основні терміни посібника в алфавітному порядку. Для кожного терміна наведено англійський відповідник і номер підрозділу, у якому його розглянуто докладно.

Алгебраїчний тип даних (algebraic data type, ADT) - складений тип, побудований з простіших типів операціями добутку (тип-добуток) і суми (тип-сума) (підрозділ 6.2.1).

Аплікативний функтор (applicative functor) - функтор з додатковими операціями загортання значення та застосування *Apply*, що поєднує кілька незалежних значень у контексті функцією кількох аргументів (підрозділ 7.7.3).

Бета-редукція (β-reduction) - правило обчислення лямбда-числення: застосування абстракції до аргументу замінюється її тілом, у якому параметр замінено аргументом (підрозділ 1.3.2).

Вираз (expression) - конструкція мови, що обчислюється до значення; на відміну від інструкції, може бути частиною іншого виразу (підрозділ 1.2.3).

Відкладене виконання (deferred execution) - обчислення елементів послідовності лише в момент їх перебору, а не в момент побудови запиту (підрозділ 2.3).

Гомоїконічність (homoiconicity) - подання програмного коду у вигляді даних, які інша програма може аналізувати й перетворювати; у мові C# реалізується деревами виразів (підрозділ 5.13.1).

Дерево виразів (expression tree) - незмінна структура даних, що описує лямбда-вираз як дерево вузлів-операцій; основа *IQueryable<T>* та Entity Framework Core (підрозділ 5.13).

Дискримінований об'єднання (discriminated union), тип-сума - тип, значення якого належить до одного з фіксованого набору варіантів, кожен з яких може мати власний склад даних (розділ 6).

Замикання (closure) - функція разом зі змінними зовнішньої області видимості, які вона захоплює (підрозділ 1.3.3).

Закрита ієрархія (closed hierarchy) - ієрархія з базовим типом, позначеним модифікатором *closed* (C# 15), нащадків якого можна оголошувати лише в тій самій збірці; компілятор перевіряє повноту виразів *switch* над нею (підрозділ 6.2.3).

Залізничне програмування (railway oriented programming) - стиль побудови ланцюжків обчислень, у якому кожен крок виконується лише на «успішній» колії, а невдача без виконання подальших кроків передається до кінця ланцюжка (підрозділ 7.1.2).

Зв'язування (bind, також *flatMap*, *SelectMany*) - операція монади, що застосовує до загорнутого значення функцію, яка сама повертає

монадичне значення, без додаткового загортання результату (підрозділ 7.2).

Згортка (fold, *Aggregate*) - функція вищого порядку, що зводить послідовність до одного значення, послідовно поєднуючи акумулятор з кожним елементом (підрозділ 5.12.1).

Зіставлення зі зразком (pattern matching) - перевірка значення на відповідність шаблону (типу, форми, властивостей) з одночасним вилученням його складових (підрозділи 1.2.6, 3.5.3).

Каррінг (currying) - перетворення функції кількох аргументів на ланцюжок функцій одного аргументу (підрозділ 9.1).

Композиція функцій (function composition) - побудова функції g ∘ f, що застосовує f, а потім g до її результату; операція асоціативна, а її нейтральним елементом є тотожна функція (підрозділ 1.3.4).

Коротке замикання (short-circuiting) - поведінка ланцюжка монадичних операцій, за якої перша невдача зупиняє виконання подальших кроків і передається до кінця ланцюжка (підрозділ 8.2.2).

Лямбда-числення (λ-calculus) - формальна система А. Черча, що описує обчислення за допомогою змінних, абстракцій і застосувань; теоретична основа функціонального програмування (підрозділ 1.3.2).

Мемоізація (memoization) - збереження результатів чистої функції для повторного використання за однакових значень аргументів (розділ 11).

Монада (monad) - узагальнений тип з операціями загортання значення та зв'язування, що задовольняють закони лівої та правої тотожності й асоціативності; описує послідовність залежних обчислень у контексті (підрозділи 7.1.2, 7.7.1).

Моноїд (monoid) - множина значень з асоціативною бінарною операцією та нейтральним елементом (підрозділ 5.12.2).

