Коноваленко І.В., Марущак П.О.

Платформа .NET та мова програмування C# 8.0

Тернопіль

2020

Коноваленко І.В., Марущак П.О.

Платформа .NET та мова програмування C# 8.0

Навчальний посібник

Тернопіль
2020

УДК 681.3
К64

Автори:
І.В. Коноваленко, канд. техн. наук, доцент,
П.О. Марущак, докт. техн. наук, професор.

Рекомендовано до друку Вченою радою
Тернопільського національного технічного університету імені Івана Пулюя.
Протокол № 10 від 20 жовтня 2020 року.

Рецензенти:
Б.П. Русин, докт. техн. наук., професор,
В.А. Кирилович, докт. техн. наук, професор.

К64 Коноваленко І.В. Платформа .NET та мова програмування C# 8.0: навчальний посібник / Коноваленко І.В., Марущак П.О. – Тернопіль: ФОП Паляниця В. А., 2020 – 320 с.

ISBN 978-617-7875-08-5

У навчальному посібнику розглянуто принципи об'єктно-орієнтованого програмування мовою C# у середовищі Microsoft Visual Studio. Описано внутрішній устрій і функціонування загального виконуючого середовища та бібліотеку базових класів платформ Microsoft .NET Framework та .NET Core.

Окремо розглянуто головні принципи розробки застосунків з графічним інтерфейсом користувача за допомогою технологій Windows Forms та Windows Presentation Foundation.

Матеріал супроводжується значною кількістю прикладів.

Навчальний посібник розроблено у відповідності до навчальної програми курсу "Об'єктно-орієнтоване програмування" і рекомендовано для студентів відповідних спеціальностей.

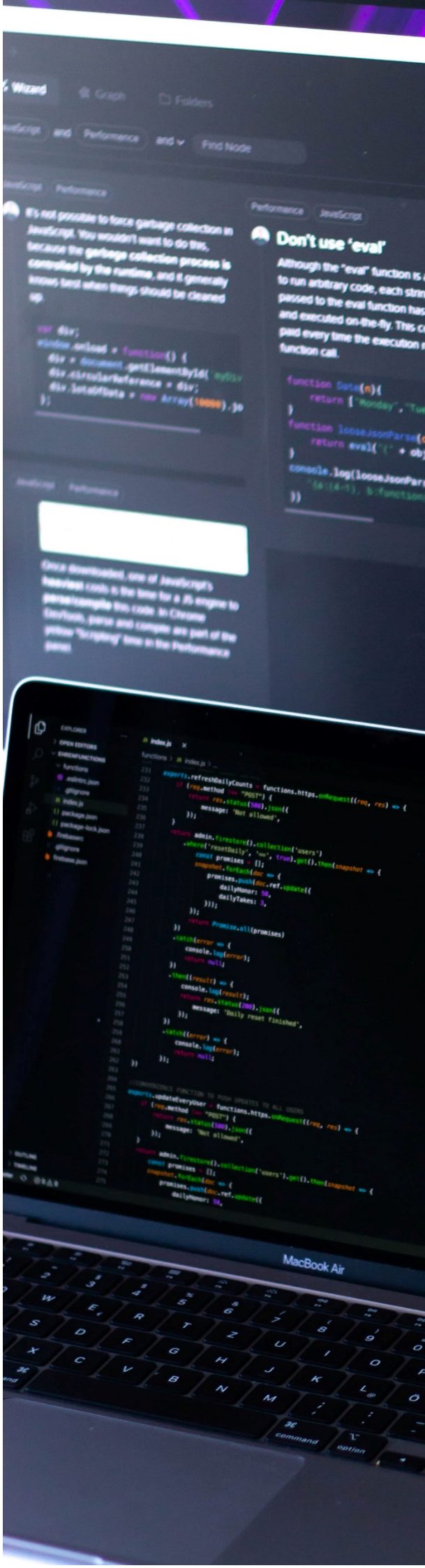
УДК 681.3

ISBN 978-617-7875-08-5

Скорочення

ПЗ	Програмне забезпечення – сукупність програм і програмних документів, необхідних для експлуатації цих програм.
BCL	(Base Class Library) – бібліотека базових класів платформи .NET Framework. Програми для .NET можуть використовувати класи BCL.
CIL	(Common Intermediate Language) – загальна проміжна мова, розроблена корпорацією Microsoft для платформи .NET Framework.
CLI	(Common Language Infrastructure) – відкрита специфікація, розроблена Microsoft і стандартизована ANSI та ECMA, яка описує виконавчий код і середовище виконання для множини високорівневих апаратно незалежних мов програмування, що можуть використовуватися на різних платформах без модифікації програмного коду.
CLR	(Common Language Runtime) – середовище виконання .NET Framework. Компонент Microsoft .NET Framework, віртуальна машина, на якій виконуються всі мови платформи .NET Framework.
MFC	(Microsoft Foundation Classes) – бібліотека, яка дає можливість розробляти віконні застосунки з графічним інтерфейсом для Microsoft Windows мовою C++ з використанням широкого набору класів. Велика частина MFC є відносно тонкою об'єктно-орієнтованою обгорткою над Windows API.
MSDN	(Microsoft Developer Network) – підрозділ компанії Microsoft, який відповідає за співпрацю з мережею зацікавлених розробників програмного та апаратного забезпечення.
Windows API	Інтерфейс прикладного програмування Windows (Application Programming Interface) – сукупність програм, протоколів та інструментів для розробки та виконання прикладних програм.
WF	(Windows Forms) – бібліотека класів, яка є одним з компонентів Microsoft .NET Framework і надає платформу для розробки віконних застосунків з графічним інтерфейсом. У .NET Framework заміщує попередню MFC.
WPF	(Windows Presentation Foundation) – графічна (презентаційна) підсистема в складі .NET Framework, яка використовується для створення сучасного графічного інтерфейсу застосунків та в повній мірі використовує можливості графічної підсистеми комп’ютера.

Вступ. Об'єктно-орієнтоване програмування



Об'єктно-орієнтоване програмування (ООП) – це технологія програмування, яка розглядає програму як множину об'єктів, що взаємодіють між собою. Програмування із загального погляду полягає у моделюванні засобами мови програмування певних процесів реального світу. Головна перевага об'єктно-орієнтованого підходу в тому, що він дозволяє наблизити структуру програми до тієї частини реальності, яка моделюється програмою.

Зручність об'єктно-орієнтованого підходу обумовлена межею можливостей технології процедурного програмування, яка використовувалась до ООП. При використанні процедурного підходу ускладнюється структура програми та утруднюється можливість її модифікації у зв'язку із зростанням кількості змінних, структур даних, функцій та зв'язків між ними. Процедурний підхід розділяє дані та код для їх опрацювання (функції). Це не відповідає картині реального світу, який складається з об'єктів, що водночас мають певні властивості та поведінку.

Обмеженість процедурного програмування особливо яскраво проявила себе при розробці великих інформаційних систем. Постійне зростання розміру і складності програм та труднощі їх реалізації привели до пошуків нових принципів програмування, які було втілено в ООП. Головною ідеєю об'єктно-орієнтованого підходу, яка забезпечила подолання згаданих труднощів, є логічне об'єднання даних та операцій над ними в одному типі даних.

У парадигмі процедурного програмування програму розглядали як перелік інструкцій для комп'ютера, які для зручності структурували у вигляді функцій та модулів. При розробці таких програм зосереджувались перш за все на оптимізації інструкцій. В об'єктно-орієнтованій парадигмі програму розглядають як сукупність об'єктів, що можуть взаємодіяти між собою за допомогою повідомлень. При цьому увага розробника змістилась з оптимізації інструкцій до організації логічно пов'язаних між собою даних та функцій в окремі захищені елементи – об'єкти.

Головні переваги об'єктно-орієнтованого програмування такі:

- Програма простіша в розумінні, оскільки краще відповідає уявленню про реальний світ.
- Краща структурованість програми.
- Більша ступінь повторного використання коду.
- Зручніша підтримка програми (внесення змін у програму в процесі її експлуатації для пристосування до нових умов).
- Менші ризики при проектуванні складних систем.

Об'єктно-орієнтоване програмування ґрунтується на понятті класів, об'єктів та успадкуванні елементів базових класів похідними класами. *Клас*

визначає абстрактний тип даних, структура якого обумовлена проблематикою розв'язуваної задачі, а *об'єкт* є екземпляром класу.

Методологія ООП складається з об'єктно-орієнтованого аналізу, проектування та програмування [1]. В процесі об'єктно-орієнтованого аналізу предметної області виявляють концептуальні сутності (об'єкти), їх характеристики та принципи взаємодії між собою.

Об'єктно-орієнтоване проектування полягає у розробці сукупності класів, які описують об'єкти та реалізують їх поведінку на основі атрибутів (характеристик) та операцій (дій) виділених концептуальних сутностей.

Об'єктно-орієнтоване програмування основане на трьох головних принципах:

- Інкапсуляція – полягає у поєднанні даних та функцій для їх обробки в єдиній захищеній програмній структурі, яку називають класом.
- Спадкування класів – перенесення структури та функціональності базового класу в похідний клас. Множина класів, пов'язаних спадкуванням, формує *ієрархію спадкування*.
- Поліморфізм – полягає у використанні однакових підходів для роботи з різними за функціональністю об'єктами.

З врахуванням сказаного можна дати таке уточнене визначення для ООП. *Об'єктно-орієнтоване програмування* – це метод програмування, оснований на поданні програми у вигляді сукупності взаємодіючих об'єктів, кожен з яких є екземпляром певного класу, а класи формують певну ієрархію спадкування [2].

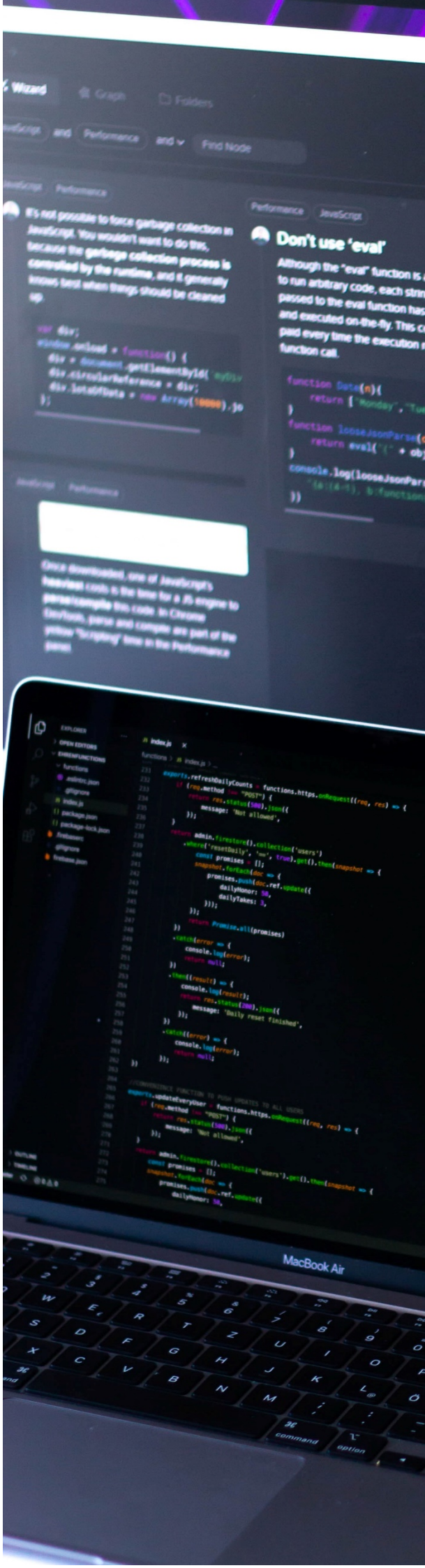
Об'єктно-орієнтовані принципи на практиці реалізують за допомогою об'єктно-орієнтованих мов програмування. Така мова а) підтримує об’єкти як абстракції даних з інтерфейсом у виді іменованих операцій і прихованим станом; б) описує об’єкти за допомогою спеціального типу – класу; в) забезпечує можливість спадкування об'єктом атрибутів та функціональності іншого об’єкта.

До об'єктно-орієнтованих мов належать Java, C++, C#, Python, Object Pascal, Delphi, Visual Basic .NET, PHP, Perl, Ruby, Scala (Scala є функціональною мовою програмування, але підтримує всі принципи ООП) та інші.

У даному навчальному посібнику розглянуто головні принципи програмування за допомогою мови C# у середовищі Microsoft Visual Studio.

Розділ 1. Microsoft .NET Framework, Microsoft .NET Core, мова C# та середовище Microsoft Visual Studio

- Microsoft .NET Framework
- Microsoft .NET Core
- Мова програмування C#
- Microsoft Visual Studio



Microsoft .NET Framework

Офіційно про розробку технології .NET Framework було оголошено 13 січня 2000 року. Microsoft було озвучено нову стратегію, яка попередньо отримала назву Next Generation Windows Services (нове покоління служб Windows). У 2002 році вийшла перша версія .NET Framework, яка мала розв'язати найпомітніші проблеми попередніх середовищ. На середину 2020 року останньою версією є .NET Framework 4.8.

Інфраструктура .NET Framework є значно цілісніше та реалізує принципи об'єктно-орієнтованого програмування значно повніше, ніж попередні технології програмування на основі бібліотеки MFC чи моделі COM. Головні відмінності такі [3]:

- Багатоплатформенність. Система працює на широкому діапазоні пристроїв, від ПК до планшетів та телефонів під керуванням ОС Windows.
- Промислові стандарти. Система використовує промислові стандарти протоколів комунікації: XML, HTTP, SOAP та інші.
- Безпека. Система створює набагато безпечніше середовище виконання.

Компоненти .NET Framework

.NET Framework містить три базові компоненти, які визначають його структуру: виконавче середовище, інструменти програмування та бібліотеку базових класів (рис. 1.1).

Середовище виконання називають Common Language Runtime (CLR). Воно керує виконанням програм, зокрема: виконує розподіл пам'яті та прибирає "сміття" (дані у пам'яті, які вже не потрібні програмі), перевіряє безпечність коду, виконує код, керує потоками виконання та обробкою помилок тощо.

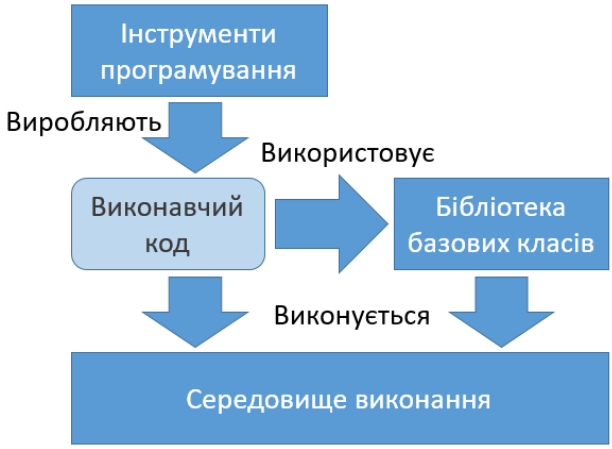


Рис. 1.1. Компоненти .NET Framework та їх взаємодія

Інструменти програмування є засобами розробки, які поєднують все необхідне для створення та налагодження програм. Вони містять:

- Інтегроване середовище розробки Microsoft Visual Studio.
- Сумісні з .NET компілятори (C#, Visual Basic .NET, F#, managed C).
- Налагоджувачі (засоби для пошуку і виправлення помилок у програмах).
- Серверні технології для WEB-розробки, такі як ASP.NET.

Бібліотека базових класів (Base Class Library, BCL) містить великий набір класів, який використовується системою .NET Framework і доступний при розробці програмного забезпечення. Сюди входять класи для роботи з різними типами даних, файлами, шифрування; класи колекцій; класи для роботи з потоками тощо.

Компіляція коду в .NET Framework

Компілятор кожної з мов .NET на основі вихідного коду програми формує вихідний файл, який називають збіркою (assembly). Цей процес ілюструє рис. 1.2. Збірка містить не безпосередній машинний код, а проміжний код мовою Common Intermediate Language (CIL, "загальна проміжна мова")[1]. Код збірки містить, крім програмного коду мовою CIL, метадані з описом типів, посилань на інші збірки, інформацію щодо безпеки виконання тощо.

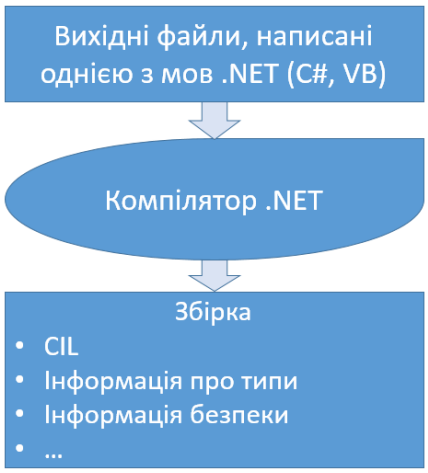


Рис. 1.2. Компіляція вихідного коду у збірку

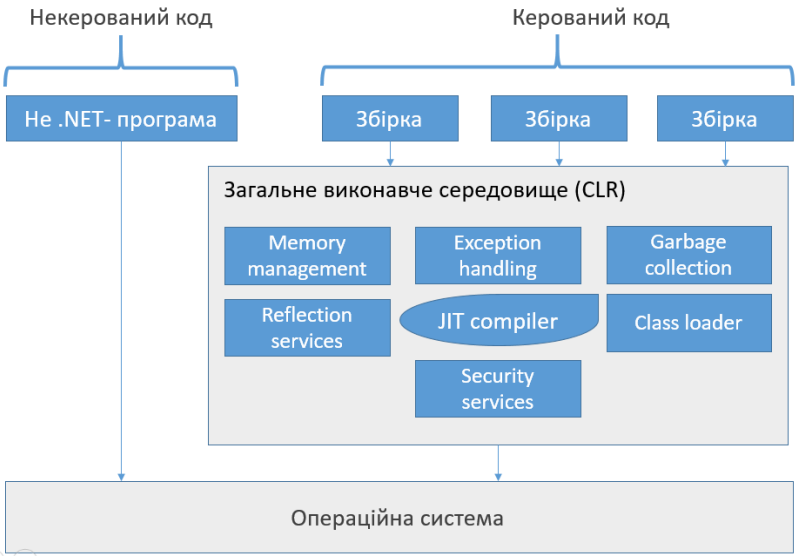


Рис. 1.3. Виконання керованого та некерованого коду

[1] Інші назви цієї проміжної мови, які вийшли з уживання – Intermediate Language (IL) та Microsoft Intermediate Language (MSIL).

Процесор комп'ютера може виконувати виключно власний машинний код. Відповідно, CIL-код збірки для виконання має бути перетворений у процесорний код. Це здійснюється тільки при виконанні збірки. Виконавчий код збірки перетворюється компілятором часу виконання (Just-in-Time compiler), який є складовою частиною виконавчого середовища CLR. Потім цей код кешується, для випадку, якщо він ще раз буде потрібний для виконання.

Так як Windows підтримує виконання застосунків, розроблених не для .NET Framework, то з точки зору системи .NET весь код ділять на дві частини:

- Керований код (managed code). Це код, написаний для .NET Framework. Він потребує середовище CLR для виконання.
- Некерований код (unmanaged code). Це код, який не виконується під контролем CLR, наприклад, динамічні бібліотеки, розроблені на C та C++.

Різницю виконання керованого та некерованого коду демонструє схема на рис. 1.3.

Описаний процес компіляції та виконання ілюструє схема на рис. 1.4 (зверніть увагу, що схема, яка ілюструє процес компіляції, рис. 1.2, є складовою частиною узагальненої схеми на рис 1.4). Програмний код, написаний однією з мов .NET Framework (наприклад, C# чи Visual Basic .NET), компілюється у проміжний код CIL. При необхідності його виконання, задіюється загальне виконавче середовище CLR, і за допомогою компілятора часу виконання формується процесорний код, який передається для виконання внутрішньому механізмові операційної системи (ОС) Windows.

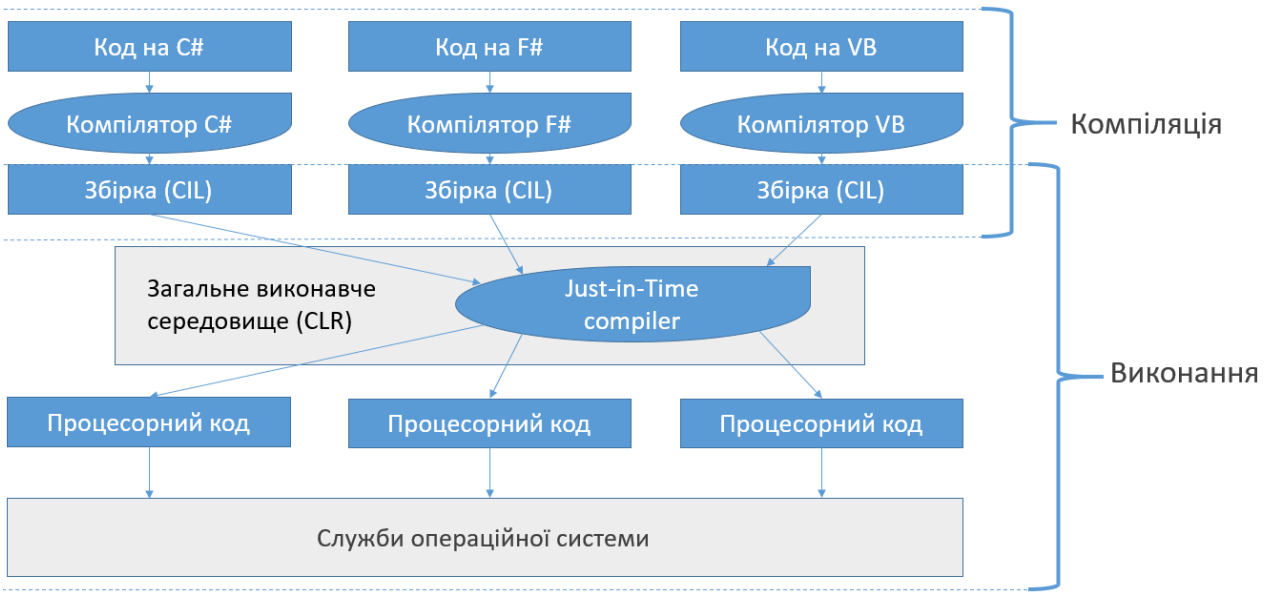


Рис. 1.4. Схема процесу компіляції коду для .NET і його виконання

Мови програмування .NET Framework

Microsoft розробила відкриту специфікацію Common Language Infrastructure (CLI)[2], яка описує особливості кодування та виконання, що дозволяють високорівневим мовам програмування використовуватися на різних платформах без внесення змін. .NET Framework (і деякі інші системи) реалізує специфікацію CLI.

Перелік мов програмування, які відповідають специфікації CLI, доволі широкий. Більшість з них компілюються безпосередньо у CIL-код загальної проміжної мови, який виконується середовищем виконання .NET Framework. Найвживанішими для програмування під .NET Framework є мови C#, Managed C++ та Visual Basic .NET.

Microsoft .NET Core

Швидкозмінні умови ринку інформаційних технологій призвели до появи ще однієї платформи, яка схожа за принципами функціонування на .NET Framework, але є окремою платформою програмування – .NET Core. .NET Framework перш за все призначений для розробки програмного забезпечення, яке працює на основі ОС Windows. Але зростання сегменту ринку пристроїв на основі операційних систем Linux та MacOS спонукало Microsoft до розробки кросплатформенного середовища з відкритим кодом, яке б підтримувало виконання застосунків на різних ОС. Таким середовищем і є .NET Core.

.NET Core дозволяє розробляти застосунки, які працюватимуть як у Windows, так і (з невеликими модифікаціями) у Linux чи MacOS. Зараз .NET Core вже підтримує ARM-процесори та може працювати навіть у системах на основі простих одноплатних комп'ютерів Raspberry Pi. Відповідно до озвучених планів Microsoft, у майбутньому платформи .NET Framework та .NET Core об'єднають у єдину платформу .NET, яка матиме переваги обох.

.NET Core поширюється на основі пакетів NuGet. Пакет є бібліотекою з кодом, яка забезпечує певний логічно завершений функціонал. Пакети розміщують у центральному репозитарії NuGet Gallery, звідки за необхідності їх завантажують.

На відміну від цього, .NET Framework, який містить вже більше 20000 класів, можна встановити тільки в повному обсязі (як на клієнтську машину, так і на комп'ютер розробника), навіть коли застосунок використовує дуже обмежений функціонал. Якщо вибрати тільки потрібні пакети, загальний обсяг

[2] Специфікація CLI стандартизована міжнародними організаціями ISO та ECMA.

виконавчого образу для застосунку .NET Core буде значно меншим, ніж такого ж за функціоналом застосунку .NET Framework.

Ще однією особливістю .NET Framework є те, що для виконання прикладної програми на клієнтській машині потрібно встановити саме ту версію, яка була використана розробником. У ряді випадків це може спричиняти складності, пов'язані з правами доступу, політикою безпеки компанії чи іншими факторами.

На відміну від цього, .NET Core можна поширювати разом з застосунком, що в принципі не викличе конфлікту версій. Якщо ж .NET Core потрібної версії вже встановлено на комп'ютері, при інсталяції застосунку можна не встановлювати додаткових системних пакетів, а використати наявні.

Більше того, кожен застосунок може мати власну копію бібліотек .NET Core. Тому в одній системі можуть виконуватися різні застосунки на основі різних версій .NET Core.

Слід зазначити, що .NET Core є проектом з відкритим кодом. Це дає програмістам більшу гнучкість у розробці та потенційно робить безпечнішими як платформу так і застосунки під час їх виконання. Вихідний код .NET Core розташовано у репозитарії GitHub (за адресою https://github.com/dotnet).

Застосунки під керуванням .NET Core можуть показати кращу продуктивність. Обидві платформи (.NET Framework і .NET Core) використовують компілятор часу виконання для перетворення CIL-коду в машинний код під час роботи. Загалом це забезпечує прийнятний рівень продуктивності. Але застосунки .NET Core можуть бути попередньо скомпільовані у нативний код операційної системи (Windows, Linux чи MacOS), що в ряді випадків може сприяти суттєвому приросту продуктивності.

.NET Core версії 1.0 випущено в 2016 році. Актуальною на середину 2020 року є версія .NET Core 3.1. Загалом доповнення до .NET Core останні роки виходять частіше, ніж до .NET Framework. Проте компанія Microsoft декларує, що платформа .NET Framework підтримуватиметься й розвиватиметься далі. Таким чином, .NET Framework і надалі гратиме важливу роль у розробці застосунків для Windows (особливо для настільних систем). Переваги .NET Core роблять її кращим вибором для розробки веб-застосунків та крос-платформенних застосунків.

Microsoft надає також крос-платформенну мобільну платформу Xamarin. В основі її також лежать бібліотеки .NET. На основі Xamarin можна розробляти застосунки для мобільних ОС Android, iOS та Windows.

Застосунки для трьох згаданих платформ: .NET Framework, .NET Core та Xamarin розробляють головним чином за допомогою мови програмування C#.

Мова програмування C#

Мова C# (вимовляється "Сі-шарп") – це багатопарадигмова об'єктно-орієнтована та компонентно-орієнтована мова програмування зі строгою типизацією, розроблена для платформи .NET Framework. Використання мови визначене стандартами ECMA-334 [4] та ISO/IEC 23270:2006 [5]. Розроблена командою Microsoft Research під керівництвом Андерса Гейлсберга.

Синтаксис мови близький до мов C++ та Java. Деякі риси (наприклад, строга статична типизація), наближують її структуру до Delphi (Object Pascal). А введення протягом останніх 5 років у C# принципів функціонального програмування у певній мірі наближують її до функціональної мови програмування Scala.

Згідно стандарту ECMA-334, цілі, поставлені при розробці мови C#, були такими:

- C# має бути простою, сучасною, об'єктно-орієнтованою мовою програмування.
- Мова має підтримувати безпечні принципи програмування, такі як строга перевірка типів, перевірка меж масиву, виявлення спроб використання неініціалізованих змінних, і автоматичне прибирання сміття.
- Можливість розробки програмних компонентів для розподілених систем.
- Мова має підтримувати переносимість коду.
- Підтримка національних мовних та інших особливостей має бути простою.

Базовий синтаксис C# схожий до інших C-подібних мов, таких як C, C++ та Java, зокрема:

- Крапку з комою використовують для позначення кінця інструкції.
- Фігурні дужки використовують для формування програмних блоків.
- Значення змінним присвоюють за допомогою знаку "=", а для їх порівняння використовують "= =".
- Квадратні дужки використовують для індексації масивів.

Проте, C# має суттєві відмінності від розглянутих мов- попередників, у тому числі:

- Краща переносимість між різними платформами, пов'язана з тим, що мова відповідає специфікації CLI.
- Строга типизація робить мову безпечнішою. C# також підтримує логічний тип boolean.
- Мова забезпечує використання властивостей у класах, роблячи роботу з об'єктами зручнішою та безпечнішою.

- Програма на C# краще структурована завдяки групуванню коду в простори імен.
- Обмежене використання вказівників робить безпечнішою роботу з пам'яттю.

Перша версія C# 1.0 вийшла разом з Microsoft Visual Studio .NET у лютому 2002 року. В кінці 2019 року випущено версію C# 8.0. Порівняно із першою версією, мову C# суттєво розвинули, доповнивши її рядом вагомих рис, які роблять її сучасною, гнучкою та перспективною для вирішення широкого кола завдань.

Існує кілька середовищ програмування, які використовують мову C#, зокрема: Microsoft Visual Studio, JetBrains Rider, MonoDevelop, SharpDevelop тощо.

Найвживанішим з них є інтегроване середовище розробки Microsoft Visual Studio, яке, крім C#, підтримує і ряд інших мов програмування, а також забезпечує великий набір класів для розв'язування різних завдань.

Microsoft Visual Studio

Microsoft Visual Studio – це інтегроване середовище розробки програмного забезпечення. Воно складається із сукупності програмних продуктів для розробки та налагодження застосунків різних видів, у тому числі консольних програм, застосунків з графічним інтерфейсом користувача, web-сайтів, web-застосунків, web-служб тощо. Visual Studio використовує такі платформи для ПЗ, як Windows API, Windows Forms, Windows Presentation Foundation, Windows Store і Microsoft Silverlight. Середовище може компілювати як керований, так і некерований код.

Visual Studio має кілька редакцій з різними можливостями, серед них:

- **Visual Studio Enterprise**. Містить повний набір інструментів розробки від Microsoft, надає середовище для розробки всіх доступних мов програмування. Забезпечує інтеграцію з сервером баз даних Microsoft SQL Server та інші можливості, які роблять легкою розробку великих корпоративних проектів.
- **Visual Studio Professional**. Містить широкий набір інструментів розробки, у якому відсутні деякі компоненти з набору Enterprise. Як і версію Enterprise, використовують для розробки великих проектів.
- **Visual Studio Community**. Це безкоштовна версія з дещо урізаними можливостями. Вона дозволяє розробляти достатньо складні проекти, підтримує мови C#, Visual Basic .NET, F#, C++, Python, Node.js та

HTML/JavaScript. Ця редакція орієнтована на індивідуальних розроблювачів і невеликі команди.

Встановлення Microsoft Visual Studio Community

Завантажити програму- установлювач Microsoft Visual Studio Community можна з сайту https://www.visualstudio.com. Щоб почати встановлення, потрібно запустити завантажений файл. На рис. 1.5 показане вікно установлювача для Microsoft Visual Studio Community 2019. При встановленні можна вибрати компоненти, які будуть потрібні для роботи, зокрема, для підтримки тих чи інших мов та технологій програмування. Програма інсталяції має зручний і зрозумілий інтерфейс, так що процес встановлення є дуже простим.

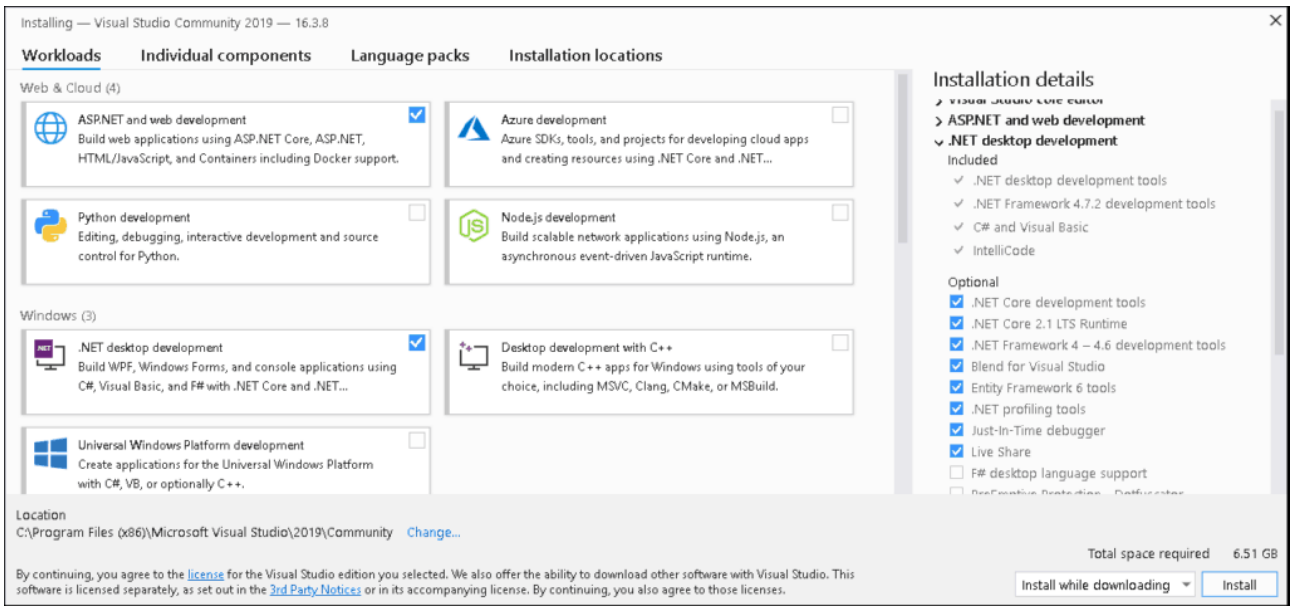


Рис. 1.5. Вікно установлювача Visual Studio

Під час встановлення за допомогою програми, отриманої із сайту, з мережі Internet завантажуватимуться додаткові дані, тому доступ до всесвітньої мережі має бути відкритим. Для встановлення без під'єднання до мережі потрібно скористатися образом інсталяційного диску[3]. Образ слід монтувати у віртуальний CD/DVD- привід або записати на фізичний носій і з нього запустити програму- встановлювач.

Встановлення Visual Studio слід виконувати від імені облікового запису адміністратора з типовими привілеями. Проте, в подальшому для роботи з Visual Studio права адміністратора не потрібні.

Рекомендовані системні вимоги для Microsoft Visual Studio Community 2019 такі:

[3] Його також можна завантажити з сайту Microsoft Visual Studio.

- Процесор з частотою 1.8 ГГц або потужніший.
- Пам'ять RAM обсягом 2 ГБ (8 ГБ рекомендовано).
- Від 800 МБ до 210 ГБ вільного простору на жорсткому диску (залежно від вибраних компонентів). Типова інсталяція потребує 20-50 ГБ.
- Для хорошої продуктивності рекомендовано як Windows, так і Visual Studio встановити на SSD.
- Відеокарта з підтримкою DirectX 9 і роздільною здатністю екрану не менше 1280×720 пікс, проте Visual Studio працюватиме краще при роздільній здатності WXGA 1366×768 пікс.

Середовище Microsoft Visual Studio

Середовище розробки Microsoft Visual Studio у його типовому вигляді (відразу після встановлення) містить такі основні частини: головне меню, панелі інструментів, вікно ToolBox з компонентами, редактор коду, дизайнер (форм), вікно Solution Explorer, вікно Properties для відображення та налаштування властивостей елементів проекту (рис. 1.6).

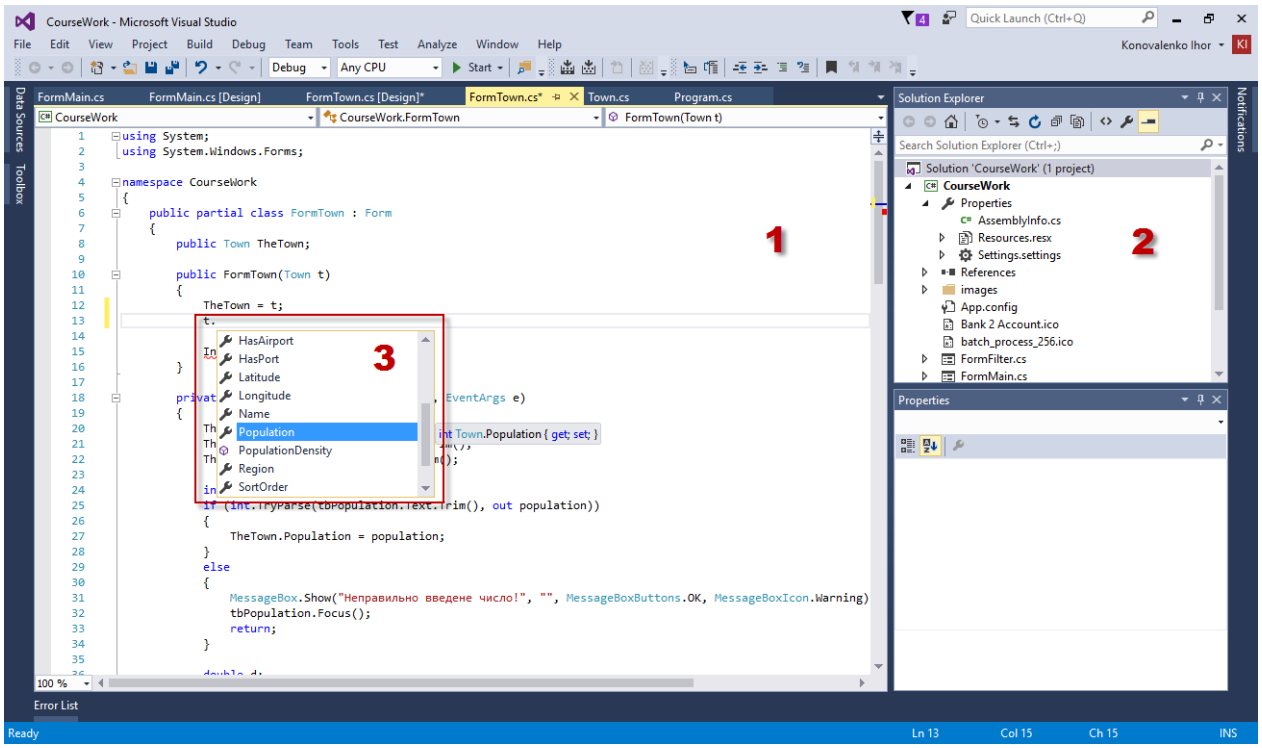


Рис. 1.6. Середовище Microsoft Visual Studio з редактором коду та прикладом роботи IntelliSense (1 – редактор коду; 2 – вікно Solution Explorer; 3 – вікно IntelliSence: при введенні назви об'єкта з'являється перелік його властивостей та методів)

У головному меню зібрано та згруповано за видами виконуваних завдань команди Visual Studio, які використовують для роботи зі складовими частинами проекту, його розробки, налаштування, пошуку помилок, налагодження тощо.

Під головним меню розташовано панелі інструментів. Загалом панелей є біля трьох десятків, але в кожен момент відображується лише кілька з них. Щоб проглянути перелік доступних панелей, ввімкнути потрібні чи вимкнути непотрібні, слід викликати контекстне меню у зоні панелей інструментів і відмітити у переліку бажані.

Сукупність файлів, створених під час розробки певного виконавчого модуля (програми, бібліотеки, web-сайту…) і необхідних для його компіляції, називають *проектом* (*project*). Ці файли можуть містити програмний код (наприклад, мовою C#), шаблони діалогових вікон, налаштування і конфігурацію проекту, його ресурси (додаткові дані типу піктограм, зображень) тощо. Кінцевий файл, у який компілюється проект, називають *збіркою*.

Пов'язані проекти об'єднують в *рішення* (*solution*). Рішення є контейнером для проекту. Так роблять, наприклад, якщо застосунок, який розробляють, містить кілька виконавчих модулів (наприклад, виконавчий файл та динамічні бібліотеки). Кожен проект входить до складу рішення. Навіть якщо передбачається, що проект буде простим і міститиме всього один виконавчий модуль, для нього створюють рішення (або додають його до існуючого рішення).

Збірка у .NET – одиниця розгортання та конфігурування. Прикладний застосунок може складатися з однієї чи кількох збірок. Найпоширеніші збірки – виконавчі файли .exe та динамічні бібліотеки .dll. Від "звичайних" .exe та .dll файлів вони відрізняються наявністю метаданих, які описують визначені у них типи, методи, події та інші елементи. Збірки містять інформацію про залежність від версій інших збірок. Функціонування .NET організовано таким чином, що збірки різних версій можуть завантажуватися і використовуватися одночасно (на відміну від "старих" динамічних бібліотек).

Вікно Solution Explorer (провідник рішень) середовища Microsoft Visual Studio (рис. 1.6, – 2) містить перелік проектів, які входять у відкрите рішення, та перелік всіх файлів, з яких складається проект. Це вікно зручно використовувати для перегляду складу проекту та відкривання вибраного файлу (для цього слід двічі клацнути мишкою на ньому). За допомогою цього вікна можна керувати складом проекту: додавати нові елементи (файли з кодом, форми тощо), змінювати назву чи видаляти існуючі у проекті файли та виконувати інші дії.

Якщо вікно провідника рішень відсутнє на екрані, його можна викликати командою меню View/ Solution Explorer. Загалом, керувати відображенням

різних вікон середовища Visual Studio можна за допомогою підменю View головного меню.

Бібліотеки класів (і, відповідно, функцій), які будуть доступні у проекті, під'єднують до проекту через посилання (references). Перелік посилань відображено у вікні Solution Explorer (у розділі References). Керувати посиланнями на бібліотеки у проекті (додавати і видаляти їх) можна за допомогою команди головного меню Project/ Add reference. Коли при створенні проекту вибирають його тип (див. таблицю 1.2), середовище автоматично під'єднує посилання, які відповідають вибраному типу.

Редактор коду

Visual Studio має зручний редактор для написання коду програми (рис. 1.6, – 1), який здійснює автоматичне форматування тексту та підтримує IntelliSense – компонент, розроблений Microsoft для автоматизованого введення коду. При розробці програми він надає підказки, що формуються на основі кількох перших введених символів. Це дозволяє швидко підбирати потрібну функцію, змінну чи тип, а також проглядати переліки доступних полів, властивостей та методів, разом з короткими описами їх параметрів. Вікно IntelliSense можна також викликати примусово, натиснувши комбінацію клавіш Ctrl+Space.

Ще один зручний механізм, який надає редактор Visual Studio – рефакторинг коду. Він полягає у зміні внутрішньої будови програми для полегшеного розуміння коду без зміни її зовнішньої поведінки.

Середовище підтримує роботу з різними мовами програмування, у тому числі C, C++ та C++/CLI, Visual Basic .NET, C# та F#. Для підтримки інших мов, (Phyton, Ruby), необхідне встановлення додаткових компонентів. Забезпечується також робота з документами XML, HTML та JavaScript.

В редакторі коду в текст програми можна встановлювати "закладки" і потім швидко переходити до потрібних місць коду. Для цього використовують набір команд Edit\Bookmarks. Особливо корисно це для великих за обсягом програм.

Редактор коду можна відкрити за допомогою клавіші F7 або команди головного меню View\ Code.

Дизайнери. Розробка форм

Visual Studio містить кілька компонентів – дизайнерів, призначених для розробки інтерфейсу застосунку:

- **Дизайнер Windows Forms.** Використовується для розробки графічного інтерфейсу звичайних віконних застосунків. При цьому реалізовано принципи візуального проектування – розміщуючи на формі[4] компоненти, можна відразу оцінити її вигляд. Visual Studio містить великий набір компонентів, які представляють всі найвживаніші елементи керування (кнопки, меню, списки, комбіновані списки, панелі інструментів тощо) і забезпечують широкі та гнучкі засоби, потрібні для розробки програм. Під час роботи застосунку форми цього виду відображує на екрані внутрішня підсистема операційної системи GDI[5] (чи GDI+, залежно від версії системи). Вона не використовує сучасних можливостей графічних карт комп'ютера, тому графіка на основі Windows Forms повільна, загалом неефективна і використовується переважно для відображення вікон (для цих цілей можливостей GDI цілком достатньо). Вікна, побудовані на основі Windows Forms, використовують для застосунків з простим інтерфейсом.
- **Дизайнер WPF.** Дозволяє розробляти форми, побудовані за технологією Windows Presentation Foundation. Такі форми описують спеціальною мовою XAML, що підвищує гнучкість їх використання і відділяє інтерфейс застосунку від його коду. Для відображення вікон цього типу на екрані використовуються потужні графічні засоби DirectX, тому графічний вивід відбувається значно швидше та ефективніше. WPF – нова технологія Microsoft, вона активно розвивається, і її використовують при проектуванні застосунків зі складним інтерфейсом.
- **Дизайнер WEB.** Використовують при розробці web-сторінок для застосунків ASP.NET. При цьому для опису скриптів використовують C# чи Visual Basic .NET. Може також генерувати діаграми класів.
- **Дизайнер класів.** Використовують для розробки класів з використанням моделювання UML. Дизайнер на основі моделі може генерувати код мовою C# чи Visual Basic .NET.

[4] Форма – це вікно застосунку на етапі його проектування.

[5] Graphics Device Interface (GDI) – підсистема операційної системи Windows, яка керує створенням та відображенням вікон застосунків. Починаючи з Windows XP, вона була вдосконалена і дістала назву GDI+.

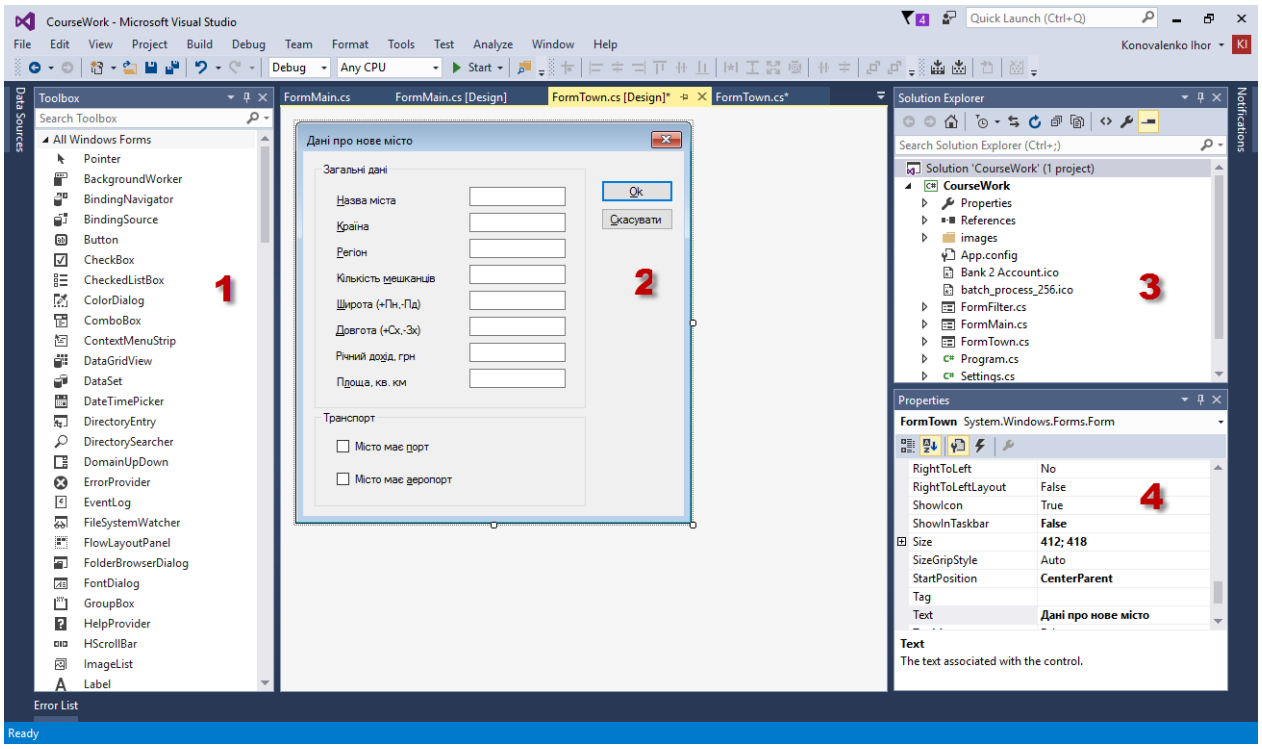


Рис. 1.7. Вікно Visual Studio з дизайнером форм Windows Forms (1 – вікно Toolbox з компонентами; 2 – дизайнер форм з формою, яка проектується; 3 – вікно Solution Explorer, провідник рішень; 4 – вікно Properties, властивості компонентів)

На рис. 1.7 показано вигляд дизайнера форм (з формою Windows Forms). Форму проектують, розставляючи на ній потрібні компоненти (текст, поля вводу, кнопки, маркери, меню тощо), які розміщені у вікні Toolbox. Компоненти згруповані за призначенням (Common Controls, Containers, Menus&Toolbars тощо). Щоб використати потрібний компонент, слід клацнути на ньому в панелі компонентів (рис. 1.7, – 1), а потім в потрібному місці дизайнера форм (рис. 1.7, – 2). Вибраний компонент з'явиться у дизайнері форм: на самій формі, або, якщо він не є частиною її інтерфейсу, то у полосі під формою.

Дизайнер можна відкрити за допомогою комбінації клавіш Shift+F7 або команди головного меню View\ Designer.

У подальшому розставлені в дизайнері форм компоненти можна переміщувати, змінювати їх розмір. Для вирівнювання, точного позиціювання та рівномірного розподілу компонентів призначена панель інструментів Layout середовища Visual Studio.

Використовуючи компоненти з вікна Toolbox, можна проектувати вікна з достатньо розвинутим дружнім інтерфейсом користувача, який забезпечить зручний доступ до функціоналу програми. Так як компоненти представляють

елементи керування давно усталеного звичного вигляду та поведінки, то розроблений за їх допомогою інтерфейс буде інтуїтивно зрозумілим.

Якщо в дизайнері форм виділити компонент (для цього слід клацнути на ньому мишкою), то у вікні Properties (рис. 1.7, – 4) буде відображено його властивості. За допомогою цього вікна їх значення можна змінювати: наприклад, задати текст напису, шрифт, його колір. Кожен вид компонентів має свою множину властивостей, які характеризують та описують його. Але всі без винятку компоненти мають властивість Name (назва компонента). Вона особлива тим, що представляє компонент у коді. Саме за цією назвою (фактично, це назва змінної певного класу) здійснюється доступ до компонента та всіх його ресурсів з тексту програми.

Кожен компонент програмно представлений певним базовим класом, за допомогою властивостей та методів якого можна керувати поведінкою компонента з коду.

Детальніше проектування вікон Windows Forms та використання відповідних компонентів розглянемо у розділі "Застосунки Windows Forms".

Документація MSDN

Microsoft надає вичерпну документацію по свої засобах, включаючи середовище Visual Studio та елементи мови програмування C#, внутрішні класи, їх методи, властивості, типи даних тощо. Ця документація поширюється через MSDN (Microsoft Developer Network) – підрозділ компанії Microsoft, який відповідає за співпрацю з мережею зацікавлених розробників програмного та апаратного забезпечення.

Вказана документація (та багато іншої інформації) доступна на сайті MSDN: https://msdn.microsoft.com.

Microsoft Visual Studio дозволяє отримувати доступ до документації MSDN прямо з редактора коду. Зокрема, щоб отримати довідку щодо певного елемента у коді, слід виділити його і натиснути F1 (рис. 1.8). Сторінка довідки (при наявності під'єднання до Internet) відкриється у браузері. Можна також виділити компонент у дизайнері форм та натиснути F1 – відкриється сторінка з довідкою по класу компонента.

Крім цього, довідку щодо потрібного елемента програми можна отримати, скориставшись полем пошуку безпосередньо на сайті MSDN.

Довідкові статті, як правило, містять вичерпну інформацію, і часто ілюстровані простими прикладами коду мовами C# та Visual Basic .NET.

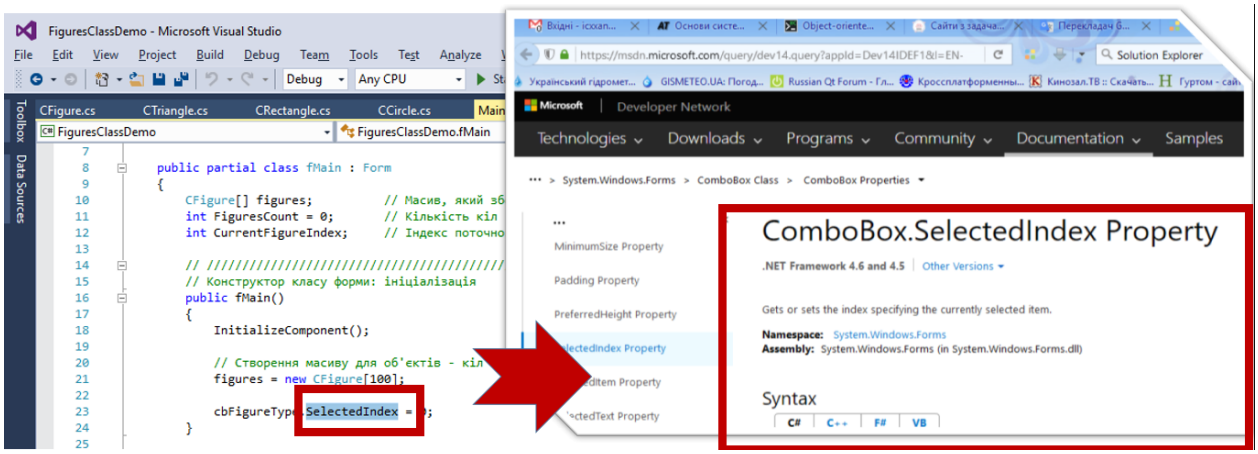


Рис. 1.8. Отримання контекстної довідки в редакторі коду: виділяємо назву властивості (на рисунку SelectedIndex), натискаємо F1, і у браузері відкривається довідка щодо виділеної властивості

Створення проекту

Створити новий проект у Visual Studio можна за допомогою команди меню File\ New\ Project (її можна швидше викликати за допомогою комбінації клавіш Ctrl+Shift+N. При створенні проекту вказують його тип та версію .NET, для якої він створюється. Середовище автоматично генерує необхідні для такого типу проекту файли, під'єднує потрібні компоненти середовища, додає необхідний базовий код та налаштовує параметри компілятора. Найвживаніші типи (шаблони) проектів для Windows приведено у таблиці 1.2.

Таблиця №1.2. Найвживаніші типи проектів Visual Studio для Windows

Шаблон	Опис
Windows Forms Application	Віконний застосунок із елементами керування Windows Forms, використовує графіку GDI+
WPF Application	Віконний застосунок із елементами керування, використовує XAML на основі графіки DirectX
Console Application	Консольна програма, яка виконується у вікні командного рядка
Class Library	Бібліотека класів .NET, до якої можна звертатися з інших програм
Empty Project	Порожній проект, містить тільки файл конфігурації та налаштування для консольного застосунку
Windows Service	Служба Windows
WPF Custom Control Library	Бібліотека спеціальних компонентів, які можуть використовуватися у застосунках WPF
Windows Forms Control Library	Бібліотека елементів керування для застосунків Windows Forms

При створенні проекту слід вказати рішення, у яке входитиме новий проект. Можна створити для проекту нове рішення, а можна додати його до вже існуючого рішення.

Після створення проекту, відповідно до вибраного типу, середовище згенерує ряд файлів з заготовками коду. Їх перелік міститься у вікні Solution Explorer, де пов'язані файли згруповані. Якщо мовою програмування вибрано C#, то частина з цих файлів матимуть розширення .cs (**c-s**harp). У них міститься код цією мовою. В подальшому розглядатимемо лише проекти з кодом мовою C#.

Папка, в яку збережено рішення з проектом, містить файли таких типів:

.sln – файл рішення. Описує його структуру.
.csproj – файл проекту на C#. Описує структуру проекту.
.cs – файл з кодом мовою C#.
.config – файл конфігурації.
.resx – файл з описом ресурсів.
.settings – файл налаштувань.

В загальному випадку перелік файлів залежить від типу вибраного проекту, але дуже часто серед них є файл Project.cs. У ньому оголошено клас Program, який представляє розроблювану програму. Цей клас містить один метод з назвою Main, який представляє точку входу у програму. Саме з інструкцій цього методу починається виконання програми.

У випадку консольного застосунку в методі Main пишуть програмний код. Для застосунків складніших типів (зокрема, Windows Forms Application чи WPF Applications) цей метод, як правило, залишають незмінним, а код розміщують в інших методах.

Налагодження програм

Під час розробки ПЗ важко уникнути помилок. Програмний засіб для їх пошуку і усунення називають *налагоджувачем* (*зневаджувачем*, англ. – debugger).

Microsoft Visual Studio містить внутрішній налагоджувач, який забезпечує зручні засоби для пошуку та виправлення помилок, зокрема: встановлення точок переривання у програмі, покрокове виконання коду, перегляд стеку та регістрів процесора тощо.

Для використання можливостей налагодження компілятор у виконавчий файл розміщує додаткову налагоджувальну інформацію. Це збільшує розмір виконавчого файлу (часто у кілька разів), а також дещо уповільнює його роботу. Тому середовище розробки пропонує дві конфігурації: Debug (її використовують під час розробки застосунку, оскільки вона забезпечує

необхідні можливості налагодження) та Release (використовують для побудови оптимізованого виконавчого файлу без налагоджувальної інформації). Потрібну конфігурацію вибирають з випадаючого списку Solution Configuration на панелі інструментів Standard (рис. 1.9). Налаштувати особливості роботи компілятора для обох конфігурацій можна у вікні властивостей проекту (на сторінках Build та Debug), яке викликають через команду меню Project\…Properties…

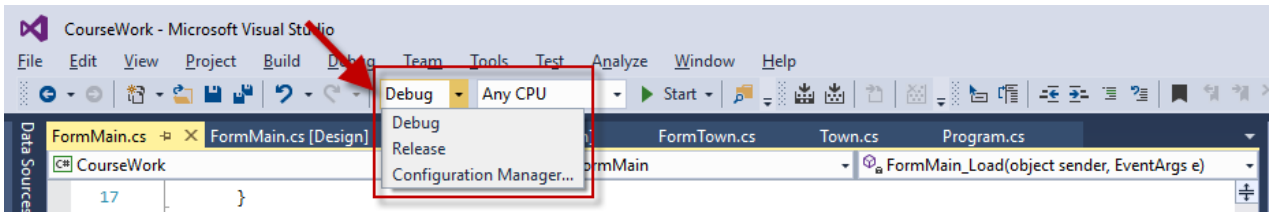


Рис. 1.9. Вибір конфігурації рішення: Debug та Release

Отже, для використання можливостей налагоджувача слід пересвідчитися, що проект скомпільовано в конфігурації Debug.

При налагодженні найчастіше використовують переривання програми у певних контрольних точках (з наступною перевіркою стану даних) та покрокове виконання програми.

Щоб розмістити у коді точку переривання програми, є такі варіанти дій:

- Помістити курсор в потрібний рядок коду і натиснути клавішу F9 (повторне натискання видалить точку переривання у цьому рядку).
- Клацнути мишкою зліва від потрібного рядка коду (рис. 1.10). Повторне клацання видалить точку переривання.

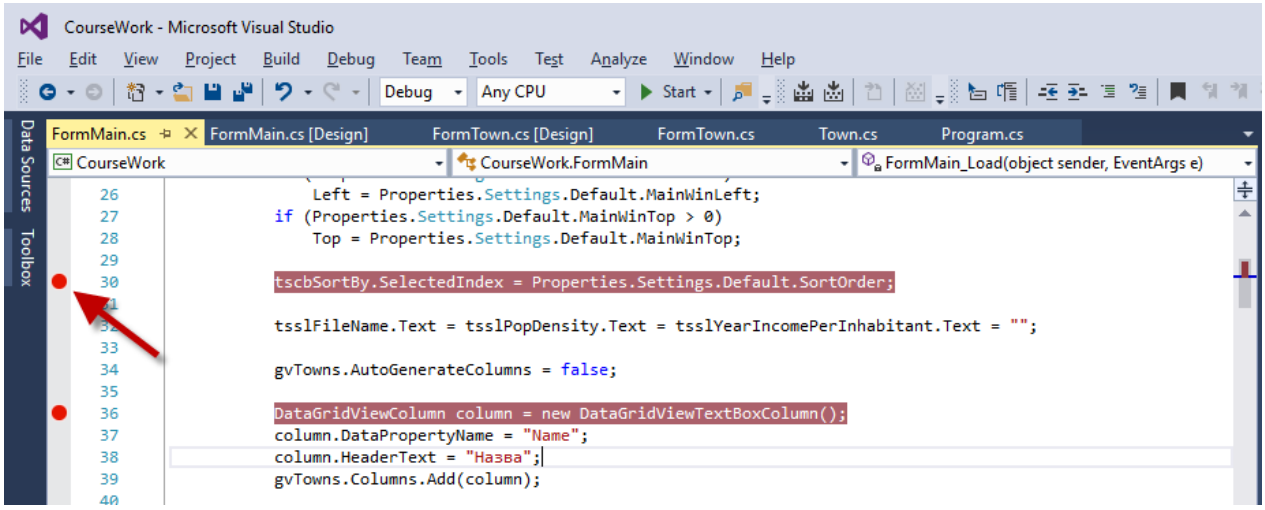


Рис. 1.10. Точки переривання у коді. Стрілкою позначено поле, у якому слід клацнути мишкою для встановлення/ видалення точки переривання

При встановленні точки переривання відповідний рядок коду виділяється (як правило, червоним кольором), а в бічному полі напроти цього рядка з'являється маркер (як правило, червона точка), як показано на рис. 1.10.

Програму з середовища Visual Studio можна запустити як у режимі налагодження, так і без нього. Щоб виконати програму, не задіюючи при цьому механізм налагодження, використовують комбінацію клавіш Ctrl+F5 (або команду меню середовища Debug\ Start Without Debugging.

Щоб запустити застосунок з під'єднанням засобів налагодження, є кілька способів:

- Натиснути на кнопку Start на панелі інструментів.
- Натиснути на клавішу F5 клавіатури.
- Вибрати команду меню Debug\ Start Debugging.

Після запуску програми в режимі налагодження середовище Microsoft Visual Studio дещо змінить свій вигляд (рис. 1.11) і переорієнтується на режим налагодження. При цьому в заголовку головного вікна Visual Studio буде присутня фраза (Debugging), а рядок стану біля нижнього краю вікна змінить свій колір (типово – з синього (див. рис.1.7) на оранжевий).

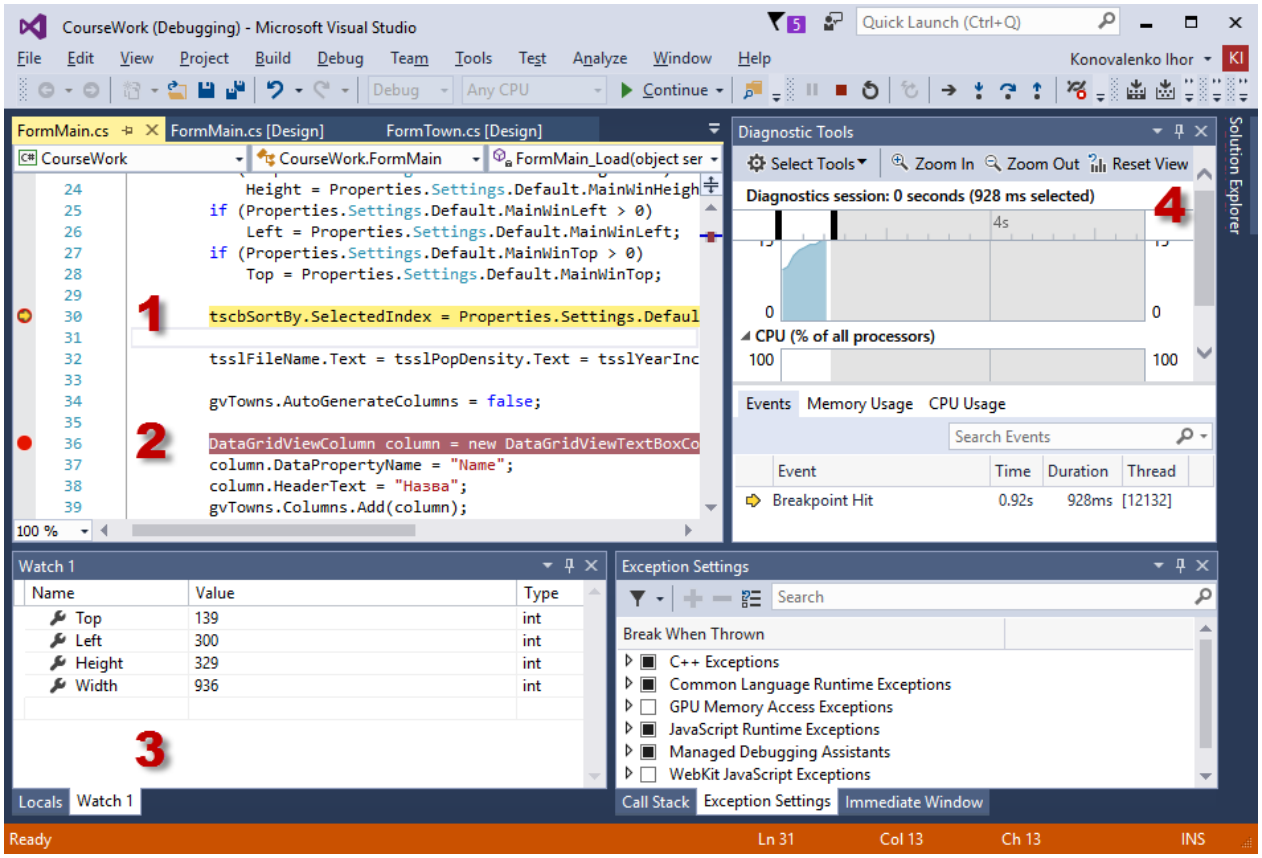


Рис. 1.11. Середовище Microsoft Visual Studio в режимі налагодження (1 – поточний рядок коду, 2 – точка переривання, 3 – вікно для перегляду значень змінних, 4 – інструменти діагностики)

Поточний рядок коду, на якому призупинено виконання програми, позначено маркером зі стрілкою в бічному полі редактора (рис. 1.11, – 1) і підсвічено іншим кольором. Значення змінних на даному етапі виконання програми можна переглядати, якщо наводити курсор миші на змінну. При цьому з'являтиметься випливаюче вікно з назвою змінної та її значенням. Крім цього, для перегляду поточних значень вибраних змінних призначене вікно Watch (рис. 1.11, – 3). Щоб додати змінну в це вікно, її назву слід виділити (наприклад, двічі клацнувши на ній мишкою) і перетягти у вікно Watch. Можна також викликати контекстне меню на цій змінній, і вибрати команду Add Watch.

У режимі налагодження середовище розробки відображує діагностичну інформацію у вікні Diagnostic Tools (рис. 1.11, – 4). Тут в режимі реального часу можна переглянути ступінь завантаження процесора під час роботи програми та прослідкувати використання пам'яті.

Середовище Microsoft Visual Studio містить панель інструментів Debug, яка стає доступною в режимі налагодження і містить кнопки для швидкого доступу до найвживаніших функцій налагодження.

Для аналізу роботи програми дуже корисним є покрокове виконання, яке дозволяє перевірити стан змінних після виконання кожної інструкції. Використовують два способи покрокового виконання:

- З входженням у функції. При цьому, якщо в поточному рядку коду є функція і для неї існує налагоджувальна інформація (для всіх функцій поточного проекту вона генерується при компіляції), то наступний крок переходить до першої інструкції всередині функції.
- Без входження у функції. У цьому випадку, якщо в поточному рядку програми є функція, вона виконується повністю і наступний крок переходить до наступної інструкції поточного блоку коду.

Покрокове виконання дозволяє аналізувати стан змінних після кожного кроку виконання програми. Порівнюючи реальні значення з тими, які очікуються згідно задуманого алгоритму, можна виявити неправильно функціонуючі рядки коду.

Щоб продовжити роботу над проектом (наприклад, для внесення змін у виявлений хибно функціонуючий код), слід закрити запущену програму. Для примусової зупинки запущеної програми слід натиснути кнопку Stop Debugging на панелі інструментів середовища або скористатись комбінацією клавіш Shift+F5.

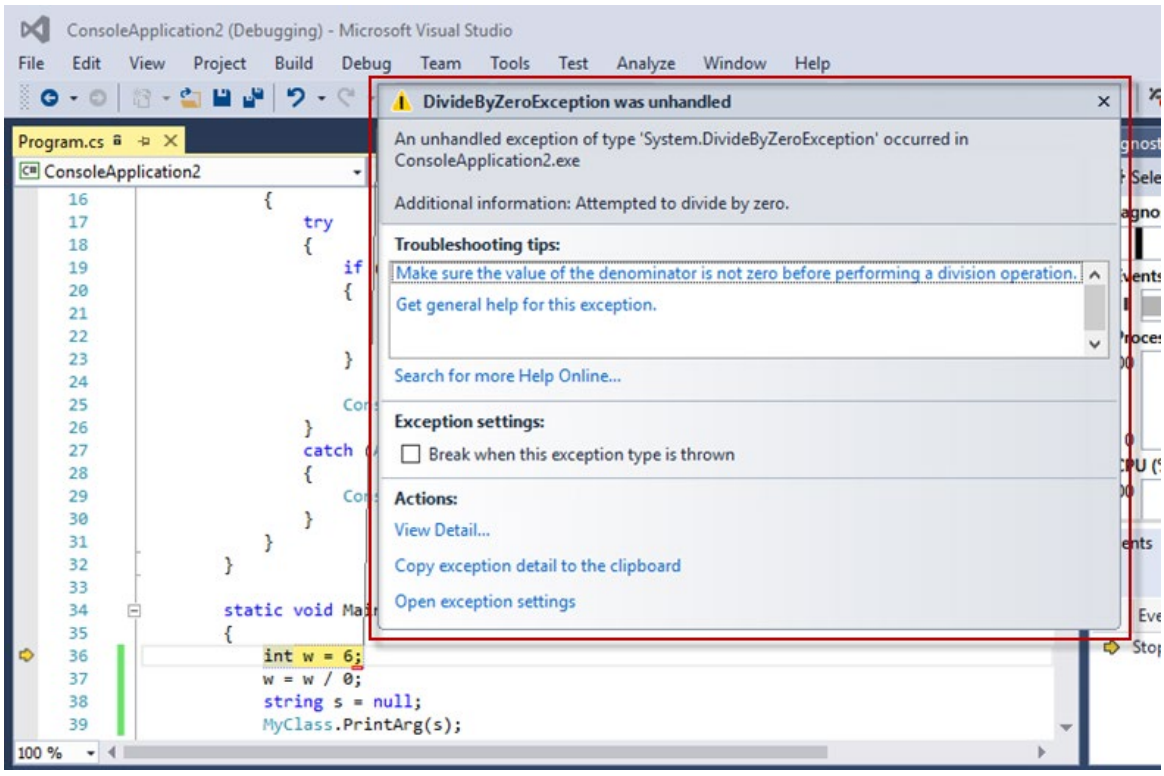


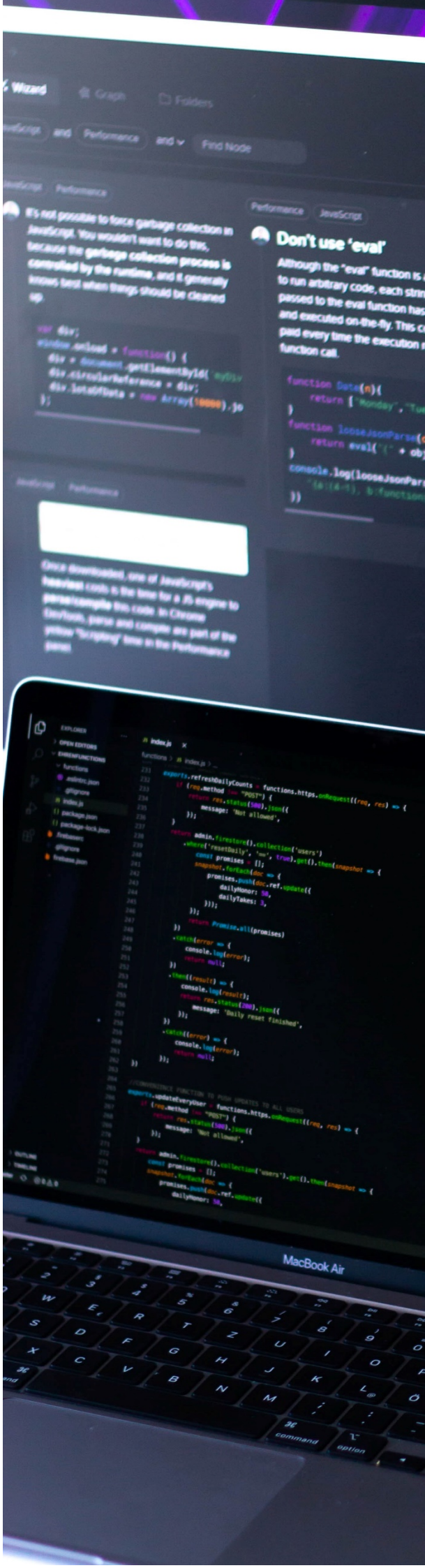
Рис. 1.12. Повідомлення про виняткові ситуацію під час виконання програми (виділено). Роботу програми зупинено на рядку 36 (у рядку 37 виконується ділення на нуль, яке спричинило помилку)

Якщо при виконанні програми трапилась виняткова ситуація (помилка часу виконання), яка не обробляється кодом, то виконання зупиняється, а Visual Studio відображує вікно з описом виняткової ситуації (рис. 1.12).

У цьому вікні вказано тип (клас) виняткової ситуації, а також її короткий опис (наприклад, на рис. 1.12 вказано тип System.DivideByZeroException та опис "Attempted to divide by zero"). Ця інформація дозволяє спростити виявлення причини помилки та її усунення. В такій ситуації, щоб перейти до роботи над проектом (і виправити помилковий код), слід вийти з режиму налагодження (Shift+F5).

Розділ 2. Основи C#

- Проста програма на C#
- Ідентифікатори
- Ключові слова
- Простори імен
- Пропуски у коді
- Коментарі
- Типи і змінні
- Оператори та вирази
- Інструкції та блоки інструкцій
- Керування потоком виконання



Проста програма на C#

Цей розділ буде основою для вивчення C#. У ньому розглядатимуться базові принципи програмування цією мовою, які ілюструватимуться простими прикладами програм, розроблених у середовищі Microsoft Visual Studio. Для розуміння принципів роботи цих програм розглянемо структуру найпростішої консольної програми на C# та загальне призначення різних її елементів. В наступних розділах деякі з цих елементів розглянемо детальніше.

Щоб створити проект консольної програми, у Visual Studio слід вибрати команду меню File\ New\ Project, і вибрати шаблон Visual C#\ Windows\ Console Application. Середовище згенерує проект консольного застосунку з текстом програми як у лістингу 2.1, за винятком рядків 9-10, які містять текстовий коментар та виклик методу WriteLine[6].

Лістинг 2.1. Файл Program.cs найпростішого консольного застосунку OurFirstApplication

```
1  using System;
2
3  namespace OurFirstApplication
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              // Наступний рядок слід дописати самому
10             Console.WriteLine("Наша перша програма :)");
11         }
12     }
13 }
```

Під час роботи ця програма виводить у консольне вікно єдину текстову фразу – "Наша перша програма :)" (рис. 2.1). Запустити програму з середовища Microsoft Visual Studio можна у двох режимах:

- У режимі налагодження (натиснувши F5). При цьому, коли консольна програма закінчує роботу, її вікно відразу закривається. І якщо код не містить спеціальних інструкцій очікування, побачити результат роботи програми не вдасться.

[6] Лістинг містить стовпчик з номерами рядків для зручності його опису. Номери рядків не є складовою частиною програми.

- Без налагодження (натиснувши Ctrl+F5). У цьому випадку, після відпрацювання консольної програми, середовище не закриває її вікна, а очікує натискання довільної клавіші клавіатури (про що повідомляє фразою "Press any key to continue…" – див. рис. 2.1).

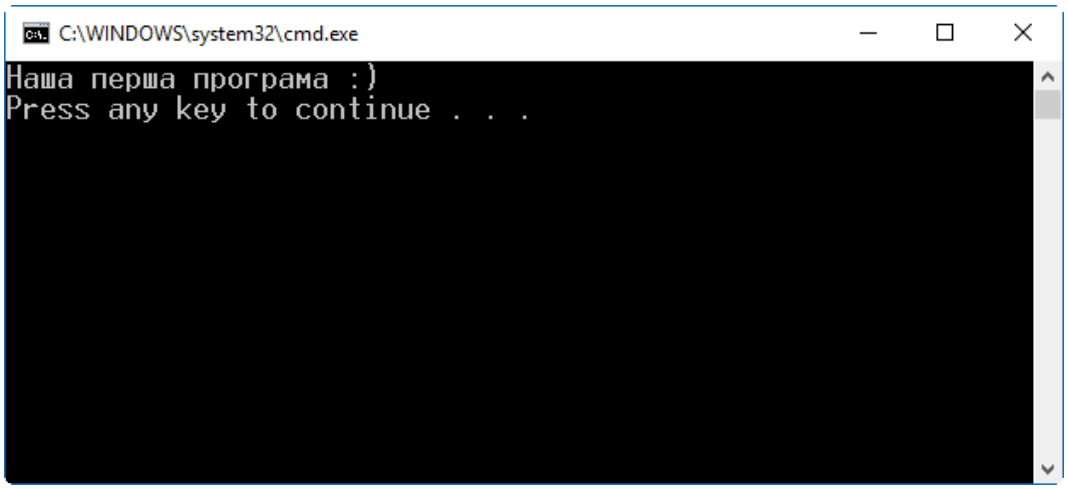


Рис. 2.1. Результат роботи програми з лістингу 2.1

Перевагою режиму налагодження є можливість використання спеціальних засобів для дослідження роботи програми (точок переривання програми, покрокове виконання програми тощо). Тому під час розробки програм найчастіше використовують саме його. Але ми розглядатимемо дуже прості (і готові) приклади, тому будемо приводити знімки екрану консольних програм, запущених без під'єднання налагоджувача (через натискання Ctrl+F5 або команди меню Debug\ Start Without Debugging).

Розглянемо код (лістинг 2.1) детальніше. Він розміщений у файлі, який за замовчуванням називається Program.cs. Рядок 9 починається з двох символів "//": так у коді позначають коментар. Текст після цих символів до кінця рядка може бути будь-яким – компілятор його не розглядає як складову програми. Таким способом у текст програми вписують пояснення до тих чи інших ділянок коду.

У таблиці 2.1 приведено детальний порядковий опис коду з лістингу 2.1.

Таблиця №2.1. Опис коду програми з лістингу 2.1

Номер рядка	Опис
1	Повідомляє компілятор, що програма використовує типи з простору імен System.
3	Оголошення нового простору імен з назвою OurFirstApplication. Цей простір імен починається відкриваючою фігурною дужкою у рядку 4 і закінчується закриваючою фігурною дужкою в рядку 13. Всі типи, оголошені у вказаних межах, є елементами простору імен OurFirstApplication.

Номер рядка	Опис
5	Оголошення нового класу з назвою Program. Все, що оголошене між парою фігурних дужок у рядках 6 та 12, належить до цього класу і формує його структуру та поведінку.
7	Оголошення методу Main, елемента класу Program. У цій програмі Main є єдиним елементом класу Program. Функція з назвою Main використовується як початкова точка програми. Реалізація методу Main знаходиться між парою відповідних фігурних дужок у рядках 8 та 11.
10	Містить єдину просту інструкцію. Цей рядок становить тіло метода Main. У C# кожна інструкція закінчується крапкою з комою. Інструкція в цьому рядку використовує клас із назвою Console, який розміщений у просторі імен System, для виводу рядка у консольне вікно. Без використання інструкції using в рядку 1 компілятор не знав би, де шукати клас Console. Метод WriteLine() класу Console використовують для виведення тексту у вікно.

Програма на C# складається з одного чи більше оголошень типів, які можуть бути розташовані в довільному порядку. У програмі OurFirstApplication оголошено тільки один тип – клас Program.

Функція Main: початкова точка програми

Кожна програма на C# повинна мати один клас з методом із назвою Main (перша літера – велика, на відміну від C та C++). У програмі з лістингу 2.1 цей метод оголошено у класі Program. Виконання будь-якої програми на C# починається з першої інструкції метода Main.

Найпростіша форма цього методу така:

```
static void Main()
{
    // Інструкції коду
}
```

Текстовий ввід/ вивід у консольному застосунку

Консольне вікно (рис. 2.1) – просте вікно з командним рядком, яке дозволяє здійснювати текстовий ввід-вивід за допомогою клавіатури. Бібліотека базових класів .NET Framework (BCL) підтримує клас Console (в просторі імен System), який містить методи для введення та виведення текстових даних у консольному вікні. Таке вікно містить *курсор* – блимаючий маркер, який позначає поточну позицію у вікні. Ввід та вивід текстових даних здійснюється саме у позиції курсора.

Для виводу тексту в консольне вікно використовують методи (функції) Write та WriteLine класу Console.

Метод Write

Метод Write виводить у консольне вікно застосунку переданий йому рядок тексту, починаючи з поточної позиції курсора. У C# (як і в C та C++) значення типу "рядок" має бути оточене подвійними лапками.

Наступний рядок коду демонструє використання метода Write:

```
Console.Write("Це просто текст");
```

Метод Write класу Console

Рядок тексту для виводу у вікно

Якщо цей рядок коду ввести у метод Main (замість рядка 10) у лістингу 2.1, то застосунок виведе у вікно фразу "Це просто текст". Після виводу тексту за допомогою методу Write не здійснюється перехід до наступного рядка. Наприклад, послідовний вивід трьох текстових фраз в одному рядку можна здійснити так:

```
Console.Write("Текст1 ");
Console.Write("Текст2 ");
Console.Write("Текст3 ");
```

Результат виконання цього коду приведено на рис. 2.2.

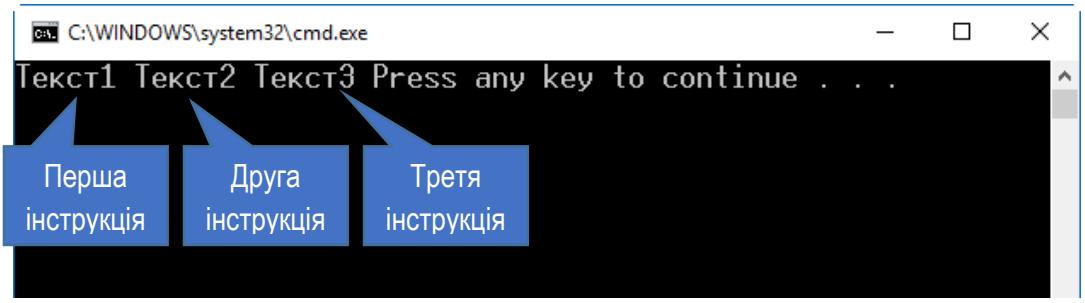


Рис. 2.2. Вивід трьох фраз за допомогою методу Write

Метод WriteLine

Метод WriteLine, який також є елементом класу Console, виконує те ж, що і метод Write, але додає у кінці виведеного рядка символ переходу до наступного рядка. Якщо переписати попередній фрагмент коду, замінивши Write на WriteLine, то отримаємо результат, як на рис. 2.3:

```
Console.WriteLine("Текст1 ");
Console.WriteLine("Текст2 ");
Console.WriteLine("Текст3 ");
```

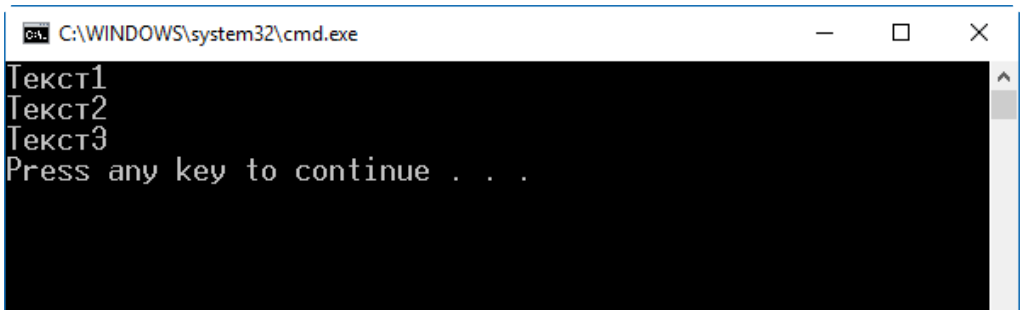


Рис. 2.3. Вивід трьох фраз за допомогою методу WriteLine

При виведенні рядків їх можна поєднувати один з одним та з числовими значеннями за допомогою знаку "+":

```
Console.WriteLine("Частота: " + 2.4 + " ГГц");
```

В результаті виконання цього коду у вікно буде виведено рядок: "Частота: 2.4 ГГц".

Форматування рядків

Текст при виведенні за допомогою методів Write та WriteLine можна відразу форматувати. У цьому випадку методу передають кілька параметрів і розділяють їх комами. Перший параметр завжди повинен містити рядок, який називають *рядком формату*. Рядок формату повинен містити *маркери підстановки*, які показують позиції, у яких будуть підставлені певні значення. Маркер підстановки є натуральним (невід'ємним цілим) числом у фігурних дужках, яке задає номер значення для підстановки. Маркери нумерують, починаючи з нуля. Значення для підстановки задають у параметрах після рядка формату.

Загальний синтаксис виклику методу для виведення форматованого рядка такий:

```
Console.WriteLine(FormatString, Value0, Value1, Value2...);
```

де FormatString – рядок формату, ValueN – значення для підстановки.

Наприклад, подана нижче інструкція містить виклик методу WriteLine з рядком формату (він містить три маркери підстановки) та трьома значеннями (10, 20 і 30) для підстановки:

Результат виводу цього фрагмента коду приведено на рис. 2.4.

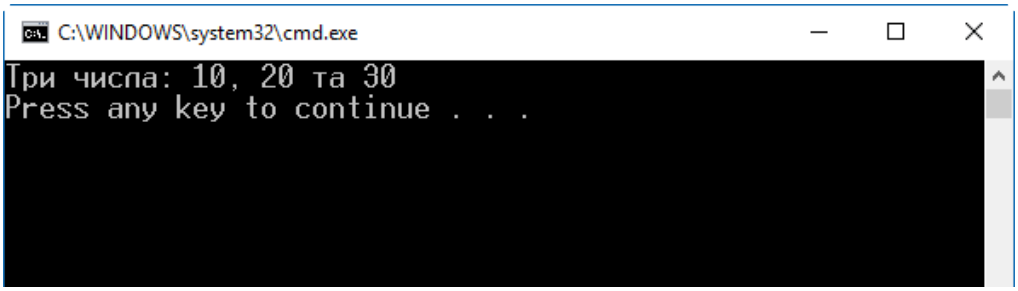
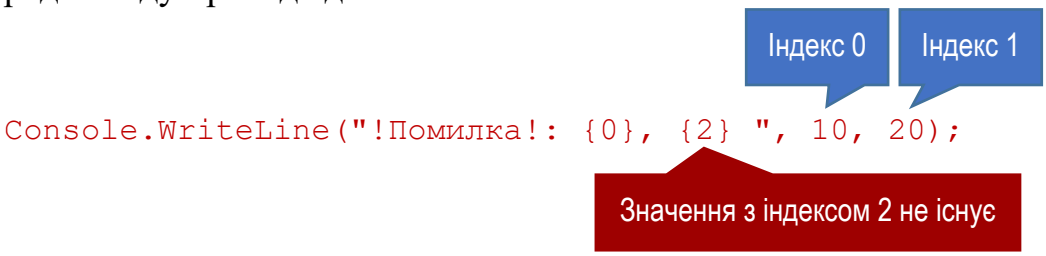


Рис. 2.4. Вивід рядка з форматуванням

У рядках формату може бути довільна кількість маркерів підстановки. При цьому їх можна розміщувати в довільному порядку, а маркер з тим же номером можна використовувати кілька разів.

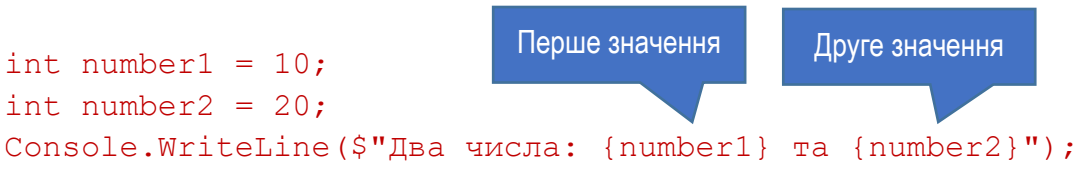
Маркер підстановки не повинен звертатися до елемента за межами переліку значень для підстановки: для кожного маркера з індексом N має існувати значення у переліку параметрів з індексом N. Якщо задати значень менше, ніж є маркерів, то програма скомпілюється і запуститься успішно, але під час виконання трапиться помилка (виняткова ситуація) з описом: "Index (zero based) must be greater than or equal to zero and less than the size of the argument list" (індекс (з основою нуль) має бути більшим або дорівнювати нулю, і меншим за розмір списку аргументів). Тому виконання наступного рядка коду приведе до помилки:



Інтерполяція рядків

Починаючи з версії C# 6.0 синтаксичні конструкції для форматування можна спростити, застосувавши *інтерполяцію рядків (string interpolation)*. Інтерполяція рядків робить код легшим для сприйняття.

Щоб вказати компілятору, що рядок слід інтерполювати, перед ним розміщують символ **$**. У самому рядку безпосередньо вказують назви змінних, оточуючи їх фігурними дужками:



Ідентифікатори

Ідентифікатори (identifiers) – це символьні слова, які використовують для позначення змінних, констант, методів, параметрів та багатьох інших програмних елементів, які розглядатимемо далі.

Прийнято давати самоописувальні ідентифікатори, через поєднання значущих слів у єдину описову назву та розділенням їх за допомогою регістру символів (наприклад, BookId, SecurityNum, FirstName тощо).

Для формування ідентифікаторів можна використовувати символи літер, цифр, а також символи "_" та "@". Сучасні компілятори підтримують використання національних символів для ідентифікаторів, але з метою уникнення плутанини та переносимості коду рекомендовано використовувати тільки символи латиниці.

При формуванні ідентифікаторів слід пам'ятати такі правила:

- Символи літер та підкреслення "_" можна розміщувати у будь-якій позиції.
- Символи цифр не можна розміщувати на початку ідентифікатора, але в будь-якому іншому місці – можна.
- Символ "@" можна розміщувати тільки на початку ідентифікатора, і не можна в інших місцях.

Ідентифікатори у C# (як і в C та C++) є чутливими до регістра. Наприклад, ідентифікатори myInt, MyInt та myint позначають різні змінні.

Проте, використання дуже схожих імен погіршує розуміння коду, сприяє допущенню помилок і робить налагодження програми важчим. Тому застосовувати такі ідентифікатори не рекомендовано: це вірно синтаксично, але заплутує код:

```
// Так робити не варто, хоча синтаксично це правильно
int totalSumm;
int TotalSumm;
int Totalsumm;
TotalSumm = totalSumm + Totalsumm;
```

Використовують різні стилі іменування[7] ідентифікаторів, які відрізняються використанням регістру символів (таблиця 2.2).

Часто для назв також використовують символ підкреслення "_":

- Для групи "особливих" ідентифікаторів назву починають підкресленням: _GetName, _GetObjectType.
- Окремі слова в ідентифікаторі розділяють символом "_": Department_Id, Language_Code, department_id, language_code.

[7] Їх ще називають угодами іменування (naming conventions).

В подальшому у прикладах коду будемо дотримуватися застосування стилів іменування за таблицею 2.2.

Таблиця №2.2. Стилі іменування

Назва стилю	Опис	Рекомендовано використовувати для:	Приклади
Pascal	Кожне слово в ідентифікаторі починають з великої літери	Назв типів, класів та їх елементів	TotalSumm, DepartmentId
Camel	Кожне слово в ідентифікаторі, крім першого, починають з великої літери	Локальних змінних та параметрів	totalSumm, departmentId
Uppercase	Всі літери великі	Тільки для абревіатур	TS, AXML

Ключові слова

Ключове слово (keyword) – це формальний символьний рядок, який визначено мовою програмування. Ключові слова не можна використовувати в ролі ідентифікаторів (тобто, для назв змінних методів тощо). Якщо за логікою програми потрібно використати ідентифікатори, які співпадають з ключовими словами C#, їх можна доповнювати символами "@" чи "_" (наприклад, @float, @int, float_, int_).

Всі ключові слова C# містять тільки малі літери. Перелік ключових слів C# приведено у таблиці 2.3.

Таблиця №2.3. Ключові слова мови C#

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	in (generic modifier)	int	interface
internal	is	lock	long
namespace	new	null	object
operator	out	out (generic modifier)	override
params	private	protected	public
readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc
static	string	struct	switch
this	throw	true	try
typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual
void	volatile	while	

Існують також *контекстні ключові слова*. Вони діють як ключові слова тільки в деяких конструкціях мови програмування. В інших місцях вони діють як звичайні ідентифікатори. Таблиця 2.4 містить перелік контекстних ключових слів мови C#.

Таблиця №2.4. Контекстні ключові слова мови C#

add	alias	ascending
async	await	descending
dynamic	from	get
global	group	into
join	let	orderby
partial (type)	partial (method)	remove
select	set	value
var	where (generic type constraint)	where (query clause)
yield		

Простори імен

Простір імен (namespace) – це сукупність оголошень типів, співставлених та пов'язаних з певною назвою. Програма з лістингу 2.1 використовує два простори імен. По-перше, вона створює новий простір імен OurFirstApplication (за замовчуванням його назва співпадає з назвою проекту, яку задали при його створенні). У ньому оголошено власний тип – клас Program. По-друге, програма використовує простір імен System, клас Console з якого вжито для виводу тексту у вікно.

Простір імен оголошують так:

```
namespace TheNameOfNameSpace
{
    //Елементи простору імен
}
```

TheNameOfNameSpace – це назва простору імен. Між фігурними дужками описують елементи простору імен – типи та класи. Простір імен є контейнером, у якому описують різні типи. Він забезпечує заданий програмістом вид групування коду, тому в одному просторі імен доцільно групувати логічно чи функціонально пов'язані типи.

Звертання до типу в просторі імен роблять через його назву, розділюючи їх ідентифікатори крапкою:

```
TheNameOfNameSpace.ClassName
```

Назва простору імен

Назва типу (класу)

Якщо код має використовувати типи з певного простору імен, то, щоб постійно не вказувати назву простора імен, можна вказати компілятору, що ми використовуємо даний простір імен. Для цього використовують ключове слово using:

```
using TheNameOfNameSpace;
```

Тоді звертатися до елемента простору імен можна коротше, опускаючи його назву:

```
using System;     // Директива використання простору імен
...
// Використовуємо повну назву, яка містить назву простору імен
System.Console.WriteLine("This is the text 1");
// Назву простору імен опускаємо
Console.WriteLine("This is the text 2");
```

Зауважимо, що починаючи з C# 6.0 є ще зручніший спосіб використання методів класу Console – за допомогою інструкції using static:

```
using static System.Console;
```

Тоді у коді можна навіть не вказувати назви класу Console:

```
WriteLine("This is the text");
```

Пропуски у коді

Пропуски у коді програми формуються символами тексту, які не відображуються на екрані. Компілятор ігнорує пропуски, але вони потрібні і використовуються програмістами для того, щоб зробити програму зрозумілішою, краще структурованою і легшою для читання.

Пропуски формують:

- Пробілами.
- Табуляцією.
- Переходом до наступного рядка.

Розглянемо два фрагменти коду:

1)
```
1  class Program
2  {
3      static void Main(string[] args)
4      {
5          Console.WriteLine("Наша перша програма :)");
6      }
7  }
```

2)
```
1  class Program{static void Main(string[] args){
2  Console.WriteLine("Наша перша програма :)");}}
```

З точки зору компілятора вони ідентичні: однаково компілюватимуться і однаково виконуватимуться. Але перший фрагмент значно легше читається: відступи зліва формують інтуїтивне сприйняття вкладеності блоків коду. Тому відразу зрозуміло, яка ділянка коду формує тіло класу Program (рядки 3-6), а яка – тіло метода Main (рядок 5).

При введенні тексту програми редактор коду Microsoft Visual Studio переважно автоматично вирівнює код, виконуючи основну роботу по форматуванню замість програміста.

Коментарі

Коментар – це довільний текстовий блок у програмі, який використовують для пояснень та приміток до відповідних ділянок коду. Коментарі, особливо у великих за обсягом та складних програмах, дозволяють швидше зрозуміти роботу реалізованого алгоритму та призначення різних його частин. Текст у коментарі ігнорується компілятором і не впливає на функціонування програми.

У C# використовують три види коментарів:

- Рядковий коментар. Він починається двосимвольним маркером "//" і продовжується до кінця рядка. Приклад використання такого коментаря приведено у лістингу 2.1 (рядок 9).
- Розмежований коментар. Він починається двосимвольним маркером "/*" і закінчується двосимвольним маркером "*/". Весь текст між цими маркерами (він може містити багато рядків) вважається коментарем. Розмежований коментар може також виділяти тільки частину одного рядка програми.
- Коментар для документації. Такий коментар починається трисимвольним маркером "///" і містить XML-текст, який використовують для автоматизованого генерування програмної документації засобами Visual Studio.

Приклади всіх згаданих коментарів приведено у лістингу 2.2. Для покращеного сприйняття тексту програми редактор коду Microsoft Visual Studio виділяє тексти коментарів іншим кольором (найчастіше – зеленим).

Лістинг 2.2. Різні види коментарів

```
1  // У рядку 1 приведено зразок однорядкового коментаря
2
3  int i ;    // Так можна закінчувати рядок програми,
4             // пояснюючи призначення коду в ньому
5
6  /* Це приклад розмежованого коментаря,
7     який може займати кілька рядків коду.
8     Коментар будь-якого виду компілятор ігнорує */
9
10 // Далі приклад розмежованого коментаря в частині рядка
11 // Наступний рядок для компілятора буде таким: int j;
12 int /*i,*/ j;
13
14 // Нижче приклад коментаря для документації:
15 /// <summary>
16 /// Цей клас призначений для ...
17 /// <\summary>
18
19 class Department
20 {
21   ...
```

Типи і змінні

Із загального погляду програма на C є сукупністю функцій та типів даних, а програма на C++ – сукупністю функцій та класів. У той же час програму на C# можна розглядати як сукупність оголошень типів. Зокрема, програма з лістингу 2.1 містить оголошення типу (класу) Program з методом Main. Таким чином, еволюція мов від C до C# ілюструє все більшу вагу об'єктно-орієнтованих принципів у програмуванні. Для розуміння програми на C# важливо розуміти, чим є типи і як з ними працювати.

Що таке тип

В загальному тип можна розглядати як *шаблон* для створення структури даних. Він сам по собі не є структурою даних, але задає характеристики об'єктів, створених за цим шаблоном. *Об'єктом*, або *екземпляром*, називають структуру даних, створену за шаблоном типу.

Тип визначають такі елементи:

- Назва типу. Вона використовується для створення об'єктів (екземплярів) типу.
- Формат (устрій) даних, які міститимуть інформацію про об'єкт.
- Призначення, поведінка та обмеження.

Наприклад, тип для позначення цілих чисел має назву int, у пам'яті займає 4 байти, і визначає поведінку цілого числа.

На основі певного типу можна створити багато об'єктів, чи екземплярів цього типу. Кожен елемент даних у програмі на C# є екземпляром типу, який визначений мовою програмування, BCL (чи іншою бібліотекою), або програмістом. При створенні екземпляра типу для нього виділяється потрібний обсяг пам'яті, структурований за шаблоном, що визначає тип.

Елементи даних та функціональні елементи типу

Деякі типи, які називають *простими типами* (int, short, long), зберігають єдину складову даних. Інші типи можуть зберігати багато складників даних. Наприклад, *масив* – це тип, який містить багато складників одного і того ж типу. До окремих його елементів можна отримати доступ через спеціальне число, яке називають індексом.

Але можуть бути і типи, які містять у складі дані різних типів. Такі складники типу називають *членами* типу, або *елементами* типу. На відміну від масиву, кожен елемент типу має свою назву, за якою до нього і звертаються.

Є два види елементів типу: елементи даних та функціональні елементи. Елементи даних зберігають дані, які стосуються екземпляра типу або типу в цілому. Функціональні елементи містять виконавчий код і визначають поведінку об'єктів цього типу.

Тип даних, який може містити іменовані елементи даних та функціональні елементи, називають *класом*.

Попередньо визначені типи

У C# визначено 16 типів, серед них 13 простих і 3 непростих (рис. 2.5). Назви всіх попередньо визначених типів містять тільки малі символи. Прості типи (див. таблицю 2.5) формують такі групи:

- Числові типи:
  - різного розміру знакові та беззнакові цілі типи (byte, sbyte, short, ushort, int, uint, long, ulong);
  - числа з плаваючою комою (float, double);
  - високоточний десятковий тип decimal, який, на відміну від типів з плаваючою комою, представляє десяткову частину числа точно.

- Символьний тип char, що представляє символ у форматі Unicode.
- Логічний (булевий) тип bool, який може приймати одне з двох значень: true (істина) чи false (хибність)[8].

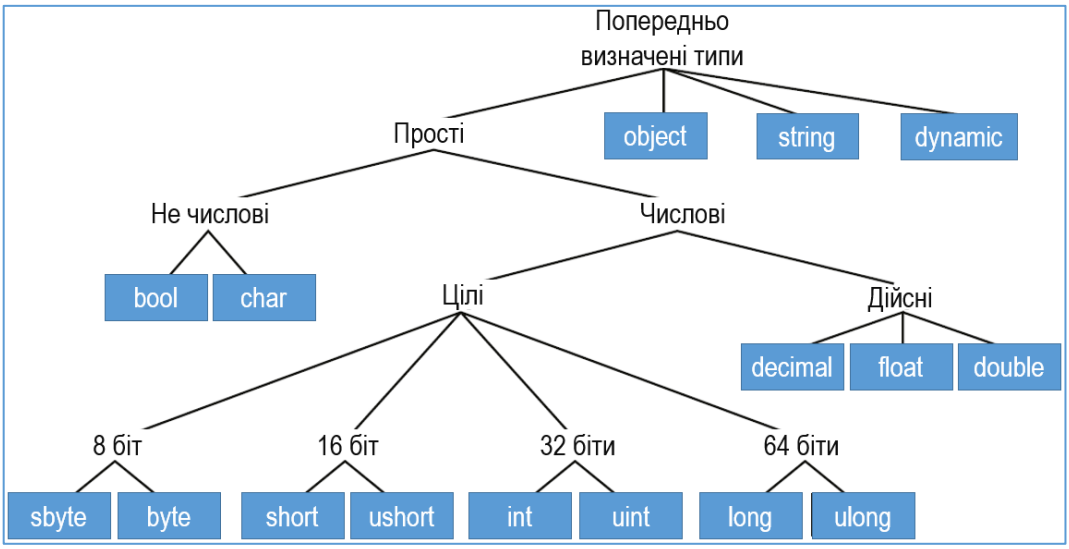


Рис. 2.5. Попередньо визначені типи C#

До непростих типів входять:

- Тип object, який є базовим для всіх інших типів – як попередньо визначених, так і користувацьких.
- Рядковий тип string, що представляє рядок у форматі Unicode.
- Тип dynamic дозволяє пропускати перевірку типу під час компіляції інструкції, в якій він застосовується. Замість цього перевірка виконується під час виконання.

Всі попередньо визначені типи (крім типу dynamic) проектуються безпосередньо на типи .NET Framework. Насправді назви типів у C# є псевдонімами для типів .NET, які визначені у просторі імен System. Тому використовувати типи даних .NET у програмі також можна, але це без потреби заплутує код. В програмі на C# перевагу слід надавати "рідним" типам.

Таким чином, оголошення змінної Count у приведеному коді рівнозначні:

```
using System;
...
int Count;      // Тип даних C#
Int32 Count;    // Той же тип даних, але .NET Framework
```

Але перевагу слід надавати першому способу, у якому використано тип даних C#.

[8] На відміну від C та C++, у C# числове значення не має логічної інтерпретації.

У таблиці 2.5 описано прості попередньо визначені типи та приведено їх відповідники з .NET Framework. Мінімальне та максимальні значення для типів, які представляють дійсні числа, подано із заокругленням. Таблиця 2.6 містить опис інших попередньо визначених типів C#.

Таблиця №2.5. Попередньо визначені прості типи C#

Тип C#	Опис	Діапазон	Тип .NET Framework
sbyte	8-бітове знакове ціле	-128…127	System.SByte
byte	8-бітове беззнакове ціле	0…255	System.Byte
short	16-бітове знакове ціле	-32768…32767	System.Int16
ushort	16-бітове беззнакове ціле	0…65535	System.UInt16
int	32-бітове знакове ціле	-2147483648… 2147483647	System.Int32
uint	32-бітове беззнакове ціле	0…4294967295	System.UInt32
long	64-бітове знакове ціле	-9223372036854775808… 9223372036854775807	System.Int64
ulong	64-бітове беззнакове ціле	0…18446744073709551615	System.UInt64
float	Число з плаваючою комою звичайної точності	-3.402×10^{-38}…3.402×10^{38}	System.Single
double	Число з плаваючою комою подвоєної точності	-1.797×10^{-308}…1.797×10^{308}	System.Double
decimal	Десятковий тип	$\pm7.923\times10^{28}$	System.Decimal
bool	Логічний тип	false, true	System.Boolean
char	Символ Unicode		System.Char

Таблиця №2.6. Попередньо визначені непрості типи C#

Тип C#	Опис	Тип .NET Framework
object	Базовий клас, від якого походять всі типи, включаючи прості типи	System.Object
string	Послідовність символів Unicode	System.String
dynamic	Використовується у збірках, написаних динамічними мовами програмування	Немає відповідника

Типи, які визначає користувач

На основі попередньо визначених типів можна створювати власні типи, призначені для опису структур даних, потрібних за алгоритмом програми. Є такі види типів, які може визначати користувач:

- Класи (class).
- Структури (struct).
- Масиви (array).
- Переліки (enum).

- Делегати (delegate).
- Інтерфейси (interface).

Тип створюють, використовуючи синтаксис оголошення типу. Він містить інформацію про вид типу даних (клас, масив тощо), його назву та оголошення всіх елементів типу (за винятком масивів та делегатів, які не мають іменованих елементів).

Після того, як тип оголошено, можна створювати та оголошувати об'єкти цього типу. Таким чином, користувацькі типи застосовують у програмі так же, як і попередньо визначені, за винятком того, що їх спочатку слід оголосити.

Перелічені користувацькі типи детально розглянемо у наступних розділах.

Розміщення екземплярів у пам'яті

Під час роботи програми її дані мають бути розташовані у пам'яті. Обсяг пам'яті, необхідний для розміщення об'єкта, залежить від конкретного типу та його елементів даних. Застосунки можуть розташовувати свої дані в оперативній пам'яті двох видів: у *стеку (stack)* та в *купі (heap)*. Як правило, програміст не має потреби безпосередньо працювати зі стеком та купою, оскільки розподілом пам'яті керують внутрішні механізми C# та .NET. Замість цього програміст у програмі оперує об'єктами, даючи вказівки створити об'єкт чи видалити його. А виконання цих інструкцій автоматично запускає внутрішній системний механізм для їх реалізації. Проте, знання основ функціонування стеку та купи дозволяє краще зрозуміти, що відбувається під час роботи програми, тому розглянемо їх детальніше.

Стек – це пам'ять, робота якої організована за принципом "останнім зайшов – першим вийшов". Стек можна уявити як коробку, в яку одна на одну складають папки з документами (рис. 2.6). Витягнути папки з неї можна тільки починаючи з кінця, причому одна за одною, в зворотному порядку.

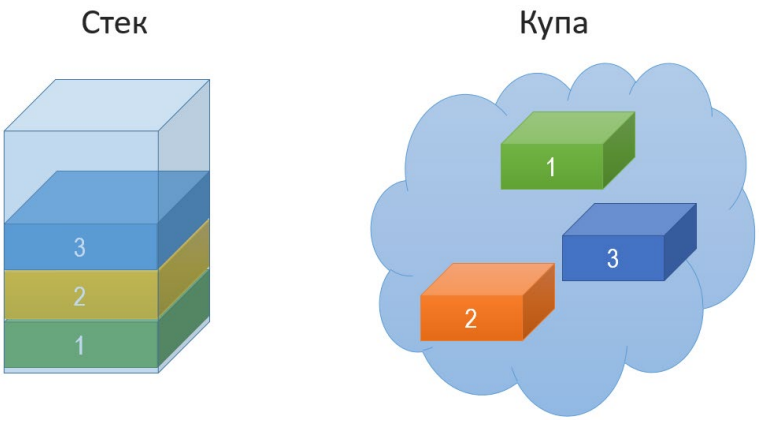


Рис. 2.6. Організація стеку та купи. 1,2,3 – структури даних

У стеку зберігаються значення змінних деяких типів, параметри методів та стан поточного середовища виконання програми. Дані в стек можна додавати тільки на його вершину.

Купа – це пам'ять, ділянки якої використовують для розміщення різних об'єктів. На відміну від стеку, дані можуть зберігатися в купі у будь-якому порядку (рис. 2.6). Порядок звертання до даних також довільний.

Екземпляри користувацьких типів (класів), які створює програма, розміщуються у купі. При цьому для них виділяється потрібна ділянка пам'яті. Але самостійно вивільняти цю пам'ять у програмі на C# не потрібно (на відміну від C та C++, де вивільнення об'єктів мав ретельно відслідковувати програміст для запобігання втрат пам'яті). Цю роботу автоматично виконує компонент .NET Framework, який називають "збирачем сміття" (garbage collector)[9]. Збирач сміття виявляє об'єкти у купі, які код більше не використовує, і знищує їх, вивільняючи використану ними пам'ять.

Значущі і посилальні типи

Тип даних визначає обсяг пам'яті, необхідний для їх зберігання. Тип також визначає, який вид пам'яті використовується для розміщення даних – стек чи купа. Всі типи ділять на дві великі категорії: *типи-значення* (або *значущі типи*) і *типи-посилання* (або *посилальні типи*). Екземпляри цих типів розміщуються у пам'яті по-різному.

Значущі типи потребують тільки однієї ділянки пам'яті, у якій зберігається реальне значення. Вони можуть розміщуватися і в стеку, і в купі. Змінні цього типу безпосередньо представляють значення, яке зберігається у відповідній ділянці пам'яті. Прикладами типів-значень є всі числові типи (`int`, `double` тощо).

Посилальні типи потребують двох ділянок пам'яті: перша містить реальні дані (зазвичай, вона більша) і завжди розташована у купі, а друга є посиланням, яке вказує на ці дані в купі. Посилання може розміщуватися як у стеку, так і в купі. Змінна цього типу представляє не дані, а посилання на них. У той час, як дані можуть займати значний обсяг пам'яті, посилання завжди мале: 32 чи 64 біти, залежно від розрядності системи. Використання цієї моделі розміщення даних забезпечує ефективнішу роботу, так як при обміні даними між різними методами не потрібно копіювати великі обсяги даних об'єкта – достатньо передавати короткі посилання.

Таблиця 2.7 групує типи даних на дві категорії – значущі та посилальні типи, окремо для попередньо визначених та користувацьких типів.

[9] Під "сміттям" мають на увазі вже не потрібні об'єкти у пам'яті.

Таблиця №2.7. Категорії типів даних у C#

	Значущі типи				Посилальні типи
Попередньо визначені типи	sbyte short long decimal	byte ushort ulong	float int char	double uint bool	object string dynamic
Типи, визначені користувачем	struct enum				array class interface delegate

Змінні

В загальному випадку будь яка програма має дані, з якими вона працює. Типи є шаблонами, за якими створюють конкретні екземпляри даних. Ці екземпляри у програмі представлені змінними. *Змінна* – це ім'я, яке під час виконання програми представляє конкретні дані у пам'яті. Кожна змінна має тип, який вказують при її оголошенні.

Якщо змінна належить до одного із значущих типів, вона безпосередньо представляє дані. Змінна посилального типу представляє лише посилання (вказівник) на дані. Ця різниця спричиняє певну відмінність у роботі зі змінними, які належать до різних категорій типів даних.

Існують такі види змінних у C#:

- Локальні змінні. Вони містять тимчасові дані, доступні у певній ділянці програми. Такі змінні не є елементами типу.
- Поля. Вони містять дані, пов'язані з типом або з екземпляром типу, і є елементами типу.
- Параметри. Це тимчасові змінні, які використовують для передачі даних від одного методу до іншого. Такі змінні не є елементами типу.

Перед тим як використовувати змінну, її слід оголосити. При оголошенні змінної їй дають назву і пов'язують з певним типом. Виконання рядка програми, у якому оголошено змінну, приводить до виділення для неї пам'яті.

Найпростіше оголошення змінної потребує задання її назви і типу:

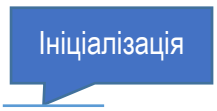

```
int VariableName;
```

Наприклад, у приведеному фрагменті коду оголошено змінні цілого типу (тип int) Population та Count, дійсну змінну Summ (тип double), і змінну town типу Town (цей тип не належить до переліку типів C# чи .NET, тому має бути описаний до оголошення відповідної змінної):

```
int Population;
int Count;
double Summ;
Town town;
```

Припустимо, що Town є класом – посилальним типом. Тоді виконання четвертого рядка цього коду приведе до виділення у пам'яті місця для посилання, а не для даних об'єкта. Посилання поки-що нікуди не вказує, але в подальшому вказуватиме на дані об'єкта (проте для них також потрібно буде виділити пам'ять). У той же час виконання перших трьох рядків спричинить до виділення у пам'яті місць, куди безпосередньо записуватимуться значення змінних Population, Count та Summ (хоча на даному етапі значення змінних ще не задані).

При оголошенні змінної можна відразу присвоїти їй значення. Присвоєння змінній початкового значення називають її *ініціалізацією*. Для ініціалізації змінної під час оголошення дописують знак "=", після якого вказують потрібне значення змінної:




```
int Percent = 100;
```

Виконання цієї інструкції приведе до виділення ділянки пам'яті для змінної Percent цілого типу та запису у неї значення 100.