Незмінність (immutability) - властивість об'єкта, стан якого після створення не змінюється (підрозділ 1.2.1).

Оптимізація хвостових викликів (tail call optimization) - перетворення хвостового рекурсивного виклику компілятором на перехід без створення нового кадру стека; у мові C# не гарантується (підрозділ 10.2).

Перетворювач монад (monad transformer) - конструкція, що поєднує два монадичні контексти (наприклад, асинхронність і помилку, *Task<Either<TL, TR>>*) і надає для поєднаного типу власні операції зв'язування (підрозділ 8.6).

Побічний ефект (side effect) - будь-яка дія функції, крім обчислення результату: зміна зовнішнього стану, введення-виведення, генерування винятку (підрозділ 1.2.4).

Потужність типу (cardinality) - кількість можливих значень типу; для типу-добутку дорівнює добутку, для типу-суми - сумі потужностей складових (підрозділ 6.2.1).

Прозорість посилань (referential transparency) - властивість виразу, за якої його можна замінити обчисленим значенням без зміни поведінки програми (підрозділ 1.2.4).

Рівнянне міркування (equational reasoning) - аналіз програми як системи рівностей шляхом підстановки виразів, можливий завдяки прозорості посилань (підрозділ 1.3.2).

Синтаксис запитів LINQ (query syntax) - форма запису *from ... in ... select*, яку компілятор перетворює на виклики *Select* і *SelectMany*; у мові C# відіграє роль монадичної do-нотації (підрозділ 7.7.5).

Тип-добуток (product type) - тип, значення якого одночасно містить значення всіх складових (кортеж, запис, структура) (підрозділ 6.2.1).

Тип-об'єднання (union type) - тип, оголошений ключовим словом *union* (C# 15), значення якого є значенням рівно одного з перелічених типів-варіантів (підрозділ 6.2.4).

Тип-запис (record) - тип із рівністю за значенням, згенерованими членами та підтримкою виразу *with*, призначений для подання незмінних даних (підрозділ 3.7.3).

Тотальна функція (total function) - функція, визначена для всіх значень області визначення; на відміну від неї, **часткова функція** (partial function) визначена лише для частини аргументів (підрозділ 1.3.1).

Трамплін (trampoline) - техніка виконання рекурсивних алгоритмів без зростання стека: функція повертає опис наступного кроку, а цикл виконує кроки по черзі (підрозділ 10.3.2).

Трансдуктор (transducer) - функція, що поєднує фільтрацію, перетворення та агрегування колекції в одну операцію (підрозділ 5.7).

Функтор (functor) - узагальнений тип з операцією *Map*, що задовольняє закони тотожності й композиції (підрозділ 7.7.2).

Функції першого класу (first-class functions) - властивість мови, за якої функції є повноправними значеннями: їх можна зберігати, передавати, повертати й створювати під час виконання (підрозділ 1.3.3).

Функціональне ядро, імперативна оболонка (functional core, imperative shell) - архітектурний підхід, за якого бізнес-логіка реалізується чистими функціями, а взаємодія із зовнішнім середовищем зосереджена в тонкому зовнішньому шарі (підрозділ 1.2.4).

Функція вищого порядку (higher-order function) - функція, що приймає інші функції як параметри або повертає функцію як результат (підрозділи 1.2.2, 5).

Хвостова рекурсія (tail recursion) - рекурсія, у якій рекурсивний виклик є останньою дією функції (підрозділ 10.2).

Часткове застосування (partial application) - фіксація частини аргументів функції з отриманням нової функції від решти аргументів (підрозділ 9.5).

Чиста функція (pure function) - функція, результат якої залежить лише від аргументів і яка не має побічних ефектів (підрозділ 1.2.4).

Член-розширення (extension member) - метод, властивість чи оператор, оголошені для наявного типу в блоці *extension* (C# 14) без зміни коду цього типу (підрозділ 3.12.1).

Thunk - функція без параметрів, що інкапсулює відкладене обчислення й виконується лише на вимогу (підрозділ 5.2).