Локальна змінна не може бути використана у програмі, доки їй не буде присвоєно якесь значення. Спроба використання змінної, якій не було присвоєно значення, приведе до помилки компіляції "Use of unassigned local variable…'".

Змінні деяких типів при оголошенні ініціалізуються автоматично. До них належать поля класів, структур та елементи масивів. Локальні змінні та параметри методів автоматично не ініціалізуються. Автоматична ініціалізація полягає у тому, що відразу після створення змінної в неї записується значення за замовчуванням. Всі числові типи ініціалізуються значенням 0, тип char – символом з кодом U+0000, тип bool значенням false. Непрості типи ініціалізуються значенням null (його розглянемо далі).

В одній інструкції можна оголошувати кілька змінних одного і того ж типу, і відразу їх вибірково ініціалізувати. При цьому опис окремих змінних розділяють комами:

```
int var1 = 10, var2, var3 = 30;
double var4, var5 = 55.55;
```

Тут оголошено три змінних цілого типу (з них перша і остання ініціалізовані), і дві змінних дійсного типу (остання з них ініціалізована).

Оголошувати в одній інструкції змінні різних типів не можна.


```
int Count, float Summ;  // Помилка! Так не можна
```

Назва змінної безпосередньо представляє значення, яке у ній зберігається. Щоб використати значення змінної у програмі, використовують її назву. Наприклад, виконання наступного коду приведе до виводу на екран числа 10 – значення змінної var1:

```
int var1 = 10;
Console.WriteLine("var1 = {0}", var1);
```

Статична і динамічна типизація

Нормальним для C# (і для багатьох інших мов програмування) є випадок, коли при оголошенні змінної вказують її тип. Він дозволяє компілятору розрахувати необхідний для збереження даних обсяг пам'яті і спланувати дії по її виділенню (у стеку чи в купі). Тип змінної не можна міняти під час виконання програми. Такий підхід називають *статичною типизацією*.

Але не всі мови підтримують статичну типизацію. Багато скриптових мов, (зокрема, IronPython та IronRuby) підтримують динамічну типизацію. При цьому тип змінної визначається тільки під час виконання. Для підтримки взаємодії програми на C# з бібліотеками, написаними на мовах з динамічною типизацією, використовують спеціальний тип даних dynamic, який забезпечує роботу зі змінною динамічного типу.

Під час компіляції компілятор не перевіряє тип змінної dynamic. Він аналізує всю інформацію про використання такої змінної і розміщує її разом зі змінною. Під час виконання ця інформація порівнюється з умовами певного типу, до якого ця змінна приводиться. Якщо ці умови не виконуються, генерується виняткова ситуація (помилка).

Ось приклад використання змінної динамічного типу:

```
dynamic var1 = 10;
var1 = var1 * 2.2;
Console.WriteLine("var1 = {0}", var1);
```

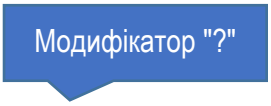
Значення null та типи, які його підтримують

При роботі з інформацією, яка представляє реальні дані, часто виникає необхідність показати, що змінна не містить коректних даних. Особливо часто така потреба виникає під час роботи з базами даних. Для посилальних типів з цією метою використовують спеціальне значення null. Це значення не тотожне значенню 0 (нуль). Воно означає відсутність даних.

Типам значень не можна присвоювати значення null. Але C# підтримує модифікатор "?", який використовують для оголошення змінної, що може приймати значення null.

Цей модифікатор допустимий при оголошенні змінних всіх простих попередньо визначених типів. Змінна такого типу з модифікатором "?" може приймати будь-яке значення з діапазону базового типу, і крім цього – значення null.

Наступний код демонструє використання змінної цілого типу, яка може містити значення null:




```
int? i = null;
i = 10;
Console.WriteLine("i = {0}", i);
```

При оголошенні змінної з модифікатором типу "?" насправді створюється змінна зовсім іншого типу – екземпляр структури System.Nullable<>, яка має свої властивості та методи.

Оголошення using

Цей спосіб оголошення змінних було додано починаючи з C# 8.0. Якщо при оголошенні змінної на початку додати ключове слово using, то компілятор вивільнить ресурси, зайняті змінною, при виході з програмного блоку, де її оголошено. Це зручно використовувати, наприклад, при роботі з файлами, які потрібно закривати по закінченню сеансу роботи з ними:

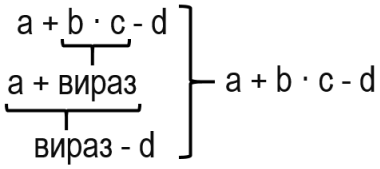
```
using var file = new System.IO.StreamWriter("WriteLines2.txt");
```

Оператори та вирази

Оператор – це символ (або група символів), які представляють операцію, що повертає єдиний результат. *Операнд* – це елемент вхідних даних для операції. Оператор виконує наступне:

- Отримує операнди на вході.
- Виконує дію.
- Повертає значення результату.

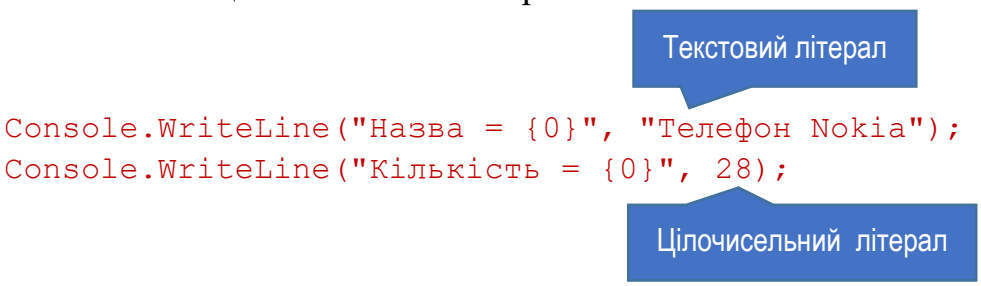
Вираз – це рядок, який містить оператори та операнди і повертає результат певного типу. В ролі операндів можуть виступати літерали, константи, змінні, виклики методів, або інші вирази. Вирази за допомогою операторів можна поєднувати у складніші вирази:



Обчислення виразу полягає у застосуванню кожного оператора до його операндів у правильній послідовності, в результаті чого отримується результат. Він передається у те місце програми, де у коді розташовано вираз, і може бути використаний як операнд для іншого виразу.

Літерали

Літерал – це значення (числове, текстове чи інше), яке вписане безпосередньо у код програми. Наприклад, приведений код використовує текстовий та цілочисельний літерали:



Деякі типи даних мають свої власні літерали. Наприклад, тип bool має два літерали: true і false. Для змінних посилального типу використовують літерал null, який представляє значення null.

При позначенні літералів цілого типу можна вказувати суфікс, який задасть точний тип даних (таблиця 2.8).

Таблиця №2.8. Суфікси для літералів цілих типів

Суфікс	Тип	Зауваження
Без суфікса	int, uint, long, ulong	
U, u	uint, ulong	
L, l	long, ulong	Використовувати маленьку літеру "l" не рекомендовано, бо її можна сплутати з цифрою "1"
ul, uL, Ul, UL, lu, lU, Lu, LU	ulong	

Приклади цілих літералів із вказуванням типу:

```
123     // int
123L    // long
123U    // unsigned int
123UL   // unsigned long
```

Суфікси для літералів, які представляють дійсні числа, приведено у таблиці 2.9.

Таблиця №2.9. Суфікси для літералів дійсних типів

Суфікс	Тип
Без суфікса	double
F, f	float
D, d	double
M, m	decimal

Символьні літерали містять символ в одинарних лапках. Вони можуть представляти: окремий символ, просту ESC-послідовність[10], шістнадцяткову ESC-послідовність, або код символу Unicode. Найвживаніші символьні літерали приведено в таблиці 2.10.

Таблиця №2.10. Символьні літерали

Літерал	Опис
\0	Null
\a	"Обережно" (звуковий сигнал)
\b	Backspace (стирає попередній символ)
\t	Горизонтальна табуляція
\n	Новий рядок
\r	Перехід на початок рядка
\"	Подвійна лапка
\'	Одинарна лапка
****	Символ "\"

Рядкові літерали є двох типів: звичайні та дослівні. Такі літерали оточують подвійними лапками. Вони можуть містити символи, символьні літерали (див. табл. 2.10) та ESC-послідовності. Перед дослівними літералами ставлять символ "@". У них символьні літерали та ESC-послідовності не обчислюються. Приклад використання рядкових літералів:

```
Console.WriteLine("Add \t \'Interest\'");
Console.WriteLine(@"Add \t \'Interest\'");
```

В результаті виконання цих інструкцій отримаємо:

```
Add      'Interest'
Add \t \'Interest\'
```

У першому рядку використано табуляцію (\t), в другому відповідний літерал відображується, ніби він є складовою частиною тексту.

[10] ESC-послідовність – це керуюча послідовність символів, яка використовується для керування виводом тексту на екран чи принтер.

Пріоритети операторів

Якщо вираз містить кілька операторів, то порядок їх виконання визначає пріоритет категорії, до якої належить оператор (таблиця 2.11).

Таблиця №2.11. Пріоритети операторів: від найвищого до найнижчого

Категорії операторів	Оператори
Первинні	a.x, f(x), a[x], x++, x--, new, typeof, checked, unchecked
Унарні	+, -, !, ~, ++x, --x, (T)x
Мультиплікативні	*, /, %
Адитивні	+, -
Зсув	<<, >>
Відношення і тип	<, >, <=, >=, is, as
Рівність	==, !=
Логічне "І"	&
Логічне "Виключаюче АБО"	^
Логічне "АБО"	|
Умовне "І"	&&
Умовне "АБО"	||
Умова	? :
Присвоєння	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

При побудові виразу, якщо слід змінити пріоритетність виконання різних його операторів, використовують круглі дужки. Тоді першими обчислюються вирази в дужках, починаючи з найглибшого рівня вкладеності.

Прості арифметичні оператори

Для базових арифметичних операцій використовують оператори згідно таблиці 2.12. Їх використання відповідає загальновідомим правилам математики.

Оператор % виконує ділення операндів, але повертає не результат ділення, а залишок. Наприклад, виконання рядка програми

```
Console.WriteLine( 11 % 3);
```

приведе до виводу на екран числа 2 – залишку від ділення 11 на 3.

Таблиця №2.12. Оператори простих арифметичних операцій

Оператор	Опис
+	Додавання
-	Віднімання
*****	Множення
/	Ділення
%	Залишок при діленні

Оператори для порівняння

Оператори для порівняння порівнюють операнди і повертають результат логічного типу (bool). Їх перелічено та описано у таблиці 2.13.

Таблиця №2.13. Оператори для порівняння

Оператор	Назва	Опис
<	Менше	Повертає true, якщо перший операнд менший за другий, і false в протилежному випадку
>	Більше	Повертає true, якщо перший операнд більший за другий, і false в протилежному випадку
<=	Менше або дорівнює	Повертає true, якщо перший операнд менший або дорівнює другому, і false в протилежному випадку
>=	Більше або дорівнює	Повертає true, якщо перший операнд більший або дорівнює другому, і false в протилежному випадку
==	Дорівнює	Повертає true, якщо перший операнд дорівнює другому, і false в протилежному випадку
!=	Не дорівнює	Повертає true, якщо перший операнд не дорівнює другому, і false в протилежному випадку

Оператори інкременту та декременту

Оператори для інкременту (приросту на 1) та декременту (зменшення на 1) приведено у таблиці 2.14. Вони мають дві форми: префіксну та постфіксну. У префіксній формі оператор розміщують перед операндом, а в постфіксній – після операнда. Значення операнда після застосування обох форм оператора одинакове. Різниця тільки в результаті, який оператор повертає у вираз, в якому він використаний.

Таблиця №2.14. Оператори для інкременту та декременту

Оператор	Назва	Опис
++	Префіксний інкремент: ++var	Збільшує значення змінної на 1 і записує його у цю змінну. Повертає нове значення змінної.
	Постфіксний інкремент: var++	Збільшує значення змінної на 1 і записує його у цю змінну. Повертає попереднє значення змінної, до інкременту.
--	Префіксний декремент: --var	Зменшує значення змінної на 1 і записує його у цю змінну. Повертає нове значення змінної.
	Постфіксний декремент: var--	Зменшує значення змінної на 1 і записує його у цю змінну. Повертає попереднє значення змінної, до декременту.

Наступний приклад демонструє використання обох форм операторів інкременту та декременту (результат його роботи приведено на рис. 2.7):

```
int x = 1, y;
Console.WriteLine("Початкове значення x: {0}", x);
y = x++;
Console.WriteLine("y = x++: x: {0}, y: {1}", x, y);
x = 1;
y = ++x;
Console.WriteLine("y = ++x: x: {0}, y: {1}", x, y);
x = 1;
y = x--;
Console.WriteLine("y = x--: x: {0}, y: {1}", x, y);
x = 1;
y = --x;
Console.WriteLine("y = --x: x: {0}, y: {1}", x, y);
```

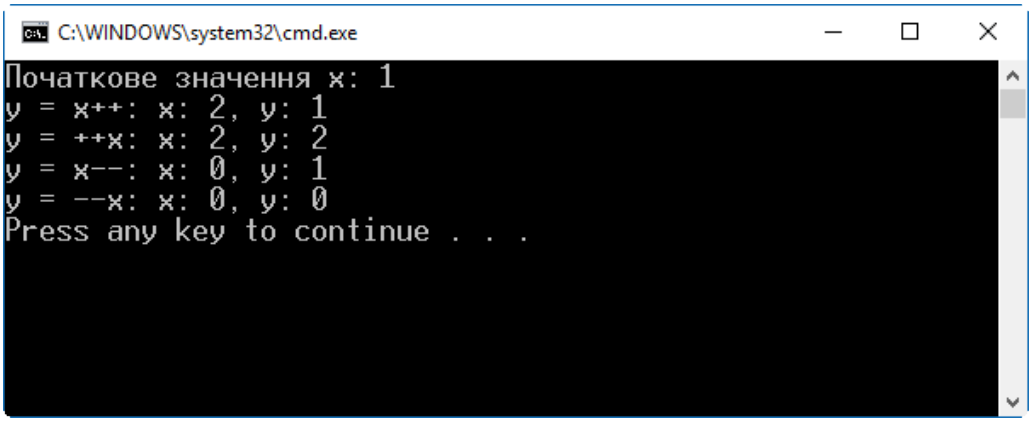


Рис. 2.7. Демонстрація використання обох форм операторів ++ та --

Логічні оператори для умов

Логічні операції "І", "АБО", "НЕ" використовують для поєднання умов при порівнянні. Перелік логічних операцій для умов приведено в таблиці 2.15.

Таблиця №2.15. Логічні оператори для поєднання умов

Оператор	Назва	Опис
&&	Логічне "І"	Повертає true, якщо обидва операнди дорівнюють true; повертає false в протилежному випадку
||	Логічне "АБО"	Повертає true, якщо хоча б один операнд дорівнює true; повертає false в протилежному випадку
!	Логічне "НЕ"	Повертає true, якщо операнд дорівнює false; повертає false, якщо операнд дорівнює true

Наприклад, вираз $((x>3)\&\&(y<10))$ поверне значення true (істина), якщо значення x більше 3 і одночасно (так як використано логічне "І") значення y менше 10. В іншому випадку результатом виразу буде false. Таким чином, щоб оператор && повернув true, потрібне виконання умов в обох його операндах.

Вираз $((x>3)||(y<10))$ поверне значення true (істина), якщо значення x більше 3 або (так як використано логічне "АБО") значення y менше 10. Щоб оператор || повернув true, потрібне виконання тільки однієї з умов його операндів.

Вираз $(!(x>3)||(y<10))$ поверне значення true (істина), якщо значення x не більше 3 (використано логічне "НЕ"), або (використано логічне "АБО") значення y менше 10.

За допомогою поєднання логічних операторів у програмі можна формувати умови довільної складності.

Наступний фрагмент коду демонструє використання логічних операторів для формування умови з використанням інструкції if (детально її розглянемо далі):

```
double x, y;
y = x = 0;
if (((x > 10) && (y <= 20)) || (x == y))
    Console.WriteLine("Ця умова виконується!");
```

Побітові оператори

Інколи при роботі з даними є необхідність побітової їх обробки. З цією метою використовують логічні побітові оператори та оператори побітового зсуву (таблиця 2.16).

Таблиця №2.16. Побітові оператори

Оператор	Назва
&	Побітове "І"
|	Побітове "АБО"
^	Побітове "Виключаюче АБО"
~	Побітове "НЕ"
<<	Зсув бітів вліво
>>	Зсув бітів вправо

Оператори присвоєння

Оператори присвоєння обчислюють значення виразу справа від оператора і присвоюють це значення змінній зліва від оператора. Ці оператори перелічено у таблиці 2.17.

Таблиця №2.17. Оператори присвоєння

Оператор	Опис
=	Базовий простий оператор присвоєння. Результат виразу справа від оператора присвоюється змінній зліва
Складні оператори присвоєння	
*=	y *= x еквівалентно до y = y * x
/=	y /= x еквівалентно до y = y / x
%=	y %= x еквівалентно до y = y % x
+=	y += x еквівалентно до y = y + x
-=	y -= x еквівалентно до y = y - x
<<=	y <<= x еквівалентно до y = y << x
>>=	y >>= x еквівалентно до y = y >> x
&=	y &= x еквівалентно до y = y & x
^=	y ^= x еквівалентно до y = y ^ x
|=	y |= x еквівалентно до y = y | x

Для складних операторів присвоєння на місці x (див. таблицю 2.17) може бути інший вираз. Наприклад:

Зауважте: дужки

```
x *= y - z;      // Еквівалентно до x = x * (y - z)
x /= y + z;      // Еквівалентно до x = x / (y + z)
```

Операція присвоєння є виразом, тому повертає результат в те місце програми, де вона записана. Тому результат операції присвоєння також може бути частиною виразу:

```
z = y = x = 5;
```

Тут спочатку опрацьовується вираз (x=5), в результаті чого змінній x присвоюється значення 5. Далі результат 5 присвоюється змінній y, і в кінці – змінній z.

Умовний оператор

Умовний оператор ? : надає зручний та швидкий спосіб вибору одного із двох значень, залежно від результату певної умови. Синтаксис використання цього оператора такий:

```
Condition ? Expression1 : Expression2
```

Вираз для умови (Condition) повинен повертати результат логічного типу (bool). Якщо вираз умови дорівнює true, то умовний оператор повертає значення виразу Expression1, а в протилежному випадку – значення виразу Expression2.

Приклад використання цього оператора:

```
int i  = x > y ? 10 : 20;
```

В результаті виконання цього оператора, якщо значення x більше від y, то змінній і буде присвоєно значення 10. Коли x не більше за y (тобто x менше або дорівнює y), то змінній і буде присвоєно значення 20.

Інструкції та блоки інструкцій

Інструкція (statement) – це ділянка коду, що описує дію, яку має виконати програма. Інструкції бувають прості (оголошення, розрахунок, присвоєння, виклик методу тощо) та складні (умова, цикл тощо). У кінці простої інструкції завжди є крапка з комою.

Розглянемо фрагмент коду:

```
int i  = 10;
i = i * 2;
```

Він складається із двох простих інструкцій. У першій оголошено змінну цілого типу і та ініціалізовано її значенням 10. У другій поточне значення змінної і множиться на 2, а результат присвоюється тій же змінній і.

Загалом інструкції можна поділити на такі категорії:

- Інструкції оголошення. Описують оголошення типів або змінних.
- Вкладені інструкції. Виконують якісь дії або керують потоком виконання програми.
- Помічені інструкції. Керують переходами всередині потоку виконання.

Блок інструкцій – це послідовність інструкцій, оточена парою відповідних фігурних дужок (відкриваючою на початку і закриваючою в кінці блоку). В ряді випадків блок може бути порожнім (не містити інструкцій). Синтаксично блок обчислюється як єдина вкладена інструкція.

На основі попереднього фрагменту коду можна створити такий блок:

```
{
    int i  = 10;
    i = i * 2;
}
```

Блок можна використовувати всюди, де синтаксис мови передбачає інструкцію, але дія, яку має виконати програма, містить кілька інструкцій. На відміну від простих інструкцій, блок не завершують символом крапки з комою.

Деякі програмні конструкції потребують блоку (наприклад, метод). У цьому випадку, замість блоку не можна підставити окрему інструкцію.

```
{
    int i  = 10;
    i = i * 2;
}
```

Кінець простої інструкції містить ";"

Кінець блоку без символа ";"

Програма може містити також *порожні інструкції*, які містять тільки крапку з комою. Такі інструкції можна використовувати в коді у тих місцях, де синтаксис мови передбачає вбудовану інструкцію, але логіка алгоритму не потребує ніяких дій. Наприклад:

```
if( x < y )          // Перевіряємо умову
{                    // Якщо умова виконується:
    ;                // Порожня інструкція: нічого не робимо
}
else
{                    // Якщо умова не виконується:
    c = a + b;       // Проста інструкція: обчислюємо вираз
}
```

Інструкція може містити вираз, який розраховує певне значення. У програмі це значення вкладається у те місце, де розташований вираз. Тому говорять, що вираз *повертає* (у програму) розраховане значення певного типу. Останній рядок попереднього фрагменту коду обчислює суму значень змінних a та b, повертає результат (суму) і записує його у ділянку пам'яті, представлену змінною c.

Керування потоком виконання

Сукупність інструкцій, які виконуються під час роботи програми, називають її *потоком виконання*. Зазвичай інструкції виконуються лінійно, послідовно одна за другою. Але виключно лінійне виконання інструкцій не може забезпечити потреб програмування. Потрібні засоби, які при потребі дозволять перенаправити потік виконання в іншу ділянку коду. Інструкції, які дозволяють це робити, називають інструкціями для керування потоком виконання.

Складні інструкції, які складаються з інших інструкцій, називають ще програмними *конструкціями*. C# забезпечує кілька конструкцій для керування потоком виконання, характерні і для інших сучасних мов програмування. До них належать:

- Конструкції умовного виконання. Вони виконують або пропускають секцію коду залежно від заданої умови. Є такі інструкції умовного виконання:
  - if
  - if … else
  - switch
- Циклічні конструкції. Вони забезпечують багатократне виконання однієї і тієї ж ділянки коду. До них належать інструкції:

- while
- do
- for
- foreach

- Інструкції переходу. Вони дозволяють перейти від однієї секції коду до іншої. Це виконують такі інструкції:
  - break
  - continue
  - return
  - goto
  - throw

Конструкції для умовного виконання та циклів містять вираз з умовою, яким задають, де саме продовжуватиметься виконання програми. Цей вираз повинен повертати результат типу bool. Вираз для умови може бути складний. Його можна формувати, поєднуючи прості умови за допомогою логічних операторів &&, || та !.

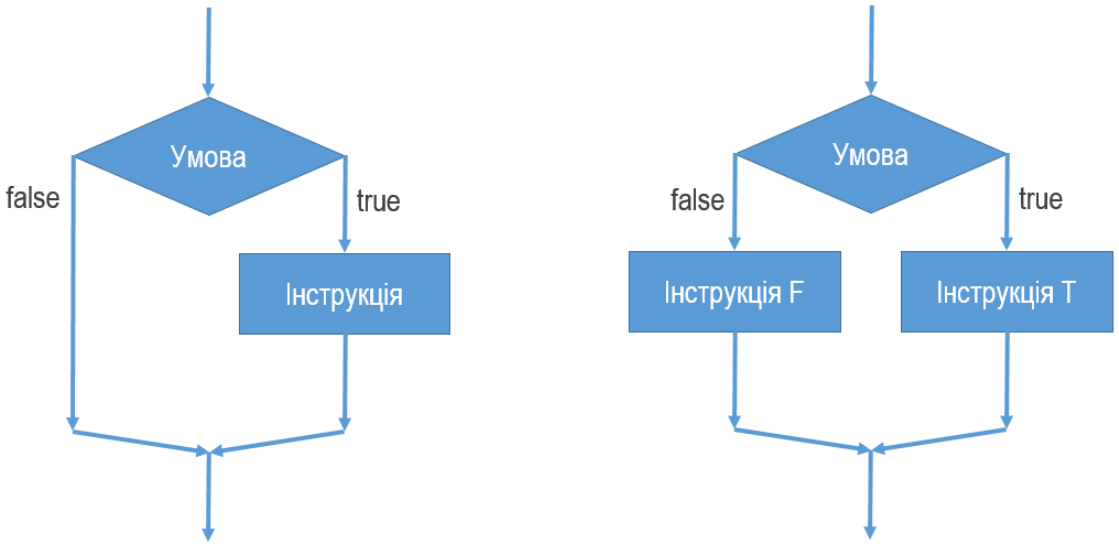
Інструкція if

Інструкція if реалізує виконання за умовою (рис. 2.8, а). Синтаксис інструкції такий:

```
if (TestExpression)
    Statement
```

Вираз TestExpression задає умову, від якої залежить перемикання на одну з двох віток потоку виконання. Вираз повинен повертати результат типу bool. Якщо TestExpression дорівнює true, то виконується інструкція Statement. Якщо умова обчислюється як false, то інструкція Statement пропускається. На місці інструкції Statement можна розмістити блок інструкцій, якщо логіка алгоритму передбачає більше однієї операції.

Лістинг 2.3. містить кілька прикладів використання інструкції if. Перший фрагмент коду (рядки 4…5) стосується випадку, коли при виконанні умови потрібно виконати тільки одну інструкцію. Якщо задовольняється умова (x<=10), то виконується інструкція в рядку 5, а після неї – наступні інструкції. Коли умова не виконується, інструкція в рядку 5 пропускається, і виконання продовжується з першої після рядка 5 інструкції (у даному випадку – 10).



а б

Рис. 2.8. Схема виконання інструкцій if (а) та if...else (б)

Другий фрагмент (рядки 10…14) ілюструє ситуацію, коли при виконанні умови, заданої в конструкції if, потрібно виконати кілька інструкцій. Якщо задовольняється умова (x>100), то виконується блок інструкцій (рядки 11…14); в іншому випадку блок пропускається, і виконання продовжується з першої після закінчення блоку інструкції.

Лістинг 2.3. Приклади використання інструкції if

```
1  // Приклад з однією інструкцією. Якщо при виконанні умови
2  // потрібно виконати тільки одну інструкцію, її не оточують
3  // фігурними дужками
4  if (x <= 10)
5      y = x + 50;       // Єдина інструкція
6
7  // Приклад з кількома інструкціями. Якщо при виконанні умови
8  // слід виконати більше однієї інструкції, то формують блок,
9  // оточуючи його фігурними дужками
10 if (x > 100)
11 {
12     x = x - 5;        // Кілька інструкцій оточують дужками,
13     y = x + z;        // формуючи з них блок інструкцій
14 }
```

Інструкція if...else

Інструкція if...else розділяє потік виконання на дві вітки, і перемикання на одну з них керується виконанням умови (рис. 2.8, б). Синтаксис конструкції такий:

```
if (TestExpression)
    StatementT
else
    StatementF
```

Якщо задовольняється умова, задана виразом TestExpression (тобто TestExpression повертає true), то виконується інструкція StatementT. В іншому випадку виконується інструкція StatementF. Якщо алгоритм передбачає виконання більше однієї інструкції при виконанні (не виконанні) умови, то замість одиничної інструкції використовують блок інструкцій.

Приклади використання інструкції if…else приведено в лістингу 2.4.

Конструкція на основі if…else розділяє потік виконання на дві вітки. Якщо потрібно створити більше віток, можна використати вкладені інструкції (if…else) (рядки 11…16).

Лістинг 2.4. Приклади використання інструкції if…else

```
1  // Конструкція if...else
2  if (x < 10 )
3      y = x - 1;       // Єдина інструкція
4  else
5  {                    // Блок інструкцій
6      x = x - 5;
7      y = x + z;
8  }
9
10 // Вкладені конструкції if...else
11 if (TestExpression1)
12     Statement1
13 else if (TestExpression2)
14     Statement2
15 else
16     Statement3
```

Інструкція switch

Інструкція switch забезпечує множинне розгалуження потоку виконання (рис. 2.9). Вона містить різні секції коду, кожна з яких починається відповідною міткою з ключовим словом case. Будь-яка секція коду повинна закінчуватися інструкцією break, або іншою інструкцією переходу (return, continue, goto, чи throw). Зі всіх інструкцій переходу найчастіше використовують break. Вона припиняє виконання секції і переходить до кінця конструкції switch.

Кожна мітка у конструкції switch має таку структуру:

Конструкція switch виконується так:

Спочатку обчислюється вираз TestExpression, який вказано в інструкції switch.

Якщо значення виразу TestExpression дорівнює значенню константи ConstantX однієї з міток, то виконуються інструкції з секції, відміченої цією міткою. Коли ні одна з міток не містить константи, яка дорівнює значенню виразу TestExpression, виконуються інструкції з секції default.

При необхідності секцію default можна не включати у конструкцію switch. Але, якщо вона є, то має закінчуватися, як і інші секції, інструкцією break, або іншою інструкцією переходу (return, continue, goto, чи throw)[11].

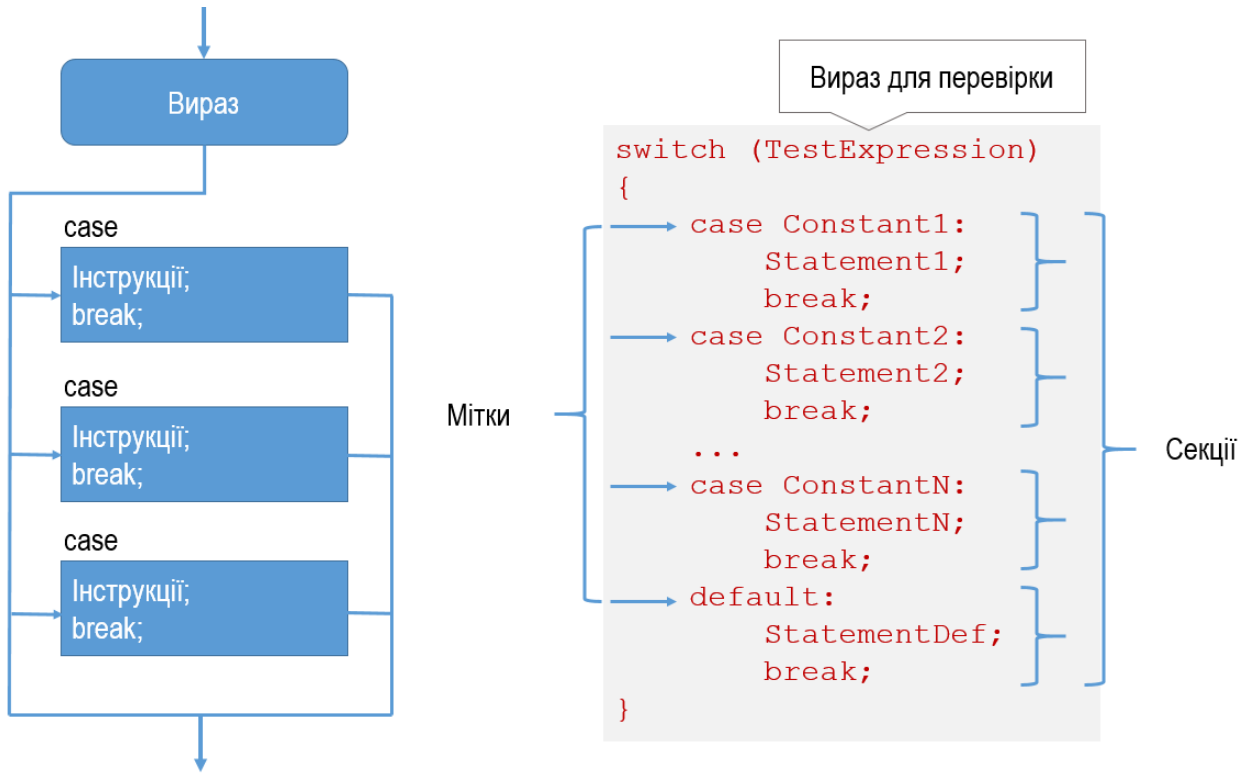


Рис. 2.9. Структура інструкції switch

Цикл while

Конструкція while формує цикл, у якому умова, що визначає його виконання (чи невиконання), перевіряється на початку циклу (рис. 2.10, а).

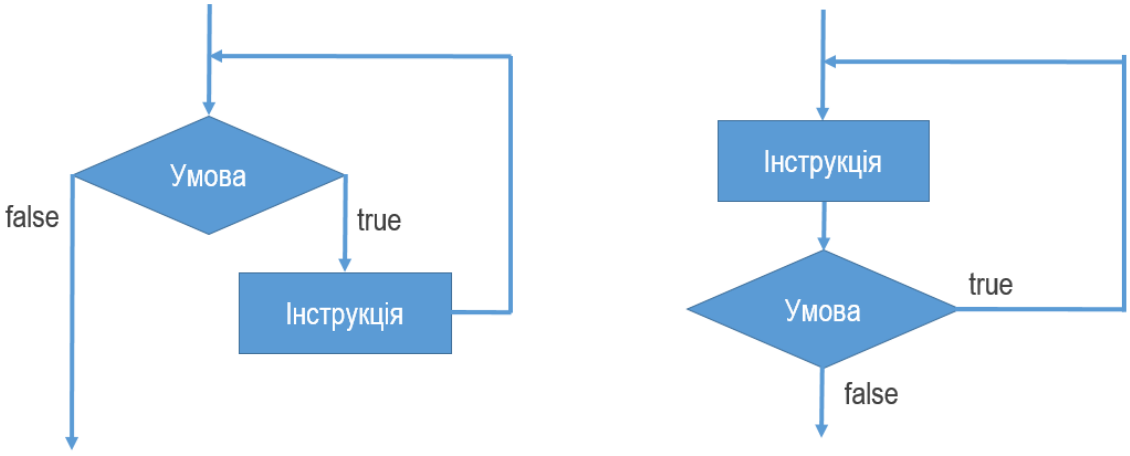
[11] Детально про інструкції переходу йтиме мова далі в цьому розділі.

Циклічну конструкцію, в якій умову виконання циклу перевіряють на початку циклу, називають *циклом з передумовою*.

Синтаксис використання інструкції while такий:

```
while (TestExpression)
    Statement
```

Вираз для перевірки умови виконання циклу TestExpression повинен повертати результат типу bool. При досягненні у потоці виконання інструкції while спочатку обчислюється значення виразу TestExpression. Якщо результат TestExpression дорівнює true, виконується інструкція Statement, що формує тіло циклу. Після цього знову розраховується значення виразу TestExpression, і все повторюється спочатку. Якщо вираз TestExpression повертає значення false, виконання циклу припиняється, і виконання програми переміщується до першої інструкції після конструкції циклу.



а б

Рис. 2.10. Структура циклів while (а) та do (б)

Коли в циклі слід виконувати більше однієї інструкції (як правило, так і є), на місці інструкції Statement слід сформувати блок інструкцій.

Зауважимо, що коли при першому входженні у цикл while умова циклу не задовольняється, то тіло циклу не буде виконане жодного разу.

Можлива і протилежна ситуація: якщо умова циклу виконується завжди, то потік виконання, зайшовши у цикл, ніколи з нього не вийде, постійно виконуючи його інструкції. Такої ситуації слід уникати, оскільки вона приведе до "зависання" програми. Тому при проектуванні циклічних конструкцій слід ретельно продумати умови припинення роботи циклу.

У лістингу 2.5 приведено приклад використання циклу while. В рядку 1 оголошено дійсні змінні x, xMax та dX, які відразу ініціалізовано певними значеннями. Умова виконання циклу (x < xMax) забезпечує постійне циклічне виконання тіла циклу (блок у рядках 4…7), доки значення змінної x менше за

значення xMax. У тілі циклу спочатку виводимо на екран поточне значення x (рядок 5), потім прирощуємо x на значення dX (рядок 6)[12]. Результат виконання програми з лістингу 2.5 приведено на рис. 2.11.

Лістинг 2.5. Приклад використання циклу while

```
1  double x = 0, xMax = 5.5, dX = 0.99;
2
3  while (x < xMax)
4  {
5      Console.WriteLine("x = {0}", x);
6      x += dX;
7  }
```

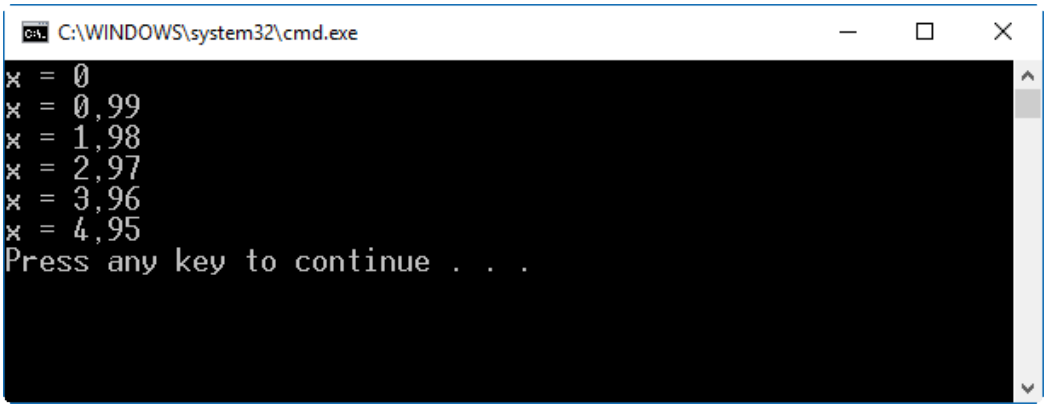


Рис. 2.11. Результат роботи циклу while з лістингу 2.5

Цикл do

Цикл do, на відміну від while, перевіряє умову виконання циклу в кінці (рис. 2.10, б). Циклічну конструкцію, у якій умову перевіряють в кінці циклу, називають *циклом з післяумовою*. Синтаксис використання циклу do такий:

```
do
    Statement
while (TestExpression);        // Кінець циклу
```

При відпрацюванні такого циклу спочатку виконується інструкція (або інструкції) Statement циклу, а потім обчислюється умова циклу – вираз TestExpressison. Якщо вираз TestExpressison повертає значення true, інструкція Statement виконується ще раз. Цикл повторюється, доки TestExpressison дорівнює true. Коли вираз із умови циклу поверне результат false, виконання інструкцій Statement припиняється, і виконуються інструкції, розташовані після циклу.

[12] Оператор += описано у таблиці 2.17.

Особливістю циклу do є те, що тіло циклу виконується хоча б один раз в будь-якому випадку, навіть якщо до входження у конструкцію циклу умова циклу не задовольняється.

Приклад використання циклу do приведено в лістингу 2.6, а результат його виконання – на рис. 2.12.

Лістинг 2.6. Приклад використання циклу do

```
1  double x = 1, xMax = 5.5, dX = 1.5;
2  do
3  {
4      Console.WriteLine("x = {0}", x);
5      x += dX;
6  } while (x <= xMax);       // <-- Крапка з комою потрібна
```

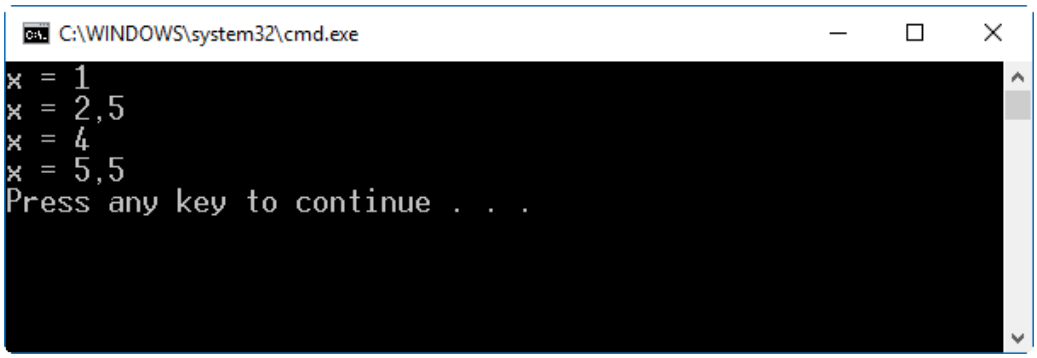


Рис. 2.12. Результат виконання циклу do з лістингу 2.6

Цикл for

Конструкція на основі інструкції for виконує цикл з передумовою, але додатково містить вирази для ініціалізації та ітерації. Схему роботи такого циклу зображено на рис. 2.13. Його синтаксис:

Вирази розділені символом " ; "

```
for (Initializer ; TestExpression ; IterationExpression)
    Statement
```

Перед виконанням циклу відбувається ініціалізація циклу – один раз виконується вираз Initializer. Далі обчислюється умова циклу TestExpression. Якщо вираз TestExpression повернув true, виконується інструкція у тілі циклу. Після цього виконується ітераційний вираз IterationExpression. Далі виконання знову переходить до початку циклу, і повторно обчислюється умова TestExpression. Доки вираз умови циклу повертає true, виконується інструкція (інструкції) у тілі циклу, після чого завжди виконується ітераційний вираз. Виконання циклу припиняється, коли TestExpression повертає false.

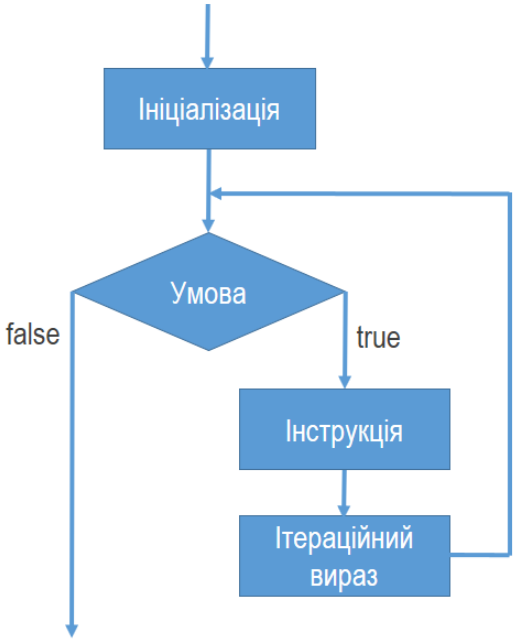


Рис. 2.13. Схема циклу for

Вирази Initializer, TestExpression та IterationExpression необов'язкові. Їх позиції можна залишити порожніми, але розділювачі (символи ";") мають бути обов'язково. Якщо вираз TestExpression не вказати, то вважатиметься, що тестовий вираз завжди повертає true, – тоді слід передбачити інший спосіб припинення циклу.

Приклад використання циклу for приведено у лістингу 2.7, а результат його виконання – на рис. 2.14.

Лістинг 2.7. Приклад використання циклу for

```
1  for (int i = 1; i <= 5; i++)
2  {
3      Console.WriteLine("i = {0}", i);
4  }
```

У цьому коді на початку циклу змінна i ініціалізується значенням 1. Цикл виконується, доки значення i <= 5. В кінці кожного проходження тіла циклу за допомогою ітераційного виразу (i++) значення змінної циклу i збільшується на 1.

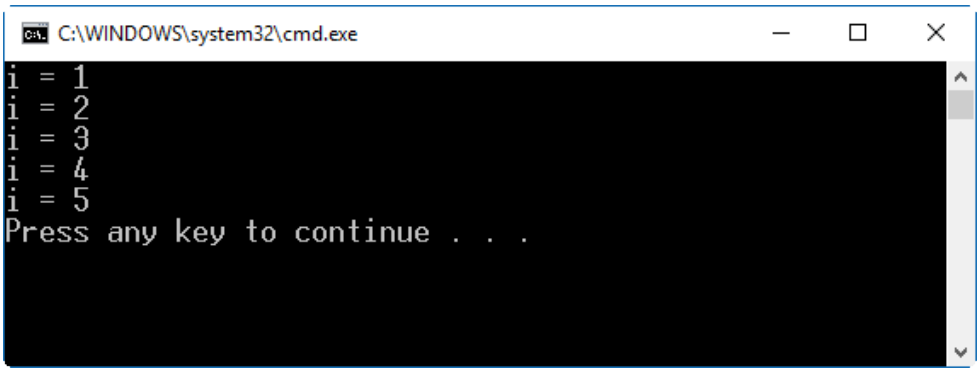


Рис. 2.14. Результат виконання циклу for з лістингу 2.7

Інструкції переходу

Коли потік виконання програми досягає інструкції переходу, виконання програми безумовно (без перевірки будь-якої умови) переноситься до іншої ділянки програми.

У C# є такі інструкції переходу:

- break;
- continue;
- return;
- goto;
- throw;

Інструкція break

Інструкцію break можна використовувати у конструкціях while, do, for, foreach та switch для переривання виконання конструкції. Якщо програма містить вкладені конструкції, то break припиняє виконання найглибшої вкладеної конструкції. Таким чином, якщо є два цикли, один вкладений в інший, і break викликається у тілі вкладеного циклу, то відбувається вихід із вкладеного циклу, але не із зовнішнього.

У лістингу 2.8 приведено приклад використання break для припинення виконання нескінченного циклу.

Лістинг 2.8. Використання інструкції break для переривання циклу

```
1  int x = 0;
2  while (true)
3  {
4      x++;
5      if (x == 3) break;
6  }
```

Цикл (рядки 2…6) містить умову, яка завжди повертає значення true (вона задана в рядку 2). Без додаткових заходів такий цикл виконуватиметься "вічно", що спричинить "зависання" програми. Щоб не допустити цього, в циклі перевіряємо значення змінної x (яка, починаючи з нуля, при кожному проходженні циклу прирощує своє значення на 1). Коли x досягає значення 3, виконання циклу припиниться, оскільки виконається інструкція break.

Інструкція continue

Інструкцію continue використовують у циклах while, do, for та foreach. Вона спричиняє перехід до початку (найглибшого вкладеного) циклу.

Розглянемо приклад її використання у лістингу 2.9. Цикл, як задано у конструкції for, виконується 7 разів (рядок 1). Але для перших трьох разів, оскільки x буде меншим або дорівнюватиме 3, виконуватиметься умова if (рядок 3), після чого в результаті спрацювання інструкції continue виконання переходитиме на початок циклу. Тому інструкції з тіла циклу, розташовані після рядка 4, не виконуватимуться. Таким чином, вивід на екран значення x (в рядку 7) буде тільки для тих x, які більші від 3. Результат виконання лістингу 2.9 приведено на рис. 2.15.

Лістинг 2.9. Використання інструкції continue у циклі for

```
1  for (int x = 1; x <= 7; x++) // Цикл виконується 7 разів
2  {
3      if (x <= 3)          // Перші 3 рази
4          continue;        // відразу переходимо до початку циклу
5
6      // Цей рядок виконуватиметься, якщо x >= 3
7      Console.WriteLine("x = {0}", x);
8  }
```

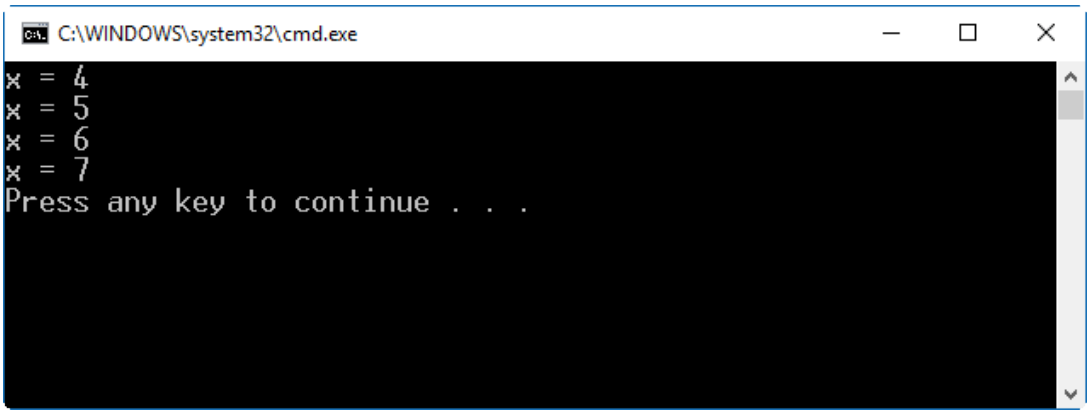


Рис. 2.15. Результат виконання коду з лістингу 2.9

Лістинг 2.10. Використання інструкції continue у циклі while

```
1  int x = 1;
2  while (x <= 7)
3  {
4      if (x <= 3)
5      {
6          x++;
7          continue; // Перехід на початок циклу
8      }
9      // Цей рядок виконуватиметься, якщо x >= 3
10     Console.WriteLine("x = {0}", x);
11     x++;
12 }
```

У лістингу 2.10 приведено приклад використання інструкції continue у циклі while. Результат його роботи буде такий самий, як і в попередньому випадку (рис. 2.15).

Мітки

Мітка – це інструкція, яка містить ідентифікатор з двокрапкою. Її використовують для безумовного переходу до того місця коду, яке вона позначає. Синтаксис використання мітки такий:

```
LabelName: Statement
```

де LabelName – назва мітки. Помічена інструкція Statement виконується так, ніби мітки не існує. Мітки оголошують лише всередині блоків коду. Область видимості мітки охоплює блок коду, в якому вона оголошена, та всі вкладені у нього (внутрішні) блоки коду.

Для переходу з будь-якого місця коду (в області видимості мітки) до виконання поміченої інструкції використовують інструкцію goto:

```
goto LabelName;
```

Після переходу на помічену інструкцію вона виконується, і далі виконання програми продовжується з цього місця коду.

Зауважимо, що інструкція goto часто заплутує код, оскільки сприяє поганій структуризації коду і як наслідок – погіршує його розуміння. Вважається хорошим тоном програмування по можливості не використовувати інструкцію goto.

Лістинг 2.11 демонструє дію мітки. У рядку 3 програма входить у "нескінченний" цикл (умова циклу завжди true). У циклі виводиться на екран поточне значення змінної i – ітератора циклу. А в рядку 7 виконується перевірка, чи значення i дорівнює 6. Якщо так, то за допомогою інструкції goto

здійснюється перехід до інструкції в рядку 14, яка помічена міткою в рядку 13. Результат роботи коду з лістингу 2.11 показано на рис. 2.16.

Лістинг 2.11. Використання мітки

```
1  int i = 0;
2  Console.WriteLine("Починаємо нескінченний цикл");
3  while (true)
4  {
5      i++;
6      Console.WriteLine("i = {0}", i);
7      if (i == 6)
8      {
9          Console.WriteLine("Пора закінчувати...");
10         goto TheEnd;
11     }
12 }
13 TheEnd:
14 Console.WriteLine("Кінець програми...");
```

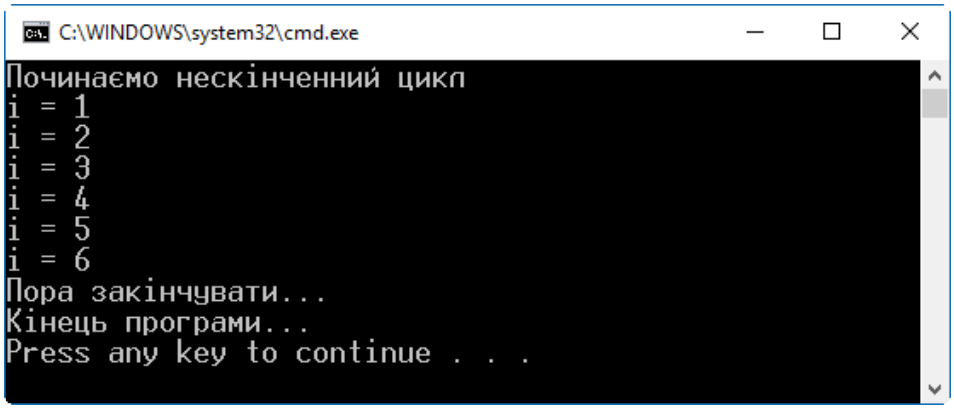


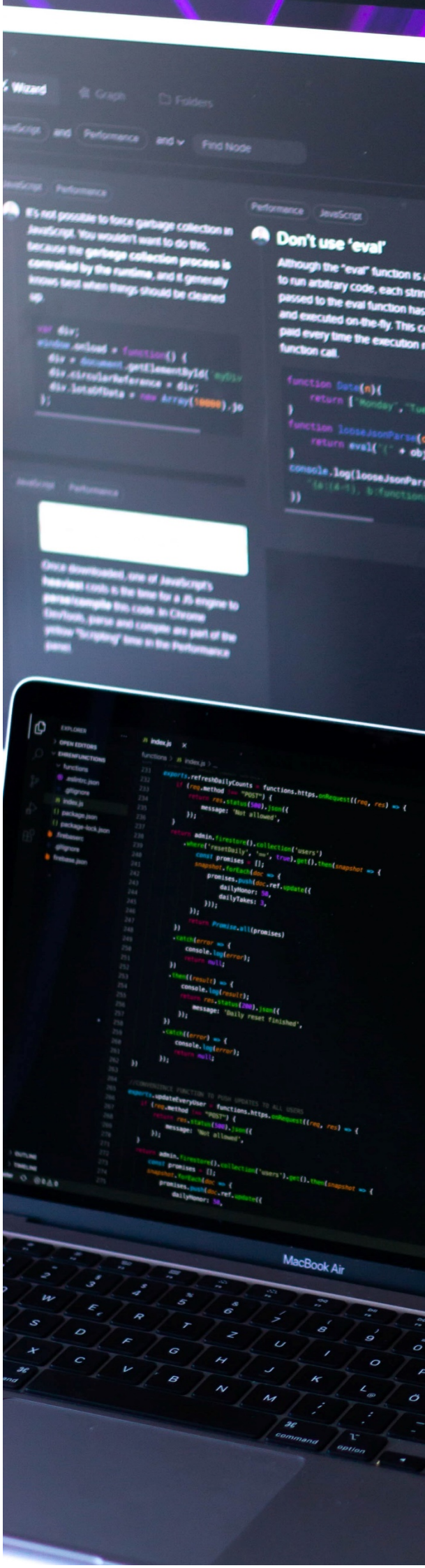
Рис. 2.16. Результат виконання коду з лістингу 2.11

Інші інструкції

Мова C# містить також інші інструкції, які керують потоком виконання програми. Серед них – foreach, try, throw, finally, return. Вони тісно пов'язані з різними особливостями програмування, тому розглянемо їх під час обговорення відповідних питань у наступних розділах.

Розділ 3. Класи: опис об'єктів

- Класи
- Методи
- Властивості
- Статичні елементи класу та константи
- Додаткові особливості елементів класу



Класи

Розглядаючи у попередньому розділі користувацькі типи даних, ми вже згадували, що один з них має назву "клас". З погляду об'єктно-орієнтованих принципів програмування клас – найважливіший тип даних, оскільки саме він реалізує головні принципи ООП: інкапсуляцію, спадкування, поліморфізм. Тому на розгляд класів звернемо особливу увагу.

Клас – це спеціальний тип даних, що використовується для опису об’єкта і визначає його загальні характеристики та дії, які він може виконувати. У класі поєднують сукупність логічно пов'язаних даних та функцій, які в програмі представляють об'єкт реального світу.

Клас представляє структуру, яка крім даних може містити виконавчий код. Складові частини класу називають *елементами класу* (або *членами класу*).

Елементи класу, які зберігають дані про об'єкт, називають *елементами даних*. Вони моделюють характеристики реального об'єкта, якого представляє клас. До елементів даних відносять поля та константи класу.

Елементи класу, які містять виконавчий код, називають *функціональними елементами*. Вони моделюють поведінку реального об'єкта. До функціональних елементів класу відносять методи (включаючи конструктори та деструктори), властивості, оператори, індексатори та події.

Типова програма на C# є групою взаємодіючих між собою об'єктів, більшість з яких є екземплярами якихось класів.

Оголошення класу

Клас є користувацьким типом даних. Проаналізувавши суттєві з точки зору програми характеристики та поведінку модельованого об'єкта, програміст повинен спроектувати клас, який їх відображуватиме. На практиці це роблять шляхом оголошення класу.

Оголошення класу означує його елементи даних та функціональні елементи. При цьому не створюється екземпляр класу (об'єкт). Клас, як інші типи, задає шаблон, за яким можна створювати екземпляри цього класу.

Нижче приведено узагальнений синтаксис оголошення класу:


```
class ClassName
{
    // Оголошення елементів класу
}
```

Оголошення класу починають ключовим словом class, після якого вказують назву класу (у приведеному зразку – ClassName). Далі між фігурними

дужками розміщують оголошення елементів класу (їх ще називають членами класу). Цю частину оголошення класу називають *тілом класу*. Елементи класу можна оголошувати у тілі класу в будь-якій послідовності.

Елементи класу

Найважливішими елементами класу є поля та методи. Поля є елементами даних, а методи – функціональними елементами.

Поля – це змінні, які належать класові. Вони можуть бути довільного типу (як попередньо визначеного C#, так і користувацького). Подібно до звичайних змінних, поля зберігають дані.

Найпростіший синтаксис для оголошення поля такий:


```
Type FieldName;
```

Нижче приведено приклад оголошення класу з одним полем:

```
class TheClass
{
    int TheField;
}
```

У цьому прикладі клас із назвою TheClass містить одне поле цілого типу, яке називається TheField.

Зауважимо, що, на відміну від C чи C++, у C# не може бути глобальних змінних, оголошених за межами типу. Всі поля належать типам і повинні бути оголошеними відповідно до правил синтаксису для даного типу.

Поле ініціалізують так само, як і звичайну змінну: при його оголошенні після назви ставлять знак "=" і тоді вказують значення чи вираз для ініціалізації:

```
class TheClass
{
    int TheField = 12;
}
```

Ініціалізація поля

Якщо поле класу явно не ініціалізують, то компілятор забезпечує його неявну ініціалізацію значенням за замовчуванням (для всіх числових типів це нуль, для посилальних типів – null). Використання явної та неявної ініціалізації демонструє приклад:

```
class TheClass
{
    int Field1 = 12;          // Ініціалізується значенням 12
    int Field2;               // Ініціалізується значенням 0
    string Field3 = "aaa";    // Ініціалізується значенням "aaa"
```

```
    string Field4;          // Ініціалізується значенням null
}
```

Як і звичайні змінні, поля класу одного типу можна оголошувати в одній інструкції, вибірково їх ініціалізуючи:

```
int Field1, Field2 = 22;
double Field3 = 4.5, Field4;
```

Метод – це іменований блок виконавчого коду (функція), який належить класові і може викликатися для виконання з різних частин програми, і навіть з-за меж даної програми. Коли метод викликають, виконується його код, після чого продовжують виконуватися інструкції, розташовані після інструкції виклику методу. Деякі методи повертають значення у місце, з якого їх викликано.

Найпростіший синтаксис для оголошення методу містить:

- Тип результату методу. Якщо метод не повинен повертати результату, замість типу вказують void.
- Назва методу. Це ідентифікатор, який використовують для його виклику.
- Перелік параметрів, розміщений у круглих дужках. Перелік параметрів може бути порожнім, але дужки повинні бути обов'язково.
- Тіло методу, оточене фігурними дужками. Воно містить виконавчий код методу.

Наступний приклад ілюструє оголошення простого методу. Тут оголошено клас Program з одним методом PrintHi. Як видно з оголошення, метод не повертає результату (як тип результату вказано void) і не має параметрів. Тіло метода містить один рядок, який виводить на екран фразу "Hi!".

```
class Program
{
    void PrintHi()
    {
        Console.WriteLine("Hi!");
    }
}
```

Тип результату

Список параметрів (порожній)

Зауважимо, що, на відміну від C чи C++, у C# не може бути глобальних функцій, оголошених за межами типу.

Створення екземпляра класу

Оголошення класу не приводить до створення об'єкта – екземпляра класу. Воно задає шаблон, за яким можна створювати екземпляри. Після того, як клас оголошено, можна створювати будь-яку кількість екземплярів.

Клас є посилальним типом даних, тому потребує двох ділянок пам'яті: для посилання на дані та для даних. Посилання на дані зберігається у змінній, оголошеній з типом класу. Оголошена, але не ініціалізована змінна класу має невизначене посилання.

Розглянемо лістинг 3.1, який містить код з оголошенням класу та змінної цього класу. У рядках 1…4 оголошено клас з назвою StringPrinter (тіло класу пропущено). Далі (рядки 6…13) оголошено клас Program з методом Main, у тілі якого (в рядку 10) оголошено змінну sp класу StringPrinter. При цьому в пам'яті (у стеку) виділяється місце для посилання (на майбутній екземпляр класу). Оскільки змінну sp не ініціалізовано, то посилання є невизначеним.

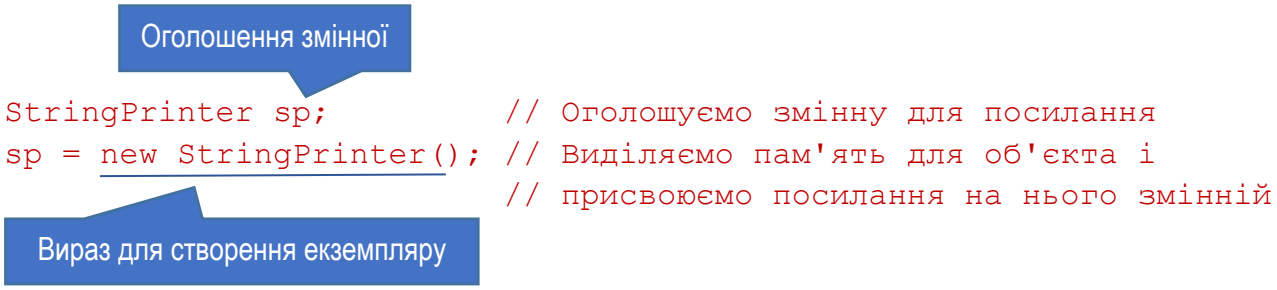
Лістинг 3.1. Оголошення класу та змінної класу

```
1  class StringPrinter
2  {
3      ...
4  }
5
6  class Program
7  {
8      static void Main(string[] args)
9      {
10         StringPrinter sp;
11         sp = new StringPrinter();
12     }
13 }
```

Оголошення змінної класового типу спричиняє виділення пам'яті для посилання, але не для даних об'єкта. Щоб виділити пам'ять для реальних даних об'єкта, використовують оператор new. Оператор new виділяє та ініціалізує пам'ять для екземпляра вказаного класу. Синтаксис його використання такий:

Оператор new створює об'єкт – екземпляр заданого класу та повертає посилання на нього.

Отже, щоб працювати у програмі з екземпляром класу, достатньо оголосити змінну його типу, створити екземпляр і зберегти у змінній посилання на екземпляр. Це ілюструє приклад:



Повніше код для створення екземпляра класу показано в лістингу 3.1. У рядку 11 створюється екземпляр класу StringPrinter і посилання на нього присвоюється змінній sp, оголошеній в попередньому рядку.

Дві інструкції, які оголошують змінну класу і створюють об'єкт, можна поєднати в одну, використавши вираз для ініціалізації змінної:

Елементи екземпляра і статичні елементи

У програмі можна створити довільну кількість екземплярів певного класу. Всі вони існуватимуть окремо і матимуть свій власний набір значень елементів класу. Елементи класу, які пов'язані з окремим екземпляром, називають *елементами екземпляра* (*instance members*). За замовчуванням, всі елементи класу є елементами екземпляра.

Але можна також оголосити елементи класу, які будуть пов'язані не з екземпляром класу, а з класом у цілому. Такі елементи класу називають *статичними* (*static members*). Їх при оголошенні позначають модифікатором static. Зокрема, метод Main має бути статичним, так як викликається до створення будь-якого екземпляра класу. Статичні елементи класу розглянемо пізніше.

Лістинг 3.2 ілюструє оголошення класу Student з полем FullName (рядки 1…4) і створення трьох екземплярів цього класу, на які посилаються змінні student1, student2 та student3 (рядки 10…12). Кожен із цих екземплярів може містити своє значення для поля FullName.

Лістинг 3.2. Створення кількох екземплярів класу

```
1  class Student
2  {
3      string FullName;
4  }
5
6  class Program
7  {
```

```
 8      static void Main(string[] args)
 9      {
10          Student student1 = new Student();
11          Student student2 = new Student();
12          Student student3 = new Student();
13      }
14  }
```

Модифікатори доступу

В межах класу будь-який його функціональний елемент має доступ до всіх інших елементів класу. Але доступ до елемента класу з інших частин програми (за межами класу) можна обмежити. При оголошенні елемента класу можна задати *модифікатор доступу*, який визначатиме, які саме частини програми матимуть доступ до елемента класу.

Мова C# передбачає такі модифікатори доступу:

- private
- public
- protected
- internal
- protected internal

Найвживаніші є перших два модифікатора. Їх використання розглянемо зараз, а призначення наступних модифікаторів – у подальших розділах.

Синтаксис оголошення елементів класу з використанням модифікатора доступу такий:

```
// Для поля
AccessModifier Type Identifier;
// Для метода
AccessModifier ReturnType MethodName()
{
    ...
}
```

Тут AccessModifier позначає модифікатор доступу (наприклад, public).

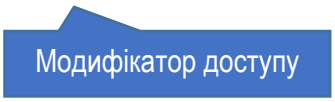
Приватні елементи класу (оголошені з модифікатором private) доступні тільки всередині класу, у якому вони оголошені. Інші класи не можуть ні "бачити" їх, ні працювати з ними. Приватний рівень доступу забезпечується за замовчуванням, тому, коли при оголошенні елемента класу модифікатор доступу не вказують, то елемент буде приватним. Але, щоб явно вказати приватний рівень доступу, використовують модифікатор private:

```
int Var1;           // Неявно оголошений приватний рівень доступу
private int Var2;   // Явно оголошений приватний рівень доступу
```

Модифікатор доступу

Публічні елементи класу (оголошені з модифікатором public) доступні всім іншим об'єктам програми. Цей рівень доступу слід вказувати тільки явно:

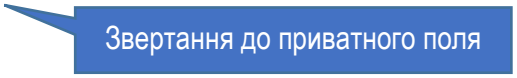
```
public int Var3;
```



Лістинг 3.3 демонструє оголошення класу з приватними та публічними елементами. У рядку 3 оголошено приватне поле цілого типу StudentId. У рядках 4…6 оголошено три публічні поля рядкового типу. Публічний метод PrintInternalInfo оголошено в рядках 8…12, а метод GetGeneralInfo – у рядках 14…17.

Лістинг 3.3. Оголошення приватних та публічних елементів класу Student

```
1  class Student
2  {
3      private int StudentId = 100;
4      public string Name;
5      public string Group;
6      public int Course;
7
8      public void PrintInternalInfo()
9      {
10         Console.WriteLine("Студент: {0}, Id: {1}",
11             Name, StudentId);
12     }
13
14     public string GetGeneralInfo()
15     {
16         return Name + ", " + Group + ", " + Course;
17     }
18 }
```



Використання елементів класу всередині класу

Всередині класу його методи мають доступ до всіх його елементів – як публічних, так і приватних. Звертаються до них безпосередньо за назвою. Так, у лістингу 3.3 в методі GetGeneralInfo (рядок 15) на основі полів Name, Group та Course формується загальний опис виду "прізвище, група, курс".

Використання елементів класу за межами класу

Для доступу до публічного елемента класу за його межами використовують назву змінної, яка представляє об'єкт, і назву елемента,

розділені крапкою. Наприклад, у цьому коді створюється екземпляр student1 класу Student (лістинг 3.3) і присвоюється значення його полю Name:

У третьому рядку виконується звертання до метода GetGeneralInfo об'єкта student1 (рядки 14…17 лістингу 3.3), і результат його виконання присвоюється змінній рядкового типу student1Info.

Лістинг 3.4 демонструє використання попередньо розглянутого класу Student у програмі. У ньому приведено код функції Main програми. У рядку 3 створюємо об'єкт, а в рядках 4…6 присвоюємо значення його полям. Рядок 7 за допомогою методу GetGeneralInfo отримує загальну інформацію про об'єкт. У рядках 8 та 9 на екран виводиться інформація про об'єкт. Результат роботи програми, яка об'єднує код лістингів 3.3 та 3.4, приведено на рис. 3.1.

Лістинг 3.4. Робота з екземпляром класу Student у програмі

```
1  static void Main(string[] args)
2  {
3      Student student1 = new Student();
4      student1.Name = "Петренко Петро";
5      student1.Group = "КА-41";
6      student1.Course = 4;
7      string student1Info = student1.GetGeneralInfo();
8      student1.PrintInternalInfo();
9      Console.WriteLine(student1Info);
10 }
```

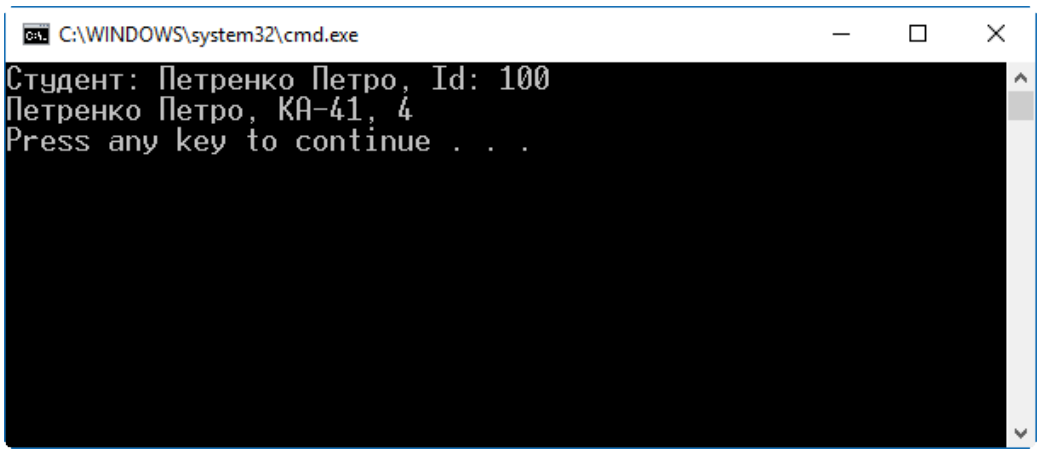


Рис. 3.1. Результат роботи програми з лістингів 3.3 та 3.4

Методи

Метод – це іменований блок коду. Методи забезпечують функціональність об'єктів. Об'єкт без методів є пасивною структурою, у якій можна зберігати дані, але вона не виконає ніяких операцій над даними.

Методи містять більшу частину коду, який забезпечує діяльність програми. Решту реалізують інші функціональні елементи класу: властивості та оператори. При написанні методів використовують попередньо розглянуті типи та конструкції для керування потоком виконання програми: перевірку умови, цикли, переходи тощо.

Код, представлений у методі, можна виконати з будь-якого місця програми в області видимості, використовуючи назву методу (та назву об'єкта). Під час роботи програми з методом можна обмінюватися даними, передаючи їх у метод та отримуючи їх від нього.

Оголошення методу

Оголошення методу містить дві головних секції: заголовок методу та тіло методу (рис. 3.2). Заголовок методу задає його характеристики: тип результату (якщо він є), назву, дані, які передаються у метод і отримуються від нього. Тіло методу містить послідовність інструкцій, які виконують необхідні дії. Виконання завжди починається з першої інструкції методу і послідовно продовжується до виходу з нього.

Найпростіша форма заголовку методу така:

```
MethodType MethodName (Param1Type Param1, Param2Type Param2);
```

Тип результату | Назва методу | Перелік параметрів

де MethodType – тип результату методу, MethodName – назва методу. У дужках вказано список параметрів. У списку параметри розділені комами, а кожен параметр описаний парою "тип параметра" – "назва параметра". Якщо у метод не потрібно передавати ніяких даних, список параметрів буде відсутнім (як на рис.3.2).

Після заголовку розташовують тіло метода, яке є звичайним програмним блоком, оточеним фігурними дужками. У ньому, як і в інших блоках, можна оголошувати локальні змінні, використовувати вкладені блоки, інструкції для керування потоком виконання, а також викликати інші методи.

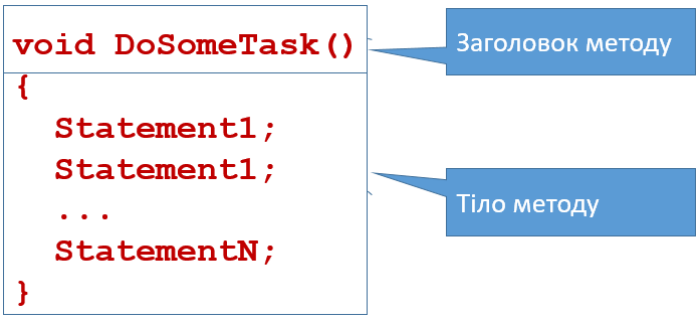


Рис. 3.2. Структура методу

Локальні змінні

Локальні змінні – це змінні, які діють тільки в обмеженому блоці коду. Подібно до полів, їх також використовують для зберігання даних. Але поля зберігають дані, які стосуються об'єктів, а локальні змінні – тимчасові проміжні дані.

Оголошують локальні змінні так само, як і інші змінні. При оголошенні вказують тип, назву змінної, а також (необов'язково) вираз для ініціалізації.

На відміну від полів класу, область видимості та життєвий цикл локальної змінної обмежені блоком коду, в якому вона оголошена. Змінна доступна після її оголошення, і перестає існувати при виході з її блоку коду. Порівняння полів та локальних змінних приведено у таблиці 3.1.

Таблиця №3.1. Порівняльна таблиця полів та локальних змінних

	Поле екземпляру	Локальна змінна
Час життя	Починається із створенням екземпляру, закінчується при знищенні екземпляру	Починається у місці блоку, де змінна оголошена. Закінчується при виході з цього блоку
Неявна ініціалізація	Ініціалізується значенням за замовчуванням для типу	Немає неявної ініціалізації. При спробі використання неініціалізованої змінної виникне помилка компілятора

Дуже часто локальні змінні ініціалізують відразу при їх оголошенні, наприклад:

```
int count = 1;
double max = 100.0D;
Student student1 = new Student();
```

При цьому інформацію про тип компілятор може отримати як через безпосереднє вказування, так і через ініціалізуюче значення. Так, в першому випадку тип значення 1 легко розпізнати як int, значення 100.0D – як тип double, а student1 – як тип Student. В таких випадках явне вказування типу є надлишковим.

Щоб запобігти надлишковості, C# надає можливість використання ключового слова var замість явно заданого типу змінної:

```
var count = 1;
var max = 100.0D;
var student1 = new Student();
```

Ключове слово var не є ознакою особливого виду змінної. Воно є синтаксичним скороченням оголошення змінної, яке повідомляє компілятору, що інформацію про тип змінної слід отримати з ініціалізаційної секції. Якщо замість типу вказати var, але не провести ініціалізації змінної, виникне помилка компіляції. Таким чином, ключове слово var не змінює строго типизованої природи C#.

Тіло методу може містити вкладені блоки. Локальні змінні можна оголошувати у вкладених блоках. В будь-якому випадку, область видимості та час життя змінної обмежені тим блоком, у якому вона оголошена (та блоками, які вкладені в нього), як показано на рис. 3.3.

Зауважимо, що в C та C++ можна було оголосити локальну змінну, а потім у вкладеному блоці оголосити ще одну локальну змінну з тою ж назвою. При цьому "внутрішня" змінна у вкладеному блоці приховувала "зовнішню". У C# оголосити другу змінну з тією ж назвою в області видимості першої не можна, незалежно від рівня вкладеності блоків.

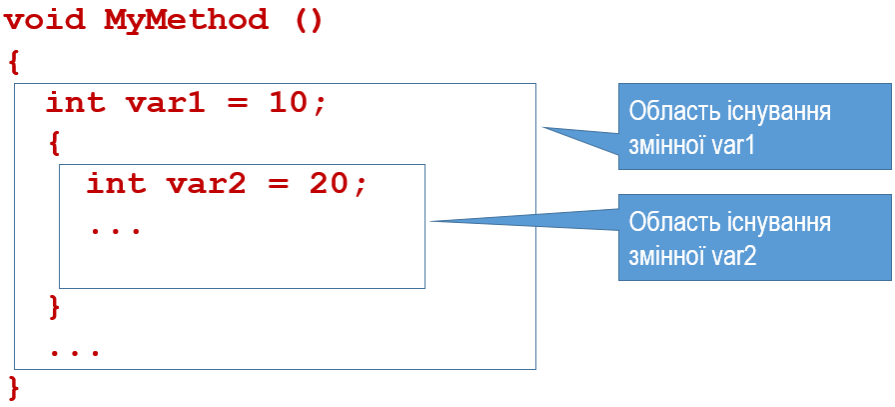


Рис. 3.3. Видимість локальних змінних у вкладених блоках

Локальні константи

Локальні константи дуже схожі на локальні змінні, але їх значення, задане при оголошенні, програмно не можна змінювати. Локальні константи, як і змінні, можуть бути оголошеними тільки у блоках.

Оголошення константи відрізняється від оголошення змінної використанням ключового слова const безпосередньо перед її типом та обов'язковою ініціалізацією під час оголошення.

Загальний синтаксис оголошення константи такий:

Ключове слово — Значення для ініціалізації

```
const Type Identifier = Value;
```

У лістингу 3.5 приведено приклад використання константи, яка представляє число π. Локальну константу з назвою PI та типом double оголошено в рядку 3. У рядку 6 вона використовується у виразі, який розраховує площу круга. Метод DisplayCircleAreas у циклі розраховує площу кругів з радіусом від 10 до 15 і виводить розраховані значення у консольне вікно. Результат виконання розглянутого коду приведено на рис. 3.4.

Лістинг 3.5. Приклад використання локальної константи

```
1  static void DisplayCircleAreas()
2  {
3      const double PI = 3.1416;
4      for (int radius = 10; radius <= 15; radius++)
5      {
6          double area = radius * radius * PI;
7          Console.WriteLine("R = {0}, Площа кола: {1}",
8              radius, area);
9      }
10 }
11
12 static void Main(string[] args)
13 {
14     DisplayCircleAreas();
15 }
```

Оголошення константи

Використання константи

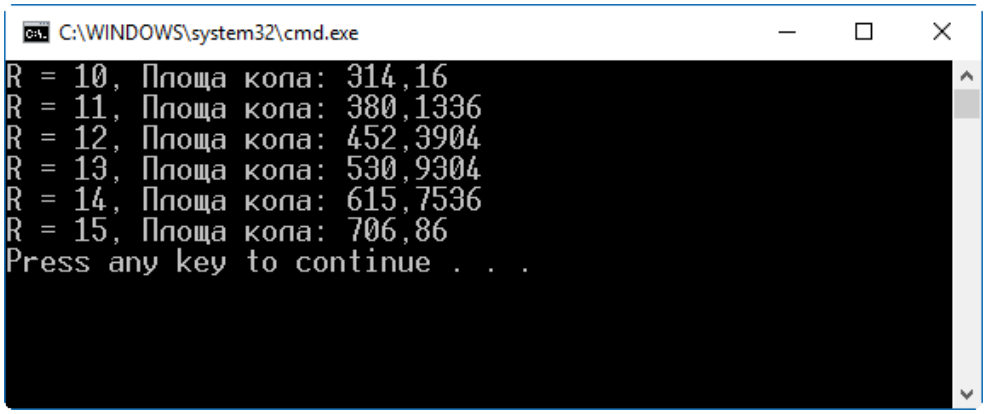


Рис. 3.4. Результат роботи програми з лістингу 3.5

Виклик методу

Перехід до виконання методу називають *викликом* методу. Методи можна викликати з будь-якого місця програми (в області їх видимості), у тому числі з інших методів. Метод, з якого викликають інший метод, називають *викликаючим*. Метод, який викликають, називають *викликаним* методом.

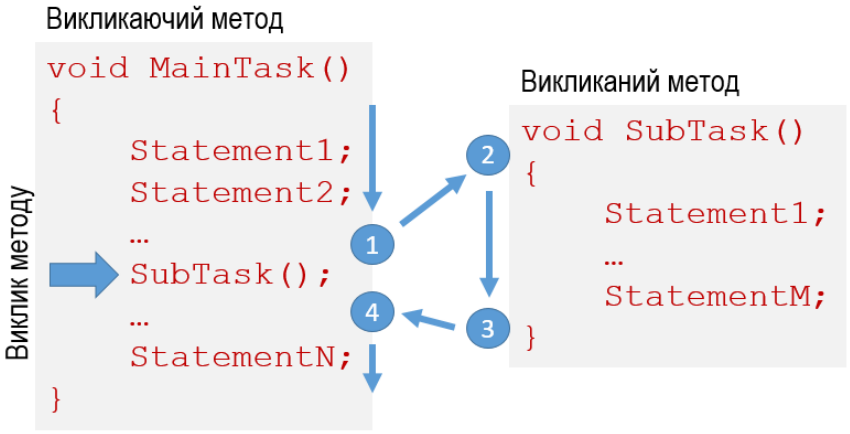


Рис. 3.5. Потік виконання при виклику методу

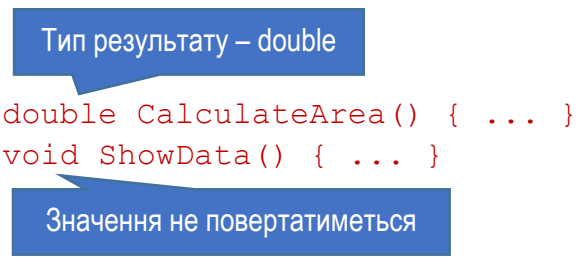
Схема на рис. 3.5 ілюструє послідовність дій при виклику методу. Спочатку послідовно виконуються інструкції викликаючого методу MainTask (стрілка до позиції 1). Якщо в інструкції трапляється виклик методу (товста стрілка), то виконання поточного методу призупиняється (позиція 1), і починає виконуватися викликаний метод SubTask (ділянка між позиціями 2 та 3). Коли викликаний метод закінчує свою роботу, потік виконання знову переходить до викликаючого методу (ділянка 3 – 4), і він продовжує виконуватися (стрілка після позиції 4) до свого закінчення.

Повернення значення з методу

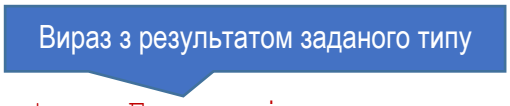
Метод може повертати до викликаючого методу якесь значення. Повернене значення вставляється у викликаючий код в місце, де записаний вираз із викликом методу.

Щоб метод міг повернути значення, для нього при оголошенні потрібно задати тип поверненого значення. Якщо метод не повинен повертати ніякого значення, слід задати замість типу поверненого значення void.

У наступному прикладі оголошено два методи. Перший з них повертає значення типу double, а другий – не повертає значення взагалі.

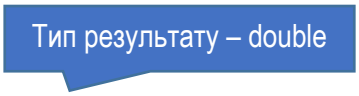


Якщо метод оголошено з типом поверненого значення, то він повинен використовувати інструкцію return з виразом, тип якого відповідає оголошеному типу. Виконання такого методу слід припиняти тільки викликом інструкції return з виразом потрібного типу:




```
return Expression;
```

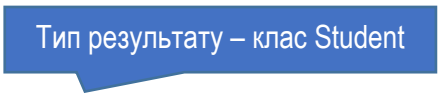
Лістинг 3.6. Оголошення методу, який повертає значення типу double




```
1  double CalculateArea(double r)
2  {
3      const double PI = 3.1416;
4      double Area = PI * r * r;
5
6      return Area;
7  }
```

У лістингу 3.6 приведено оголошення методу CalculateArea, який обчислює площу круга із заданим радіусом і повертає її як результат. У рядку 1 в заголовку метода вказано тип результату – double. В коді оголошено змінну Area цього ж типу (рядок 4), і присвоєно їй розраховане значення площі. У рядку 6 за допомогою інструкції return вміст змінної Area повертається викликаючому коду.

Лістинг 3.7. Оголошення методу, який повертає об'єкт класу Student (див. лістинг 3.3)




```
1  Student CreateStudent1()
2  {
3      Student student = new Student();
4      student.Name = "Петренко Петро";
5      student.Group = "КА-41";
6      student.Course = 4;
7
8      return student;
9  }
```

Метод може повертати не тільки прості значення, а й об'єкти визначених користувачем типів. Наприклад, код у лістингу 3.7 описує метод, який повертає

об'єкт класу Student (точніше – посилання на об'єкт, оскільки клас є посилальним типом даних). Сам клас Student описано у лістингу 3.3. В рядку 3 оголошено тимчасову змінну student класу Student і створено відповідний об'єкт. Далі (рядки 4…6) полям об'єкта присвоєно якісь значення. Рядок 8 повертає створений об'єкт student викликаючому коду.

Повернення з методів без типу (void)

Метод, для якого при оголошенні не вказано типу його результату (вказано void), закінчує свою роботу при виконанні своєї останньої інструкції. Але часто трапляються умови, при яких необхідно припинити виконання методу швидше. Для цього також використовують інструкцію return, але без виразу. У будь-якому місці методу з нього можна вийти за допомогою інструкції return:

```
return;
```

Таку форму інструкції return (без виразу) можна використовувати тільки у методах void. При спробі використати її у методах, які повертають результат, виникне помилка компіляції.

Розглянемо приклад коду в лістингу 3.8. Код методу виконується у звичному порядку, послідовно від першої інструкції. У рядку 4 за допомогою інструкції if перевіряємо якусь умову (вона задана виразом Condition1). Якщо вона виконується, спрацьовує інструкція return у рядку 5, і метод припиняє свою роботу. Програма продовжує виконання наступних інструкцій викликаючого коду. Якщо умова у рядку 4 не виконується, то інструкція return не виконується, і метод продовжує роботу.

Лістинг 3.8. Припинення роботи методу

Тип результату – відсутній

```
1  void SomeMethod()
2  {
3      ...
4      if (Condition1)          // Якась умова
5          return;              // Повернення до викликаючого коду
6      ...
7      if (Condition2)          // Інша умова
8          return;              // Повернення до викликаючого коду
9      ...
10 }            // Кінець методу: повернення до викликаючого коду
```

Далі лістинг 3.8 містить ще одну перевірку з наступним припиненням роботи методу при виконанні умови (рядки 7…8). Якщо ні одна інструкція

return не спрацювала, то виконуються всі інструкції методу, і після виконання останньої відбувається перехід до викликаючого коду.

Параметри

Дуже часто для роботи методу йому потрібно передати додаткові дані. Це роблять за допомогою параметрів методу. Параметри дозволяють також отримати з методу більше даних, ніж одне повернене значення. *Параметри* – це спеціальні змінні, за допомогою яких у метод передають дані (та отримують їх).

Розрізняють *формальні* та *фактичні параметри*.

Формальні параметри (formal parameters) – це локальні змінні, оголошені у заголовку методу. Вони представляють дані, що передаються у метод. Перелік параметрів у заголовку вказують у дужках парами "тип-назва", розділяючи їх комами. Метод може мати довільну кількість параметрів.

Ось приклад оголошення методу зі списком параметрів:

```
public double CalcRectangleArea(double a, double b)
{
    ...
}
```

Перелік параметрів

Формальні параметри – це змінні, тому в них можна зберігати дані (відповідно до типу), присвоювати їм значення та читати його. На відміну від локальних змінних у методі, параметри оголошені за межами методу і ініціалізуються до того, як його перша інструкція спрацює (винятком є спеціальний вид параметрів – *вихідні* параметри).

Область видимості параметрів – все тіло методу. У лістингу 3.9 оголошено метод CalcRectangleArea з двома параметрами дійсного типу. Функція CalcRectangleArea розраховує площу прямокутника зі сторонами a та b, значення яких передають у параметрах.

Лістинг 3.9. Метод з двома параметрами

```
1  double CalcRectangleArea(double a, double b)
2  {
3      return a * b;
4  }
```

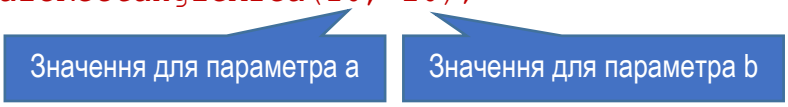
Під час виклику методу, до того, як він почне виконуватися, його параметри мають бути ініціалізовані. Вирази або змінні, які використовують для ініціалізації формальних параметрів, називають *фактичними параметрами* (*actual parameters*).

Порядок і тип фактичних параметрів повинен відповідати порядку і типу формальних параметрів. При виклику методу кожен фактичний параметр

розміщують на місце відповідного формального. Кожен фактичний параметр повинен мати тип, який відповідає заданому типу відповідного формального параметра.

Ось приклад виклику методу CalcRectangleArea з лістингу 3.9:

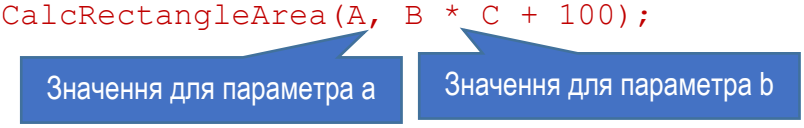
```
double area = CalcRectangleArea(10, 20);
```



У цьому прикладі оголошено змінну area типу double і присвоєно їй результат виконання методу CalcRectangleArea. При виклику в метод передають два фактичних параметра. Значення 10 присвоюється формальному параметру a (він у списку перший), значення 20 присвоюється параметру b (він другий).

Фактичними параметрами при виклику метода можуть бути не тільки безпосередні значення (як у попередньому прикладі), але і вирази:

```
double A = 30, B = 40, C = 0.5;
double area = CalcRectangleArea(A, B * C + 100);
```



Параметри значущих та посилальних типів

Якщо тип параметра належить до значущих типів, то значення фактичного параметра копіюється у формальний параметр. Таким чином, метод працює з копією переданих йому даних. Всі зміни, які метод вносить у цю копію, не впливають на початкові дані фактичного параметра.

Розглянемо приклад з лістингу 3.10. Тут оголошено метод Doubling, який отримує через параметр число, множить його на 2 і результат виводить на екран.

Лістинг 3.10. Передача в метод параметра значущого типу

```
1  static void Doubling(double a)
2  {
3      a = a * 2;
4      Console.WriteLine("В методі: a = {0}", a);
5  }
6
7  static void Main(string[] args)
8  {
9      double aa = 10;
10     Console.WriteLine("До виклику aa = {0}", aa);
11     Doubling(aa);
12     Console.WriteLine("Після виклику aa = {0}", aa);
13 }
```

У методі Main оголошено змінну aa, ініціалізовано її значенням 10 і передано через параметр у метод Doubling. Не зважаючи на те, що всередині методу передане значення подвоюється, після виходу з нього значення змінної aa залишається незмінним – 10 (див. рис. 3.6).

По-іншому ведуть себе дані, передані через параметри посилальних типів. При передачі у метод параметр-посилання також копіюється, але копія посилання вказує на той самий об'єкт у пам'яті, що й посилання у фактичному параметрі. Тому всі зміни з таким об'єктом, зроблені всередині методу, відобразяться у викликаючому методі.

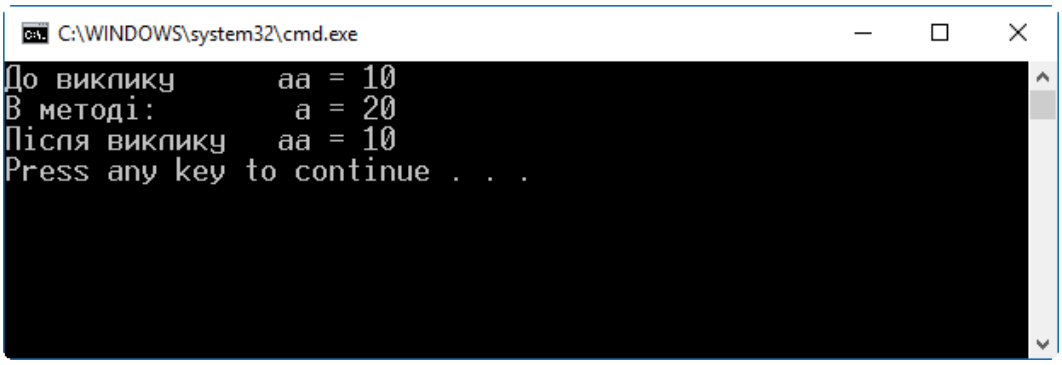


Рис. 3.6. Результат роботи програми з лістингу 3.10. Зміни, внесені методом у переданий параметр значущого типу, не зберігаються після виходу з методу

У лістингу 3.11 приведено приклад передачі у метод ChangeName параметра student – посилання на об'єкт Student (див. лістинг 3.3). У тілі методу полю Name присвоюється значення "Васильченко Василь", яке виводиться на екран.

Лістинг 3.11. Передача в метод параметра посилального типу

```
1  static void ChangeName(Student student)
2  {
3      student.Name = "Васильченко Василь";
4      Console.WriteLine("В методі: \t" + student.Name);
5  }
6
7  static void Main(string[] args)
8  {
9      Student stud = new Student();
10     stud.Name = "Петренко Петро";
11     Console.WriteLine("До виклику: \t" + stud.Name);
12     ChangeName(stud);
13     Console.WriteLine("Після виклику: \t" + stud.Name);
14 }
```

У методі Main створюється об'єкт – екземпляр класу Student, після чого в його поле Name записується значення "Петренко Петро". На екран виводиться початкове значення поля Name об'єкта. Далі викликається метод ChangeName (йому через параметр передається створений об'єкт), і в кінці на екран ще раз виводиться значення поля Name об'єкта. Як видно з рис. 3.7, зміни, зроблені методом ChangeName, зберігаються після закінчення його роботи.

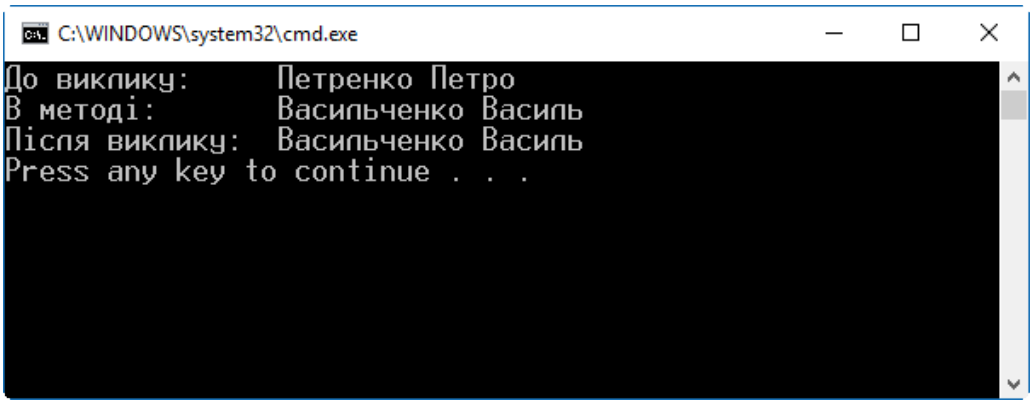


Рис. 3.7. Результат роботи програми з лістингу 3.11. Зміни, внесені методом у переданий параметр посилального типу, зберігаються після виходу з методу

Параметри-посилання

Параметри-посилання (*reference parameters*) дозволяють передати у метод посилання на змінну будь-якого типу (у тому числі значущого). Такі параметри оголошують та передають у метод з модифікатором ref. При цьому фактичним параметром повинна бути тільки змінна, і їй слід присвоїти значення до того, як вона буде використана в ролі фактичного параметра.

Синтаксис оголошення та використання параметра-посилання такий:

```
                Модифікатор ref
void TheMetod(ref double par1)   // Оголошення методу
{ ... }
double a = 100.5;                // Змінна для фактичного параметра
TheMetod(ref a);                 // Виклик метода з ref-параметром
          Модифікатор ref
TheMetod(ref 100.5);             // Так робити не можна!
          Помилка! Має бути змінна!
```

У попередньому пункті ми бачили, що при передаванні у метод параметра значущого типу його значення копіюється у формальний параметр, і метод працює не з початковими даними, а з їх копією. При використанні

параметра-посилання він веде себе подібно до параметрів посилальних типів. Формальний параметр-посилання діє як псевдонім для змінної актуального параметра, посилаючись на ту ж ділянку пам'яті з даними. Тому всі зміни, які вносяться у передані дані, збережуться після виходу з методу і будуть доступними через фактичний параметр.

У лістингу 3.10 ми розглядали передачу в метод параметра значущого типу, і бачили, що після виходу з методу зміни, які вносилися у формальний параметр, не зберігалися (рис. 3.6). Замінимо в лістингу 3.10 параметр методу Doubling на параметр-посилання (лістинг 3.12). Зауважимо, що єдина відмінність лістингу 3.12 від 3.10 – використання у рядках 1 та 11 модифікатора ref, який перетворює звичайний параметр на параметр-посилання.

Лістинг 3.12. Передача в метод параметра-посилання значущого типу

```
1  static void Doubling(ref double a)
2  {
3      a = a * 2;
4      Console.WriteLine("В методі: a = {0}", a);
5  }
6
7  static void Main(string[] args)
8  {
9      double aa = 10;
10     Console.WriteLine("До виклику aa = {0}", aa);
11     Doubling(ref aa);
12     Console.WriteLine("Після виклику aa = {0}", aa);
13 }
```

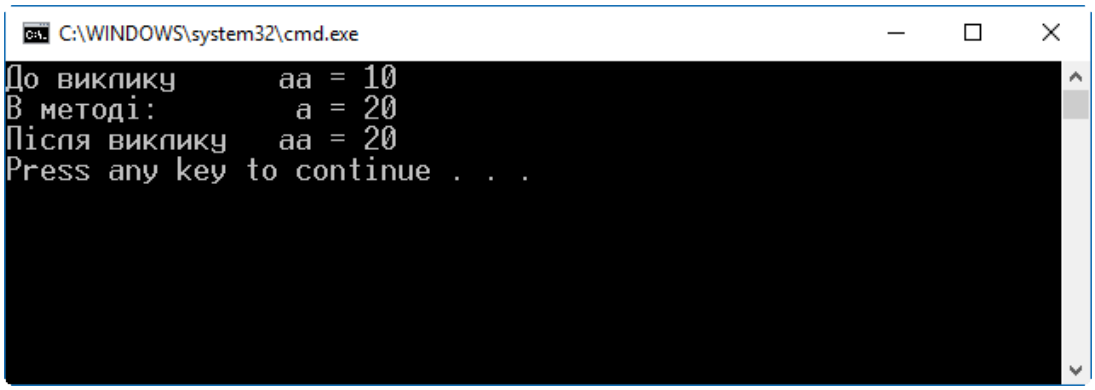


Рис. 3.8. Результат роботи програми з лістингу 3.12. Зміни, внесені методом у переданий параметр-посилання, зберігаються після виходу з методу

Виконавши цей код, бачимо, що зміни, внесені у переданий ref-параметр, зберігаються після відпрацювання методу, який їх зробив (порівняйте рис. 3.8 та рис. 3.6).

Підсумовуючи сказане, можна зробити такі висновки:

- Якщо викликаючому коду передане у метод значення не потрібно змінювати, його слід передавати як звичайний параметр.
- Якщо викликаючий код передбачає, що значення переданого параметра може (чи повинно) змінюватися викликаним методом, то його слід передавати як параметр-посилання з модифікатором ref.

Посилальні типи у звичайних та ref-параметрах

Ми вже бачили, що переданий у метод об'єкт посилального типу можна змінювати всередині викликаного метода, незалежно від того, чи використано звичайний параметр, чи параметр-посилання (див. лістинг 3.11). Проте, передавання параметра посилального типу з модифікатором ref має суттєву відмінність. Вона проявляється, якщо всередині викликаного метода за переданим посиланням не просто змінюють дані об'єкта, а створюють новий об'єкт.

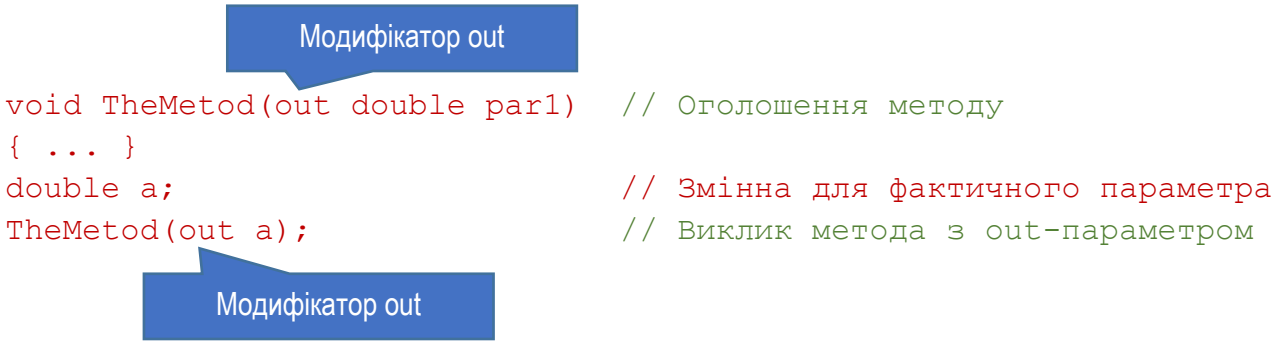
Розглянемо два випадки:

1. Передавання об'єкта посилального типу через звичайний параметр. У викликаному методі через формальний параметр можна змінювати актуальні дані об'єкта. Але, якщо створити новий об'єкт і присвоїти формальному параметру посилання на нього, то це розірве зв'язок між формальним та фактичним параметром. Новий об'єкт (відповідно і його дані) не збережеться після виходу з викликаного методу.
2. Передавання об'єкта посилального типу через параметр-посилання. Якщо у викликаному методі створити новий об'єкт і присвоїти його формальному параметру, то зміниться і посилання в фактичному параметрі – воно також почне вказувати на новий об'єкт. Після виходу з викликаного методу новий об'єкт, тепер пов'язаний зі змінною фактичного параметра, збережеться. Старий об'єкт, як структура, на яку більше не вказує ніяке посилання, буде знищений збирачем сміття .NET Framework.

Вихідні параметри

Вихідні параметри призначені для передачі даних з методу у код, з якого він був викликаний. Їх поведінка дуже схожа на параметри-посилання. Вихідні параметри оголошують та викликають з модифікатором out. Як і параметр-посилання, вихідний параметр повинен бути змінною: передавати вирази цим параметром неможливо.

Синтаксис використання вихідного параметру такий:



Подібно до параметра-посилання, формальний параметр для вихідного параметра діє як посилання на фактичний параметр. І формальний, і фактичний параметр є назвами для однієї і тієї ж ділянки пам'яті з реальними даними. Тому всі зміни з формальним параметром відразу відображаються у реальних даних і зберігаються після виходу з методу. Вихідний параметр відрізняється від параметра-посилання такими рисами:

- Всередині методу з вихідного параметра можна читати дані тільки після того, як присвоїти йому якесь значення. Це означає, що ініціалізація вихідного параметра перед його передачею в метод є недоречною: це значення у методі все одно не може бути отримане. Таким чином, передати у метод дані через вихідний параметр неможливо.
- Всередині методу вихідному параметру обов'язково має бути присвоєне значення перед закінченням його роботи.

У лістингу 3.13 продемонстровано використання вихідного параметра для отримання результату роботи методу.

Лістинг 3.13. Отримання значення із методу через вихідний параметр

```
1  static void CalcSummOut(double a, double b, out double summ)
2  {
3      summ = a + b;
4  }
5
6  static void Main(string[] args)
7  {
8      double aa = 10, bb = 20, cc;
9      CalcSummOut(aa, bb, out cc);  // Вживаємо модифікатор out
10     Console.WriteLine("Сума {0} та {1} = {2}", aa, bb, cc);
11 }
```

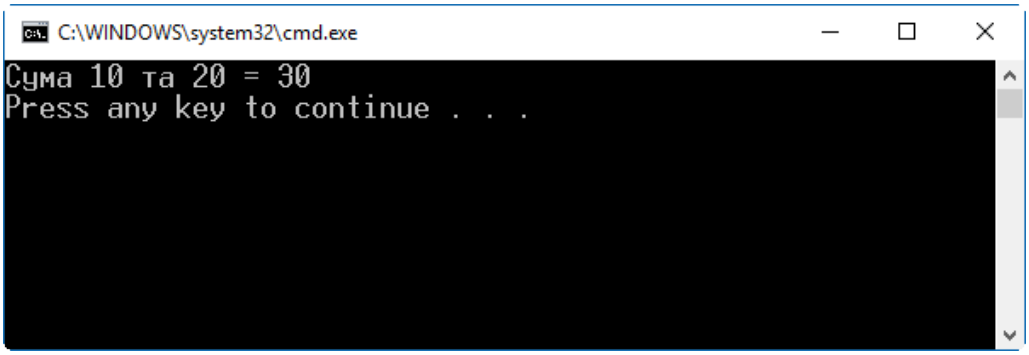


Рис. 3.9. Результат роботи програми з лістингу 3.13. Передача даних з метода через вихідний параметр

У рядках 1…4 оголошено метод CalcSummOut, який містить два звичайних параметра (a та b), і один вихідний (summ). В тілі метода обчислюється сума значень a та b, результат якої записується у вихідний параметр summ. Таким чином, вихідний параметр повертає у викликаючий код суму переданих значень. У методі Main викликається метод CalcSummOut (рядок 9). Як вихідний параметр йому передається змінна cc. З рис. 3.9 видно, що отримане через вихідний параметр значення справді містить суму переданих при виклику чисел.

Починаючи з версії C# 7.0, вихідний параметр можна не оголошувати окремим рядком, а безпосередньо під час виклику методу. У наступному фрагменті викликається метод TryParse класу Int32, якому передається вихідний параметр result, оголошений безпосередньо під час виклику.

Модифікатор out

```
Int32.TryParse(input, out int result);
```

Областю його використання є, як і для звичайної локальної змінної, поточний блок коду.

Якщо ж вихідний параметр не потрібний у подальшому, замість нього можна використати символ відкидання (_):

Відкинути вихідний параметр

```
bool isCorrectCount = Int32.TryParse(input, out _);
```

Масив параметрів

У всіх попередньо розглянутих прикладах кількість формальних та фактичних параметрів строго однакова: для кожного формального параметра існує один відповідний фактичний параметр. Але інколи наперед невідомо, скільки значень потрібно передати у метод. Тоді допомогти може масив

параметрів. Цей механізм відмінний від масиву, який передають через параметр, але дуже схожий. Роботу масивів розглянемо в одному з наступних розділів.

Масив параметрів дозволяє передати нуль або більше фактичних параметрів за допомогою одного формального параметра. При використанні масиву параметрів слід пам'ятати:

- У переліку параметрів метода може бути тільки один масив параметрів.
- Якщо він є, то має бути останнім у переліку.
- Всі параметри масиву повинні мати один тип.

Щоб оголосити масив параметрів, перед назвою формального параметра вказують модифікатор params, а після типу параметра розміщують пару квадратних дужок. Синтаксис оголошення методу з масивом параметрів такий:

Лістинг 3.14. Передача у метод масиву параметрів

```
1  static double CalcAnySumm(params double[] numbers)
2  {
3      double summ = 0;
4      foreach (double d in numbers)
5          summ += d;
6      return summ;
7  }
8
9  static void Main(string[] args)
10 {
11     double a = 10, b = 20, c = 30, d = 40, e = 50;
12     Console.WriteLine("Сума чисел ({0}, {1}) = {2}",
13         a, b, CalcAnySumm(a, b));
14     Console.WriteLine("Сума чисел ({0}, {1}, {2}) = {3}",
15         a, b, c, CalcAnySumm(a, b, c));
16     Console.WriteLine(
17         "Сума чисел ({0}, {1}, {2}, {3}) = {4}",
18         a, b, c, d, CalcAnySumm(a, b, c, d));
19     Console.WriteLine(
20         "Сума чисел ({0}, {1}, {2}, {3}, {4}) = {5}",
21         a, b, c, d, e, CalcAnySumm(a, b, c, d, e));
22 }
```

Масив параметрів

У лістингу 3.14 приведено код методу CalcAnySumm, якому передається масив параметрів numbers. Сам метод оголошено в рядках 1…7. Він розраховує суму всіх переданих у масиві параметрів значень. В тілі метода організовано цикл foreach, який проходить по всіх елементах масиву та додає їх до локальної змінної summ. В кінці її значення повертається у викликаючий код як результат методу.

У методі Main відбувається кілька викликів методу CalcAnySumm з різною кількістю переданих значень (рядки 13, 15, 18, 21). На рис. 3.10 приведено результат роботи розглянутого коду. Незалежно від кількості переданих параметрів, метод CalcAnySumm правильно розраховує їх суму.

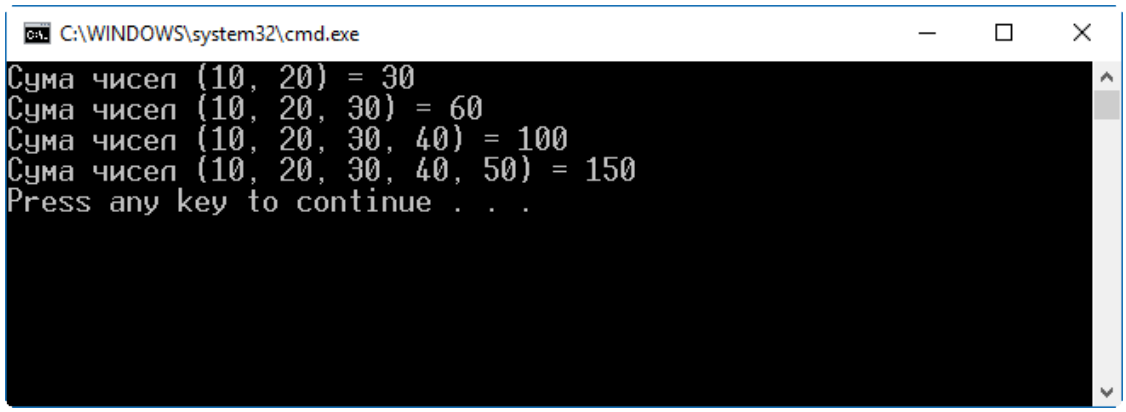


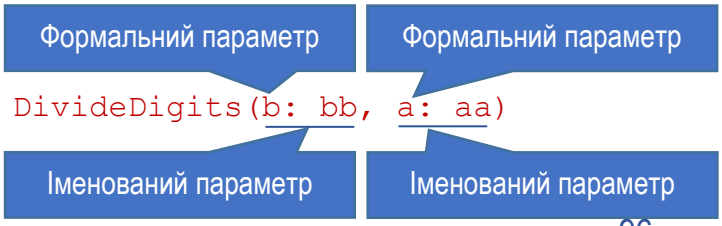
Рис. 3.10. Результат роботи програми з лістингу 3.14. Передача у метод масиву параметрів і розрахунок їх суми

Іменовані параметри

Перед цим ми розглядали випадки, коли під час виклику методу фактичні параметри підставляють виключно на місце відповідного формального параметра. Такі параметри називають *позиційними параметрами* (*positional parameters*).

У C# є можливість використання параметрів іншого виду – *іменованих* параметрів. Іменовані параметри дозволяють при виклику методу записувати фактичні параметри в довільному порядку, вказуючи при цьому назву формального параметра.

Для використання іменованих параметрів в оголошенні методу не потрібно нічого змінювати. Але під час виклику методу перед кожним фактичним параметром слід вказувати назву формального параметра з двокрапкою:



У лістингу 3.15 оголошено метод DivideDigits (рядки 1…4), який ділить два передані йому числа і повертає результат. В коді метода Main два рази викликається функція DivideDigits (рядки 10 та 12). Під час першого виклику спочатку вказують параметр a, потім b. Під час другого виклику порядок протилежний: спочатку b, потім a. Таким чином, використовуючи іменовані параметри, можна змінювати порядок розміщення параметрів під час виклику методу.

Лістинг 3.15. Використання іменованих параметрів

```
 1 static double DivideDigits(double a, double b)
 2 {
 3     return a / b;
 4 }
 5
 6 static void Main(string[] args)
 7 {
 8     double aa = 20, bb = 10;
 9     Console.WriteLine("{0} / {1} = {2}", aa, bb,
10         DivideDigits(a: aa, b: bb));
11     Console.WriteLine("{0} / {1} = {2}", aa, bb,
12         DivideDigits(b: bb, a: aa));
13 }
```

Необов'язкові параметри

Необов'язкові параметри можна не вказувати при виклику метода. Щоб вказати, що параметр не обов'язковий, для нього в оголошенні слід задати значення за замовчуванням. При цьому використовують такий же синтаксис, як і при ініціалізації змінної. Коли при виклику метода не вказують необов'язкового параметра, то приймається, що він дорівнює значенню за замовчуванням, вказаному при оголошенні методу.

Ось видозмінений метод з лістингу 3.15, у якому вказано необов'язковий параметр b із значенням за замовчуванням 2.

```
static double DivideDigits(double a, double b = 2)
{
    return a / b;
}
```

Якщо викликати цей метод, не вказуючи параметра b, то відбуватиметься ділення єдиного переданого параметра на 2 (значення за замовчуванням для b):

```
double half = DivideDigits(20);
```

Для параметрів посилальних типів значенням за замовчуванням може бути тільки null. Параметри з модифікаторами ref, out та params не можуть бути необов'язковими.

Всі обов'язкові параметри в оголошенні методу повинні розміщуватися перед необов'язковими. Якщо при цьому метод містить масив параметрів (з модифікатором params), то його слід розмістити після всіх необов'язкових параметрів.

При виклику метода з необов'язковими параметрами не можна пропускати їх у довільному порядку. Якщо якийсь необов'язковий параметр не вказують, то слід пропускати і всі інші необов'язкові параметри до кінця списку.

Методи як вирази

Якщо метод складається з єдиного виразу, його оголошення можна спростити, застосувавши синтаксис *елементів класу- виразів (expression-bodied members)*. Синтаксис елементів-виразів було введено у C# 6.0 та розширено у C# 7.0. Щоб застосувати оголошення такого виду, потрібно, щоб виконувались такі умови:

- Код повинен містити єдиний вираз, який закінчується крапкою з комою.
- Фігурні дужки для позначення тіла методу не застосовують.
- Перелік параметрів та вираз, який формує тіло методу, розділяють за допомогою лямбда-оператора (=>).

Використання синтаксису елементів-виразів дозволяє зробити код компактнішим та легшим для читання. Наприклад, нехай маємо метод, який повертає поштову адресу на основі параметрів з назвою міста та вулиці:

```
public string GetAddress(string cityName, string streetAddr)
{
    return cityName + ", " + streetAddr;
}
```

Цей же метод можна оголосити як вираз:

Лямбда-оператор

```
public string GetAddress(string cityName, string streetAddr) =>
    cityName + ", " + streetAddr;
```

Тіло методу

Локальні функції

Припустимо, що у коді є невеликий метод SmallMethod, який буде викликатися виключно з якогось іншого методу TheMethod. Всі інші методи класу не повинні мати доступу до SmallMethod. Такий метод можна оголосити як локальну функцію. Її код розміщують безпосередньо в тілі метода. Локальним функціям притаманні всі особливості методів, але при їх оголошенні не вказують модифікатори доступу. Хоча немає ніяких обмежень на розмір локальних функцій, їх первинне застосування – замістити невеликі методи, щоб не засмічувати ними оголошення класу.