Список використаних джерел

1. Painter S. J. Functional Programming with C#: Create More Supportable, Robust, and Testable Code. Sebastopol : O'Reilly Media, 2023. 400 p.
2. Bueno C. C# Functional: Monads from Zero to Hero. 2024. 85 p.
3. Buonanno E. Functional Programming in C#: How to Write Better C# Code. 2nd ed. Shelter Island : Manning, 2022. 448 p.
4. Wlaschin S. Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#. Raleigh : Pragmatic Bookshelf, 2018. 312 p.
5. Abraham I. Get Programming with F#: A Guide for .NET Developers. Shelter Island : Manning, 2018. 592 p.
6. Albahari J. C# 12 in a Nutshell: The Definitive Reference. Sebastopol : O'Reilly Media, 2023. 1084 p.
7. Seemann M. Code That Fits in Your Head: Heuristics for Software Engineering. Boston : Addison-Wesley, 2021. 384 p.
8. Lipovača M. Learn You a Haskell for Great Good! A Beginner's Guide. San Francisco : No Starch Press, 2011. 400 p.
9. Church A. The Calculi of Lambda-Conversion. Princeton : Princeton University Press, 1941. 77 p.
10. Curry H. B., Feys R. Combinatory Logic. Vol. 1. Amsterdam : North-Holland, 1958. 417 p.
11. Schönfinkel M. Über die Bausteine der mathematischen Logik. Mathematische Annalen. 1924. Vol. 92. P. 305-316.
12. McCarthy J. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. Communications of the ACM. 1960. Vol. 3, No. 4. P. 184-195.
13. Hoare C. A. R. Null References: The Billion Dollar Mistake : доповідь на конференції QCon London. 2009.
14. Wlaschin S. Railway Oriented Programming. F# for Fun and Profit. URL: https://fsharpforfunandprofit.com/rop/ (дата звернення: 13.11.2026).
15. Bernhardt G. Functional Core, Imperative Shell : відеоматеріал. Destroy All Software, 2012. URL: https://www.destroyallsoftware.com/screencasts/catalog/functional-core-imperative-shell (дата звернення: 13.11.2026).
16. C# documentation. Microsoft Learn. URL: https://learn.microsoft.com/dotnet/csharp/ (дата звернення: 13.11.2026).
17. Language Integrated Query (LINQ). Microsoft Learn. URL: https://learn.microsoft.com/dotnet/csharp/linq/ (дата звернення: 13.11.2026).

18. F# documentation. Microsoft Learn. URL: https://learn.microsoft.com/dotnet/fsharp/ (дата звернення: 13.11.2026).
19. BenchmarkDotNet : документація. URL: https://benchmarkdotnet.org (дата звернення: 13.11.2026).
20. OneOf : репозиторій бібліотеки. URL: https://github.com/mcintyre321/OneOf (дата звернення: 13.11.2026).
21. LanguageExt : репозиторій бібліотеки. URL: https://github.com/louthy/language-ext (дата звернення: 13.11.2026).
22. Functional.Maybe : репозиторій бібліотеки. URL: https://github.com/AndreyTsvetkov/Functional.Maybe (дата звернення: 13.11.2026).
23. CSharpFunctionalExtensions : репозиторій бібліотеки. URL: https://github.com/vkhorikov/CSharpFunctionalExtensions (дата звернення: 13.11.2026).
24. Okasaki C. Purely Functional Data Structures. Cambridge : Cambridge University Press, 1998. 220 p.
25. Wadler P. Theorems for Free! Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA'89). New York : ACM, 1989. P. 347-359.
26. Milewski B. Category Theory for Programmers. Blurb, 2019. 498 p.
27. Mac Lane S. Categories for the Working Mathematician. 2nd ed. New York : Springer, 1998. 314 p.
28. Pierce B. C. Types and Programming Languages. Cambridge, MA : MIT Press, 2002. 623 p.
29. What's new in C# 15. Microsoft Learn. URL: https://learn.microsoft.com/dotnet/csharp/whats-new/csharp-15 (дата звернення: 13.11.2026).

Томка Юрій Ярославович

ФУНКЦІОНАЛЬНЕ 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