Наступний блок коду містить локальну функцію addNumbers, яка розраховує суму двох цілих чисел. При оголошенні методу використано синтакс елемента- вираза.

```
using static System.Console;
...
static void Main(string[] args)
{
    int n1 = 10;
    int n2 = 20;

    int addNumbers(int num1, int num2) => num1 + num2;

    WriteLine($"Сума {n1} та {n2} дорівнює {addNumbers(n1, n2)}");
}
```

Локальна функція

Починаючи з версії C# 8.0, локальні функції можна оголошувати статичними (з ключовим словом static). Оскільки статичні елементи не можуть посилатися на змінні часу виконання, то оголошення локальної функції статичною убезпечує розробника від посилання з локальної функції на локальні змінні метода, у якому ця функція оголошена.

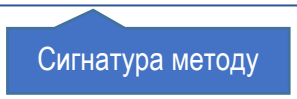
Перевантаження методів

Клас може містити кілька методів з однаковою назвою. Такі методи називають *перевантаженими* (*overloaded*). Але кожен з перевантажених методів повинен мати різну сигнатуру. *Сигнатура метода* містить його назву, кількість, типи і порядок параметрів, а також їх модифікатори. Тип поверненого значення не є частиною сигнатури. Назви формальних параметрів також не є частиною сигнатури – лише їх тип.

Перевантаження методів використовують, якщо потрібно мати множину методів, які виконують схожі чи однакові дії над параметрами різних типів.

У приведеному прикладі підкреслено елементи заголовка, які входять до складу сигнатури:

```
int  CalcSumm(int a, int b) { ... }
```



В лістингу 3.16 приведено приклад класу з чотирма перевантаженими методами CalcSumm.

Лістинг 3.16. Перевантаження методів

```
1  class TheClass
2  {
3     long CalcSumm(int a, int b) { return a + b; }
4     long CalcSumm(int c, int d, int e) { return c + d + e; }
5     long CalcSumm(float a, float b) { return (long)(a + b); }
6     long CalcSumm(long c, long d) { return c + d; }
7  }
```

Конструктори екземпляру

Конструктор екземпляру – це спеціальний метод класу, який виконується, коли створюється новий екземпляр класу. Конструктор використовують для ініціалізації екземпляру. Щоб можна було створити екземпляр класу з коду, який не належить цьому класові, конструктор має бути оголошений з модифікатором доступу public (інакше він буде недоступний – за замовчуванням використовується модифікатор private).

Назва конструктора завжди співпадає з назвою класу. Конструктор не може повертати значення, при його оголошенні не вказують навіть ключового слова void. В усьому іншому конструктор – звичайний метод. Зокрема, він може мати параметри. Найчастіше їх використовують для передачі значень, якими слід ініціалізувати поля створюваного об’єкта.

Клас завжди має хоча б один конструктор. Якщо при оголошенні класу явно не задано жодного конструктора, то використовується конструктор за замовчуванням, який не має параметрів і має порожнє тіло (не робить нічого, тільки створює об'єкт). Розглянуті до цього приклади створення екземпляра (наприклад, лістинги 3.1, 3.2) використовують конструктор за замовчуванням.

Якщо у класі явно оголошено хоча б один конструктор, то компілятор не створює додаткового, і конструктор за замовчуванням стає недоступним. Тоді при спробі його використати виникне помилка компіляції. Якщо конструктор такого виду все ж потрібен у програмі, його слід описати самому, не вказуючи параметрів та створивши для нього порожнє тіло.

Конструктори можуть бути перевантаженими. Це дозволяє створити множину конструкторів, які можна використовувати для різних режимів

ініціалізації новоствореного об'єкта. Потрібний конструктор вказують при створенні екземпляра після оператора new.

Лістинг 3.17. Клас з двома конструкторами

```
1  class ClassX
2  {
3      public double X;
4
5      public ClassX()
6      {
7
8      }
9      public ClassX(double theX)
10     {
11         X = theX;
12     }
13 }
14
15 static void Main(string[] args)
16 {
17     ClassX cx1 = new ClassX();
18     ClassX cx2 = new ClassX(2.5);
19     Console.WriteLine("cx1.X = " + cx1.X);
20     Console.WriteLine("cx2.X = " + cx2.X);
21 }
```

У лістингу 3.17 приведено приклад оголошення класу з двома конструкторами. Клас містить одне поле X. Перший конструктор (рядки 5…8) імітує функції конструктора за замовчуванням. Він не має ні параметрів, ні жодної інструкції. При його виклику створиться екземпляр класу, а поле X буде ініціалізоване значенням за замовчуванням для типу double (нулем). Другий конструктор (рядки 9…12) має один параметр, через який передають значення для поля X. Передане значення присвоюється полю X. Зауважимо, що якби другого конструктора не було б, то можна не оголошувати і першого. Тоді клас мав би неявний конструктор за замовчуванням, який повністю ідентичний першому конструктору.

В методі Main приведено приклад створення двох екземплярів класу ClassX. Перший екземпляр (cx1) створюється викликом першого конструктора без параметрів. Його поле X відразу після створення дорівнюватиме 0 (рис. 3.11). Другий екземпляр (cx2) створюється за допомогою другого конструктора, – йому передається числове значення. Його поле X після створення дорівнюватиме 2.5 (рис. 3.11).

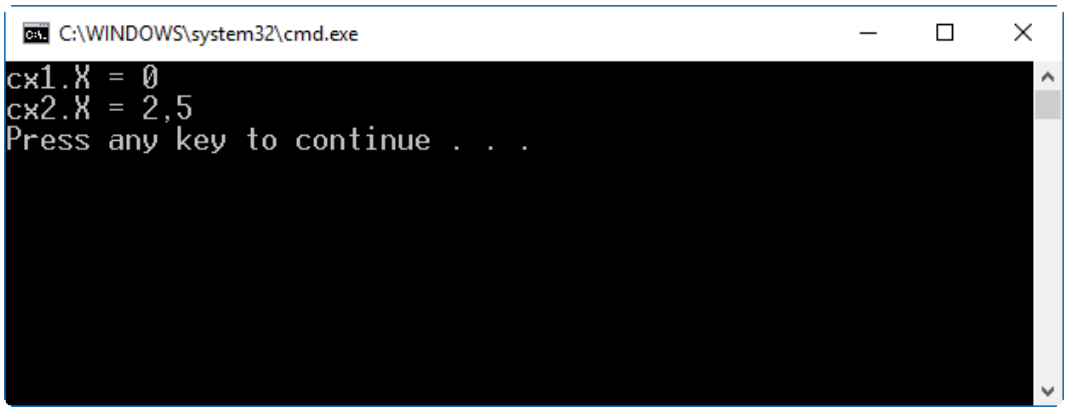


Рис. 3.11. Ілюстрація роботи двох конструкторів (лістинг 3.17)

Деструктори

Деструктори використовують для вивільнення вже не потрібних ресурсів, які використовував екземпляр класу. До таких ресурсів належать об'єкти у пам'яті, дескриптори відкритих файлів тощо. Деструктор викликається при знищенні екземпляру. Якщо програма використовує тільки ресурси .NET Framework, то рідко виникає необхідність створення деструктора, бо ресурсами керує сама інфраструктура.

Клас може мати тільки один деструктор. Назва деструктора співпадає з назвою класу, але має префікс "~" (символ "тильда"). Деструктор не має параметрів і модифікаторів доступу. В програмі на C# деструктор явно не викликають. Система сама його викликає у процесі збирання сміття, коли виявляє більше не потрібні програмі ресурси.

Ось приклад оголошення деструктора для класу ClassX з лістингу 3.17:

```
~ClassX()
{
    // Вивільнення ресурсів
}
```

Якщо у цьому немає необхідності, деструктор не потрібно оголошувати. Оголошення порожнього деструктора без потреби задіює зайві системні ресурси. У деструкторі слід вивільняти тільки зовнішні щодо .NET ресурси, якими володіє об'єкт (наприклад, ресурси, отримані через Windows API).

На відміну від C++, деструктор у C# не викликається негайно при знищенні об'єкта. Насправді, немає способу вияснити, коли саме він спрацює, оскільки його запускатиме підсистема .NET Framework. Тому, якщо екземпляр класу може містити сторонні ресурси, які слід вивільнити відразу при його знищенні, то цей функціонал слід окремо запрограмувати (наприклад, використавши інтерфейс IDisposable).

Властивості

Властивості – це елементи класу, які представляють дані екземпляра чи класу. Використання властивості дуже схоже на використання поля: операції читання та запису однакові. Властивість, як і поле, має назву та тип; їй можна присвоювати значення і з неї можна читати значення. Проте, на відміну від поля, вона є функціональним елементом класу і містить виконавчий код. Властивість не потребує пам'яті для розміщення даних.

Властивість є іменованою парою двох методів, які називають *методами доступу*:

- Метод доступу set використовують, щоб присвоїти властивості значення.
- Метод доступу get використовують, щоб прочитати її значення.

Властивість безпосередньо не має виділеного місця для зберігання даних. Методи доступу визначають тільки те, що треба зробити з переданими у властивість даними і які дані треба повернути при звертанні до властивості.

Оголошення властивості

Синтаксис оголошення властивості такий:

```
PropertyType PropertyName
{
    set { // Код для запису  значення властивості }
    get { // Код для читання значення властивості }
}
```

PropertyType задає тип властивості, а PropertyName – її назву. Ця частина оголошення властивості така ж, як і оголошення поля.

Методи доступу мають свій синтаксис і призначення. Для них явно не вказують ні тип результату, ні параметри. Метод set має один неявний параметр з назвою value, який містить значення, що присвоюють властивості. Цей метод не повертає результату. Метод get не має параметрів, але повертає значення, яке відповідає оголошеному типу властивості. Структуру оголошення властивості ілюструє рис. 3.12.

Неявний параметр з назвою value можна використовувати у тілі метода доступу set як звичайний параметр у звичайних методах. Метод get повинен закінчуватися інструкцією return зі значенням відповідного типу.

Методи get та set можна оголошувати в довільному порядку. Крім них, властивість не може мати ніяких інших методів.

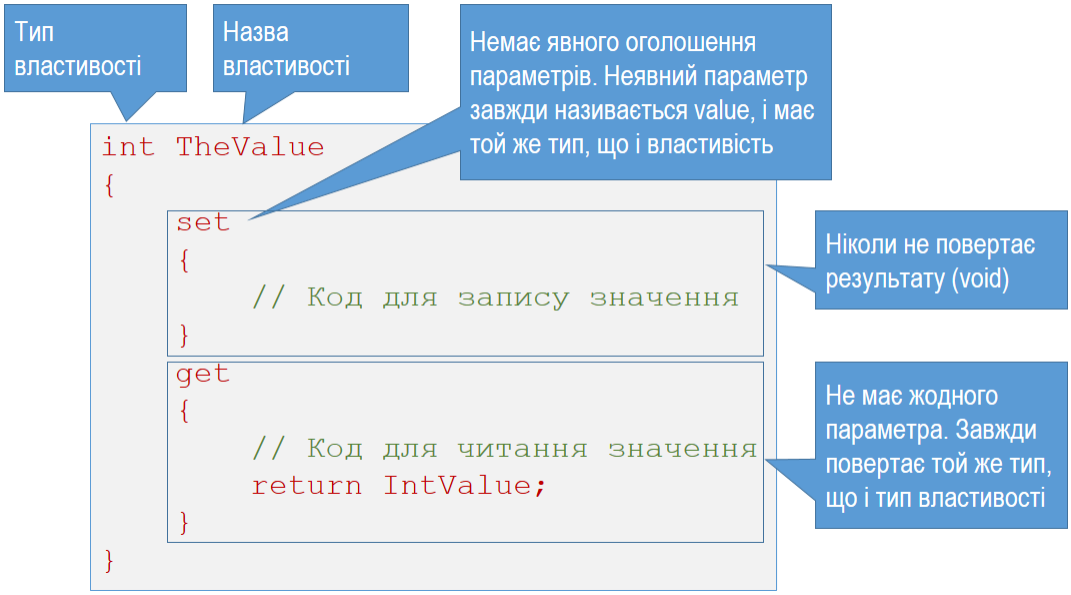


Рис. 3.12. Схема оголошення властивості

Лістинг 3.18 ілюструє оголошення класу ClassX з властивістю X. У приведеному прикладі метод доступу set (рядки 7…10) просто присвоює полю _x значення параметра value, а метод get (рядки 11…14) повертає значення поля _x. Зауважимо, що поле _x має той же тип – double – що і властивість X. Для поля _x не вказано модифікатора доступу, тому воно є приватним (до нього не буде доступу з-за меж класу), а властивість матиме повний доступ (використано модифікатор public).

Лістинг 3.18. Оголошення класу з властивістю

```
1  class ClassX
2  {
3      double _x;
4
5      public double X
6      {
7          set
8          {
9              _x = value;
10         }
11         get
12         {
13             return _x;
14         }
15     }
16 }
```

В загальному випадку властивості володіють рядом переваг порівняно з полями:

- Так як властивості є функціональними елементами, а не елементами даних (як поля), вони дозволяють провести перетворення над даними чи їх перевірку при записі та читанні.
- Семантично поля та властивості зовсім різні (хоча схожі у використанні). Ряд внутрішніх засобів C# та .NET Framework працюють тільки з властивостями.

Тому в загальному випадку слід надавати перевагу властивостям, особливо якщо клас входить до складу бібліотеки, яку використовують інші збірки.

Використання властивості

Використання властивостей аналогічне до використання полів. Так, якщо створити екземпляр класу ClassX (див. лістинг 3.18), то до властивості X можна звертатись як до звичайного поля: вказати назву об'єкта та назву властивості, розділивши їх крапкою:

```
ClassX cx = new ClassX();
cx.X = 200.5;
Console.WriteLine("Властивість X = {0}", cx.X);
```

Звертання до властивості

У першому рядку створено екземпляр cx класу ClassX. В другому рядку властивості X об’єкта cx присвоєно числове значення. Третій рядок містить звертання до метода WriteLine, який виводить на екран значення, що читається з властивості X.

Зауважимо, що напряму викликати методи доступу set та get не можна. Вони викликаються неявно, під час операцій читання/ запису.

Властивість часто пов'язують з полем – як у розглянутому класі ClassX. Загальною практикою є використання приватного поля у класі для зберігання даних і публічної властивості для забезпечення контрольованого доступу до властивості з-за меж класу (як у лістингу 3.18). Поле, пов'язане з властивістю, називають *підтримуючим полем* (*backing field*) властивості. Так, у розглянутому прикладі поле _x є підтримуючим полем властивості X.

Для кращого розуміння коду використовують кілька підходів до іменування властивостей та підтримуючих полів. Найчастіше використовують такі угоди щодо іменування (див. таблицю 2.2):

1. Для назв властивості і поля використовують один рядок, але для поля використовують стиль іменування camel (кожне слово, крім першого, в ідентифікаторі починають з великої літери), а для властивості – pascal (кожне слово починають з великої літери):

```
private int theData; // стиль camel
public int TheData   // стиль pascal
{
    get { return theData; }
    set { theData = value; }
}
```

2. Для назви властивості використовують стиль pascal, а для поля – стиль camel з початковим підкресленням.

```
private int _theData; // стиль camel з підкресленням
public int TheData    // стиль pascal
{
    get { return _theData; }
    set { _theData = value; }
}
```

Дотримання таких угод значно полегшує читання коду: функціонально пов'язані елементи мають схожі назви. У лістингу 3.19 використано перший варіант іменування, а в лістингу 3.18 – другий.

Звичайно, методи доступу не обмежуються простою передачею значення з/ до підтримуючого поля. У цих методах можна виконувати перевірку даних, які записують у властивість, або додаткове форматування даних при читанні тощо. Приклад у лістингу 3.19 демонструє оголошення властивості Percent, яка повинна містити значення відсотків (від 0 до 100). Підтримуючим полем для властивості є percent

Лістинг 3.19. Перевірка значення при запису у властивість

```
1  private int percent = 10;       // Поле
2  public  int Percent             // Властивість
3  {
4      set
5      {
6          percent = value > 100 ? 100 : value;
7          percent = percent < 0 ? 0 : percent;
8      }
9      get
10     {
11           return percent;
12     }
13 }
```

При запису значення у властивість в методі set виконується перевірка значення value, яке присвоюють властивості, на відповідність допустимому діапазону [0…100]. З цією метою два рази використано умовний оператор "? :" (рядки 6 і 7). Спочатку перевіряємо, чи значення параметра value більше від 100, і якщо так, то задаємо для поля верхню допустиму межу – 100. В іншому випадку зберігається поточне значення value. Після цього перевіряємо, чи отримане значення менше 0, і якщо так, то задаємо для поля нижню допустиму межу – 0.

Метод get просто повертає значення поля percent.

Підтримуючі поля для властивостей використовують часто, але інколи значення властивості обчислюють на основі інших даних. Для таких обчислюваних властивостей підтримуючі поля взагалі не потрібні – їх значення формуються кодом.

Властивості тільки для читання і тільки для запису

Один з методів доступу властивості (але не обидва нараз) при оголошенні можна не вказувати. Властивість з єдиним методом доступу має певну особливість:

- Властивість, яка має тільки метод get, є властивістю *тільки для читання* (*read-only*). Вона забезпечує отримання даних але не дозволяє задати їх.
- Властивість, яка має тільки метод set, є властивістю *тільки для запису* (*write-only*). Вона дозволяє записати дані, але не дасть їх отримати.

Хоча б один метод доступу повинен існувати. В іншому випадку виникне помилка компіляції.

У лістингу 3.20 приведено оголошення класу CurrencyExchange, який призначений для перетворення валют (рядки 1…12). Клас містить два поля: SummUSD (сюди записують суму в базовій валюті (доларах)) та CourseUAH (банківський курс гривні відносно базової валюти). Властивість тільки для читання SummUAH дозволяє отримати суму у гривнях. При читанні значення властивості відповідна сума розраховується у методі доступу get (рядок 9). Для коректної роботи перед читанням значення властивості SummUAH слід присвоїти значення полям SummUSD та CourseUAH.

У методі Main показано використання описаного класу. Спочатку створюємо екземпляр класу, потім у поля SummUSD та CourseUAH записуємо значення (суму та курс). Далі (у рядках 19…20) читаємо значення властивості SummUAH (ця сума розраховується автоматично) об'єкта ce, і виводимо її у вікно. Рис. 3.13 ілюструє роботу цієї програми.

Лістинг 3.20. Клас з властивістю тільки для читання

```
1  class CurrencyExchange
2  {
3      public decimal SummUSD;
4      public decimal CourseUAH;
5      public decimal SummUAH
6      {
7          get
8          {
9              return SummUSD * CourseUAH;
10         }
11     }
12 }
13
14 static void Main(string[] args)
15 {
16     CurrencyExchange ce = new CurrencyExchange();
17     ce.SummUSD = 10;
18     ce.CourseUAH = 25.322393M;
19     Console.WriteLine("USD = {0:0.000}, UAH = {1:0.000}",
20         ce.SummUSD, ce.SummUAH);
21 }
```

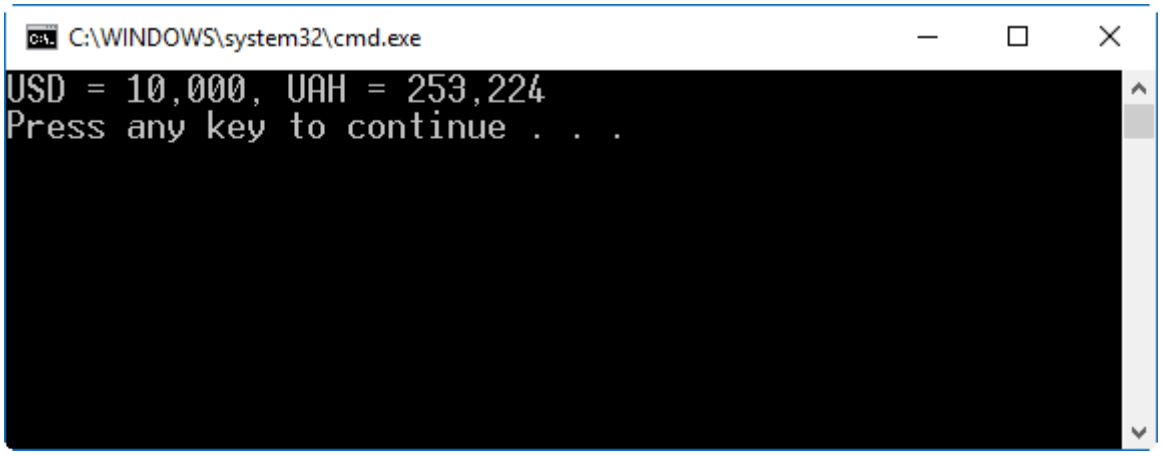


Рис. 3.13. Результат роботи програми з лістингу 3.20

Властивості тільки для читання можна оголошувати за допомогою синтаксису елементів класу- виразів. При цьому мають дотримуватися ті ж умови, що і для методів. Назву метода get під час оголошення не вказують.

Наприклад:

```
public double TheReadOnlyProperty => 123.45;
```

Властивості з автоматичною реалізацією

Для зберігання даних властивості підтримуючі поля використовують дуже часто. При цьому методи доступу нерідко просто транслюють значення у поле. Тому C# забезпечує *властивості з автоматичною реалізацією* (*auto-implemented properties*), для яких автоматично створюється приховане підтримуюче поле та методи доступу, на основі заданого типу властивості.

При оголошенні такої властивості у тілі кожного метода доступу вказують порожню інструкцію (крапку з комою). Ніякого додаткового коду методи доступу не повинні містити. Такий метод get діятиме як просте читання з прихованого підтримуючого поля, а метод set – як запис у нього.

Приховане підтримуюче поле властивості невидиме і недоступне для програмного коду. Тому властивості тільки для читання та тільки для запису у таких полях не можуть бути реалізовані і не підтримуються компілятором.

Наступний код демонструє оголошення властивості з автоматичною реалізацією у класі:

```
class TheClass
{
    public int TheValue { set; get; }
}
```

Порожня інструкція

Порожня інструкція

Властивості з автоматичною реалізацією забезпечують дуже зручний механізм для оголошення простих властивостей. Загалом при проектуванні класів властивостям варто надавати перевагу над полями (коли це виправдано). Так, у класі CurrencyExchange з лістингу 3.20 поля можна замінити на властивості, як показано у лістингу 3.21. Властивості SummUSD та CourseUAH описано як властивості з автоматичною реалізацією. Вони не мають ні явного підтримуючого поля, ні коду для методів доступу.

Лістинг 3.21. Допрацьований клас CurrencyExchange: тільки властивості

```
1  class CurrencyExchange
2  {
3      public decimal SummUSD
4      {
5          get; set;
6      }
7      public decimal CourseUAH
8      {
9          get; set;
10     }
11     public decimal SummUAH
12     {
13         get
```

Властивість з автоматичною реалізацією

Властивість з автоматичною реалізацією

```
14          {
15              return SummUSD * CourseUAH;
16          }
17      }
18  }
```

Властивості з автоматичною реалізацією можна також ініціалізувати значеннями відразу при оголошенні. При цьому значення, яким ініціалізують властивість, повинно бути відомим на момент компіляції коду:

```
public double Length { get; set; } = 100.0;
```

Властивість з автоматичною реалізацією може бути тільки для читання. Для цього при її оголошенні вказують лише метод get:

```
public string CompanyName { get; }
```

Модифікатори доступу для методів set та get

За замовчуванням обидва методи доступу мають однаковий рівень доступу, який визначений рівнем доступу властивості. Так, якщо властивість оголошена як public, то методи set та get також матимуть рівень public.

Але, при оголошенні властивості для методів доступу можна вказати власні модифікатори доступу. Наприклад, можна оголосити властивість з приватним методом set і публічним get (лістинг 3.22). Тоді записати дані у таку властивість можна буде тільки всередині самого класу, але читати дані з властивості зможуть всі. Цей підхід використовують доволі часто, оскільки він реалізує зручний механізм інкапсуляції даних.

Лістинг 3.22. Задання модифікатора доступу для метода set властивості

```
1  class Person
2  {
3      public int PersonCode { get; private set; }
4      public Person(int code)
5      {
6          PersonCode = code;
7      }
8  }
```

Модифікатор доступу

У лістингу 3.22 властивість PersonCode має саме таку реалізацію методів set та get (рядок 3). Значення для цієї властивості задає конструктор (рядки 4…7). Але з-за меж класу цього зробити неможливо.

Методу доступу властивості можна задавати модифікатор доступу тільки тоді, коли властивість має обидва методи – set і get. При цьому модифікатор можна задавати тільки для одного з них.

Модифікатор доступу для методу доступу повинен задавати строгіше обмеження, ніж модифікатор властивості. Тобто, для public-властивості можна задати private-метод set (чи get), але не навпаки.

Наступний порядок модифікаторів доступу відображує зростання обмеженнь: public > (protected чи internal) > private.

Статичні елементи класу та константи

Крім елементів, які належать окремому екземплярові, клас може містити елементи (поля, методи тощо), які пов'язані з класом в цілому. Їх називають *статичними* елементами класу і позначають при оголошенні модифікатором static. Для роботи зі статичними елементами не потрібно створювати об'єкт – екземпляр класу. Вони існують навіть тоді, коли не створено жодного екземпляра.

Статичні поля

Статичне поле для розміщення даних використовує ділянку пам'яті, яка спільна для всіх екземплярів класу. Таким чином, якщо один екземпляр змінює вміст цього поля, воно змінюється для всіх екземплярів. Якщо статичне поле при оголошенні ініціалізується, то вказане значення присвоюється йому до першого звертання.

У цьому прикладі оголошено клас, який містить одне поле екземпляра і одне статичне поле:

```
class MyClass
{
    int Field1;           // Поле екземпляра
    static int Field2;    // Статичне поле
}
```

При звертанні до статичного поля, на відміну від поля екземпляра, вказують не назву екземпляра, а назву класу (екземпляр може навіть не існувати):

```
MyClass.Field2 = 12;
```

Статичні функціональні елементи

Статичні функціональні елементи (зокрема, методи), як і статичні поля, не прив'язані до екземпляра класу. Їх можна викликати навіть тоді, коли екземпляр не існує. Відповідно, статичні функціональні елементи не можуть

звертатися до елементів екземпляра, а тільки до інших статичних елементів класу.

У лістингу 3.23 приведено програму, яка містить оголошення класу зі статичним полем та статичним методом, і продемонстровано роботу з ними. Клас ClassX оголошено у рядках 1…9. Він містить статичне поле X (рядок 3) і статичний метод DoubleX (рядки 5…8). Метод просто множить значення поля X на 2.

Лістинг 3.23. Клас зі статичними елементами

```
1  class ClassX
2  {
3      static public double X;
4      public double Y;
5      static public void DoubleX()
6      {
7          X *= 2;
8      }
9  }
10
11 static void Main(string[] args)
12 {
13     ClassX.X = 10;
14     Console.WriteLine("До виклику методу: {0}", ClassX.X);
15     ClassX.DoubleX();
16     Console.WriteLine("Після виклику методу: {0}", ClassX.X);
17 }
```

В методі Main спочатку присвоюємо статичному полю X значення 10 (при цьому звертаємось до нього через назву класу, а не через назву екземпляру). Далі виводимо поточне значення X на екран, викликаємо статичний метод DoubleX (рядок 15), і ще раз виводимо X. Зауважимо, що при звертанні до статичного метода вказуємо назву класу (ClassX), а не назву екземпляра (він не існує). Результат роботи цієї програми зображено на рис. 3.14.

Статичний метод DoubleX може звертатися тільки до статичних полів. Тому створити аналогічний статичний метод, який подвоюватиме значення поля Y, не можна – воно не є статичним.

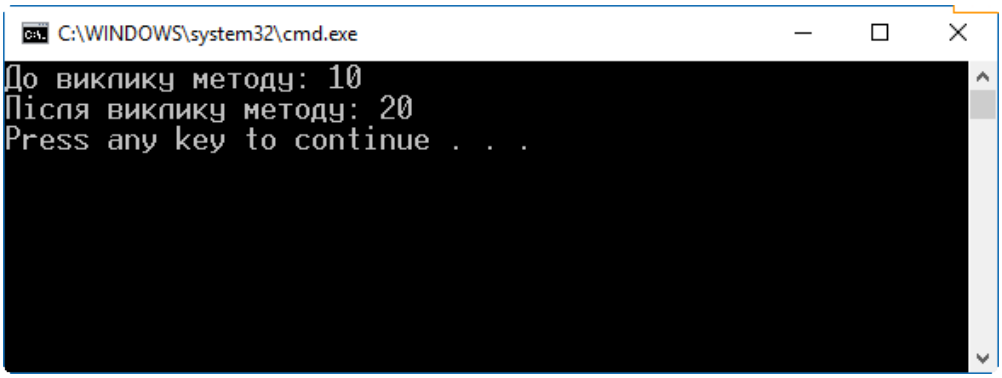


Рис. 3.14. Результат роботи програми з лістингу 3.23

Константи класу

Константи класу оголошують так само, як і локальні константи, але в оголошенні класу. Такі константи є елементами класу і до них можна звертатися через його назву. Ось приклад оголошення константи у класі:

```
class TheClass
{
    const int BasicNum = 100;      // Оголошення константи типу int
}
```

Ключове слово

Ініціалізація

Тут в тілі класу оголошено константу з назвою BasicNum цілого типу зі значенням 100.

Значення для константи обчислюється під час компіляції. Тому для її ініціалізації можна вказати вираз, який може бути розрахований у цей час, наприклад:

```
class TheClass
{
    const int BasicNum  = 100;
    const int BasicNum2 = BasicNum * 2; // Обчислюється на основі
}                                       // попереднього значення
```

Зауважимо, що, на відміну від C та C++, у C# неможливо оголосити глобальну константу. Кожна константа має бути частиною типу.

Константи класу особливі тим, що вони доступні як для всіх екземплярів класу, так і для статичних функціональних елементів, які не пов'язані з екземпляром і можуть викликатися тоді, коли жодного екземпляру не існує. У цьому константи класу схожі на статичні поля, але з ширшою зоною доступності. Але, на відміну від полів, константи не зберігаються у пам'яті, а просто підставляються компілятором у код.

Хоча константи у дії схожі на статичні поля, оголосити статичну константу не можна. Тому наступний код приведе до помилки компіляції:

```
static const int A = 50; // Помилка! Статичних констант немає
```

Статичні конструктори

Конструктори також можуть бути статичними. Тоді як конструктори екземпляра ініціалізують поля кожного нового екземпляру, статичний конструктор ініціалізує статичні елементи рівня класу.

Статичні елементи класу ініціалізуються перед першим звертанням до будь-якого статичного елемента і перед створенням екземпляру класу в програмі.

Статичний конструктор оголошують так само, як конструктор екземпляра, але дописують спочатку ключове слово static. Клас може містити тільки один статичний конструктор, і він не повинен мати параметрів. Модифікаторів доступу для статичного конструктора не вказують.

Клас може мати одночасно і статичний конструктор, і конструктор екземпляра. Як і інші статичні методи, статичний конструктор не має доступу до елементів екземпляра.

Явно у програмі не можна викликати статичний конструктор. Він викликається автоматично, до створення першого екземпляра класу і до першого використання будь-якого статичного елемента.

Статичні властивості

Властивість також можна оголосити статичною, використавши ключове слово static. Методи доступу статичної властивості, як і інші статичні методи, можуть звертатися виключно до інших статичних елементів класу. Елементи екземпляра для них недоступні. Звертатися до статичних властивостей можна тільки через назву класу, а не назву об'єкта.

Використання статичних методів класу

Якщо певний простір імен містить тип (клас) зі статичними елементами, для зручності його використання в коді призначена інструкція using static. До методів вказаного у ній класу можна звертатися коротше, не вказуючи назви самого класу. Цей функціонал було додано починаючи з версії C# 6.0.

Таким чином можна описати використання статичного класу Math, у якому реалізовано методи для різноманітних математичних функцій:

```
using static System.Math;
```

Тоді замість звичного звертання до елементів класу Math через його назву:

```
double d = Math.Sin(Math.PI) + Math.Cos(Math.PI);
```

можна викликати методи цього класу безпосередньо:

```
double d = Sin(PI) + Cos(PI);
```

Інколи такий підхід дозволяє суттєво спростити код програми, зробивши його легшим для розуміння.

Додаткові особливості елементів класу

Поля тільки для читання

Поле може виступати як "константа", значення якій присвоюється один раз під час виконання програми і після цього не змінюється. Таке поле оголошують з модифікатором readonly. Поле тільки для читання схоже на константу, але відрізняється від неї наступним:

- Константу слід ініціалізувати виключно при її оголошенні, а поле readonly, крім цього, можна ініціалізувати у конструкторі класу. Якщо поле є ще й статичним, то ініціалізацію слід робити у статичному конструкторі.
- Тоді як значення для константи має бути відомим на час компіляції, значення для поля readonly можна обчислювати під час виконання програми. Це дозволяє створювати поля з постійними, але наперед невідомими даними.

Ключове слово this

У функціональних елементах класу (методах екземпляру, конструкторах, методах доступу до властивостей тощо) часто потрібно звертатися до елементів поточного екземпляра. Ключове слово this використовують для посилання на поточний екземпляр класу.

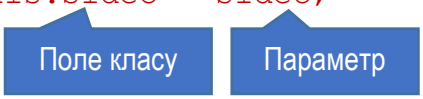
Так як статичні елементи класу не зв'язані з екземплярами (а з класом у цілому), ключове слово this у статичних функціональних елементах не можна використовувати.

Посилання на поточний екземпляр this використовують, щоб відрізнити елемент класу від інших даних (локальних змінних, елементів інших класів тощо). Також, це посилання можна передавати як параметр методам інших класів, якщо вони потребують посилання на даний об'єкт.

У лістингу 3.24 приведено оголошення класу Triangle. Він містить три поля: SideA, SideB та SideC.

Лістинг 3.24. Використання посилання на поточний екземпляр this

```
1  class Triangle
2  {
3      public double SideA;
4      public double SideB;
5      public double SideC;
6
7      public Triangle(double SideA, double SideB, double SideC)
8      {
9          this.SideA = SideA;
10         this.SideB = SideB;
11         this.SideC = SideC;
12     }
13 }
```



Значення для цих полів ініціалізуються конструктором класу (рядки 7…12). При цьому назви трьох параметрів конструктора співпадають з назвами відповідних полів класу, для яких передаються дані. Щоб відрізнити поле від параметра з такою ж назвою, у конструкторі перед назвою поля вказано ключове слово this (рядки 9…11). Якщо його не вказати, то значення параметра SideA буде присвоєне не полю SideA, а цьому самому параметру. Таким чином, поля об'єкта залишаться не ініціалізованими.

Часткові класи та часткові типи

Оголошення класу можна розбити на кілька частин. Такий клас називають *частковим* (*partial class*). Різні частини можуть бути розміщені в одному, або різних файлах. Але компілюються вони завжди в єдиний клас.

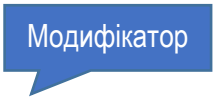
Частини класу описують елементи одного і того ж класу. Всі часткові оголошення класу позначають модифікатором partial перед ключовим словом class. У всьому іншому оголошення частини класу таке ж, як і оголошення звичайного класу.

Лістинг 3.25 містить приклад оголошення часткового класу. Перша частина (рядки 1…8) містить оголошення властивостей PersonCode та Person, а друга (рядки 12…15) – оголошення поля Name.

Під час компіляції з оголошених частин класу формується один цілісний клас. Його структура та зміст повністю відповідають тому випадку, коли б всі елементи класу були оголошені разом.

Лістинг 3.25. Оголошення часткового класу з двох частин

```
1  partial class Person        // Перша частина класу
2  {
3      public int PersonCode { get; private set; }
4      public Person(int code)
5      {
6          PersonCode = code;
7      }
8  }
9
10 ...
11
12 partial class Person        // Друга частина класу
13 {
14     public string Name;
15 }
```

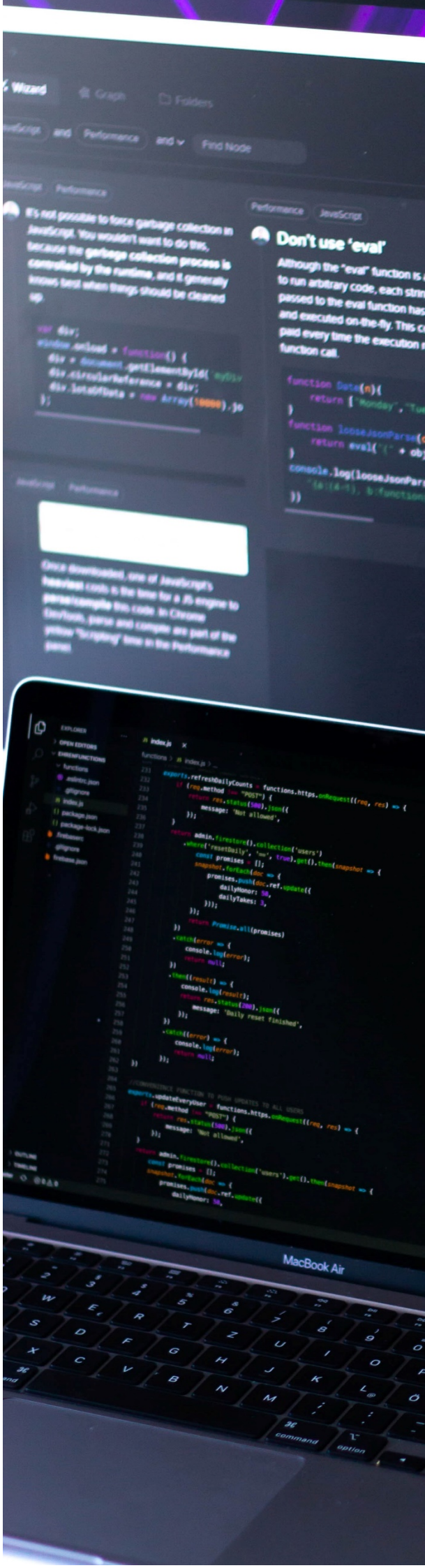


Microsoft Visual Studio використовує часткові класи, коли формує стандартні шаблони застосунків. Так, при створенні нового проекту Windows Forms, який представляє стандартний віконний застосунок з графічним інтерфейсом користувача, створюється два часткових класи для кожної форми. Одна частина містить опис компонентів для форми. Її не редагують вручну, оскільки ця частина оголошення перегенеровується Visual Studio при внесенні змін у дизайнері форм. Друга частина містить частковий клас, у якому ми програмуємо власну поведінку застосунку. Це ж стосується і форм WPF, ASP.NET тощо.

Крім часткових класів, C# підтримує часткові структури та інтерфейси.

Розділ 4. Використання класів

- Спадкування класів
- Використання успадкованих елементів
- Приховування елементів базового класу
- Посилання на базовий клас
- Віртуальні методи та їх заміщення
- Виконання конструкторів
- Модифікатори доступу класу
- Використання класів з інших збірок
- Модифікатори доступу елементів класу
- Абстрактні елементи та класи
- Запечатані класи
- Статичні класи



Спадкування класів

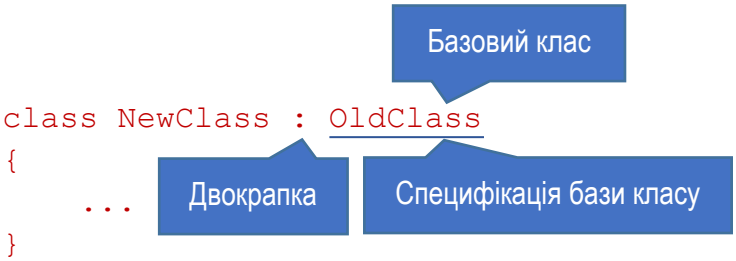
Спадкування – один із трьох фундаментальних принципів об'єктно-орієнтованого програмування. Саме завдяки йому можливе створення ієрархічних груп класів. Спадкування класів дозволяє оголосити новий клас, який містить весь функціонал іншого класу і розширює його. Вже існуючий клас, який називають *базовим класом*, стає основою для нового класу, який називають *похідним*. Похідний клас містить всі елементи базового класу, і крім цього, свої власні елементи.

Щоб оголосити похідний клас, при оголошенні після його назви через двокрапку вказують назву базового класу. В загальному випадку, крім базового класу, можна вказати інші типи (цей випадок розглянемо пізніше). Цю частину оголошення класу називають *специфікацією бази класу*.

Описуючи взаємовідносини між такими класами, говорять, що похідний клас є прямим *нащадком* вказаного базового класу. А базовий клас є його *предком*. Класи, пов'язані через механізм спадкування, формують *ієрархію* класів, яку часто зображують у вигляді дерева, з'єднуючи прямих предків і нащадків лініями.

Похідний клас містить елементи базового, і додатково – свої власні елементи. Тому кажуть, що похідний клас розширює функціонал базового класу. Похідний клас не може видалити ні одного успадкованого елемента, але може приховувати їх і оголошувати нові.

У цьому прикладі на основі існуючого класу OldClass оголошено новий клас із назвою NewClass:



Під час створення об'єкта похідного класу, перед виконанням інструкцій його конструктора викликається конструктор базового класу. Він створює об'єкт базового класу, який стає складовою частиною об'єкта похідного класу. Його називають частиною об'єкта, представленою базовим класом.

Всі без винятку класи (і оголошені в бібліотеках C#, і користувацькі класи), навіть якщо у їх оголошенні не задано специфікації бази, є похідними від єдиного базового класу – object. Клас object єдиний, який не є похідним від інших: він розташований на вершині ієрархії наслідування.

Класи, для яких не задано специфікації бази, неявно спадкують клас object. Опускання специфікації при оголошенні класу є скороченням синтаксису, так що два наступних оголошення є семантично еквівалентними:

```
class TheClass
{
    …
}
```

```
class TheClass : object
{
    …
}
```

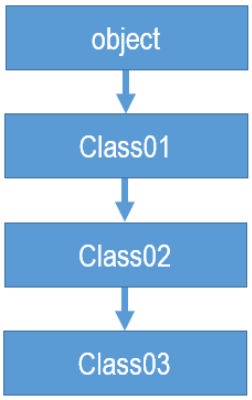
У C# кожен клас може мати тільки один базовий клас. Це називають *одиничним спадкуванням*. Інші мови (наприклад, C++, Delphi) підтримують *множинне спадкування*, при якому специфікація бази класу може містити більше одного класу. Але у C# для підвищення надійності коду від цього відмовилися.

Не зважаючи на те, що кожен клас може мати тільки одного прямого предка, немає обмежень на кількість рівнів спадкування. Тобто, базовий клас може в свою чергу бути нащадком іншого класу, той – ще якогось, і так далі, аж до вершини ієрархії – класу object.

Отже, по суті всі класи є похідними. Але терміни *базовий клас* та *похідний клас* використовують для пари класів, безпосередньо пов'язаних спадкуванням. На рис. 4.1 показано приклад ієрархії класів. Стрілки показують напрям від базового класу до похідного.

```
class Class01
{ ... }

class Class02 : Class01
{ ... }

class Class03 : Class02
{ ... }
```

а)



б)

Рис. 4.1. Кілька пов'язаних через спадкування класів (а) та їх дерево ієрархії (б)

Використання успадкованих елементів

До успадкованих елементів класу звертаються так, ніби вони оголошені у самому похідному класі (за винятком успадкованих конструкторів, але специфіку їх використання розглянемо далі). Розглянемо код лістингу 4.1, де як приклад приведено оголошення двох класів – Rectangle (базового) та ColoredRectangle (похідного).

Лістинг 4.1. Оголошення похідного класу

```
1  class Rectangle
2  {
3      public double sideA = 10, sideB = 20;
4      public void PrintBasicInfo()
5      {
6          Console.WriteLine("sideA={0}, sideB={1}",
7              sideA, sideB);
8      }
9  }
10
11 class ColoredRectangle : Rectangle
12 {
13     public string color = "жовтий";
14     public void PrintExtendedInfo()
15     {
16         Console.WriteLine("sideA={0}, sideB={1}, color={2}",
17             sideA, sideB, color);
18     }
19 }
20
21 static void Main(string[] args)
22 {
23     ColoredRectangle colRect = new ColoredRectangle();
24     colRect.sideA = 100;
25     colRect.PrintBasicInfo();
26     colRect.PrintExtendedInfo();
27 }
```

Базовий клас

Базовий клас оголошено у рядках 1…9. Він містить два поля (sideA, sideB), та один метод PrintBasicInfo, який виводить значення полів у вікно. Похідний клас оголошено у рядках 11…19. Він має додаткове поле color та метод PrintExtendedInfo. У методі Main створюємо екземпляр похідного класу colRect і присвоюємо його полю sideA (з базового класу) значення 100. У наступних двох рядках викликаємо метод PrintBasicInfo базового класу та метод PrintExtendedInfo похідного класу. Перший метод виводить значення полів базового класу, а другий – значення всіх полів (і базового, і похідного класів). Таким чином, похідний клас має доступ до елементів обох класів, що видно і з результату роботи програми на рис. 4.2.

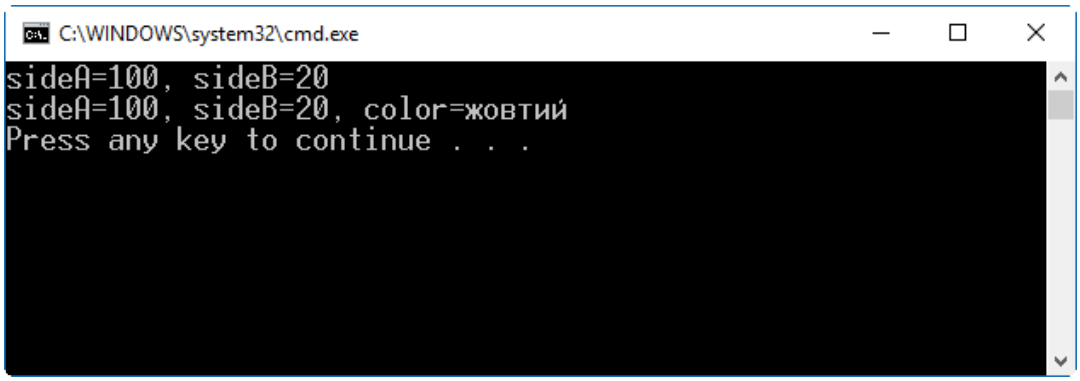


Рис. 4.2. Результат роботи програми з лістингу 4.1

Приховування елементів базового класу

У похідному класі не можна видалити жодного успадкованого елемента. Але можна приховати елемент базового класу новим елементом з такою ж назвою. Коли клас має успадкований метод, який не підходить для виконання його специфічного завдання, ми можемо оголосити інший метод з такою ж назвою і реалізувати в ньому потрібну поведінку.

Щоб приховати елемент даних, слід оголосити новий елемент того ж типу і з такою ж назвою. Щоб приховати функціональний елемент класу (наприклад, метод), потрібно оголосити новий функціональний елемент з такою самою сигнатурою (сигнатура містить назву елемента та послідовність і типи параметрів). Статичні елементи класу також можна приховувати.

Оголошуючи елемент класу, який приховує успадкований елемент, використовують модифікатор new. Він дозволяє явно вказати, що відбувається навмисне приховування. Без цього модифікатора компіляція програми буде також успішною, але компілятор згенерує попередження про неявне приховування успадкованого елемента.

Розглянемо лістинг 4.2. Він містить раніше розглянуті класи Rectangle та ColoredRectangle (див. лістинг 4.1). Базовий клас Rectangle залишився незмінним, а у похідному класі ColoredRectangle приховано поле sideA (рядок 13) та метод PrintBasicInfo (рядок 15). При їх оголошенні використано модифікатор new. Тепер кожен із класів містить метод з назвою PrintBasicInfo. Для базового класу він виводить на екран значення полів базового класу, а для похідного – значення полів похідного класу.

У рядку 25 "новому" полю sideA (з похідного класу) присвоюється значення 100. Далі викликається метод PrintBasicInfo (з похідного класу, оскільки змінна colRect має його тип), який виводить на екран значення полів. Зауважимо, що значення для поля sideA береться із похідного класу, а значення для поля sideB – з базового (рис. 4.3).

Лістинг 4.2. Приховування елементів базового класу

```
1  class Rectangle
2  {
3      public double sideA = 10, sideB = 20;
4      public void PrintBasicInfo()
5      {
6          Console.WriteLine("sideA={0}, sideB={1}",
7              sideA, sideB);
8      }
9  }
10
11 class ColoredRectangle : Rectangle
12 {
13     public new double sideA;           // Приховування поля
14     public string color = "жовтий";
15     public new void PrintBasicInfo()   // Приховування методу
16     {
17         Console.WriteLine("sideA={0}, sideB={1}, color={2}",
18             sideA, sideB, color);
19     }
20 }
21
22 static void Main(string[] args)
23 {
24     ColoredRectangle colRect = new ColoredRectangle();
25     colRect.sideA = 100;
26     colRect.PrintBasicInfo();
27 }
```

Модифікатор new

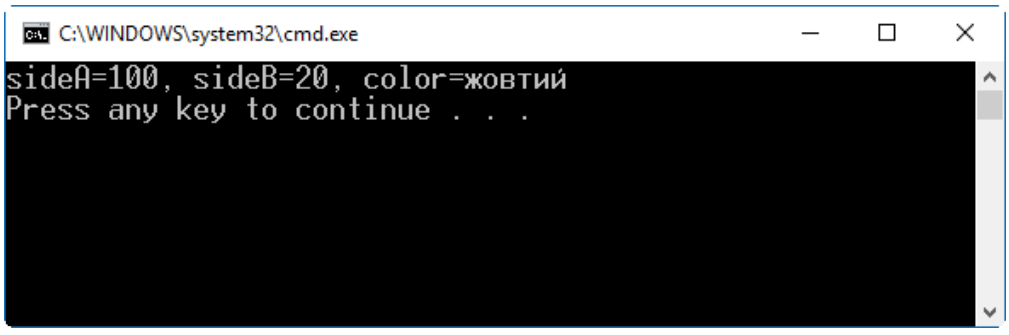


Рис. 4.3. Результат роботи програми з лістингу 4.2

Під час приховування елемента базового класу він насправді зберігається. Тому об'єкт colRect з попереднього лістингу має доступ як до поля sideA свого класу, так і до поля sideA успадкованого класу. Це два різних поля. Те ж стосується і методів: об'єкт colRect може викликати метод PrintBasicInfo з обох класів.

Для доступу до прихованого елемента базового класу з коду похідного класу використовують ключове слово base, після якого через крапку вказують назву потрібного елемента базового класу.

Зокрема, звернутись до поля базового класу sideA з методу похідного класу можна так:

```
base.sideA = 300;
```

Ключове слово

Аналогічно можна викликати метод базового класу:

```
base.PrintBasicInfo();
```

Ключове слово

Змінимо у лістингу 4.2 код методу PrintBasicInfo, замінивши поля для виводу:

```
public new void PrintBasicInfo()
{
    Console.WriteLine("base.sideA={0}, sideA={1}",
        base.sideA, sideA);
}
```

Тепер виводитимуться значення обох полів: і базового, і похідного класів (рис. 4.4). Зауважимо, що ці значення різні. Поле базового класу містить значення 10, задане йому при ініціалізації (рядок 3 лістингу 4.2), а полю похідного класу значення 100 присвоюється у коді методу Main (рядок 25).

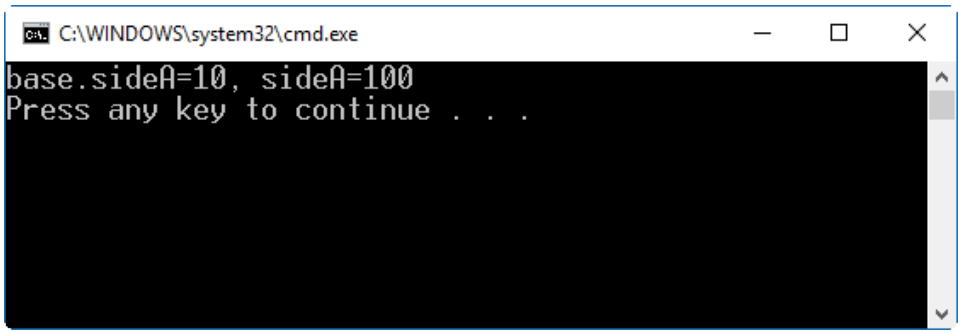


Рис. 4.4. Доступ до прихованого поля базового класу

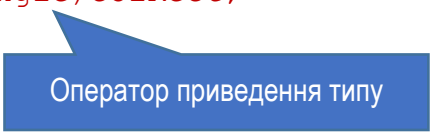
Посилання на базовий клас

Екземпляр похідного класу містить екземпляр базового класу та додаткові елементи свого класу. Посилання на екземпляр похідного класу вказує на цілий об'єкт, включаючи частину, отриману від базового класу. Тому, маючи посилання на похідний об'єкт, можна отримати посилання на включений у нього об'єкт базового класу. З цією метою використовують *приведення типу*

посилання до типу базового класу за допомогою *оператора приведення типу*. Цей оператор складається із розміщеної в круглих дужках назви типу (класу), до якого слід привести дані.

Для отримання посилання на об'єкт базового класу перед посиланням розміщують оператор приведення типу (тобто оточену круглими дужками назву базового класу). Щоб отримати посилання на об'єкт базового класу, який входить до складу об'єкта colRect з лістингу 4.2, можна використати такий код:

```
Rectangle baseObj = (Rectangle)colRect;
baseObj.sideA = 100;
baseObj.PrintBasicInfo();
```



У першому рядку оголошено змінну базового класу baseObj і присвоєно їй посилання на об'єкт базового класу, який входить до складу екземпляру похідного класу colRect. У другому рядку, використовуючи отримане посилання, полю sideA базового класу (адже baseObj вказує на "базову" частину) присвоюється значення 100. Третій рядок містить виклик метода PrintBasicInfo базового класу. Зауважимо, що об'єкт colRect з лістингу 4.2 містить два методи PrintBasicInfo (і два поля sideA): один отриманий з базового класу, інший оголошено в класі самого об'єкта. Але, отримавши через приведення типу посилання на базову частку об'єкта, ми викликаємо елементи саме базового класу.

За допомогою посилання на частину об'єкта, отриману від базового класу, неможливо отримати доступ до іншої частини об'єкта: це посилання "бачить" тільки об'єкт базового класу.

Для приведення типу об'єкта одного класу до іншого, крім оператора приведення типу можна використовувати оператор as. Синтаксис його використання такий:

```
derivedObj as BaseClass
```

Тут derivedObj – змінна-посилання на об'єкт похідного класу, а BaseClass – назва базового класу. Вказаний вираз поверне посилання на частину об'єкта derivedObj, представлену класом BaseClass. При цьому клас BaseClass повинен бути одним з предків класу об'єкта derivedObj в ієрархії класів.

Нижче (у першому рядку) приведено приклад використання оператора as для отримання посилання на базову частину об'єкта:

```
Rectangle baseObj = colRect as Rectangle;
Rectangle baseObj = (Rectangle)colRect;
```

Другий рядок виконує те ж саме за допомогою оператора приведення типу. Ці дві інструкції виконують однакові дії – отримують посилання на екземпляр базового класу. Але, якщо в об'єкті неможливо виділити частину,

представлену вказаним базовим класом, то при використанні оператора приведення виникне виняткова ситуація. У той же час вираз на основі оператора as поверне значення null. Оператор as, на відміну від оператора приведення типу, можна використовувати тільки для перетворень, пов'язаних з посиланнями.

Віртуальні методи та їх заміщення

Перед цим ми бачили, що посилаючись на об'єкт базового класу, успадкований похідним об'єктом, ми отримували звертання тільки до елементів базового класу. Так було у всіх методах, які ми розглядали у прикладах до цього. Але класи містять механізм, який розширює їх функціональність. *Віртуальні методи* дозволяють використовувати посилання на базовий клас для звертання до елементів похідних класів. При виклику віртуальних методів використовують не оголошений на час компіляції клас змінної, а фактичний клас об'єкта під час виконання.

Ми вже знаємо, що у кожному похідному об'єкті є базова частина, яка представлена його базовим класом. Якщо викликати віртуальний метод через базову частину об'єкта, то в класі, який відповідає фактичному типу об'єкта, підшуковуватиметься відповідний заміщуючий метод. І, якщо він є, то буде виконано його, а не метод базового класу. У цьому полягає важлива особливість віртуальних методів, яку часто використовують при розробці ієрархії взаємопов'язаних класів. Таким чином об'єкт, представлений змінною одного класу, веде себе як об'єкт іншого класу.

Саме за допомогою віртуальних методів та їх заміщення забезпечується реалізація поліморфізму – одного з трьох головних принципів об'єктно-орієнтованого програмування. Поліморфізм забезпечує "багатоформенність" об'єкта, імітуючи зміну його "форми" (типу) залежно від умов виконання.

Щоб оголосити віртуальний та заміщуючий методи у базовому та похідному класах, потрібно забезпечити виконання таких умов:

- Метод похідного класу та метод базового класу повинні мати однакову сигнатуру та тип.
- Метод базового класу слід позначити як virtual (віртуальний метод).
- Метод похідного класу потрібно позначити як override (він заміщує віртуальний метод).

Наступний приклад демонструє оголошення віртуального методу у базовому класі та його заміщення в похідному класі за допомогою модифікаторів virtual та override:

```
class TheBaseClass                          // Базовий клас
{
    virtual public void Print()             // Віртуальний метод
    ...
```

Ключове слово virtual

```
class TheDerivedClass : TheBaseClass  // Похідний клас
{
    override public void Print()        // Заміщуючий метод
    ...
```

Ключове слово override

Розглянемо приклад, який демонструє поведінку віртуального методу при використанні різних посилань на об'єкт (лістинг 4.3). Базовий клас ClassA оголошено у рядках 1…7. Він містить один віртуальний метод PrintLine, який виводить у консольне вікно фразу, що пояснює, який саме метод викликано. Метод оголошено з модифікатором virtual (рядок 3) – саме це робить його віртуальним. Похідний клас ClassB оголошено у рядках 9…15. Базовим класом для нього є ClassA (рядок 9).

Похідний клас також має один метод PrintLine, який заміщує віртуальний метод базового класу з такою ж назвою. Щоб вказати, що реалізовано заміщення, використано модифікатор override (рядок 11). Заміщуючий метод виводить у вікно фразу, яка інформує, що викликано саме його.

У методі Main реалізовано чотири випадки виклику віртуального методу (вони позначені відповідними коментарями). Розглянемо кожен з них детальніше.

1. Виклик віртуального методу об'єкта базового класу. У рядку 19 оголошено змінну базового класу A та створено екземпляр цього класу. У рядку 21 викликається метод PrintLine. При цьому очікувано (як і в усіх попередніх прикладах) спрацьовує метод базового класу (рис. 4.5, перший рядок), оскілки змінна A представляє об'єкт базового класу. Якщо спробувати звернутися через змінну A до методу похідного класу, виконавши приведення типу об'єкта A до похідного класу (закоментований рядок 22), то під час виконання виникне помилка "Object reference not set to an instance of an object". Це зрозуміло: об'єкт A не має ніяких відомостей про похідний клас і не може звертатися до його елементів.
2. Виклик заміщуючого методу в об'єкті похідного класу. У рядку 25 оголошено змінну B похідного класу та створено екземпляр цього класу. Далі викликається метод PrintLine (рядок 27). В даному випадку спрацює метод похідного класу (рис. 4.5, другий рядок). Це також очікувано: об'єкт похідного класу викликає свій власний метод.

Лістинг 4.3. Демонстрація роботи віртуального методу

```
1  class ClassA                                    // Базовий клас
2  {
3      public virtual void PrintLine()
4      {
5          Console.WriteLine("Метод базового класу");
6      }
7  }
8
9  class ClassB : ClassA                           // Похідний клас
10 {
11     public override void PrintLine()
12     {
13         Console.WriteLine("Метод похідного класу");
14     }
15 }
16
17 static void Main(string[] args)
18 {   // 1)
19     ClassA A = new ClassA();
20     Console.Write("Об'єкт базового класу -> ");
21     A.PrintLine();
22     //(A as ClassB).PrintLine(); // !!! Помилка !!!
23
24     // 2)
25     ClassB B = new ClassB();
26     Console.Write("Об'єкт похідного класу -> ");
27     B.PrintLine();
28
29     Console.WriteLine("-----------------------------------");
30     // 3)
31     ClassA AinB = (ClassA)B;
32     Console.Write("Базова частка в об'єкті " +
33         "похідного класу -> ");
34     AinB.PrintLine();
35
36     // 4)
37     ClassA BfromA = new ClassB();
38     Console.Write("Об'єкт похідного класу, " +
39         "створений через змінну базового -> ");
40     BfromA.PrintLine();
41 }
```

Віртуальний метод

Заміщуючий метод

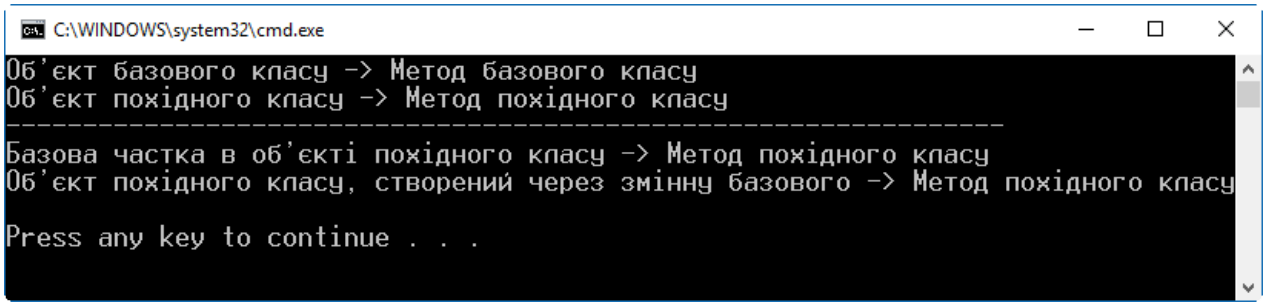


Рис. 4.5. Результат роботи програми з лістингу 4.3. Демонстрація роботи віртуального та заміщуючого методів

3. Виклик віртуального методу базової частки об'єкта похідного класу. На відміну від двох попередніх випадків, коли віртуальний метод працював аналогічно до звичайного (не віртуального), випадки 3 і 4 демонструють особливості роботи віртуальних методів. Отримаємо базову частку AinB об'єкта похідного класу, створеного при розгляді попереднього пункту (рядок 31), і викличемо її метод PrintLine (рядок 34). Для звичайних методів це привело б до виклику методу базового класу. Але, оскільки метод PrintLine базової частки об'єкта є віртуальним, а сам об'єкт належить похідному класові і містить заміщуючий метод, то виконається метод похідного класу, а не базового (рис. 4.5, перший рядок після горизонтального розділювача). Якщо ж з оголошень методів забрати модифікатори virtual та override, перетворивши ці методи з віртуальних у звичайні, то виклик метода PrintLine у рядку 34 приведе до спрацювання методу базового класу, адже AinB є об'єктом базового типу.
4. Об'єкт похідного класу, створений через змінну базового. Цей випадок по суті не відрізняється від попереднього, але демонструє елегантніший (і загальноприйнятий) спосіб реалізації поліморфізму. У рядку 37 оголошено змінну **базового** класу BfromA (зліва від оператора "="), створено екземпляр **похідного** класу (справа від оператора "="), і присвоєно оголошеній змінній посилання на базову частину новоствореного похідного об'єкта. Якщо тепер викликати метод PrintLine (рядок 40), то спрацює, як і в попередньому випадку, метод похідного класу (рис. 4.5, другий рядок після горизонтального розділювача), не зважаючи на те, що і змінна BfromA, і об'єкт, який вона безпосередньо представляє, мають базовий клас.

Оголошуючи віртуальні методи та заміщуючи їх, слід пам'ятати, що заміщуваний та заміщуючий методи повинні мати однакові модифікатори доступу. Якщо, наприклад, віртуальний метод є приватним, то заміщуючий також повинен бути приватним.

Не можна заміщувати статичний метод, або не віртуальний метод (оголошений без модифікатора virtual).

Крім методів, у класі можна заміщувати також властивості, події та індексатори.

Заміщення заміщуючих методів

Методи можна заміщувати на будь-якому рівні ієрархії класів. Тобто, заміщуючий метод (позначений модифікатором override) може бути заміщений іншим методом в іншому похідному класі. Його також позначають модифікатором override. Якщо викликається віртуальний метод, то серед ієрархії класів завжди підшуковується його найновіша для класу об'єкта заміщуюча версія. Коли в одному з похідних класів є метод з такою ж сигнатурою, як і віртуальний метод базового класу, але не позначений модифікатором override, то реалізується не механізм заміщення, а механізм приховування, розглянутий раніше.

Розглянемо приклад у лістингу 4.4. Тут оголошено чотири класи: ClassA, ClassB, ClassC і ClassD. Базовим для кожного з них (крім першого) є попередній клас у списку. Всі вони мають метод PrintLine, який виводить на екран повідомлення, з якого він класу. В першому класі метод PrintLine оголошено віртуальним (virtual), у всіх наступних – заміщуючим (override).

Лістинг 4.4. Заміщення методів

```
1  class ClassA
2  {
3      public virtual void PrintLine()
4      { Console.WriteLine("Метод класу A"); }
5  }
6
7  class ClassB : ClassA
8  {
9      public override void PrintLine()
10     { Console.WriteLine("Метод класу B"); }
11 }
12
13 class ClassC : ClassB
14 {
15     public override void PrintLine()
16     { Console.WriteLine("Метод класу C"); }
17 }
18
19 class ClassD : ClassC
20 {
21     public override void PrintLine()
22     { Console.WriteLine("Метод класу D"); }
```

```
23  }
24
25  static void Main(string[] args)
26  {
27      ClassA ab = new ClassB();
28      ab.PrintLine();
29
30      ClassA ac = new ClassC();
31      ac.PrintLine();
32
33      ClassB bc = new ClassC();
34      bc.PrintLine();
35  }
```

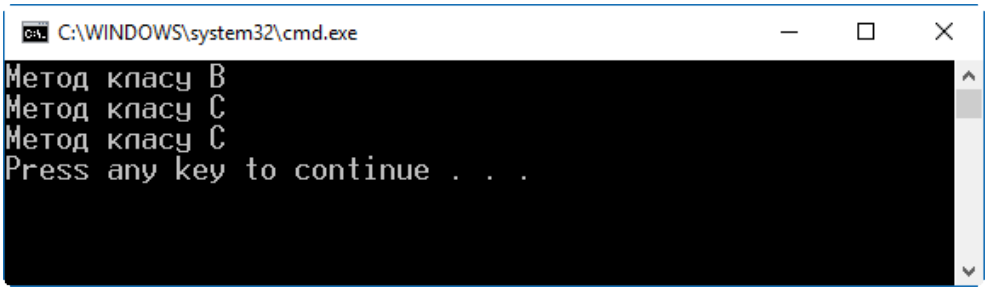


Рис. 4.6. Результат роботи програми з лістингу 4.4. Демонстрація заміщення методів у похідних класах

В методі Main у рядку 27 створюємо екземпляр класу B і отримуємо посилання на його базову частку класу A, з якої викликаємо віртуальний метод. Це приведе до виклику методу з класу B (рис. 4.6), оскільки він для екземпляра класу B є найновішою заміщуючою версією методу.

У рядку 30 створюємо екземпляр класу C і отримуємо посилання на його базову частку класу A, з якої знову викликаємо віртуальний метод. Тепер буде викликано метод з класу C, оскільки він для екземпляра класу C є найновішою заміщуючою версією методу.

В рядку 33 створюємо екземпляр класу C отримуємо посилання на його базову частку класу B, і викликаємо віртуальний метод. Це знову приведе до виклику методу з класу C, як і в попередньому випадку. Хоча тип базової частки для одного і того ж об'єкта інший, найновіша заміщуюча версія методу та ж.

Заміщення властивостей

Властивості заміщують так само, як і методи. У базовому класі властивість позначають як віртуальну модифікатором virtual, а в похідному – як заміщуючу модифікатором override. Приклад заміщення віртуальної властивості

приведено в лістингу 4.5. Принцип дії віртуальних властивостей такий самий, як і для віртуальних методів: підшуковується найновіша заміщуюча версія для класу екземпляра. Результат виконання приведеної програми приведено на рис. 4.7.

В обох випадках (і для об'єкта похідного класу, і для його базової частки) повертається значення властивості з похідного класу.

Лістинг 4.5. Заміщення властивостей

```
1  class ClassA
2  {
3      private int _field1 = 10;
4      virtual public int TheProperty
5      {
6          get { return _field1; }
7      }
8  }
9
10 class ClassB : ClassA
11 {
12     private int _field2 = 20;
13     override public int TheProperty
14     {
15         get { return _field2; }
16     }
17 }
18
19 static void Main(string[] args)
20 {
21     ClassB b = new ClassB();
22     ClassA a = b as ClassA;
23     Console.WriteLine(b.TheProperty);
24     Console.WriteLine(a.TheProperty);
25 }
```

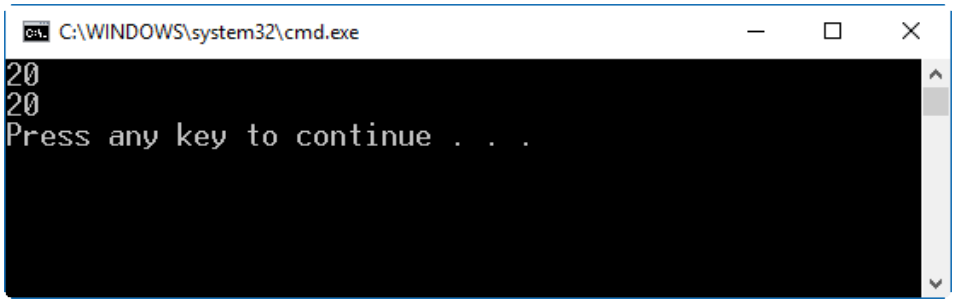


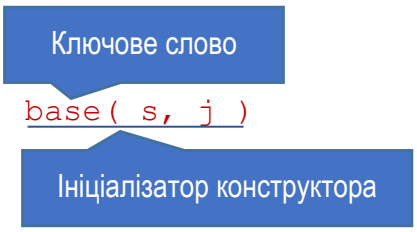
Рис. 4.7. Результат роботи програми з лістингу 4.5. Демонстрація заміщення властивостей у похідних класах

Виконання конструкторів

Коли створюється екземпляр похідного класу, то перед виконанням його конструктора викликається конструктор базового класу. Він створює об'єкт базового класу, який стає складовою частиною об'єкта похідного класу. Якщо в ієрархії спадкування є більше двох класів, то виконуються конструктори всіх попередніх класів, починаючи з найнижчого рівня. Таким чином, результуючий об'єкт, подібно до матрьошки, міситиме у собі об'єкти всіх попередніх успадкованих базових класів.

При цьому за замовчуванням використовується конструктор без параметрів. Але конструктори можуть бути перевантаженими, або базовий клас може мати їх кілька. Щоб явно вказати, який саме конструктор слід використовувати для створення екземпляра базового класу, використовують ініціалізатор конструктора на основі ключового слова base.

Потрібний конструктор базового класу вказують так: після заголовку конструктора похідного класу ставлять двокрапку, далі розміщують ключове слово base і перелік параметрів для конструктора базового класу в дужках:

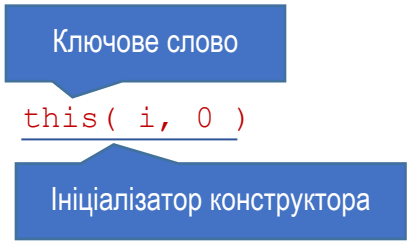



```
public TheDerivedClass(int j, string s) : base( s, j )
{
    ...
```

У даному випадку ініціалізатор вказує, що при виклику конструктора TheDerivedClass похідного класу для конструювання його базової частини слід викликати конструктор базового класу, який має два параметри, перший з яких є рядкового типу, а другий – цілого. Самі параметри (та інформація про їх тип) беруться з переліку параметрів конструктора похідного класу.

Якщо для конструктора не вказано ініціалізатора, то для ініціалізації частини об'єкта, представленої базовим класом, використовуватиметься конструктор базового класу без параметрів. Він повинен існувати, інакше виникне помилка компіляції.

Ще один вид ініціалізатора для конструктора вказує не конструктор базового класу, а конструктор цього ж класу, який має виконуватися перед виконанням даного конструктора. Такий ініціалізатор має структуру, аналогічну до попередньо розглянутого, але замість ключового слова base використовують ключове слово this:




```
public TheClass(int i) : this( i, 0 )
{
    ...
```

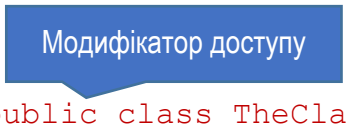
Даний запис означає, що при виклику однопараметричного конструктора спочатку має виконатися двопараметричний конструктор цього ж класу, в якого обидва параметри – цілі числа. Для першого параметра передається значення параметра і однопараметричного конструктора, а для другого параметра – значення 0.

Така ініціалізація конструктора може бути корисна, якщо при проектуванні класів вникає необхідність у кількох конструкторах, кожен з яких під час ініціалізації нового об'єкта має виконати якісь спільні для всіх дії. Такі спільні інструкції доцільно виділити в окремий конструктор, щоб не повторювати один і той же код. Тоді цей спільний для всіх конструктор слід вказати в ініціалізаторі інших конструкторів.

Модифікатори доступу класу

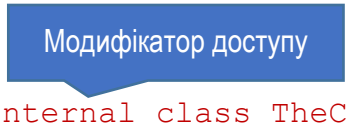
Класи можуть використовуватися іншими класами системи .NET Framework. Щоб вказати рівень доступності класу, його оголошення доповнюють модифікатором доступу. Існує два рівні доступності класів: загальний (public) та внутрішній (internal):

- Клас, позначений як public, буде доступний з коду будь-якої збірки у системі. Якщо потрібно, щоб клас був видимий іншим збіркам, при його оголошенні слід вказати модифікатор public:




```
public class TheClass { ... }
```

- Клас, позначений як internal, може використовуватися виключно класами збірки, у якій він оголошений. Цей рівень доступності використовується за замовчуванням. Тому, якщо при оголошенні класу явно не задано модифікатора доступу, то інші збірки його не бачитимуть. Але можна також явно вказати, що клас буде внутрішнім:




```
internal class TheClass { ... }
```

Використання класів з інших збірок

Класи з інших збірок можна використовувати так, ніби вони оголошені в поточній збірці. Для цього потрібно забезпечити виконання таких умов:

- Клас повинен мати загальний рівень доступності (позначений як public).
- У проект Visual Studio потрібно додати посилання на збірку, яка містить клас. Це можна зробити у розділі References вікна Solution Explorer. Слід вибрати команду Add Reference… з контекстного меню, викликаного на вузлі References, далі натиснути кнопку Browse… і вибрати потрібну збірку. Всі її типи з загальним рівнем доступності будуть видимі в поточному проекті.

Щоб було зручніше використовувати класи та інші типи зі збірки, не вказуючи їх повні назви, на початку файлу з текстом програми розміщують директиву using та вказують простір імен під'єднаної збірки. Використання класів з під'єднаної збірки нічим не відрізняється від використання власних класів.

Зауважимо, що додавання посилання на іншу збірку та використання директиви using – різні речі. Додане посилання показує, де (у якому файлі) компілятор має шукати типи, а директива using дозволяє посилатися на типи за короткими назвами.

Збіркою, з якої можна використовувати клас, може бути як динамічна бібліотека (dynamic-link library, dll) з розширенням .dll, так і виконавчий файл (з розширенням .exe).

Модифікатори доступу елементів класу

Кожен елемент класу може мати свій власний модифікатор доступу, який задаватиме рівень його видимості в системі. Ми вже розглядали два рівні: public (надає повний доступ до елемента) та private (робить елемент доступним тільки в межах класу, в якому він оголошений). За замовчуванням, якщо не вказати ніякого модифікатора доступу, використовується приватний рівень.

Для елементів класу можна задати такі модифікатори доступу:

- public
- private
- protected
- internal
- protected internal

Щоб вказати потрібний рівень доступності елемента класу, при його оголошенні вказують відповідний модифікатор, як вже було розглянуто на прикладі модифікаторів для загального та приватного доступу.

Припустимо, у нас є клас A. Тоді інший клас B може мати (чи не мати) доступ до елементів класу A, залежно від таких умов:

- Чи клас B похідний від класу A, чи ні.
- Чи клас B оголошено у тій же збірці, що й A, чи в іншій збірці.

Розглянемо вплив кожного модифікатора доступу у всіх комбінаціях вказаних умов.

Модифікатор public

Цей рівень не має обмежень. Всі класи всередині збірки та за її межами мають доступ до такого елемента. Це ілюструє схема на рис. 4.8.

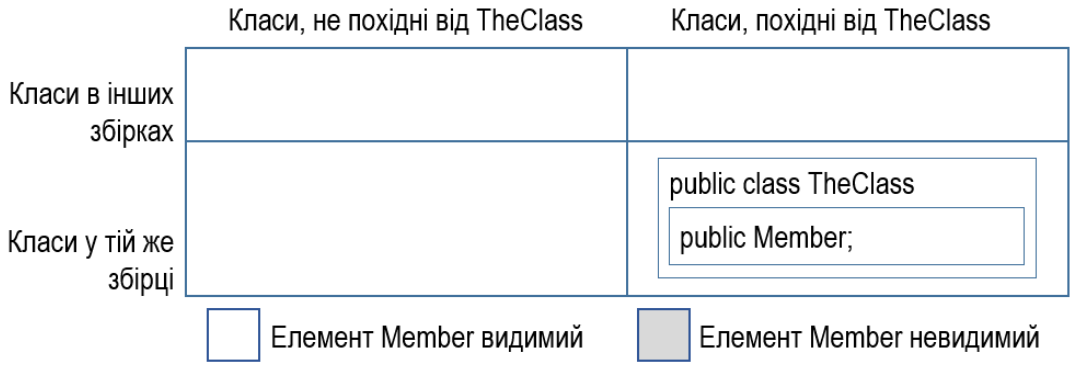


Рис. 4.8. Доступність елементів з модифікатором public

Модифікатор private

Цей рівень накладає найбільше обмежень. Приватний елемент видимий виключно у своєму класі. Він не доступний навіть у похідному класі тієї ж збірки. Область доступності елемента класу з модифікатором private ілюструє рис. 4.9.

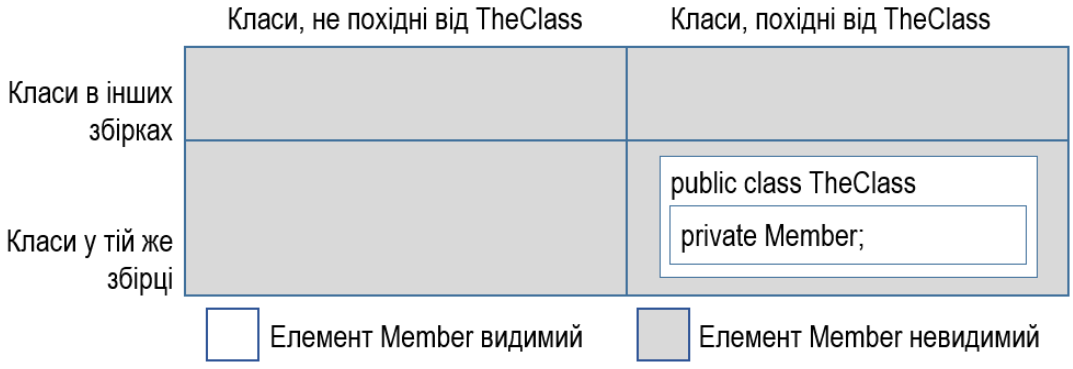


Рис. 4.9. Доступність елементів з модифікатором private

Модифікатор protected

Цей рівень робить елемент доступним у всіх похідних класах. Навіть похідні класи з інших збірок також мають доступ до такого елемента (рис. 4.10).

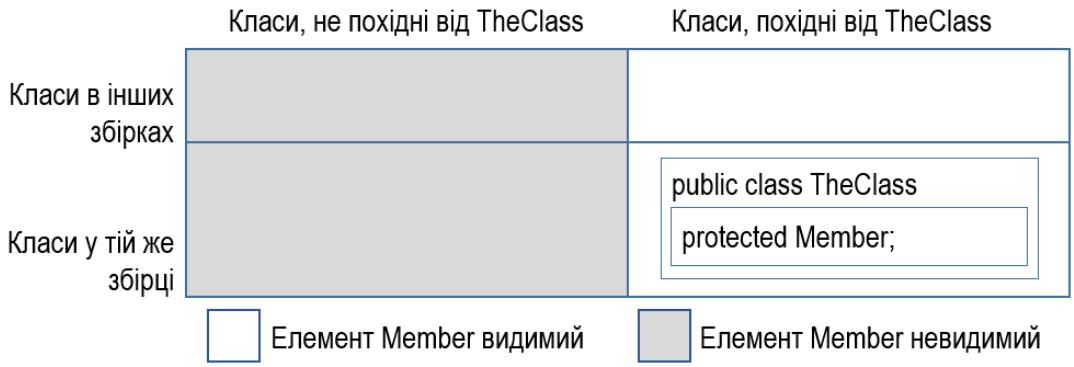


Рис. 4.10. Доступність елементів з модифікатором protected

Модифікатор internal

Елементи, позначені як внутрішні (internal), видимі для всіх класів збірки. Але класи інших збірок доступу до них не мають. Навіть похідні класи в інших збірках не бачитимуть такого елемента (рис. 4.11).

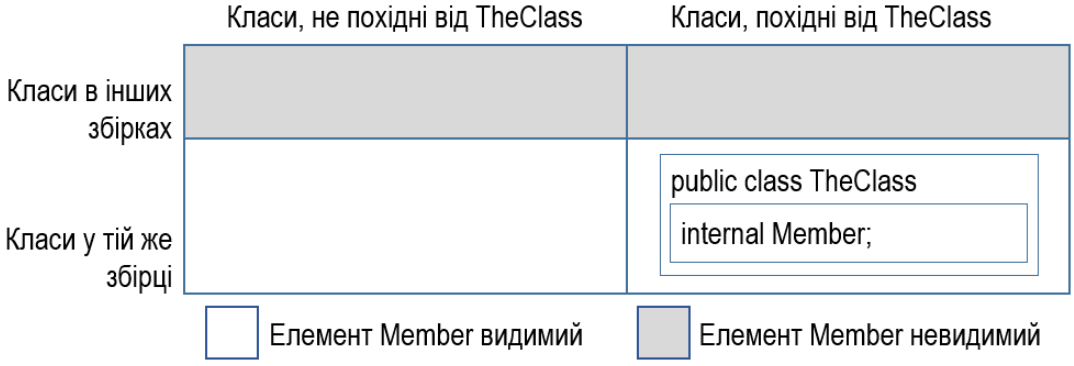


Рис. 4.11. Доступність елементів з модифікатором internal

Модифікатор protected internal

Елементи, позначені як protected internal, видимі для всіх похідних класів та всіх класів збірки. Область видимості такого елемента є об'єднанням областей видимості, заданих модифікаторами protected та internal (рис. 4.12).

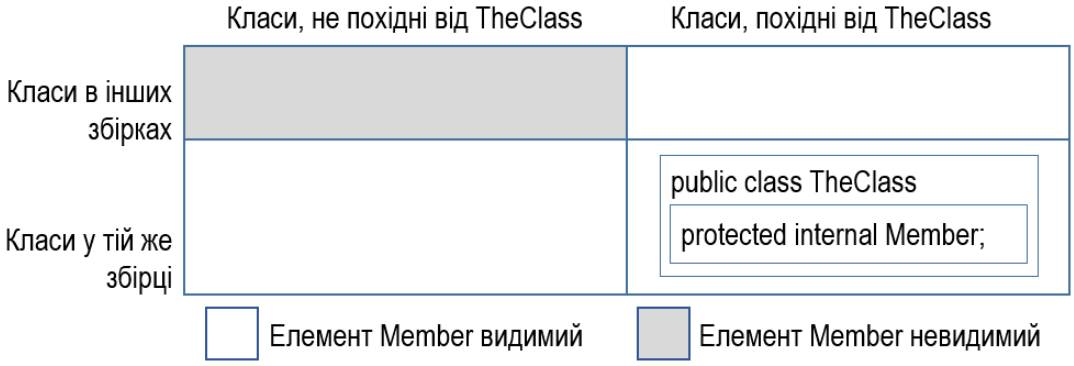


Рис. 4.12. Доступність елементів з модифікаторами protected internal

Підсумки

У таблиці 4.1 зведено інформацію про призначення модифікаторів доступу елементів типу. Рівні доступу в таблиці приведено в порядку зменшення обмежень: від найобмеженішого private до найвільнішого public.

Таблиця №4.1. Модифікатори доступу елементів класу

Модифікатор	Опис доступності елемента класу
private	Доступний тільки у класі
internal	Доступний для всіх класів збірки
protected	Доступний для всіх похідних класів
protected internal	Доступний для всіх похідних класів та всіх класів збірки
public	Доступний для всіх

Абстрактні елементи та класи

Абстрактні класи

Абстрактні класи – це класи, призначені виключно для спадкування їх іншими класами. Створити екземпляр такого класу неможливо. В абстрактному класі описують (але не реалізують) найзагальніші характеристики об'єкта, які повинні бути уточнені шляхом конкретної їх реалізації у похідному класі.

Абстрактний клас оголошують з модифікатором abstract:

Ключове слово abstract

```
abstract class TheClass
{
    ...
}
```

Абстрактний клас може містити звичайні та абстрактні елементи. Він може бути похідними від іншого абстрактного класу. У класі, похідному від

абстрактного, слід реалізувати всі успадковані абстрактні елементи, позначивши їх ключовим словом override (якщо тільки клас сам не є абстрактним).

Абстрактні елементи

Абстрактний елемент класу – це елемент, призначений для заміщення у похідному класі. Абстрактними можуть бути методи та інші функціональні елементи (властивості, події, індексатори). Елементи даних класу не можуть бути абстрактними.

Абстрактний елемент позначають модифікатором abstract. Він не повинен мати реалізації: його код представляють однією порожньою інструкцією (крапкою з комою):


```
abstract public void TheMethod();
abstract public int TheProperty
{
    get;
    set;
}
```

Абстрактні елементи класу можна оголошувати тільки в абстрактному класі. Хоча вони і мають бути заміщені в похідному класі з використанням директиви override, але при їх оголошенні не можна використовувати модифікатор virtual. Віртуальні елементи, на відміну від абстрактних, повинні мати код реалізації. У похідному класі віртуальні елементи можуть бути заміщеними, а абстрактні – повинні бути заміщеними.

Лістинг 4.6. Абстрактний клас з абстрактним методом

```
1  abstract class TheAbstractClass            // Абстрактний клас
2  {
3      public void PrintIdentify()
4      {
5          Console.WriteLine("Надруковано     з     абстрактного
6  класу");
7      }
8      abstract  public  void  AbstractMethod();  //  Абстрактний
9  метод
10 }
11
12 class TheClass : TheAbstractClass
13 {
14     public override void AbstractMethod()
```

```
15      {
16          Console.WriteLine("Надруковано з похідного класу");
17      }
18  }
19
20  static void Main(string[] args)
21  {
22      TheClass tc = new TheClass();
23      tc.PrintIdentify();
        tc.AbstractMethod();
    }
```

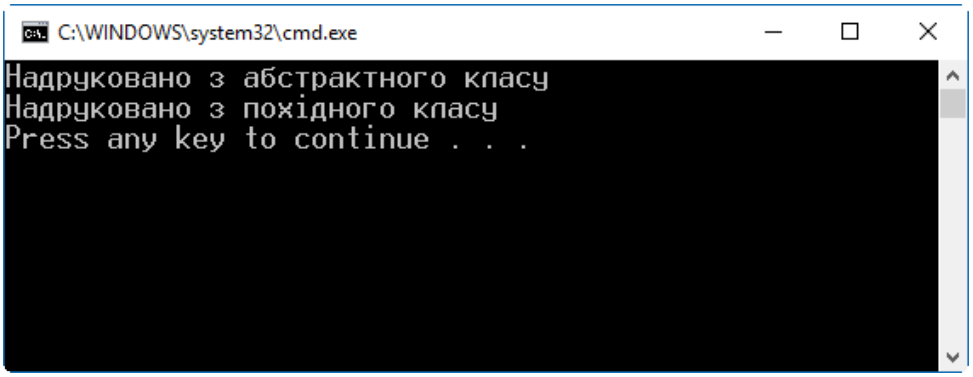


Рис. 4.13. Результат виконання програми з лістингу 4.6

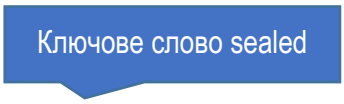
У лістингу 4.6 приведено приклад, який містить оголошення абстрактного класу TheAbstractClass (рядки 1…8). Він має два методи: звичайний метод PrintIdentify (рядки 3…6) та абстрактний метод AbstractMethod (рядок 7). Далі оголошено похідний клас (рядки 10…16), в якому заміщено та реалізовано абстрактний метод (рядки 12…15).

В методі Main створюється екземпляр класу TheClass та викликаються два його методи: один успадкований з абстрактного базового класу, а другий – з похідного класу, реалізований для заміщення абстрактного. При виконанні ця програма виведе у консольне вікно дві фрази, які позначають виконання обох методів (рис. 4.13).

Запечатані класи

Запечатані класи забороняють успадковувати їх. Вони є протилежністю до абстрактних. Якщо з абстрактного класу неможливо створити екземпляр, і його перед використанням обов'язково слід успадкувати, то запечатаний клас призначений винятково для створення екземплярів, його неможливо успадкувати. Спроба використати запечатаний клас в ролі базового для іншого класу спричинить помилку компіляції.

Запечатаний клас оголошують з модифікатором sealed:




```
sealed class TheClass
{
    ...
}
```

Статичні класи

Статичним називають клас, у якого всі елементи статичні. Такі класи використовують для групування даних та функцій, які не зв'язані з екземпляром. Наприклад, статичний клас доречно використати для проектування математичної бібліотеки, яка містить набір математичних значень та функцій.

Статичний клас при оголошенні позначають модифікатором static. Всі його елементи також слід оголосити статичними. Такий клас може мати статичний конструктор, але не може мати жодного конструктора екземпляра, оскільки створити екземпляр такого класу неможливо. Статичний клас за замовчуванням запечатаний. Його не можуть спадкувати інші класи. До елементів статичного класу можна звертатися через його назву.

У лістингу 4.7 приведено приклад оголошення та використання статичного класу IntelectualMath, який містить кілька методів для математичних операцій. У класі оголошено два статичних поля типу double (рядки 3…4): одне з них представляє число π, а інше – число Ейлера. Крім цього, клас містить два статичних методи. Перший з них, метод XPi (рядки 6…9), повертає результат множення числа π на заданий аргумент, а другий метод (XE) робить те саме з числом Ейлера.

В методі Main продемонстровано звертання до всіх чотирьох статичних елементів класу IntelectualMath: значення полів та результати виконання методів виводяться у консольне вікно (рис. 4.14).

Лістинг 4.7. Приклад статичного класу

```
1  static class IntelectualMath
2  {
3      public static double Pi = 3.14159265;
4      public static double E  = 2.71828183;
5
6      public static double XPi(double x)
7      {
8          return x * Pi;
9      }
```

```
10      public static double XE(double x)
11      {
12          return x * E;
13      }
14  }
15
16  static void Main(string[] args)
17  {
18      Console.WriteLine("Pi = {0}", IntelectualMath.Pi);
19      Console.WriteLine("E  = {0}", IntelectualMath.E);
20      Console.WriteLine("12*Pi = {0:0.00000}",
21          IntelectualMath.XPi(12));
22      Console.WriteLine("12*e  = {0:0.00000}",
23          IntelectualMath.XE(12));
24  }
```

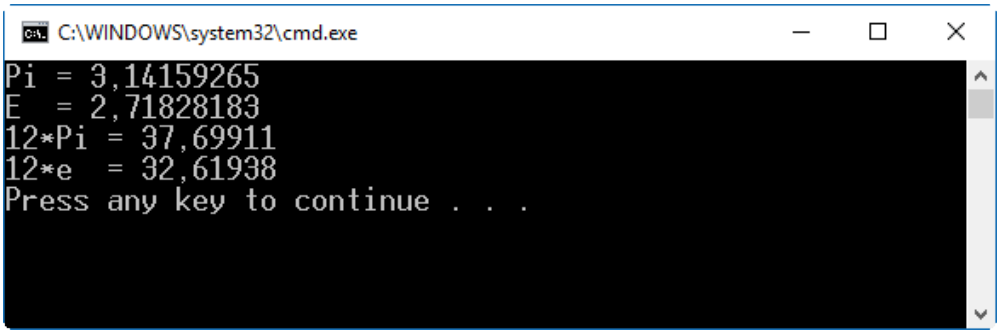
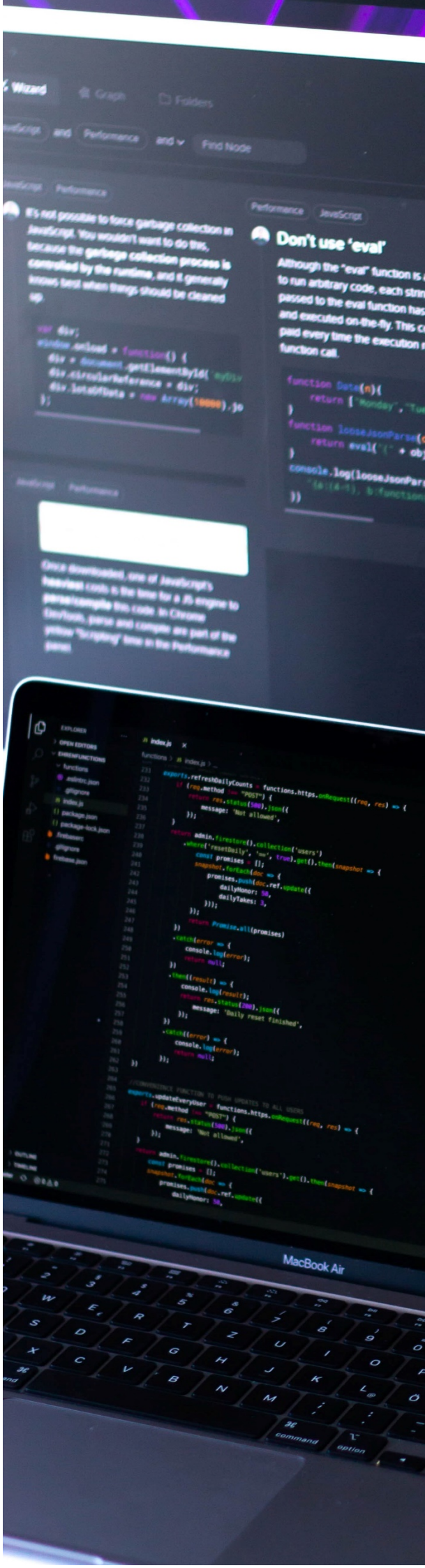


Рис. 4.14. Результат виконання програми з лістингу 4.7

Розділ 5. Виняткові ситуації

- Виняткові ситуації
- Класи виняткових ситуацій
- Обробка виняткових ситуацій
- Секція catch
- Секція finally
- Вкладені блоки try
- Генерування винятків
- Створення власних винятків



Виняткові ситуації

Помилки, які виникають при розробці та експлуатації програм, можуть бути наступних видів: синтаксичні, логічні та динамічні.

Синтаксичні помилки виникають при порушеннях синтаксису мови програмування. Вони виявляються на етапі компіляції програми самим компілятором і найлегше виправляються. Слід зауважити, що відсутність синтаксичних помилок не гарантує коректної роботи програми: помилковий алгоритм можна реалізувати синтаксично правильно.

Логічні помилки є наслідком реалізації неправильного алгоритму і проявляються при виконанні програми. Наявність логічних помилок може не приводити до системних збоїв у роботі програми, але вона видаватиме невірні результати. Ці помилки не виявляються в автоматизованому режимі, так як можуть бути знайдені лише шляхом аналізу конкретного реалізованого алгоритму.

Динамічні помилки виникають при виконанні програми та є наслідком неправильної організації роботи елементів програми чи операційної системи. Динамічні помилки також називають помилками часу виконання (*runtime error*). Прикладом такої помилки може бути ділення на нуль:

```
int x = 3, y = 0;
x = x / y;
```

Цей код скомпілюється успішно, але якщо під час виконання значення y дорівнюватиме 0, то виникне помилка часу виконання "Attempted to divide by zero" (рис. 5.1).

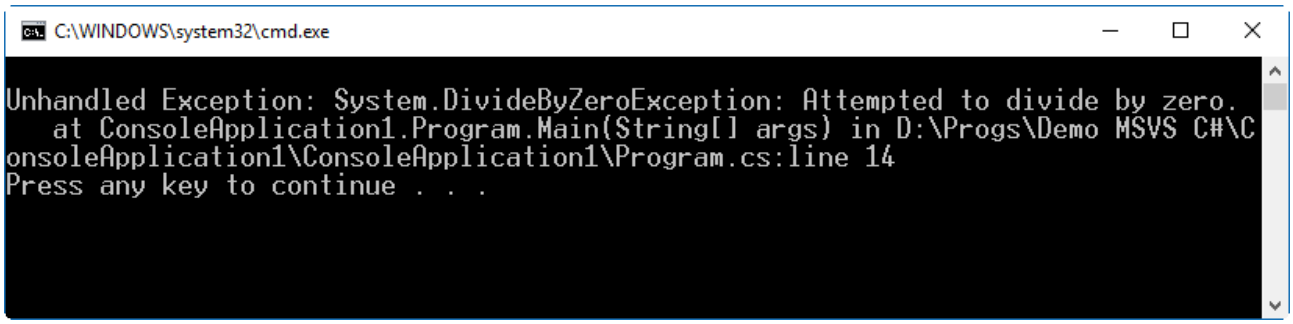


Рис. 5.1. Повідомлення про виняткову ситуацію ділення на нуль у консольному застосунку

Для обробки динамічних помилок введено поняття виняткової ситуації.

Виняткова ситуація – це помилка у програмі, яка виникає внаслідок порушення системних або програмних обмежень. Виняткова ситуація (її ще

називають *винятком*) викликає переривання або припинення виконання програми. Обробка виняткової ситуації полягає у нейтралізації динамічної помилки, яка її визвала. Виняткова ситуація залишається активною, доки не буде оброблена. Якщо програма не містить коду для обробки виняткової ситуації, то система припинить виконання програми.

Коли виникає виняткова ситуація, система автоматично генерує об'єкт виняткової ситуації, який містить дані про неї.

Класи виняткових ситуацій

У програмах можуть траплятися винятки багатьох різних типів. Бібліотека базових класів .NET Framework (BCL) визначає велику кількість класів, які представляють виняткові ситуації конкретного виду. Коли трапляється виняткова ситуація, середовище виконання .NET (CLR) створює об'єкт виняткової ситуації певного типу і шукає прийнятний пункт catch для її обробки.

Всі винятки походять від класу Exception, оголошеного у просторі імен System. Цей клас інкапсулює загальні характеристики виняткової ситуації. Об'єкт виняткової ситуації містить властивості (доступні лише для читання) з інформацією про виняток. У таблиці 5.1 приведено найвживаніші з них.

На основі класу Exception створено велику кількість класів, які описують виняткові ситуації певного виду: помилки при роботі з числами із плаваючою комою, помилки при роботі з пам'яттю, помилки дискового вводу-виводу тощо.

Таблиця №5.1. Найвживаніші властивості класу Exception

Властивість	Тип	Опис
Message	string	Містить повідомлення про помилку, яке пояснює причину винятку
StackTrace	string	Містить інформацію, яка описує, де сталась помилка
InnerException	Exception	Якщо поточний виняток згенерований іншим винятком, ця властивість містить посилання на попередній виняток
HelpLink	string	Винятки, описані програмно, можуть задати у цій властивості посилання (URN, URL) на інформацію, яка пояснює причину винятку
Source	string	Якщо ця властивість не встановлена програмно, то вона містить назву збірки, де виник виняток

Обробка виняткових ситуацій

Для обробки виняткових ситуацій використовують інструкцію try. Вона дозволяє позначити блок коду, в якому відслідковуватимуться виняткові ситуації, а також забезпечує код для їх обробки. Інструкція try може складатися з таких трьох секції (рис. 5.2):

- Блок try містить код, у якому відслідковуватимуться винятки.
- Секція catch містить одну чи кілька інструкцій catch, кожна з яких має свій блок коду для обробки винятків. Інструкції catch називають ще оброблювачами виняткових ситуацій.
- Блок finally містить код, який виконається при будь-яких умовах: трапиться виняток, чи ні.

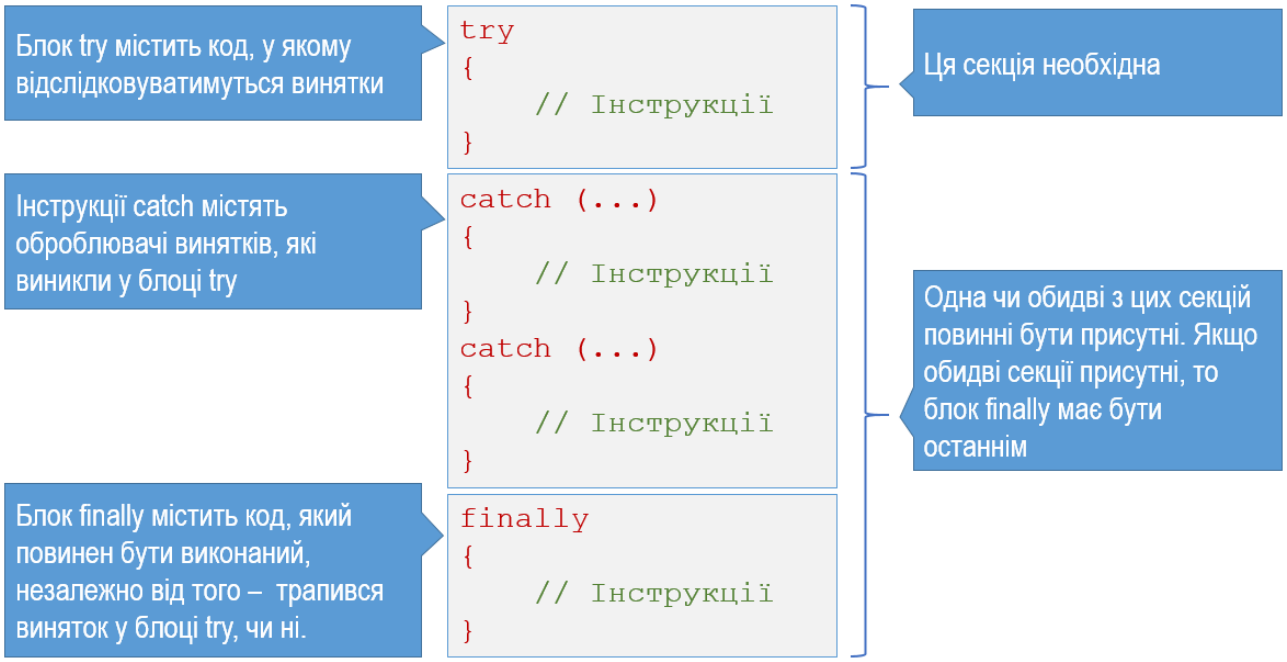


Рис. 5.2. Структура конструкції try

Конструкція try може містити одну з секцій (catch чи finally), або обидві. Блоки try, як і інші програмні блоки, можуть бути вкладеними один в інший, але не можуть перетинатися.

Послідовність спрацювання інструкцій коду при виникненні виняткової ситуації така. Як завжди, послідовно виконуються інструкції блоку try. Якщо при відпрацюванні однієї з них трапилась виняткова ситуація, то всі наступні інструкції до кінця блоку try пропускаються, і виконання переходить у відповідний блок catch. Після відпрацювання всіх його інструкцій продовжується виконання інструкцій програми, розташованих за секцією обробки винятків. Якщо у блоці try виняткової ситуації не трапилось, то блок

catch пропускається. Останнім виконується блок finally, причому незалежно від того, трапиться виняткова ситуація у блоці try чи ні.

Розглянемо код з лістингу 5.1, у якому відслідковується та обробляється виняткова ситуація ділення на нуль. Блок try, у якому відслідковуються виняткові ситуації, розміщений у рядках 5…8. Рядок 7 містить інструкцію ділення на нуль (оскільки значення змінної y дорівнює нулю). На відміну від попередньо розглянутого випадку, робота програми не буде припинена. Спочатку буде виконано блок catch, який у даному випадку містить код з виводом на екран повідомлення. Далі виконання програми буде продовжено з першої інструкції за блоком catch. Результат виконання програми приведено на рис. 5.3. Якщо з розглянутого коду забрати обробку виняткових ситуацій, то на рядку 7 виконання програми припиниться, і система виведе повідомлення про виняткову ситуацію, як на рис. 5.1. Якщо змінній y присвоїти відмінне від нуля значення, то секція catch взагалі не виконуватиметься.

Зауважимо, що єдина відмінність коду в лістингу 5.1 від попередньо розглянутого – наявність інструкцій для обробки винятків.

Лістинг 5.1. Обробка виняткових ситуацій

```
1  static void Main(string[] args)
2  {
3      int x = 3, y = 0;
4
5      try                 // Початок обробки виняткових ситуацій
6      {
7          x = x / y;      // Тут виникає виняткова ситуація
8      }
9      catch
10     {                   // Код для обробки виняткових ситуацій
11         Console.WriteLine("Обробляємо виняток...");
12     }
13
14     Console.WriteLine("І продовжуємо роботу...");
15 }
```

Інструкція try

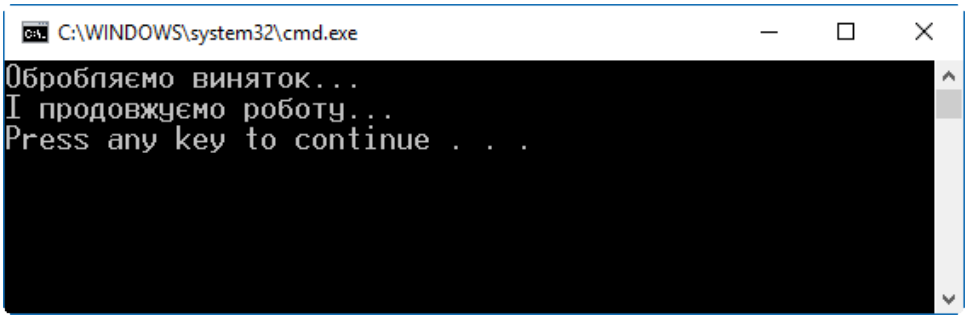


Рис. 5.3. Результат виконання програми з лістингу 5.1

Секція catch

Кожна інструкція catch обробляє виняткову ситуацію. Є три форми реалізації catch, які дозволяють задіяти різні рівні обробки винятків.

1. Загальна інструкція catch не містить параметрів і підходить для виняткових ситуацій будь-якого типу. Синтаксис такої інструкції catch найпростіший:

```
catch
{
    // Інструкції
}
```

2. Спеціальна інструкція catch отримує назву класу винятку як параметр і підходить для всіх винятків вказаного класу, або похідних класів.

```
catch (ExceptionType)
{
    // Інструкції
}
```

3. Спеціальна інструкція catch з об'єктом містить після назви класу ідентифікатор, який діє як локальна змінна у блоці коду catch. Ця змінна є посиланням на об'єкт виняткової ситуації, за допомогою якого можна отримати інформацію про виняток.

```
catch (ExceptionType ExceptionVariable)
{
    // Інструкції
}
```

За допомогою загальної інструкції catch можна обробляти будь-які винятки, але неможливо розрізнити тип винятку. Вона забезпечує найзагальнішу обробку всіх винятків, які тільки можуть трапитися. Використання загальної інструкції catch продемонстровано у лістингу 5.1.

Спеціальну інструкцію catch використовують виключно для обробки винятків вказаного класу, та похідних від нього класів. Це сильно звужує область застосування, але дозволяє конкретизувати причину помилки та виконати дії по її виправленню. Приклад використання спеціальної інструкції catch приведено в лістингу 5.2. У ньому використано той же код, що і минулому прикладі, але змінено форму інструкції catch.

Спеціальна інструкція catch з об'єктом дає найповнішу інформацію про виняток. Вона підходить для обробки винятків вказаного класу, та похідних від нього класів. Через вказану для цієї інструкції змінну програміст отримує доступ до об'єкта виняткової ситуації, створеного виконавчою підсистемою .NET Framework (CLR). Зокрема, у програмі можна прочитати значення властивостей цього об'єкта (таблиця 5.1). Приклад використання цієї форми catch наведено у лістингу 5.3. У рядку 8 вказано тип винятку

(DivideByZeroException) та змінну e, яка представлятиме системний об'єкт винятку. Далі у блоці коду catch за допомогою цього об'єкта отримуються та виводяться у вікно дані про виняток (рядки 11…13), що видно з рис. 5.4. Звертаються до об'єкта виняткової ситуації так само, як і до об'єкта – екземпляра будь-якого класу: вказують назву змінної об'єкта та назву його властивості, розділяючи їх крапкою.

Лістинг 5.2. Спеціальна інструкція catch

```
1  static void Main(string[] args)
2  {
3      int x = 3, y = 0;
4      try
5      {
6          x /=  y;
7      }
8      catch (DivideByZeroException)
9      {
10         Console.WriteLine("Була спроба ділення на нуль!");
11     }
12
13     Console.WriteLine("Але продовжуємо роботу...");
14 }
```

Вказано клас винятку

Лістинг 5.3. Спеціальна інструкція catch з об'єктом винятку

```
1  static void Main(string[] args)
2  {
3      int x = 3, y = 0;
4      try
5      {
6          x /=  y;
7      }
8      catch (DivideByZeroException e)
9      {
10         Console.WriteLine("Стався виняток з описом:");
11         Console.WriteLine("Message: {0}", e.Message);
12         Console.WriteLine("Source: {0}", e.Source);
13         Console.WriteLine("Stack: {0}", e.StackTrace);
14     }
15     Console.WriteLine();
16     Console.WriteLine("Але продовжуємо роботу...");
17 }
```

Клас винятку

Змінна винятку

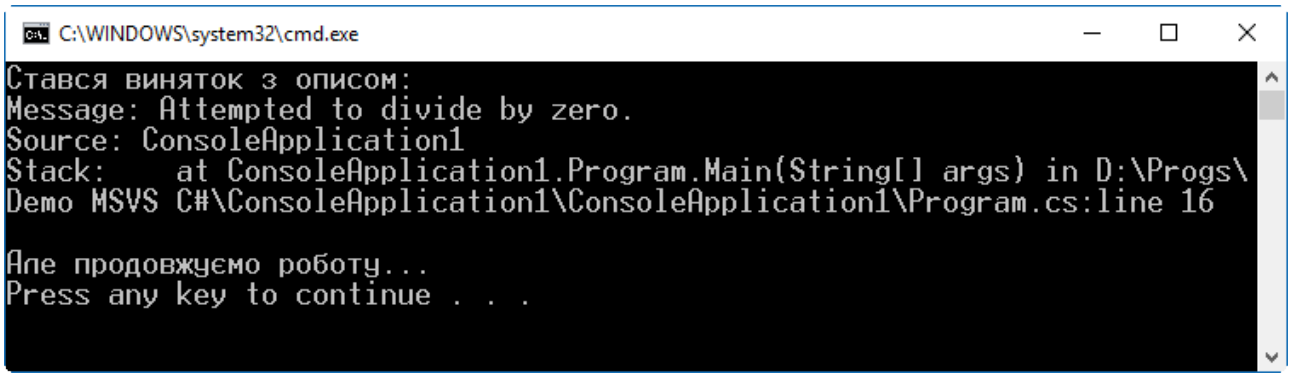


Рис. 5.4. Результат виконання програми з лістингу 5.3

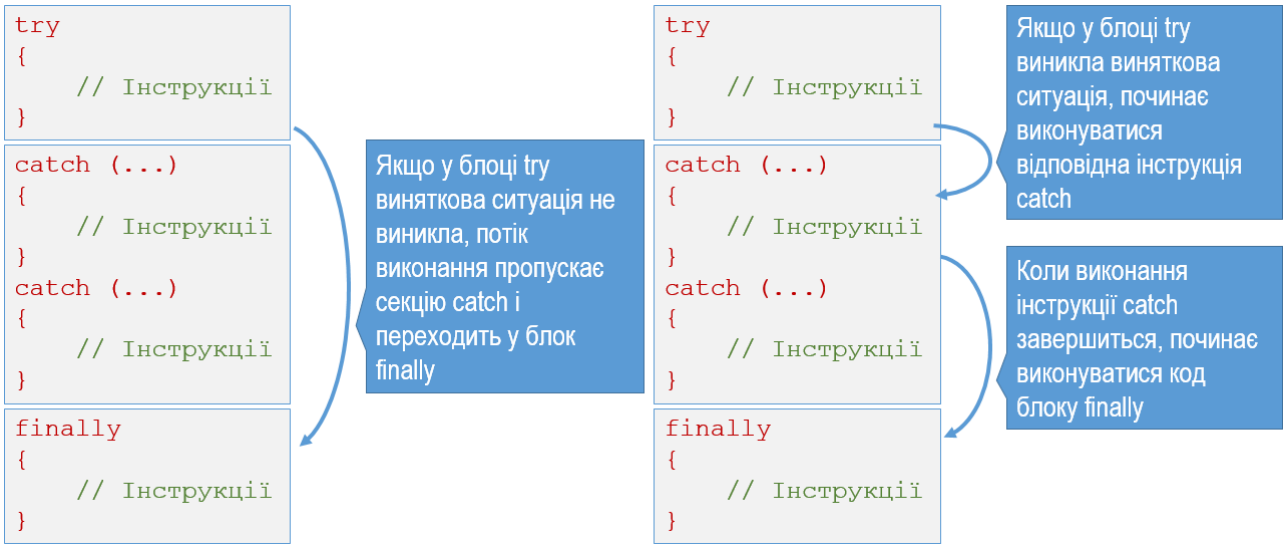
Секція catch може містити кілька інструкцій catch, які розміщують одна за одною. Коли виникає виняткова ситуація, система переглядає список інструкцій catch в записаному порядку, і виконається перша з них, що задовольнятиме типу винятку. Всі наступні блоки catch будуть проігноровані. Тому інструкції catch слід розташовувати в порядку від специфічних до загальних типів виняткових ситуацій. В іншому випадку спочатку виконуватиметься інструкція, яка оброблятиме узагальнені винятки, а інструкції для специфічних типів винятків ніколи не виконуватимуться. Отже, якщо у переліку інструкцій секції catch є інструкція загальної форми, вона завжди повинна бути останньою, після всіх спеціальних інструкцій.

Блоки try можуть бути вкладеними один в інший. Коли у секції try внутрішнього блоку сталась виняткова ситуація, але для винятку даного типу не знайдено потрібної інструкції catch, то система переглядає, чи містить зовнішній блок підходящу інструкцію catch. Якщо її знайдено, то виконається вона (але після секції finally внутрішнього блоку, – коли вона є). Якщо потрібної інструкції catch не знайдено ні на одному рівні вкладеності, система припинить виконання програми.

Секція finally

Блок finally містить інструкції, які повинні виконуватися завжди, незалежно від того, виникала виняткова ситуація, чи ні. Якщо при виконанні інструкцій у блоці try виняткової ситуації не трапилось, то потік виконання ігнорує всі інструкції catch і переходить у блок finally (рис. 5.5, а). Якщо у блоці try сталась виняткова ситуація, то спочатку виконується придатна для її обробки інструкція catch (якщо вона є), а потім виконуються інструкції блоку finally (рис. 5.5, б).

Якщо потік виконання ввійшов у секцію try, то блок finally буде виконано в будь-якому випадку, навіть коли секція try містить інструкцію return, що припиняє роботу поточного методу.



а) б)

Рис. 5.5. Виконання блоку finally, коли виняткова ситуація не виникла (а) та виникла (б)

Вкладені блоки try

Блоки для обробки виняткових ситуацій також можуть бути вкладеними один в інший. У лістингу 5.4 приведено приклад з двома вкладеними блоками try. Внутрішній блок описано в рядках 6…9. Саме в ньому у рядку 8 міститься інструкція, виконання якої приведе до виникнення винятку (ділення на нуль). Внутрішній блок для обробки винятків містить спеціальну секцію catch (рядки 10…13), яка обробляє винятки одного типу (IndexOutOfRangeException, вихід за межі діапазону). Так як виняток ділення на нуль має інший тип (DivideByZeroException), то він не може бути оброблений даною інструкцією catch. Крім цього, внутрішній блок для обробки винятків містить секцію finally (рядки 14…17), код якої виводить повідомлення у вікно.

Зовнішній блок try (рядки 3…18) оточує внутрішній, і закінчується інструкцією catch для обробки винятків ділення на нуль. Таким чином інструкція catch, потрібна для обробки винятку з внутрішнього блоку, міститься у зовнішньому блоці.

При виникненні виняткової ситуації у рядку 8 системою спочатку переглядатимуться оброблювачі catch внутрішнього блоку, придатні для обробки винятку ділення на нуль. Оскільки їх немає (наявний оброблювач може

опрацьовувати тільки винятки іншого типу – IndexOutOfRangeException), то виконається блок finally (він виконується завжди), і система продовжить пошук оброблювача для виняткової ситуації у зовнішньому блоці. Зовнішній блок справді містить такий оброблювач (рядки 19…22).

Лістинг 5.4. Вкладені блоки try

```
1  static void Main(string[] args)
2  {
3      try
4      {
5          int x = 3, y = 0;
6          try
7          {
8              x /= y;     // Тут виникає виняткова ситуація
9          }
10         catch (IndexOutOfRangeException)
11         {
12             Console.WriteLine("Вихід за межі діапазону!");
13         }
14         finally
15         {
16             Console.WriteLine("Внутрішній блок finally");
17         }
18     }
19     catch (DivideByZeroException)
20     {
21         Console.WriteLine("Ділення на нуль!");
22     }
23
24     Console.WriteLine();
25     Console.WriteLine("І продовжуємо роботу...");
26 }
```

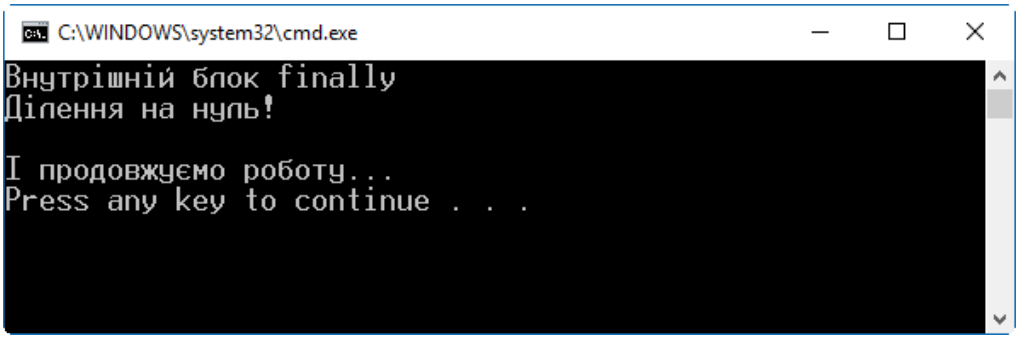


Рис. 5.6. Результат виконання програми з лістингу 5.4 (вкладені блоки try)

Таким чином, при появі винятку в рядку 8 виконання внутрішнього блоку try перерветься, виконається секція finally внутрішнього блоку (рядок 16), потім –

блок catch зовнішнього блоку (рядок 21), і далі продовжиться виконання інструкцій програми, розташованих після зовнішнього блоку (рядок 24 і далі). Цю послідовність видно з виводу програми у консоль (рис. 5.6).

Генерування винятків

У попередньо розглянутих прикладах ми розглядали перехоплення винятків, автоматично згенерованих системою. Але в коді можна явно генерувати виняткові ситуації. Для цього використовують інструкцію throw. Синтаксис її використання такий:

```
throw exceptionObject;
```

де exceptionObject – посилання на об'єкт винятку (класу Exception або похідного класу).

Припустимо, що арифметичний вираз дещо складніший, ніж у рядку 8 лістингу 5.4, і містить кілька операцій ділення з різними знаменниками. Тоді, при кожній операції ділення може виникнути виняток одного і того ж класу – DivideByZeroException. Щоб розрізнити, яке саме значення спричинило виняток, можна використати підхід, реалізований у лістингу 5.5.

Лістинг 5.5. Програмне генерування винятку

```
1  static void Main(string[] args)
2  {
3    try
4    {
5        int x = 3, y = 10, z = 0;
6
7        if (y == 0) throw new DivideByZeroException("y == 0!");
8        if (z == 0) throw new DivideByZeroException("z == 0!");
9        x = (x / y + 100) / z;
10   }
11   catch (DivideByZeroException e)
12   {
13       Console.WriteLine("Помилка: {0}", e.Message);
14   }
15
16   Console.WriteLine("Але продовжуємо роботу...");
17 }
```

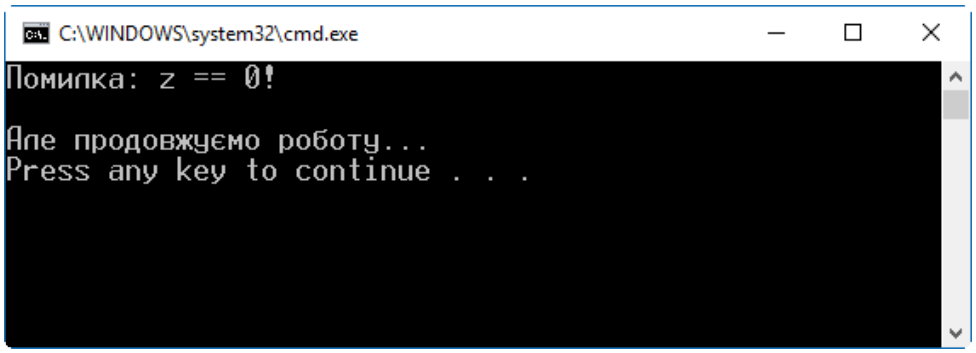


Рис. 5.7. Результат виконання програми з лістингу 5.5

Арифметичний вираз, який розраховується у рядку 9, містить два знаменника: y та z. Перед розрахунком обидва значення (y та z) перевіряються на рівність нулю. Якщо одна зі змінних має нульове значення (рядки 7,8), то за допомогою оператора new створюємо об'єкт винятку – екземпляр класу DivideByZeroException. При цьому передаємо у його конструктор значення для властивості Message (вона містить опис винятку, див. таблицю 5.1), у якому вказано назву змінної, яка спричинить виняток (y чи z). Таким чином, для різних нульових знаменників опис винятку буде різний (порівняйте рядки 7 та 8). На рис. 5.7 показано результат виконання розглянутої програми. Якщо в лістингу 5.5 у рядку 5 змінити значення змінних так, щоб значення y було нульовим, то зміниться і вивід на екран (буде вказано іншу назву змінної).

Зауважимо, що в розглянутому коді при нульових знаменниках вираз у рядку 9 ніколи на віть не почне виконуватися: виконання умови в рядку 7 чи 8 згенерує виняток ще до цього.

Інструкцію throw можна також викликати без об'єкта виняткової ситуації. Це можна робити тільки у блоці catch. При цьому інструкція наново генерує поточний виняток, і система продовжить шукати інший оброблювач, придатний для обробки цього винятку.

Створення власних винятків

Незважаючи на те, що описані у C# винятки охоплюють найпоширеніші програмні помилки, обробка виняткових ситуацій не обмежується тільки ними. В програмі можна описувати свої класи виняткових ситуацій, які відповідатимуть потребам реалізованого алгоритму. Щоб створити свій клас для опису виняткової ситуації, потрібно оголосити клас, похідний від класу Exception. Класам, які описують винятки, прийнято давати назву, яка закінчується словом Exception.

У лістингу 5.6 оголошено клас виняткових ситуацій з назвою InvNameException (рядки 1…7), похідний від класу Exception. Щоб для екземпляру

цього класу при створенні можна було задати своє значення властивості Message з описом винятку, реалізовано конструктор (рядки 3…6) з одним параметром message, в якому розміщують опис винятку. Для конструктора використано ініціалізатор base, якому передаватиметься параметр-рядок конструктора[13]. Це приведе до виклику конструктора базового класу (Exception) з одним параметром рядкового типу. Такий конструктор присвоїть властивості Message об'єкта винятку вказаний при виклику рядок, який описуватиме виняток. Зауважимо, що властивість Message тільки для читання, і безпосередньо їй присвоїти значення неможливо.

Лістинг 5.6. Програмне генерування винятку

```
1  class InvNameException : Exception
2  {
3      public InvNameException(string message) : base(message)
4      {
5
6      }
7  }
8
9  static void Main(string[] args)
10 {
11     try
12     {
13         string Name = "a";
14         if ((Name == null) || (Name.Length < 2))
15             throw new InvNameException("Неправильна назва!");
16         Console.WriteLine("Назва: {0}", Name);
17     }
18     catch (InvNameException e)
19     {
20         Console.WriteLine("Помилка: {0}", e.Message);
21     }
22
23     Console.WriteLine("Але продовжуємо роботу...");
24 }
```

Базовий клас – Exception

Ініціалізатор конструктора

У розглянутій програмі виняток InvNameException призначений для відслідковування неправильної назви, яку представляє змінна Name (рядок 13). Припустимо, що коректною є назва, для якої виконуються умови: а) назва повинна бути задана (не дорівнювати null); б) назва повинна бути довшою за

[13] Ініціалізатори конструкторів було розглянуто в попередньому розділі.

один символ. Таким чином, змінній Name при оголошенні присвоєно некоректне значення (закоротке).

У рядку 14 перевіряємо значення змінної Name на виконання двох вказаних умов (вони поєднані логічним оператором "АБО"). Якщо одна з умов не виконується, створюється екземпляр класу InvNameException. При цьому йому передається опис помилки ("Неправильна назва!"). Якщо генерується виняток, то всі наступні інструкції блоку try (рядок 16) вже не виконуються. Керування потоком коду передається у спеціальний блок catch з об'єктом. Цей блок призначений для обробки винятків нашого конкретного типу і виводить у вікно опис згенерованого винятку (рис. 5.8). Опис було передано в конструктор при створенні об'єкта винятку. Далі програма продовжує виконуватися з інструкції, розміщеної після блоку для обробки виняткових ситуацій (рядок 23).

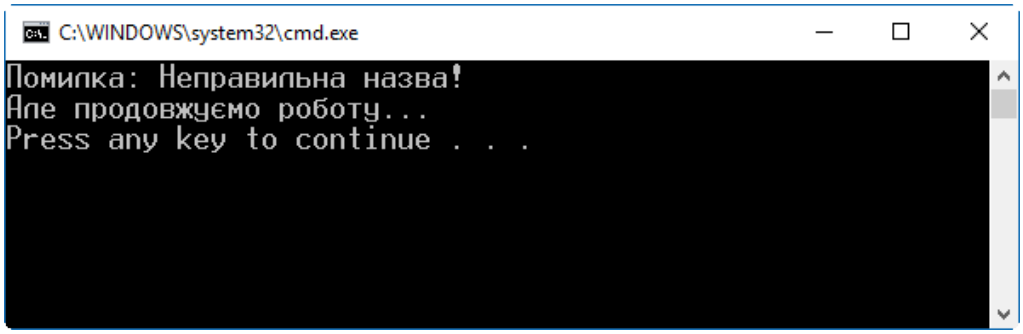
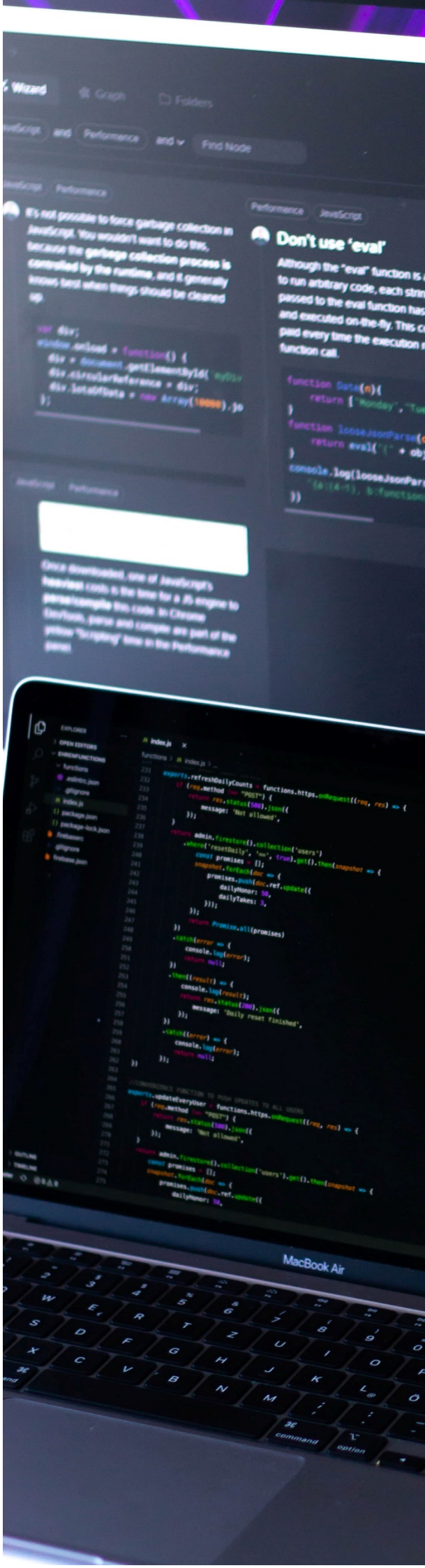


Рис. 5.8. Результат виконання програми з лістингу 5.6

Якщо назва буде правильною, то виняток не буде згенеровано, а виконається рядок 16, в результаті чого значення змінної Name буде виведено у вікно. Блок catch не виконуватиметься взагалі, а потік виконання відразу перейде до інструкцій за блоком обробки винятків.

Розділ 6. Структури

- Структури
- Присвоєння структур
- Конструктори структур
- Структури як параметри та результати методів
- Додаткові дані щодо структур



Структури

Структури – це типи даних, які описує програміст. Вони дуже схожі на класи. Структури, як і класи, можуть містити елементи даних та функціональні елементи. Але, на відміну від класів, структури належать до значущих типів (класи – до посилальних).

Структуру використовують для опису даних зі статичним устроєм, який задають одноразово при оголошенні [14]. Тому щодо структур не застосовний механізм спадкування. Говорячи термінами класів, структури є неявно запечатаними, і їх неможливо спадкувати іншими структурами (чи класами).

Синтаксис для оголошення структур дуже схожий на оголошення класів:


```
struct TheStructName
{
    // Оголошення елементів структури
}
```

Структури, як і класи, можуть містити статичні елементи.

У лістингу 6.1 приведено оголошення структури Point з двома полями – X та Y (рядок 1…5). Ця структура представляє точку на екрані (піксель) з цілочисельними декартовими координатами. Далі у методі Main оголошено дві змінні структурного типу – first та second. Їх полям присвоєно певні значення, після чого дані з обох структур виводяться на екран (рис. 6.1).

Лістинг 6.1. Оголошення та використання структури Point

```
1  struct Point                        // Оголошення структури
2  {
3      public int X;
4      public int Y;
5  }
6  class Program
7  {
8      static void Main()
9      {
10         Point first, second;    // Оголошення змінних-структур
11         // Присвоєння значень полям структур
12         first.X = 10;
13         first.Y = 10;
```

[14] У той же час класи використовують для опису даних з динамічним устроєм, який можна змінювати за допомогою механізмів спадкування та поліморфізму.

```
14          second.X = first.X + 20;
15          second.Y = first.Y + 20;
16          // Вивід на екран значень полів структур
17          Console.WriteLine("first: {0}, {1}",
18              first.X, first.Y);
19          Console.WriteLine("second: {0}, {1}",
20              second.X, second.Y);
21      }
22  }
```

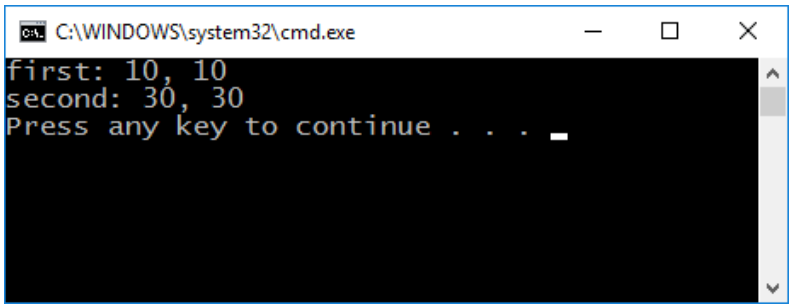


Рис. 6.1. Вивід лістингу 6.1

Як і всі змінні значущих типів, змінна структури безпосередньо містить її дані. Тому:

- змінна структури не може приймати значення null;
- дві змінних-структури не можуть посилатися на один і той же об'єкт даних.

Лістинг 6.2. Структура та клас з однаковими полями

```
1   class CPoint
2   {
3       public int X;
4       public int Y;
5   }
6   struct SPoint
7   {
8       public int X;
9       public int Y;
10  }
11  class Program
12  {
13      static void Main()
14      {
15          CPoint cp = new CPoint();
16          SPoint sp;
17          ...
```

Наприклад, лістинг 6.2 містить оголошення класу CPoint та структури SPoint з однаковими полями. Рис. 6.2 ілюструє, як змінні цих двох типів розташовані у пам'яті.

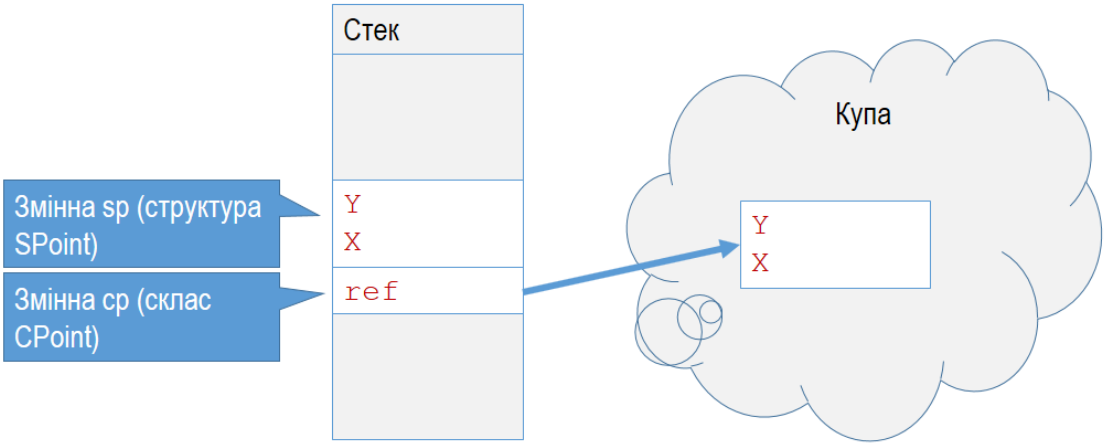


Рис. 6.2. Розміщення у пам'яті змінної структури та класу

При оголошенні полів структури їх (на відміну від класів) не можна ініціалізувати:

```
struct SPoint
{
    public int X = 10;   // Буде помилка !
    public int Y;
}
```

Виникне помилка компіляції!

Оскільки структури не підтримують механізму спадкування, то використання ряду модифікаторів для них неможливе. Зокрема, модифікатори protected, internal, abstract та virtual у структурах не підтримуються.

Проте, за внутрішнім механізмом C#, структури є похідними від класу System.ValueType, для якого базовим є клас object. Тому два пов'язаних зі спадкуванням ключових слова при оголошенні елементів структур все ж можна використовувати: new та override. Їх використовують при оголошенні елементів, назва яких співпадає з назвою елементів базового класу System.ValueType.

Присвоєння структур

Присвоєння однієї структури іншій копіює всі *значення* від однієї структури до іншої. Така поведінка повністю відмінна від присвоєння змінних класу, при якому копіюється лише *посилання* на об'єкт. Це ілюструє лістинг 6.3. Тут оголошено дві змінних класу (рядок 1) та дві змінних-структури (рядок 2). Типи структури та класу взято з лістингу 6.2.

Лістинг 6.3. Присвоєння змінних структури та змінних класу

```
1  CPoint cp1 = new CPoint(), cp2 = null; // Екземпляри класу
2  SPoint sp1, sp2;                       //          Екземпляри
3  структури
4  cp1.X = sp1.X = 5;        // Присвоєння значень полям
5  cp1.Y = sp1.Y = 5;        //   першого   об'єкту   та   першої
6  структури
7
   cp2 = cp1;                // Присвоєння змінних класу
   sp2 = sp1;                // Присвоєння змінних структури
```

В результаті присвоєння змінної першого об'єкта змінній другого об'єкта (рядок 6) отримуємо два посилання на один і той же об'єкт даних у пам'яті. Самі ж дані представлено у єдиному екземплярі (рис. 6.3). Після присвоєння змінної першої структури змінній другої структури (рядок 7) отримуємо дві однакові структури даних у пам'яті. Таким чином, дані у пам'яті буде продубльовано.

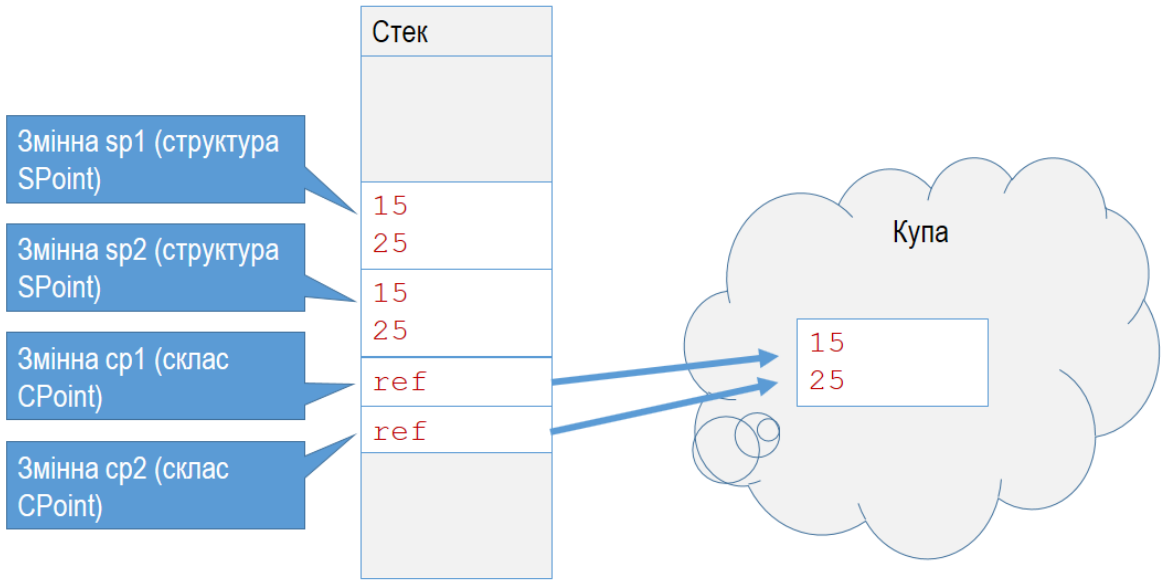


Рис. 6.3. Присвоєння змінних структури та класу

Конструктори структур

Структури можуть містити статичні конструктори, але не можуть мати деструкторів. Кожна структура має конструктор за замовчуванням без параметрів. Такий конструктор присвоює кожному елементу структури значення за замовчуванням відповідно до його типу. Значущим елементам присвоюється відповідне до типу значення, а посилальним – null.

Визначений мовою C# конструктор без параметрів неможливо видалити чи перевизначити. Але у структурі можна оголосити нові конструктори з параметрами. Зауважимо, що ситуація з конструкторами у структурах та класах різна. Для класів підтримується неявний конструктор без параметрів тільки тоді, коли інших конструкторів не оголошено.

Якщо у структурі оголошено конструктор, то в ньому обов'язково потрібно ініціалізувати всі поля структури. В іншому випадку виникне помилка компіляції.

Щоб викликати конструктор (у тому числі конструктор за замовчуванням), використовують оператор new. Як вже наголошувалося, при цьому окрема від самої змінної пам'ять у купі не виділяється.

У лістингу 6.4 продемонстровано структуру з конструктором, який має два параметри: aX та aY (рядки 5…9). Далі у методі Main оголошено два екземпляри структури: перша (p1) з використанням конструктора за замовчуванням і друга (p2) з використанням оголошеного двопараметричного конструктора.

При цьому поля структури p1 будуть ініціалізовані значеннями за замовчуванням для їх типу (int), тобто нулями. Поля структури p2 будуть ініціалізовані переданими конструктору значеннями (у даному випадку – 5 та 50).

Лістинг 6.4. Конструктор структури

```
1  struct SPoint
2  {
3      public int X;
4      public int Y;
5      public SPoint(int aX, int aY)
6      {
7          X = aX;
8          Y = aY;
9      }
10 }
11 ...
12 static void Main()
13 {
14     SPoint p1 = new SPoint();
15     SPoint p2 = new SPoint(5, 50);
16 }
```

Виклик неявного конструктора

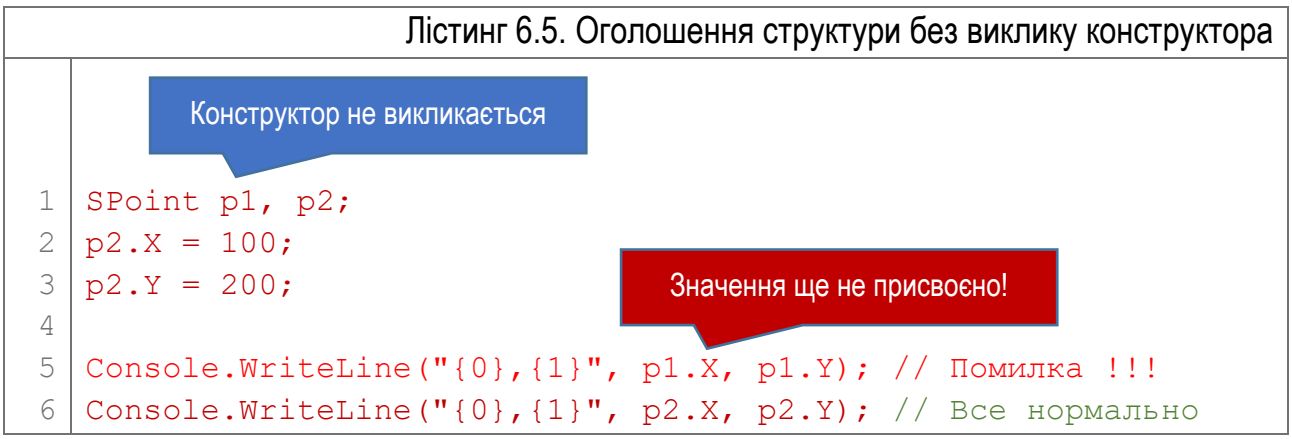
Виклик оголошеного конструктора

Можна також використовувати екземпляр структури без виклику конструктора (і без використання оператора new). У цьому випадку:

- елементи даних не можна використовувати, доки їм явно не буде присвоєно значення;
- не можна викликати жодного функціонального елемента структури, доки всім елементам даних не буде присвоєно значення.

У лістингу 6.5 приведено оголошення двох структур без виклику конструктора (і без оператора new). Далі полям другої структури присвоєно значення, а поля першої структури залишено неініціалізованими. При компіляції такого коду у рядку 5 виникне помилка (спроба доступу до неініціалізованих полів структури).

Лістинг 6.5. Оголошення структури без виклику конструктора




```
1  SPoint p1, p2;
2  p2.X = 100;
3  p2.Y = 200;
4
5  Console.WriteLine("{0},{1}", p1.X, p1.Y); // Помилка !!!
6  Console.WriteLine("{0},{1}", p2.X, p2.Y); // Все нормально
```

Структура може містити статичні конструктори. Вони повинні ініціалізувати статичні елементи даних і не можуть посилатися на елементи екземпляра. Статичний конструктор автоматично викликається у таких випадках:

- при виклику явно оголошеного конструктора;
- при звертанні до статичного елемента структури.

Структури як параметри та результати методів

Структури, як і інші типи даних, можуть бути використані для параметрів та результатів методів. При цьому мають місце такі особливості:

- Якщо метод повертає результат типу "структура", то створюється копія структури з методу і передається у викликаючий код.
- Якщо структуру використовують як параметр, то створюється копія дійсної структури, яка і передається у метод. Ця копія використовується при виконанні методу.
- Якщо структуру використовують як посилальний (ref) або вихідний (out) параметр, то у метод передається посилання на структуру, так що у вихідній структурі можуть бути змінені елементи даних.

У лістингу 6.6 описано метод Distance, який містить два формальних параметри – структури p1 та p2. Перед викликом цього методу у програмі оголошено та ініціалізовано дві структури (pa та pb), які виступають в ролі дійсних параметрів методу Distance.

Лістинг 6.6. Передача структур як параметрів методу

```
1  static public double Distance(SPoint p1, SPoint p2)
2  {
3      int dX = p2.X - p1.X;
4      int dY = p2.Y - p1.Y;
5      return Math.Sqrt(dX * dX + dY * dY);
6  }
7  static void Main()
8  {
9      SPoint pa = new SPoint(10, 10), pb = new SPoint(30, 30);
10     double d = Distance(pa, pb);
11 }
```

Структури в ролі параметрів

Додаткові дані щодо структур

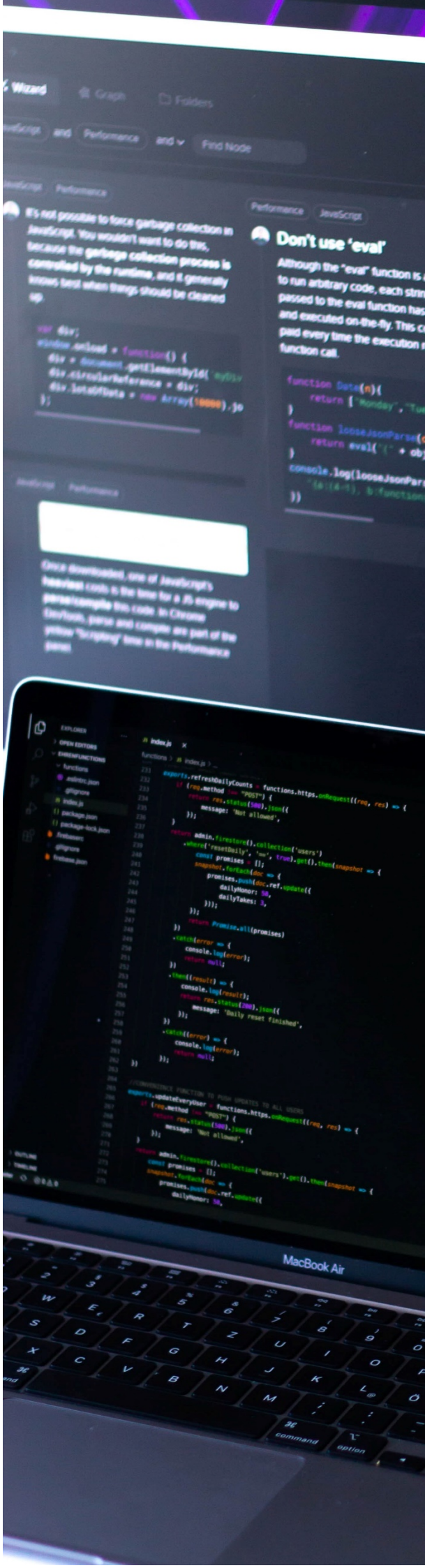
Загалом, виділення пам'яті для структури потребує менше обчислювальних витрат, ніж створення екземпляра класу. Тому використання структур замість класів в ряді випадків може покращити продуктивність програми. Проте, операції запаковування (boxing) та розпаковування (boxing) при роботі зі структурами потребують додаткових обчислювальних витрат.

Зауважимо, що попередньо визначені типи даних (int, uint, short та інші), хоча і розглядаються як цілісні примітиви у C# та .NET, насправді у внутрішньому механізмі .NET реалізовані як структури.

Структури можна оголошувати як часткові (partial). Структури також можуть впроваджувати інтерфейси.

Розділ 7. Масиви

- Масиви
- Види масивів
- Одновимірні та прямокутні масиви
- Зубчаті масиви
- Порівняння прямокутних та зубчатих масивів
- Інструкція foreach
- Коваріантність масивів
- Використання успадкованих елементів класу Array



Масиви

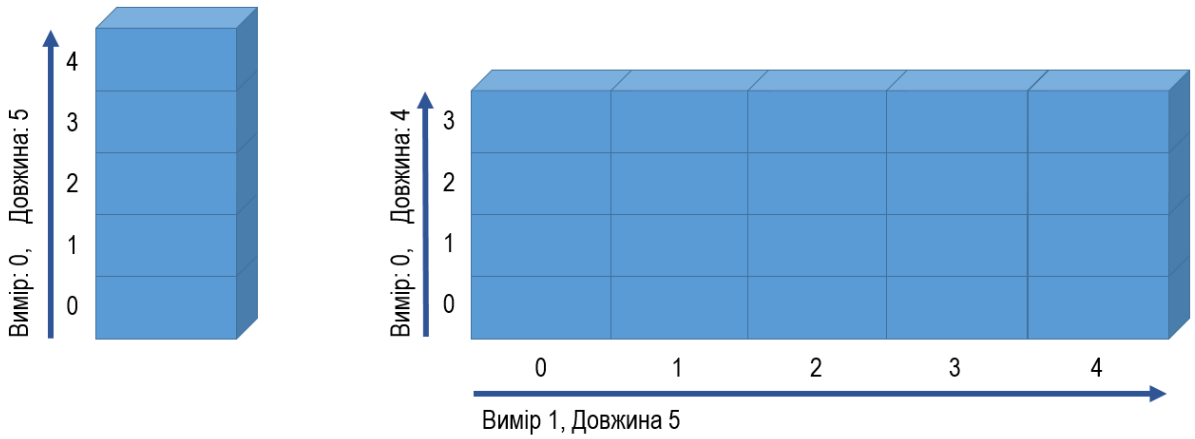
Масив – це множина однотипних елементів даних, представлених однією змінною. До окремих елементів масиву звертаються через його назву та один чи кілька індексів, які вказують у квадратних дужках після назви:

```
TheArray[12]
```

Назва масиву | Індекс елемента

Всі елементи масиву повинні мати однаковий тип. Масив може мати будь-яку кількість *вимірів*. Кількість вимірів масиву називають його *рангом*. Кожен вимір масиву має свою *довжину*, яка задає кількість позицій у цьому вимірі. Загальну кількість елементів масиву у всіх вимірах разом називають *довжиною масиву*.

Розмір створеного масиву є фіксованим. C# не підтримує динамічних масивів, довжину яких можна змінювати під час роботи програми[15]. Всі індекси масивів починаються з нуля. Це означає, що якщо довжина виміру масиву дорівнює N, то індекс елементів у цьому вимірі може змінюватися від 0 до N-1. На рис. 7.1 показано виміри та довжини двох прикладів масивів: одновимірного та двовимірного.



Одновимірний масив int[5]
Ранг: 1
Довжина масиву: 5

Двовимірний масив int[4,5]
Ранг: 2
Довжина масиву: 20

Рис. 7.1. Виміри та розміри масивів

Екземпляр масиву – це об'єкт, похідний від класу Array з простору імен System. Клас Array визначено у бібліотеці базових класів .NET. Він має свій

[15] Для роботи з сукупністю даних, кількість яких може змінюватися під час роботи програми, у C# використовують колекції, розглянуті далі.

набір методів та властивостей. Наприклад, властивість Rank цього класу повертає кількість вимірів масиву, а властивість Length – довжину масиву (загальну кількість всіх елементів).

Види масивів

Масиви можуть бути різних видів. За кількістю вимірів розрізняють:

- Одновимірні масиви. Їх можна уявити як лінійну послідовність елементів – *вектор*.
- Багатовимірні масиви. Вони побудовані так, що кожен елемент первинного вектора (одновимірного масиву) також є масивом, який називають *субмасивом*. Елементами кожного субмасива також можуть бути інші субмасиви.

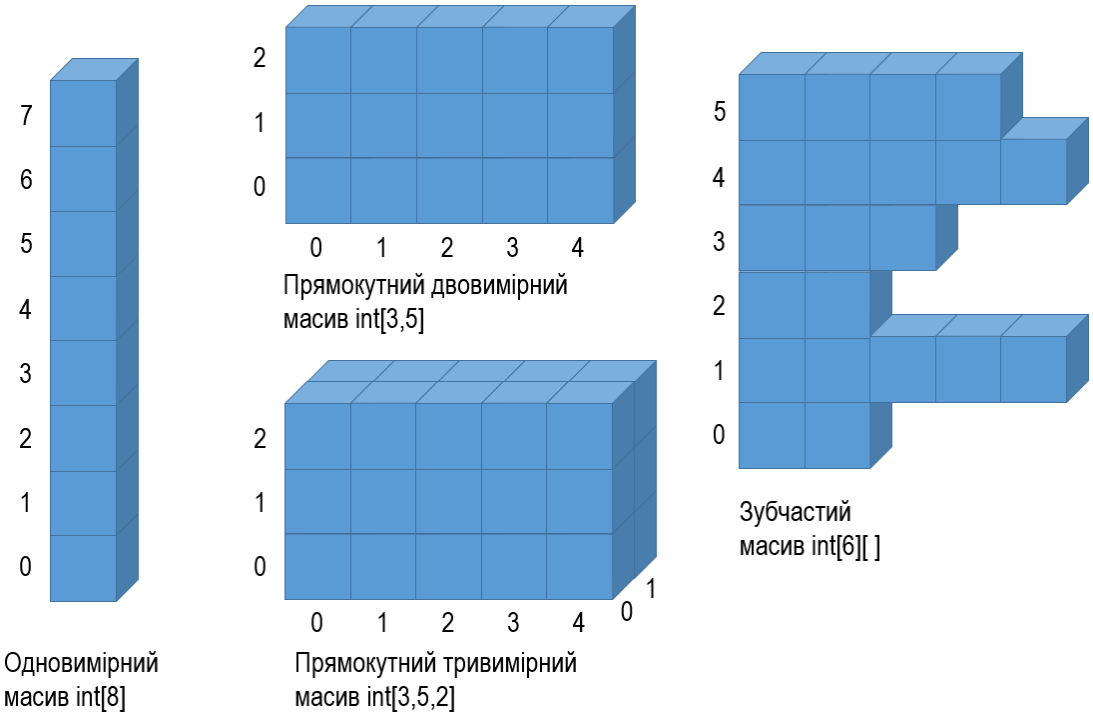


Рис. 7.2. Види масивів: одновимірний, прямокутний та зубчастий

Крім цього, розрізняють такі два типи багатовимірних масивів:

- Прямокутні масиви. У ньому всі субмасиви одного виміру мають однакову довжину. При роботі з такими масивами завжди використовують тільки одну пару квадратних дужок, незалежно від кількості вимірів:
  ```
  theArray[3, 5, 9]
  ```
- Зубчасті масиви. У такому масиві кожен субмасив є незалежним від інших субмасивів. Його субмасиви можуть мати різну довжину. При

роботі з таким масивом використовують окремі пари квадратних дужок для кожного виміру:

```
jagArray[3][5][9]
```

Різні види масивів проілюстровано на рис. 7.2.

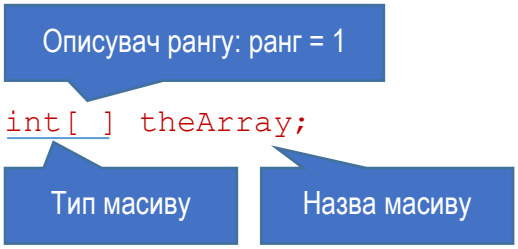
Масив є посилальним типом, тому його змінна є посиланням на ділянку пам'яті з даними. У той же час елементами масиву можуть бути як значущі, так і посилальні типи. Масив є *значущого типу*, якщо його елементи належать до значущого типу. Масив *посилального типу* містить елементи посилального типу. У ньому елементи є посиланнями на об'єкти.

Одновимірні та прямокутні масиви

При оголошенні одновимірного або прямокутного масиву використовують одну пару квадратних дужок після типу. Коми в цих дужках є *описувачами виміру* масиву. Вони задають кількість вимірів у масиві. Ранг масиву завжди на один більший за кількість ком (наприклад, одна кома означає два виміри).

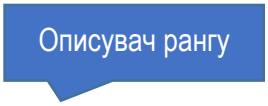
Тип масиву містить базовий тип та описувачі виміру.

При оголошенні масиву вказують його тип та назву:



Кількість вимірів масиву може бути будь-яка. Тип масиву не містить даних про довжину виміру, тому в типі не вказують конкретних довжин. Вони будуть задані при створенні екземпляра масиву. Таким чином, коли оголошено масив, кількість його вимірів є фіксованою. Але довжина вимірів може бути різною: її задають при створенні масиву.

Ось приклади оголошень масивів:




```
int[,] the2DArray;        // Двовимірний масив int
long[,,] the3DArray;      // Тривимірний масив long
double[,,,] the4DArray;   // Чотиривимірний масив double
```

Оголошення масиву приводить до виділення пам'яті тільки для посилання на масив. Щоб в масив можна було помістити дані, потрібно створити екземпляр масиву. Для цього використовують оператор new, після якого

вказують базовий тип та в квадратних дужках задають довжину кожного виміру. Ось приклади створення масивів різних типів:

```
int[,] the2DArray = new int[2,4];
long[,,] the3DArray = new long[3,5,7];
double[,,,] the4DArray = new double[3,5,7,9];
MobilePhone[] theClassArr = new MobilePhone[10];
```

Останній масив містить елементи-посилання на об'єкти класу MobilePhone.

Звертаються до елементів масиву, вказуючи у квадратних дужках індекс елемента для кожного виміру:

```
the2DArray[1,2] = 10;
the3DArray[3,0,4] = 20;
```

Лістинг 7.1 демонструє створення масиву, запис у нього даних та виведення їх на екран. У рядку 3 оголошено змінну одновимірного масиву intArr. Сам масив довжиною п’ять елементів створюємо в рядку 4. Після цього у циклі кожному елементу масиву присвоюємо певне значення. Далі (рядки 8…9) виводимо значення елементів масиву у вікно (рис. 7.3).

Лістинг 7.1. Робота з одновимірним масивом

```
1  static void Main(string[] args)
2  {
3      int[] intArr;
4      intArr = new int[5];
5      for (int i = 0; i < intArr.GetLength(0); i++)
6          intArr[i] = i * 10;
7
8      for (int i = 0; i < intArr.GetLength(0); i++)
9          Console.WriteLine("intArr[{0}] = {1}", i, intArr[i]);
10 }
```

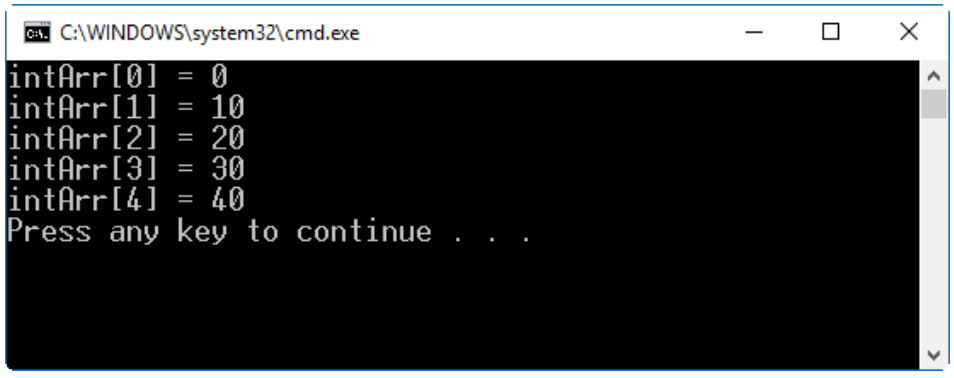


Рис. 7.3. Результат роботи програми з лістингу 7.1

Коли масив створюється, всі його елементи автоматично ініціалізуються значеннями за замовчуванням для його базового типу. Для числових даних це 0, для логічних – false, для посилальних – null.

Щоб явно ініціалізувати масив при його створенні вказують множину значень елементів у *списку ініціалізації*. Список ініціалізації розміщують у фігурних дужках, а значення у ньому розділяють комами.

Ось приклад ініціалізації одновимірного масиву:

```
int[] intArr = new int[5] { 10, 20, 30, 40, 50 };
```

Список ініціалізації

Прямокутні масиви ініціалізують, використавши на місці кожного елемента вкладений список значень у фігурних дужках.

Нижче приведено приклад оголошення, створення та ініціалізації двовимірного прямокутного масиву int2DArray та тривимірного масиву int3DArray:

```
int[,] int2DArray =
    new int[3,2] { { 10, 1 }, { 2, 10 }, { 11, 9 } };
int[,,] int3DArray = new int[4, 3, 2] {
    { {8, 6}, {5, 2}, {2, 9} },
    { {6, 4}, {3, 9}, {8, 4} },
    { {7, 2}, {1, 3}, {9, 3} },
    { {4, 6}, {3, 2}, {3, 8} }
};
```

Якщо масив оголошується, створюється і відразу ініціалізується, то вираз для створення масиву можна взагалі опустити. Інформацію про довжину компілятор отримує зі списку ініціалізації. Таким чином, наступні інструкції еквівалентні (друга є скороченим записом першої):

```
int[] intArr = new int[5] { 10, 20, 30, 40, 50 };
int[] intArr =            { 10, 20, 30, 40, 50 };
```

Зубчаті масиви

Зубчатий масив можна розглядати як масив масивів. Він може мати довільну, але більшу від 1, кількість вимірів. Кожен субмасив зубчатого масиву може мати різну довжину. Оголошення зубчатого масиву потребує пари квадратних дужок для кожного виміру. Як і для прямокутних масивів, довжини не вказують у типі масиву.

Ось приклади оголошення зубчатих масивів:

Описувачі рангу: ранг = 2

```
int[][] SomeArr;       // Ранг = 2
int[][][] OtherArr;    // Ранг = 3
```

Тип масиву

Назва масиву

Під час оголошення зубчатого масиву можна відразу створити його субмасив першого рівня (але тільки першого рівня):

```
int[][] jagArr = new int[3][];
```

Кожен з елементів зубчатого масиву є окремим масивом, який також потребує створення. Тому при створенні масиву можна вказати лише один вимір, а наступна інструкція приведе до виникнення помилки компіляції:

```
int[][] jagErrorArr = new int[3][5];     // Помилка!
```

На відміну від масивів інших типів, зубчатий масив неможливо створити весь за один крок. Оскільки він складається з інших незалежних масивів, то кожен з них слід створити окремо. Таким чином, для створення повного екземпляру зубчатого масиву потрібно:

- Створити масив першого рівня.
- Створити кожен субмасив інших рівнів окремо, присвоюючи посилання на новостворений субмасив відповідному елементу субмасиву попереднього рівня.

Наприклад, створення та ініціалізацію двовимірного зубчатого масиву можна здійснити так:

```
// 1. Створення масиву першого рівня
int[][] theArr = new int[3][];
// 2. Створення субмасивів інших рівнів
theArr[0] = new int[] { 10, 20, 30 };
theArr[1] = new int[] { 100, 200, 300, 400 };
theArr[2] = new int[] { 1000, 2000, 3000 };
```

У пункті 1 оголошено та створено масив першого рівня з трьох елементів. В пункті 2 створено три субмасиви для кожного з базових трьох елементів масива першого рівня.

Оскільки субмасиви в зубчатому масиві також є масивами, то всередині зубчатого масива можна використовувати прямокутні масиви.

Порівняння прямокутних та зубчатих масивів

Структура прямокутних та зубчатих масивів суттєво відрізняється, навіть якщо їх розмірність однакова. На рис. 7.4 приведено схему розміщення в пам'яті даних прямокутного та зубчатого масивів розмірністю 3×3 елементи. Прямокутний масив складається з одного цілісного об'єкта, тоді як зубчастий містить чотири різних об'єкти: один для масиву першого рівня, і по одному – для кожного елемента масиву першого рівня.

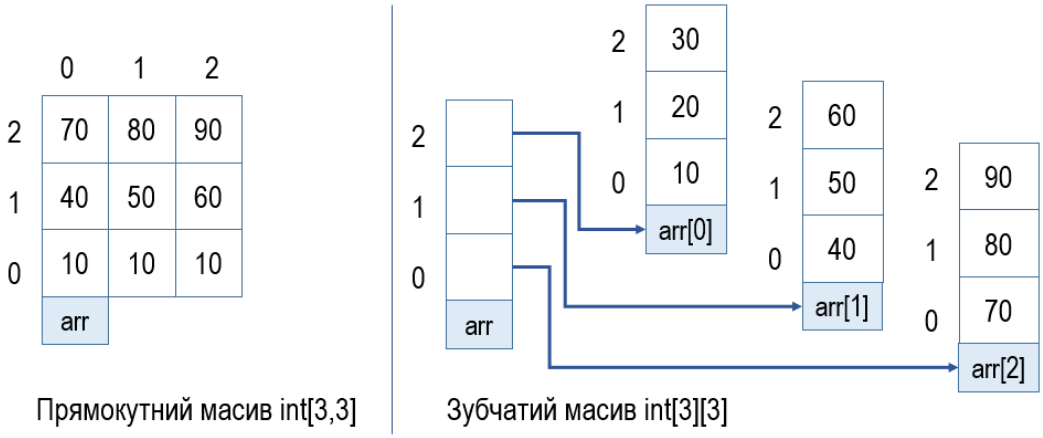


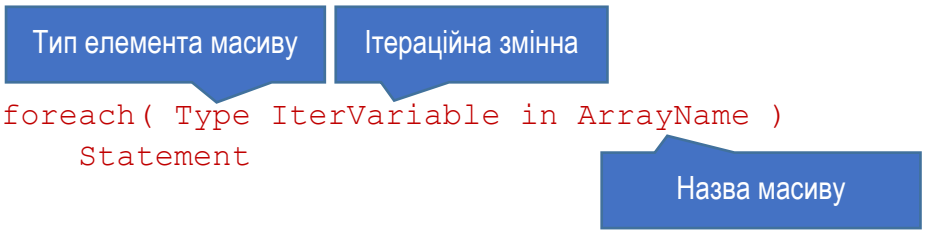
Рис. 7.4. Структура прямокутного та зубчатого масиву у пам'яті

.NET Framework містить оптимізований механізм для роботи з одновимірними масивами, який забезпечує кращу продуктивність порівняно з прямокутними. Тому інколи, особливо для масивів значних розмірів, може бути ефективніше використовувати оптимізований зубчатий масив з одновимірними субмасивами, ніж єдиний прямокутний масив.

Інструкція foreach

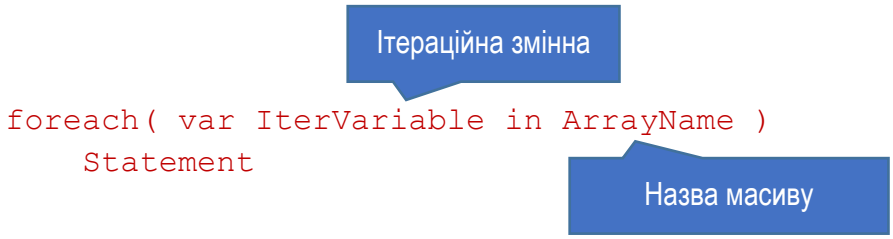
Інструкція foreach дозволяє послідовно вибирати кожен елемент масиву. Вона повторює групу вкладених операторів для кожного елемента масиву або колекції об'єктів, що реалізують інтерфейс System.Collections.IEnumerable або System.Collections.Generic.IEnumerable<T> (колекції розглянемо далі).

В інструкції foreach використовують *ітераційну змінну*, яка є тимчасовою змінною того ж типу, що й елементи масиву. Ітераційна змінна послідовно представляє кожен елемент масиву. Синтаксис використання інструкції foreach такий:



Ітераційна змінна IterVariable під час кожної ітерації (проходження) циклу містить значення відповідного елемента масиву ArrayName. Інструкція Statement, яка виконується для кожного елемента, може бути представлена блоком інструкцій.

Тип ітераційної змінної можна задати неявно, вказавши замість типу ключове слово var (тоді компілятор автоматично підбере тип змінної відповідно до типу масиву):



Але обидві приведені форми інструкції foreach семантично однакові.

Інструкція foreach працює так:

- Починаючи з першого елемента масиву, присвоює значення кожного елемента ітераційній змінній.
- Виконує пов'язану з foreach інструкцію (чи блок інструкцій). При цьому ітераційну змінну можна використовувати як доступний тільки для читання псевдонім елемента масиву.

Таким чином, інструкція foreach дозволяє пройти у циклі по всіх елементах масиву і виконати пов'язані з ними дії. Лістинг 7.2 містить приклад програми, у якій оголошено масив цілих чисел arr (рядок 3) і виведено його значення у вікно консолі за допомогою циклу foreach (рядки 4…5). В ролі ітераційної змінної використано змінну ai цілого типу. Інструкція з викликом WriteLine виконується послідовно для кожного елемента масиву. Під час першого проходження ітераційна змінна ai містить значення першого елемента масиву, потім – другого і т.д. Результат виводу програми приведено на рис. 7.5.

Лістинг 7.2. Вивід значень елементів масиву у циклі foreach

```
1  static void Main(string[] args)
2  {
3      int[] arr = { 10, 20, 30, 40, 50 };
4      foreach (int ai in arr)
5          Console.WriteLine("Значення елемента: {0}", ai);
6  }
```

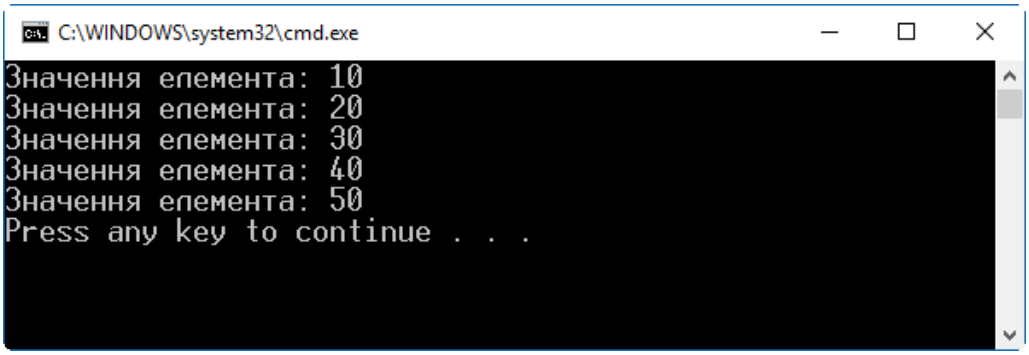


Рис. 7.5. Результат роботи програми з лістингу 7.2

Інструкцію foreach використовують для отримання інформації про елемент. Її не варто використовувати для додавання або видалення елементів, щоб уникнути непередбачених побічних ефектів, оскільки при цьому змінюється структура масиву і поведінка foreach може бути непередбачуваною. Якщо потрібно додати або видалити елементи масиву, слід використовувати цикл for.

Значення через ітераційну змінну доступне тільки для читання і не може бути змінене. Це по-різному впливає на масиви значущих та посилальних типів. У масиві значущого типу елемент за допомогою ітераційної змінної неможливо змінити. Але елементами масиву посилального типу є посилання на об'єкти з даними. Хоча само посилання все одно не можна змінити, але дані, на які вказує посилання в ітераційній змінній, доступні для редагування. Тому в масиві з даними посилального типу самі дані за посиланням можна змінювати. Але створити новий об'єкт з новим посиланням, і присвоїти це посилання ітераційній змінній неможливо.

Коли інструкція foreach застосовується до прямокутного масиву, то елементи обробляються починаючи з крайнього правого (як записано в оголошенні) виміру. Коли його значення проходить від 0 до Length-1, збільшується на одиницю наступний індекс лівіше і т.д. Це ілюструє приклад з лістингу 7.3. Якщо його виконати, то послідовно буде виведено елементи прямокутного двовимірного масиву (рис. 7.6).

Лістинг 7.3. Вивід значень елементів масиву у циклі foreach

```
1  static void Main(string[] args)
2  {
3      int[,] arr = { { 10, 20 }, { 30, 40 } };
4      foreach (int ai in arr)
5          Console.WriteLine("Значення елемента: {0}", ai);
6  }
```

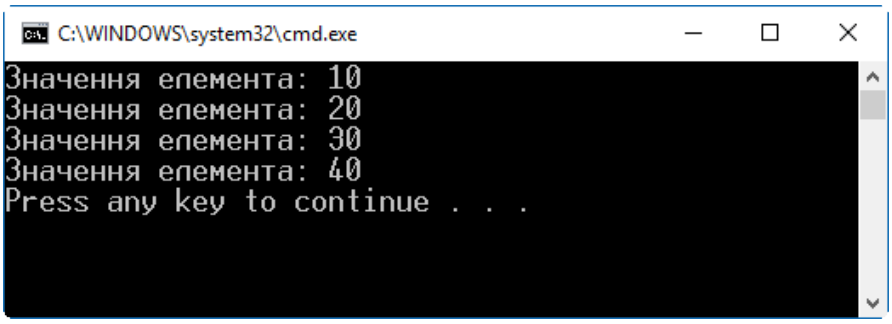


Рис. 7.6. Результат роботи програми з лістингу 7.3. Вивід елементів прямокутного двовимірного масиву

Використання інструкції foreach щодо зубчатого багатовимірного масиву дещо відрізняється від її використання для прямокутного масиву. Оскільки зубчатий масив представляє собою масиви у масиві, то для різних його вимірів потрібно використовувати окремі вкладені інструкції foreach. Це ілюструє лістинг 7.4, у якому оголошено зубчатий масив arr. Зовнішня інструкція foreach (рядки 8…13) проходить по елементах першого рівня, кожен з яких є субмасивом. Внутрішня (рядки 11…12) послідовно проходить по елементах кожного субмасиву і виводить їх значення у консоль. Перед виводом значень кожного субмасиву виводиться повідомлення про початок обробки субмасиву (рядок 10). Результат виводу програми приведено на рис. 7.7.

Лістинг 7.4. Використання циклу foreach для зубчатого масиву

```
1  static void Main(string[] args)
2  {
3      int[][] theArr = new int[3][];
4      theArr[0] = new int[] { 10, 20, 30 };
5      theArr[1] = new int[] { 100, 200, 300, 400 };
6      theArr[2] = new int[] { 1000, 2000, 3000 };
7
8      foreach (int[] arr in theArr)
9      {
10         Console.WriteLine("Субмасив:");
11         foreach(int i in arr)
12             Console.WriteLine("Значення елемента: {0}", i);
13     }
14 }
```

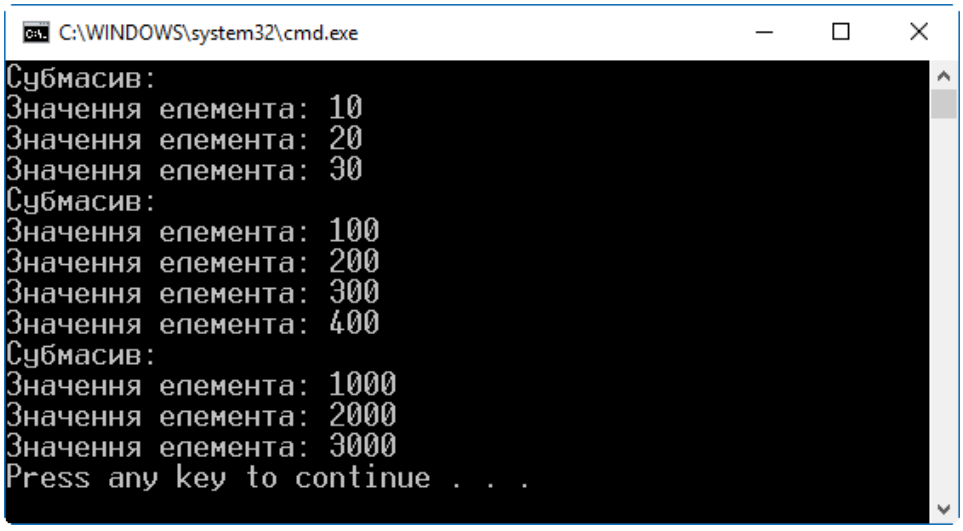


Рис. 7.7. Результат роботи програми з лістингу 7.4. Вивід елементів зубчатого двовимірного масиву

Коваріантність масивів

Елементу масиву посилального типу можна присвоїти посилання на об'єкт, тип якого відмінний від базового типу масиву. Це називають *коваріантністю* масиву. При цьому слід використати явне чи неявне приведення типу об'єкта, який записують у масив, до типу масиву. Оскільки завжди існує неявне приведення типу між похідним і базовим класом, то об'єкт похідного класу можна без додаткових операцій записувати у масив, оголошений з типом базового класу. Масиви значущих типів не є коваріантними.

Використання успадкованих елементів класу Array

Масив у C# є похідним від класу Array з простору імен System. Від свого базового класу масив спадкує ряд властивостей і методів, які полегшують роботу з масивом. Їх приведено у таблиці 7.1.

Використання деяких із них продемонстровано програмою у лістингу 7.5. У рядку 9 оголошено одновимірний масив цілих чисел та ініціалізовано його несортованим набором значень. У рядку 11 за допомогою статичного методу Sort сортуємо масив за зростанням. Після цього за допомогою статичного методу Reverse змінюємо порядок масиву на протилежний (рядок 14). В кінці виводимо загальну інформацію про масив: ранг, довжину, тип тощо (рядки 18…21). Результат виконання програми приведено на рис. 7.8.

Таблиця №7.1. Елементи класу Array

Назва елемента	Вид	Опис
Властивості		
Rank	Екземпляр	Повертає кількість вимірів у масиві
Length	Екземпляр	Повертає загальну кількість елементів у всіх вимірах
Методи		
GetLength	Екземпляр	Повертає довжину вказаного виміру
Clear	Статичний	Задає для діапазону значень масиву значення за замовчуванням для типу
Sort	Статичний	Сортує елементи одновимірного масиву
Clone	Екземпляр	Створює копію масиву. Копіюються тільки елементи масиву
IndexOf	Статичний	Повертає індекс першого входження вказаного значення в одновимірному масиві
Reverse	Статичний	Міняє порядок елементів одновимірного масиву на протилежний
GetLowerBound	Екземпляр	Повертає нижню межу (індекс) заданого виміру
GetUpperBound	Екземпляр	Повертає верхню межу (індекс) заданого виміру
GetType	Екземпляр	Повертає тип масиву

Лістинг 7.5. Використання успадкованих елементів класу Array

```
1  public static void PrintArray(int[] a)
2  {
3      foreach (int i in a) Console.Write("{0} ", i);
4      Console.WriteLine();
5  }
6
7  static void Main(string[] args)
8  {
9      int[] arr = new int[] { 30, 20, 40, 70, 60, 50 };
10     Console.Write("Несортовано: ");  PrintArray(arr);
11     Array.Sort(arr);
12     Console.Write("Сортовано: ");    PrintArray(arr);
13
14     Array.Reverse(arr);
15     Console.Write("Реверс: ");       PrintArray(arr);
16
17     Console.WriteLine();
18     Console.WriteLine("Rank = {0}, Length = {1}",
19         arr.Rank, arr.Length);
20     Console.WriteLine("GetLength(0)          =          {0}",
21 arr.GetLength(0));
22     Console.WriteLine("GetType() = {0}", arr.GetType());
   }
```

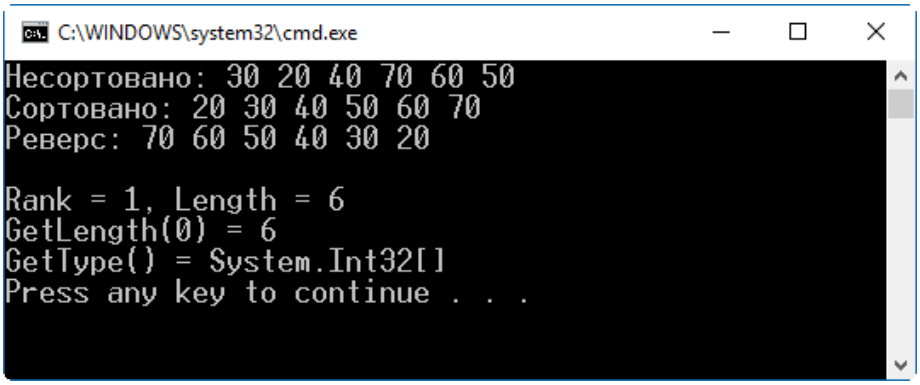


Рис. 7.8. Результат роботи програми з лістингу 7.5. Використання успадкованих елементів класу Array

Клонування масиву

Успадкований метод Clone виконує поверхову копію масиву. При цьому копіюються виключно елементи масиву. У випадку з масивом значущого типу створюється його копія і результатом є два незалежні однакові масиви. Якщо масив є посилального типу, то об’єкти, на які посилаються елементи масиву, не

копіюються. Тобто, при створенні такої копії масиву елементи нового масиву посилатимуться на ті ж самі об’єкти у пам’яті, що й елементи початкового масиву (рис. 7.9). Тому, якщо змінити об’єкт через посилання одного масиву, то ці зміни відобразяться і через посилання з тим же індексом елемента клонованого масиву.

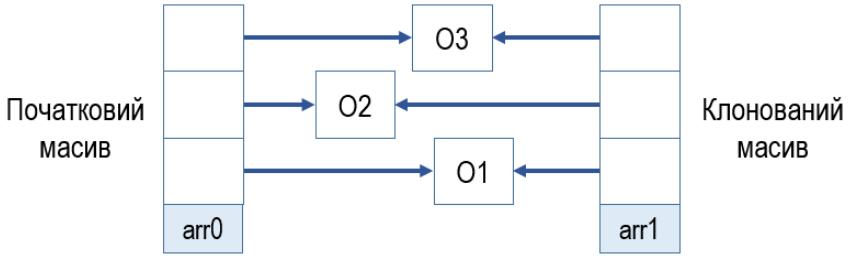
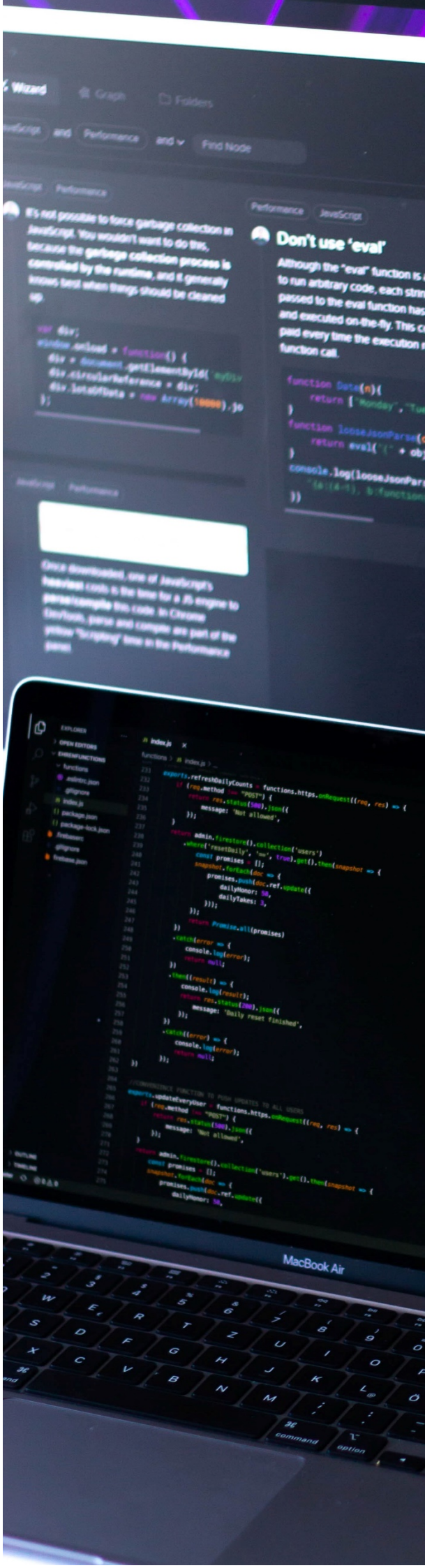


Рис. 7.9. При клонуванні масиву посилального типу елементи з однаковим індексом обох масивів вказують на той самий об’єкт

Розділ 8. Інтерфейси

- Інтерфейси
- Приклад використання інтерфейсу
- Інтерфейс як посилальний тип даних
- Реалізація кількох інтерфейсів
- Спадкування інтерфейсів



Інтерфейси

Інтерфейс – це посилальний тип, який задає множину функціональних елементів, але не реалізовує їх. Інтерфейс можуть реалізовувати (впроваджувати) класи та структури. Семантично інтерфейси схожі на абстрактні класи, але жоден метод інтерфейсу не може мати реалізації. Інтерфейс показує, що клас повинен робити, але не визначає, як саме. При впровадженні інтерфейсу у клас мають бути реалізовані всі методи інтерфейсу, але кожен клас може реалізувати їх по-іншому.

Розглянемо приклад, який висвітлить недоліки, що усуваються при використанні інтерфейсу. Припустимо, що ми маємо достатньо великий набір товарів різних видів. Кожен вид має притаманні йому характеристики. Так, товар виду "мобільний телефон" має одні характеристики, а товар виду "кухонний комбайн" – інші. На основі існуючого набору товарів потрібно скласти їх загальний список. При цьому кожен елемент списку міститиме коротку характеристику товару і формуватиметься на основі характеристик свого виду.

У лістингу 8.1 оголошено два класи. Клас MobilePhone (рядки 1…12) описує товари виду "мобільний телефон", а клас FoodProcessor (рядки 14…24) – товари виду "кухонний комбайн". Класи містять різний набір полів, які характеризують відповідний вид товару. Кожен клас має метод, який повертає опис об'єкта, що формується на основі його полів. При цьому для опису екземплярів різних класів використовуються різні поля різних типів. Метод PhoneDescription (рядки 7…11) формує опис конкретного об'єкта "мобільний телефон", а метод FoodProcessorDescription (рядки 19…23) – об'єкта "кухонний комбайн".

У методі Main створено по одному екземпляру кожного класу. Інформація про об'єкти виводиться у вікно шляхом виклику методів, які повертають їх опис (рядки 41, 42).

Зауважимо, що для розглянутого підходу формування описів об'єктів характерні наступні риси, які ускладнюють і заплутують структуру програми:

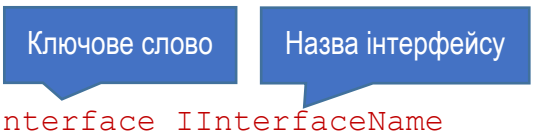
- Методи, які повертають інформацію про об'єкт, належать різним класам і можуть мати різну назву. Щоб вивести опис, потрібно пам'ятати назви прийнятних для даного випадку методів (адже клас може містити багато методів).
- Потрібний для опису об'єкта метод у класі взагалі може бути відсутнім.
- Екземпляри різних класів неможливо передати у інший метод для обробки через один параметр, оскільки вони мають різні типи. Так, у метод PrintUpperDescr можна передати тільки об'єкт класу MobilePhone.

Лістинг 8.1. Використання класів, які описують різні товари

```
class MobilePhone                       // Мобільний телефон
{
    public string Manufacturer;         // Виробник
    public string Model;                // Модель
    public double ScreenSize;           // Розмір екрану
    public double MemorySize;           // Обсяг пам'яті
    public string PhoneDescription() // Опис
    {
        return String.Format("{0} {1} {2}\" {3}Гб",
            Manufacturer, Model, ScreenSize, MemorySize);
    }
}

class FoodProcessor                     // Кухонний комбайн
{
    public string Manufacturer;         // Виробник
    public string ModelInfo;            // Модель
    public double Power;                // Потужність
    public string FoodProcessorDescription()  // Опис
    {
        return String.Format("{0} {1} {2}Вт",
            Manufacturer, ModelInfo, Power);
    }
}

static void PrintUpperDescr(MobilePhone m)
{
    Console.WriteLine(m.PhoneDescription().ToUpper());
}

static void Main(string[] args)
{
    MobilePhone mp = new MobilePhone()
        { Manufacturer = "PhoneMan", Model = "KL-289i+",
          MemorySize = 2, ScreenSize = 7 };

    FoodProcessor fp = new FoodProcessor()
        { Manufacturer = "KitchenTec",
          ModelInfo = "Astra FV-202", Power = 12 };

    Console.WriteLine(mp.PhoneDescription());
    Console.WriteLine(fp.FoodProcessorDescription());
    PrintUpperDescr(mp);
}
```

Використання інтерфейсу дозволяє реалізувати логічніший та зрозуміліший підхід. Для цього слід оголосити інтерфейс, який матиме метод, призначений для формування опису об'єкта. Після цього кожен з класів, що представляє об'єкти різних видів, повинен реалізувати цей інтерфейс. Застосування інтерфейсу дозволить уникнути всіх трьох приведених недоліків розглянутого коду.

Загальний синтаксис оголошення інтерфейсу такий:




```
interface IInterfaceName
{
    // Заголовки функціональних елементів інтерфейсу
}
```

Назви інтерфейсам прийнято починати з великої літери "I". Елементами інтерфейсу можуть бути методи, властивості, індексатори та події. Інтерфейс не може містити елементів даних та статичних елементів. В інтерфейсі розміщують виключно **оголошення** функціональних елементів, а їх **тіла** повинні бути реалізовані у класах, які спадкують інтерфейс. За змістом інтерфейс нагадує абстрактний клас. Але для інтерфейсу та його елементів не використовують ключового слова abstract.

Оголошення інтерфейсу може містити модифікатори доступу public, protected, internal чи private. Проте елементи інтерфейсу неявно є тільки публічними і не можуть містити модифікаторів доступу (включаючи public). Клас, який впроваджує інтерфейс, обов'язково повинен реалізувати **всі** елементи інтерфейсу. При цьому код для реалізації елементів інтерфейсу може як реалізуватися самим класом, так і спадкуватися від базових класів.

Хоча оголошення інтерфейсу не містить реалізації його методів, документація для інтерфейсу повинна містити опис того, що вони виконують. Програміст, який створюватиме клас з інтерфейсом, повинен знати призначення його елементів.

Щоб впровадити інтерфейс у клас, слід при оголошенні класу вказати назву інтерфейсу в переліку базових класів:


```
class TheClass : IInterfaceName
{ ... }
```

Якщо клас, у який впроваджують інтерфейс, є похідним, то його базовий клас у списку успадкованих класів повинен бути першим. Після назви базового класу через кому можна вказати довільну кількість інтерфейсів (у той же час базовий клас може бути тільки один):


```
class TheClass : TheBaseClass, IInterface1, IInterface2
{ ... }
```

У лістингу 8.2 оголошено інтерфейс IInfo (рядки 1…5) з одним методом GetDescription рядкового типу та однією властивістю ItemMark рядкового типу тільки для читання. Метод GetDescription має повертати опис об'єкта, а властивість ItemMark – "марку" об'єкта з базовою інформацією про виробника та модель. Інтерфейс IInfo містить оголошення методу та властивості без їх реалізації.

Лістинг 8.2. Оголошення та реалізація інтерфейсу

```
 1  interface IInfo                 // Оголошення інтерфейсу
 2  {
 3      string GetDescription(); // Метод
 4      string ItemMark { get; } // Властивість
 5  }
 6
 7  class MobilePhone : IInfo       // Клас реалізує інтерфейс IInfo
 8  {
 9      public string Manufacturer;
10      public string Model;
11      public double ScreenSize;
12      public double MemorySize;
13
14      public string ItemMark
15      {
16          get { return String.Format("{0} {1}",
17              Manufacturer, Model); }
18      }
19      public string GetDescription()
20      {
21          return String.Format("{0} {1} {2}\" {3}Гб",
22              Manufacturer, Model, ScreenSize, MemorySize);
23      }
24  }
25
26  class FoodProcessor : IInfo  // Клас реалізує інтерфейс IInfo
27  {
28      public string Manufacturer;
29      public string ModelInfo;
30      public double Power;
31
32      public string ItemMark
33      {
34          get { return String.Format("{0} {1}",
```

```
35              Manufacturer, ModelInfo); }
36      }
37      public string GetDescription()
38      {
39          return String.Format("{0} {1} {2}Вт",
40              Manufacturer, ModelInfo, Power);
41      }
42  }
43
44  static void PrintUpperDescr(IInfo obj)
45  {
46      Console.WriteLine(obj.GetDescription().ToUpper());
47  }
48
49  static void Main(string[] args)
50  {
51      MobilePhone mp = new MobilePhone()
52          { Manufacturer = "PhoneMan", Model = "KL-289i+",
53            MemorySize = 2, ScreenSize = 7 };
54
55      FoodProcessor fp = new FoodProcessor()
56          { Manufacturer = "KitchenTec",
57            ModelInfo = "Astra FV-202", Power = 12 };
58
59      Console.WriteLine("Описи:");
60      Console.WriteLine(mp.GetDescription());
61      Console.WriteLine(fp.GetDescription());
62
63      Console.WriteLine("Марки:");
64      Console.WriteLine(mp.ItemMark);
65      Console.WriteLine(fp.ItemMark);
66
67      Console.WriteLine("Описи великими літерами:");
68      PrintUpperDescr(mp);
69      PrintUpperDescr(fp);
70  }
```

У параметр передаємо інтерфейс

Всі класи, які описують товари різних видів, повинні реалізувати інтерфейс IInfo. Щоб у класі реалізувати інтерфейс, при оголошенні класу в переліку базових класів вказуємо назву інтерфейсу (рядки 7, 26). При цьому в класі обов'язково потрібно реалізувати всі елементи інтерфейсу (в нашому випадку їх два). Якщо цього не зробити, виникне помилка компіляції. У лістингу 8.2 обидва класи, які описують різні види товарів (класи MobilePhone та FoodProcessor), містять метод GetDescription (рядки 19…23 та 37…41). Структура цих методів (тип результату, назва, параметри) жорстко задана інтерфейсом, але реалізація може бути різна. У цьому прикладі методи GetDescription різних

класів на основі полів свого класу формують опис об'єкта. Властивість ItemMark також реалізована в обох класах (рядки 14…18 та 32…36). Зауважимо, що структура методу GetDescription та властивості ItemMark (назва, тип, кількість і тип параметрів тощо) відповідають їх опису в інтерфейсі.

Таким чином, впровадження інтерфейсу в клас примусово задає структуру класу, потрібну для взаємодії з ним. Інтерфейс для класу можна порівняти з роз'ємом для пристрою (наприклад, USB чи RS-232), який підтримує стандартний формат обміну даними. Зокрема, інтерфейс IInfo з лістингу 8.2 забезпечує "стандартизоване" отримання даних про опис та про марку об'єкта.

Крім цього, екземпляри всіх класів, які реалізують один інтерфейс, можна легко передавати в один метод для додаткової обробки даних. У лістингу 8.2 оголошено метод PrintUpperDescr (рядки 44…47) з параметром obj типу IInfo. Через цей параметр в метод можна передати будь-який клас, що реалізовує вказаний інтерфейс. При цьому посилання на об'єкт автоматично перетворюється у посилання на інтерфейс, який у ньому реалізовано. В даному випадку метод PrintUpperDescr просто виводить на екран опис переданого через параметр об'єкта великими літерами.

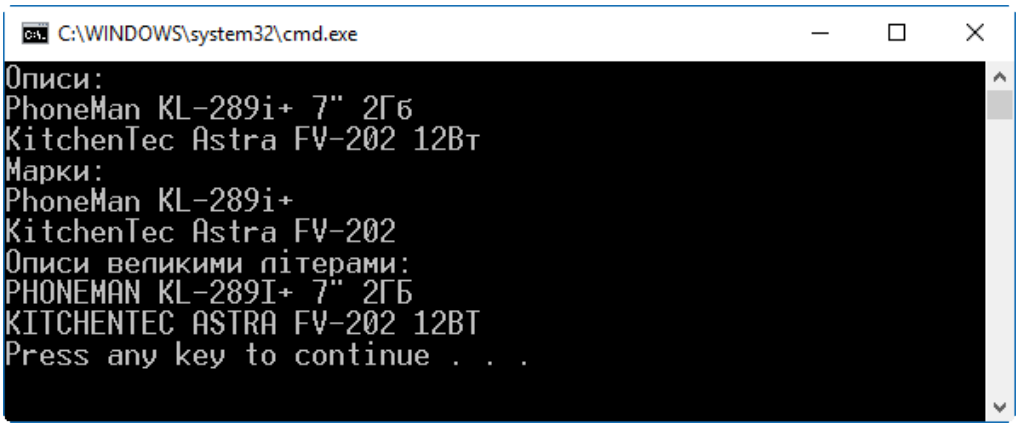


Рис. 8.1. Результат виконання програми з лістингу 8.2. Вивід даних про об'єкти через інтерфейс IInfo

У методі Main створено екземпляри класів MobilePhone і FoodProcessor (рядки 51…57) та продемонстровано використання реалізованих у класах елементів інтерфейсу IInfo. В рядках 60…61 викликаємо методи інтерфейсу GetDescription, в рядках 64…65 – виводимо у вікно значення властивостей інтерфейсу ItemMark, а в рядках 68…69 – передаємо методу PrintUpperDescr об'єкти різних типів, які реалізують інтерфейс IInfo.

На рис. 8.1 показано результат виконання програми з лістингу 8.2. Проект IInfoDemo, який ілюструє роботу з інтерфейсом IInfo, є серед прикладів до навчального посібника.

Приклад використання інтерфейсу IComparable

Бібліотека C# містить набір інтерфейсів, які можна використовувати у своїх програмах. Один із них – інтерфейс IComparable, – реалізує метод, що дозволяє порівнювати екземпляри будь-якого класу між собою. Цей інтерфейс використовується методом Sort класу Array, який представляє масив.

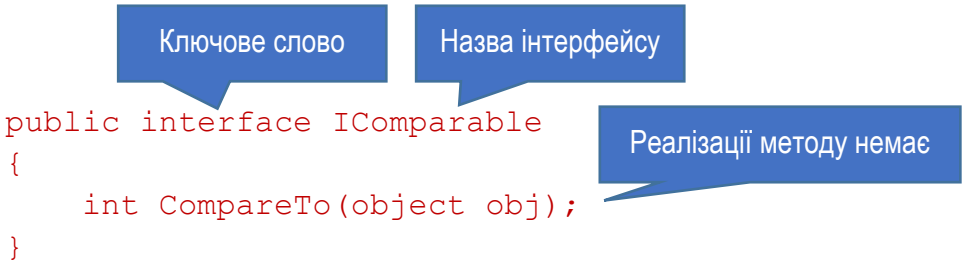
Розглянемо такий приклад коду:

```
var intArr = new[] { 20, 30, 10, 80, 50, 40, 90};
Array.Sort(intArr);
foreach (int i in intArr) Console.Write("{0} ", i);
```

У цьому фрагменті програми спочатку оголошуємо масив цілих чисел intArr і відразу ініціалізуємо його несортованою множиною значень. Далі викликаємо статичний метод Sort класу Array, який сортує масив за зростанням. І в кінці виводимо відсортований масив на екран, щоб пересвідчитися, що значення масиву справді сортовані.

Таким чином, відсортувати значення масиву дуже легко. Але якщо спробувати сортувати масив не з числовими значеннями, а з екземплярами свого класу, то при виконанні програми виникне виняткова ситуація "System.ArgumentException: At least one object must implement IComparable" (Щонайменше один з об'єктів повинен реалізувати IComparable). Справа у тому, що метод Sort "не знає", як порівняти між собою об'єкти довільного визначеного програмістом класу і як їх впорядкувати. Метод Sort класу Array залежить від інтерфейсу, який називається IComparable і визначений у BCL.

Інтерфейс IComparable оголошено так:




```
public interface IComparable
{
    int CompareTo(object obj);
}
```

Цей інтерфейс має всього один метод CompareTo. Оголошення методу задає, що він отримує один параметр типу object і повертає результат типу int.

Хоча сам інтерфейс не містить реалізації цього методу, але документація .NET описує, що цей метод має робити, якщо інтерфейс IComparable буде впроваджений у клас. Згідно документації, метод CompareTo має повертати такі значення:

- Від'ємне значення, якщо у впорядкованому (відсортованому) списку поточний об'єкт повинен бути розташований перед об'єктом, заданим параметром object.
- Додатне значення, якщо поточний об'єкт повинен бути розташований після об'єкта object.

- Нуль, якщо обидва об'єкти при впорядкуванні займають одну і ту ж позицію.

Алгоритм, використаний методом Sort класу Array, основує свою роботу на методі CompareTo інтерфейсу IComparable. Він порівнює два елементи масиву між собою. Тип int реалізує цей інтерфейс, тому сортування масиву цілих чисел пройшло успішно. Але довільний визначений у програмі клас не містить реалізації інтерфейсу IComparable, тому спроба виклику методу Sort спричинить виняткову ситуацію.

Щоб метод Sort міг сортувати екземпляри певного класу, цей клас повинен реалізувати інтерфейс IComparable. Для цього при оголошенні класу назву інтерфейсу слід вказати в переліку базових класів, а також забезпечити реалізацію методу CompareTo. Якщо клас впроваджує інтерфейс, він має реалізувати всі елементи інтерфейсу. Оскільки інтерфейс IComparable містить тільки один метод CompareTo, то потрібно реалізувати лише його.

Лістинг 8.3. Реалізація інтерфейсу IComparable у класі Computer

```
class Computer : IComparable
{
    public string MotherBoard;
    public string CPUModel;
    public double CPUGHz;

    public int CompareTo(object obj)
    {
        return CPUGHz > (obj as Computer).CPUGHz ? 1 :
            CPUGHz < (obj as Computer).CPUGHz ? -1 : 0;
    }
}

static void Main(string[] args)
{
    Computer[] compArr = new[] {
        new Computer() { MotherBoard = "GigaB JHs-298i",
            CPUModel = "Intel", CPUGHz = 2.2},
        new Computer() { MotherBoard = "KiBor 93-HxO92",
            CPUModel = "AMD", CPUGHz = 2.1},
        new Computer() { MotherBoard = "LionBoard H989",
            CPUModel = "Intel", CPUGHz = 2.7},
        new Computer() { MotherBoard = "ZeroXL KL-092z",
            CPUModel = "CoolCPU", CPUGHz = 2.0}
    };

    Console.WriteLine("Сортування масиву об'єктів:");
    Array.Sort(compArr);
```

```
29      foreach (Computer comp in compArr)
30          Console.WriteLine("{0}, {1}, {2:0.00} GHz",
31              comp.MotherBoard, comp.CPUModel, comp.CPUGHz);
32  }
```

У лістингу 8.3 описано клас Computer, у який впроваджено інтерфейс IComparable (рядки 1…12). Клас містить реалізацію методу інтерфейсу CompareTo (рядки 7…11). Клас Computer в загальному описує кілька рис об'єкта "комп'ютер". Його поле MotherBoard містить опис системної плати, поле CPUModel – модель процесора, а поле CPUGHz – частоту процесора.

Метод CompareTo інтерфейсу, реалізований у класі Computer, порівнює значення поля CPUGHz поточного об'єкта із значенням цього ж поля іншого об'єкта, переданого у параметрі obj. При цьому об'єкт obj приводимо до типу Computer за допомогою оператора as. Таким чином, реалізований у лістингу 8.3 метод CompareTo забезпечить сортування об'єктів Computer за значенням поля CPUGHz (тактовою частотою процесора).

У методі Main оголошено та ініціалізовано масив compArr (рядки 16…25), елементи якого є екземплярами класу Computer. Для сортування, як і в попередньо розглянутому випадку з масивом цілих чисел, викликаємо статичний метод Sort класу Array (рядок 28), передавши йому посилання на масив compArr. Щоб пересвідчитися, що масив відсортовано правильно, у циклі виводимо описи кожного об'єкта Computer у масиві (рядки 29…31). Результат роботи програми приведено на рис. 8.2. Дані про окремі об'єкти Computer, які містяться у масиві compArr, виведено по рядках. Бачимо, що об'єкти справді відсортовані за зростанням значення поля CPUGHz (останнє значення кожного рядка з даними).

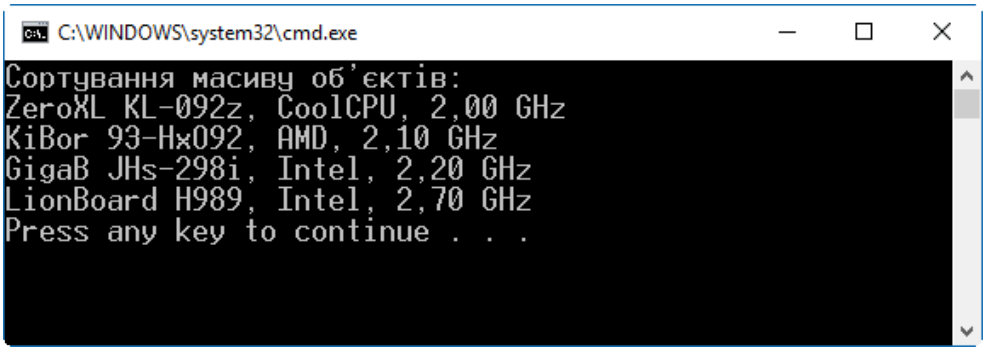


Рис. 8.2. Результат виконання програми з лістингу 8.3. Сортування об'єктів класу Computer

Проект IComparableDemo, який демонструє впровадження та використання інтерфейсу IComparable, можна завантажити з бібліотеки прикладів до навчального посібника.

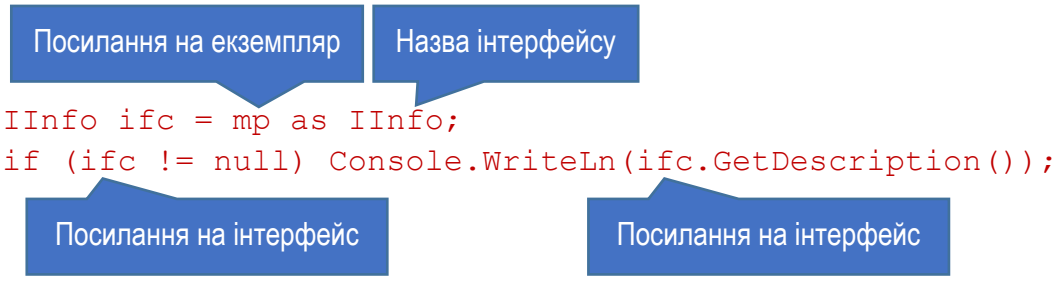
Інтерфейс як посилальний тип даних

Інтерфейс забезпечує значно більшу функціональність, ніж задання простого переліку елементів, які мають бути реалізовані класом. Він представляє окремий тип даних, подібно до класу. До інтерфейсу не можна звертатися безпосередньо через елементи класу. Але *посилання на інтерфейс* можна отримати через приведення типу посилання на об'єкт до типу інтерфейсу. Маючи посилання на інтерфейс, можна звертатися до його елементів.

Наприклад, отримати посилання на інтерфейс IInfo (з лістингу 8.2) в екземплярі класу MobilePhone можна так:

У першому рядку оголошено змінну інтерфейсного типу ifc та присвоєно їй посилання на інтерфейс IInfo, реалізований в екземплярі mp (це об'єкт класу MobilePhone). У другому рядку безпосередньо через інтерфейс IInfo об'єкта mp викликано метод GetDescription.

Замість приведення типу для отримання інтерфейсу об'єкта можна (і варто) використовувати оператор as, який спеціально призначений для перетворень посилальних типів. Якщо застосувати приведення типу до класу, який не реалізує вказаного інтерфейсу, то виникне виняток. В тій же ситуації оператор as, не спричиняючи винятку, поверне значення null. Таким чином, аналізуючи результат оператора as, можна вияснити, реалізує клас потрібний інтерфейс, чи ні. Ось приклад виконання попереднього завдання за допомогою оператора as:

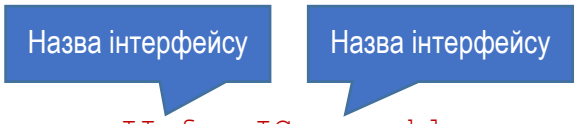


У першому рядку інтерфейсній змінній ifc присвоєно посилання на інтерфейс IInfo, реалізований в об'єкті mp. У другому рядку спочатку перевіряємо, чи посилання на інтерфейс існує (тобто не дорівнює null), і якщо так, то виводимо у вікно опис об'єкта, повернений методом GetDescription. Якщо виявиться, що об'єкт mp не реалізує інтерфейс IInfo, то змінна ifc прийме

значення null. Тоді умова у другому рядку не виконається, і метод інтерфейсу навіть не викликатиметься.

Реалізація кількох інтерфейсів

Клас (або структура) може реалізувати будь-яку кількість інтерфейсів. Тоді оголошення класу в списку базових класів повинно містити перелік всіх інтерфейсів. Наприклад, клас MobilePhone реалізує два інтерфейси: IInfo та IComparable:




```
class MobilePhone : IInfo, IComparable
{
    ...
    public int CompareTo(object obj)
    { ... }
    public string GetDescription()
    { ... }
}
```

Цей клас містить і метод CompareTo з інтерфейсу IComparable, і метод GetDescription з інтерфейсу IInfo.

Якщо в оголошенні класу вказано кілька інтерфейсів, то клас повинен реалізувати всі елементи всіх перелічених інтерфейсів. При цьому може трапитись, що різні інтерфейси мають метод з однаковою сигнатурою. Тоді у класі реалізовують тільки один метод, який має задовольняти обидва інтерфейси. Інший спосіб вирішення цього завдання – при реалізації елемента інтерфейсу в класі явно вказати назву інтерфейсу:

```
class MyClass : IIfc1, IIfc2
{
    void IIfc1.PrintDescr(string s)
    { ... }
    void IIfc2.PrintDescr(string s)
    { ... }
}
```

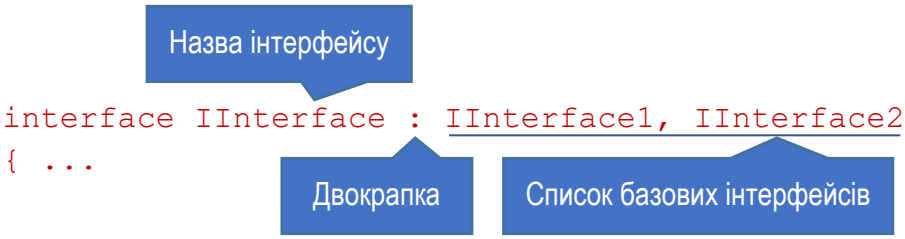
На окремі інтерфейси класу також можна отримувати посилання, як було розглянуто раніше. Наприклад, якщо є екземпляр ep класу, в якому реалізовано інтерфейси IInfo та IComparable, то посилання на обидва інтерфейси отримують так:

```
IInfo ii = mp as IInfo;
IComparable ic = mp as IComparable;
```

За допомогою цих посилань можна звертатися до відповідних елементів інтерфейсу, як було продемонстровано вище.

Спадкування інтерфейсів

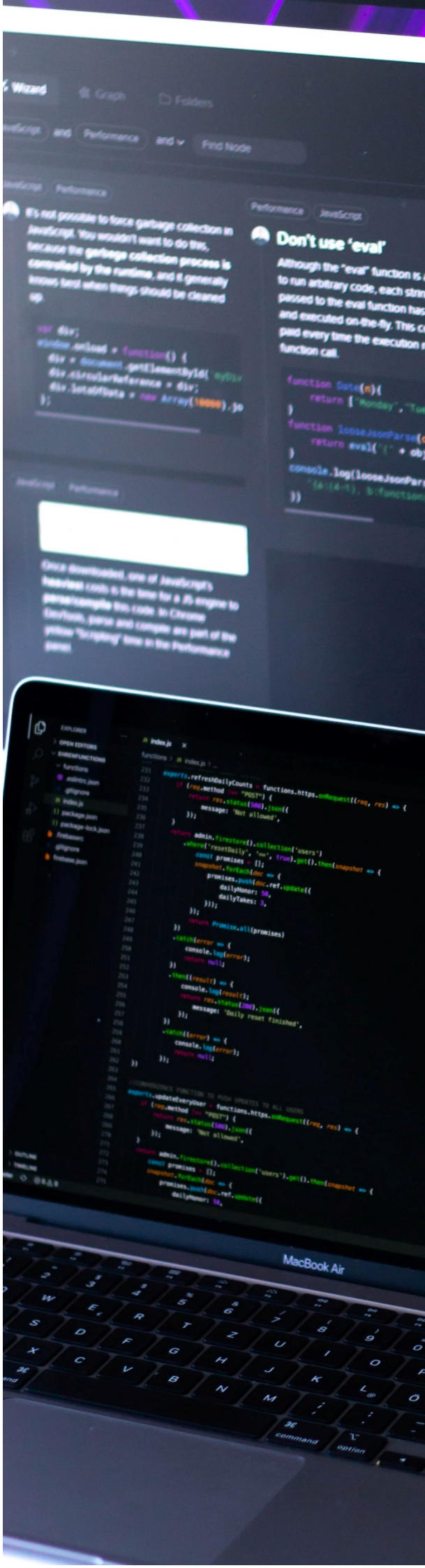
Інтерфейс, як і клас, може спадкувати структуру іншого інтерфейсу. Таким чином, на основі одних інтерфейсів можна створювати інші інтерфейси. Щоб вказати, що інтерфейс успадковує інші інтерфейси, імена базових інтерфейсів слід вказати так само, як при спадкуванні класів – в окремому переліку:



На відміну від класів, які можуть успадкувати тільки один базовий клас, інтерфейси можуть спадкувати довільну кількість базових інтерфейсів. У свою чергу, кожен базовий інтерфейс може спадкувати інші інтерфейси. Результуючий інтерфейс крім своїх елементів містить елементи всіх успадкованих інтерфейсів. Відповідно клас, який впроваджує такий інтерфейс, також повинен реалізувати всі елементи всіх успадкованих інтерфейсів.

Розділ 9. Колекції

- Колекції
- Інтерфейси узагальнених колекцій
- Класи узагальнених колекцій



Колекції

У C# *колекція* – це сукупність об'єктів, розміщених в одному контейнері, що надає можливість їх послідовної обробки. В області обробки множинних даних колекції є потужною альтернативою масивам.

Середовище .NET Framework містить інтерфейси та класи, в яких визначено і реалізовано різні типи колекцій. Колекції спрощують рішення багатьох завдань програмування завдяки тому, що пропонують готові рішення для створення цілого ряду типових, але часто трудомістких для розробки структур даних. Зокрема, .NET Framework містить колекції для підтримки динамічних масивів, зв'язаних списків, стеків, черг і хеш-таблиць. Колекції є сучасним технологічним засобом і заслуговують пильної уваги всіх, хто програмує на C#.

Головна перевага колекцій полягає в тому, що вони стандартизують обробку груп об'єктів у програмі. Колекції розроблені на основі інтерфейсів .NET. Є також можливість реалізувати власну колекцію, хоча потреба в цьому виникає рідко. Існує п'ять типів колекцій:

- Неузагальнені колекції реалізують такі структури даних, як динамічні масиви, стеки, черги та словники. Вони оперують елементами класу object. Тому неузагальнені колекції можуть зберігати дані будь-якого типу, причому одна колекція може містити дані різних типів. Класи та інтерфейси узагальнених колекцій описані в просторі імен System.Collections.
- Узагальнені колекції забезпечують узагальнену реалізацію декількох неузагальнених колекцій. Такі колекції є типізованими, тому в узагальненій колекції можуть зберігатися тільки елементи даних, сумісні за типом з колекцією. Завдяки цьому виключається випадкова розбіжність типів. Узагальнені колекції оголошено в просторі імен System.Collections.Generic.
- Спеціальні колекції оперують даними конкретного типу або реалізують спеціалізований алгоритм їх обробки. Наприклад, є колекції для роботи з рядками. Такі колекції оголошено в просторі імен System.Collections.Specialized.
- Бітові колекції призначені для порозрядних операцій над даними. Вони описані у просторі імен System.Collections.
- Паралельні колекції підтримують багатопотоковий доступ. Це узагальнені колекції, описані в просторі імен System.Collections.Concurrent.

Спеціальні, бітові та паралельні колекції використовують для вирішення специфічних завдань. Найчастіше для типових потреб використовують узагальнені колекції. Неузагальнені колекції мають дуже схожий функціонал, але, оскільки узагальнені є типобезпечними, то Microsoft рекомендує завжди при можливості використовувати саме узагальнені колекції. Неузагальнені колекції слід використовувати лише тоді, коли колекція має зберігати дані різних типів.

Базовим для всіх колекцій є наявність *перелічувача* – сукупності програмних засобів, які дозволяють послідовно переміщуватися через всі об'єкти колекції (тобто, "перелічувати" їх). Перелічувач реалізують через інтерфейси неузагальнених колекцій IEnumerator і IEnumerable, та інтерфейси узагальнених колекцій IEnumerator<T> і IEnumerable<T>. Тому будь-яка колекція має реалізовувати ці інтерфейси.

Інтерфейси узагальнених колекцій

Узагальнені колекції реалізовано за допомогою ряду інтерфейсів, які задають засоби та принципи роботи з колекціями. Ці інтерфейси описано у просторі імен System.Collections.Generic. Дані про них приведено в таблиці 9.1. Тип, з яким працює колекція, вказують у трикутних дужках (<>). На місці покажчика типу T може бути будь-який тип даних.

Таблиця №9.1. Інтерфейси узагальнених колекцій

Інтерфейс	Опис
ICollection<T>	Визначає головні властивості узагальнених колекцій
IComparer<T>	Визначає метод Compare для порівняння об'єктів колекції
IDictionary<Tkey, TValue>	Визначає колекцію, яка містить пари "ключ-значення"
IEnumerable<T>	Визначає узагальнений метод GetEnumerator, який надає перелічувач для будь-якого класу
IEnumerator<T>	Визначає методи, які дозволяють отримувати вміст колекції по черзі
IEqualityComparer<T>	Порівнює два об'єкти на рівність
IList<T>	Визначає узагальнену колекцію, до якої отримують доступ за допомогою індексатора
ISet<T>	Визначає узагальнену колекцію, яка представляє множину

Інтерфейс ICollection<T>

Інтерфейс ICollection<T> визначає ряд загальних для всіх колекцій властивостей. Властивості і методи цього інтерфейсу зібрано і пояснено в таблиці 9.2.

Таблиця №9.2. Елементи інтерфейсу ICollection<T>

Елемент	Опис
Властивості	
int Count	Містить кількість елементів, розміщених у поточній колекції. Властивість тільки для читання
bool IsReadOnly	Якщо властивість дорівнює true, то колекція доступна тільки для читання; в іншому випадку колекція доступна для читання та запису. Властивість тільки для читання
Методи	
void Add(T item)	Додає об'єкт item до колекції. Якщо колекція доступна тільки для читання, генерує виняток NotSupportedException
void Clear( )	Видаляє всі елементи з викликаючої колекції
bool Contains(T item)	Повертає true, якщо викликаюча колекція містить об'єкт, заданий параметром item. Повертає false в протилежному випадку
void CopyTo(T[ ] array, int arrayIndex)	Копіює вміст викликаючої колекції в масив array, починаючи з індекса arrayIndex
bool Remove(T item)	Видаляє з колекції перше входження об'єкта item

Інтерфейс IList<T>

Інтерфейс IList<T> визначає поведінку узагальненої колекції, яка дозволяє звертатися до її елементів за індексом з відліком від нуля. Методи Insert та RemoveAt дозволяють змінювати вміст колекції. Якщо ж вона доступна тільки для читання, то ці методи генеруватимуть виняток NotSupportedException. Методи інтерфейсу IList<T> описано в таблиці 9.3.

Таблиця №9.3. Методи інтерфейсу IList<T>

Метод	Опис
int IndexOf(T item)	Повертає індекс першого входження об'єкта item у викликаючій колекції. Якщо об'єкт не знайдено, повертає -1
void Insert(int index, T item)	Вставляє у викликаючу колекцію об'єкт item за індексом index
void RemoveAt(int index)	Видаляє з викликаючої колекції об'єкт за індексом index

Крім цього, інтерфейс IList<T> визначає індексатор

```
T this[int index] { get; set; }
```

Він дозволяє записувати і читати значення елемента колекції за індексом index, який вказують у квадратних дужках, подібно до звертання до елементів масиву.

Інтерфейс IDictionary<TKey, TValue>

Інтерфейс IDictionary<TKey, TValue> визначає поведінку колекції виду "словник", яка зберігає пари "ключ-значення". Методи інтерфейсу дозволяють перетворювати унікальні ключі у відповідні значення. Елементи інтерфейсу IDictionary<TKey, TValue> приведено в таблиці 9.4.

Таблиця №9.4. Елементи інтерфейсу IDictionary<TKey, TValue>

Елемент	Опис
Методи	
void Add(TKey key, TValue value)	Додає у викликаючу колекцію пару "ключ-значення"
bool ContainsKey(TKey key)	Повертає true, якщо викликаюча колекція містить ключ key, і false в протилежному випадку
bool Remove(TKey key)	Видаляє запис колекції з ключем key
bool TryGetValue(TKey key, out TValue value)	Намагається отримати значення із запису колекції з ключем key. Повертає true, якщо це вдалось зробити, і false в протилежному випадку. Якщо ключ key не знайдено, параметр value приймає значення за замовчуванням
Властивості	
ICollection Keys<TKey>	Отримує колекцію ключів. Властивість тільки для читання
ICollection Values<TValue>	Отримує колекцію значень. Властивість тільки для читання

Інтерфейси IEnumerable<T> та IEnumerator<T>

Єдиний метод інтерфейсу IEnumerable<T> – GetEnumerator. Він повертає колекцію. Завдяки реалізації інтерфейсу IEnumerable можна отримати вміст колекції в циклі foreach.

В інтерфейсі IEnumerator<T> визначено функції перелічувача. За допомогою методів цього інтерфейсу можна циклічно звертатися до вмісту колекції. Інтерфейс містить методи MoveNext і Reset та властивість Current. Властивість Current містить посилання на поточний об'єкт. Метод MoveNext здійснює перехід до наступного елементу колекції, а метод Reset починає перелічування з самого початку (таблиця 9.5).

Інтерфейси IComparer<T> та IEqualityComparer<T>

В інтерфейсі IComparer визначено метод Compare для порівняння двох об'єктів: x та y. Він повертає позитивне значення, якщо значення об'єкта x більше, ніж об'єкта y; негативне – якщо значення об'єкта x менше, ніж об'єкта y; і нульове – якщо порівнювані значення однакові. Інтерфейс можна використовувати для задання способу сортування елементів колекції.

В інтерфейсі IEqualityComparer визначено методи Equals та GetHashCode (таблиця 9.6).

Таблиця №9.5. Елементи інтерфейсів IEnumerable<T> та IEnumerator<T>

Елемент	Опис
Інтерфейс IEnumerable<T>	
Методи	
IEnumerator<T> GetEnumerator()	Повертає перелічувач, який дозволяє виконувати перебір елементів у колекції
Інтерфейс IEnumerator<T>	
Властивості	
T Current	Повертає поточний елемент колекції
Методи	
bool MoveNext()	Переміщує перелічувач до наступного об'єкта колекції. Повертає true, якщо перелічувач успішно переміщено до наступного об'єкта, і false, якщо досягнуто кінець колекції
void Reset()	Встановлює перелічувач у його початкове положення: перед першим об'єктом колекції

Таблиця №9.6. Елементи інтерфейсів IComparer<T> та IEqualityComparer<T>

Елемент	Опис
Інтерфейс IComparer<T>	
Методи	
int Compare (object x, object y)	Порівнює два об'єкти колекції. Повертає позитивне значення, якщо значення x більше, ніж y; негативне – якщо значення x менше, ніж y; і нульове – якщо значення однакові
Інтерфейс IEqualityComparer<T>	
Методи	
bool Equals (object x, object y)	Порівнює об'єкти колекції. Повертає значення true, якщо значення x та y однакові, і false – в протилежному випадку
int GetHashCode (object obj)	Повертає хеш-код для об'єкта obj

Інтерфейси ISet<T>

Інтерфейс ISet<T> визначає поведінку узагальненої колекції, яка реалізує себе як множина елементів. Він спадкує інтерфейси IEnumerable, IEnumerable<T> та ICollection<T>. Методи цього інтерфейсу приведено в таблиці 9.7. Параметри цих методів відносяться до типу IEnumerable<T>. Це означає, що у методи можна передати колекції інших типів. Але найчастіше обидва аргументи є колекціями типу ISet<T>.

Таблиця №9.7. Методи інтерфейсу ISet<T>

Метод	Опис
void ExceptWith(IEnumerable<T> other)	Видаляє з викликаючої колекції елементи, які містяться в іншій колекції other
void IntersectWith(IEnumerable<T> other)	Після виклику методу викликаюча колекція міститиме перетин її елементів з елементами колекції other
bool IsProperSubsetOf(IEnumerable<T> other)	Повертає true, якщо викликаюча колекція є правильною підмножиною колекції other, і false в протилежному випадку
bool IsProperSupersetOf(IEnumerable<T> other)	Повертає true, якщо викликаюча колекція є правильною надмножиною колекції other, і false в протилежному випадку
bool IsSubsetOf(IEnumerable<T> other)	Повертає true, якщо викликаюча колекція є підмножиною колекції other, і false в протилежному випадку
bool IsSupersetOf(IEnumerable<T> other)	Повертає true, якщо викликаюча колекція є надмножиною колекції other, і false в протилежному випадку
bool Overlaps(IEnumerable<T> other)	Повертає true, якщо викликаюча колекція та колекція other містять щонайменше один спільний елемент, і false в протилежному випадку
bool SetEquals(IEnumerable<T> other)	Повертає true, якщо викликаюча колекція та колекція other мають всі елементи спільними, і false в протилежному випадку. Порядок елементів не береться до уваги, і дублікати елементів у колекції other ігноруються
void UnionWith(IEnumerable<T> other)	Після виклику методу викликаюча колекція міститиме об'єднання її елементів з елементами колекції other

Класи узагальнених колекцій

Узагальнені колекції описані в таких класах:

- Dictionary<Tkey, TValue>. Колекція виду "словник", яка зберігає пари "ключ-значення".
- HashSet<T>. Зберігає ряд унікальних значень за допомогою хеш-таблиці.
- LinkedList<T>. Зберігає елементи у двонаправленому списку.

- List<T>. Колекція виду "динамічний масив", яка дозволяє працювати з її елементами як з елементами масиву. Розмір масиву можна змінювати під час виконання.
- Queue<T>. Колекція виду "черга" – список, який діє за принципом "першим зайшов – першим вийшов".
- SortedDictionary<TKey, TValue>. Колекція виду "сортований словник". Містить пари "ключ-значення".
- SortedList<TKey, TValue>. Колекція виду "сортований список", яка містить пари "ключ-значення".
- SortedSet<T>. Створює сортовану множину.
- Stack<T>. Колекція виду "стек" – список, який діє за принципом "першим зайшов – останнім вийшов".

Класи узагальнених колекцій визначено в просторі імен System.Collections.Generic.

Клас List<T>

Клас List<T> реалізує динамічний масив, який може змінювати розмір під час виконання програми. У C# стандартні масиви мають фіксовану довжину, яка не може змінюватися після створення масиву. Але часто конкретна довжина масиву є невідомою. Для таких ситуацій і призначений клас List<T>.

Він визначає масив змінної довжини, який складається з посилань на об'єкти і може динамічно збільшувати і зменшувати свій розмір. Колекцію типу List<T> створюють з початковим розміром. Якщо цей розмір перевищується, то вона автоматично розширюється. При видаленні об'єктів колекція автоматично скорочується. Колекції класу List<T> широко застосовують в практиці програмування на C#.

В класі List<T> реалізовано інтерфейси ICollection, ICollection<T>, IList, IList<T>, IEnumerable і IEnumerable<T>. Клас List<T> має такі конструктори:

```
public List()
public List(IEnumerable <T> collection)
public List(int capacity)
```

Перший конструктор створює порожню колекцію класу з початковим розміром за замовчуванням. Другий конструктор створює колекцію типу List, яка відразу міститиме елементи з заданої колекції collection. Третій конструктор створює колекцію типу List з розміром, заданим у параметрі capacity.

Розмір колекції динамічного масиву може збільшуватися автоматично при додаванні в неї елементів.

Крім методів, оголошених у реалізованих класом інтерфейсах, клас List<T> визначає ряд власних методів. Найвживаніші з них приведено у таблиці 9.8.

Таблиця №9.8. Найвживаніші методи класу List<T>

Метод	Опис
public void AddRange(IEnumerable<T> collection)	Додає елементи колекції collection в кінець викликаючої колекції
public int BinarySearch(T item)	Шукає у викликаючій колекції значення, задане параметром item. Повертає індекс знайденого елемента. Якщо значення не знайдено, повертає від'ємне значення. Викликаюча колекція має бути сортована.
public int BinarySearch(T item, IComparer<T> comparer)	Шукає у викликаючій колекції значення item, використовуючи для цього спосіб порівняння, заданий параметром comparer. Повертає індекс знайденого елемента. Якщо значення не знайдено, повертає від'ємне значення. Викликаюча колекція має бути сортована.
public int BinarySearch(int index, int count, T item, IComparer<T> comparer)	Шукає у викликаючій колекції значення item, використовуючи для цього спосіб порівняння, заданий параметром comparer. Пошук проводиться починаючи з індекса index серед наступних елементів, кількість яких задає параметр count. Повертає індекс знайденого елемента. Якщо значення не знайдено, повертає від'ємне значення. Викликаюча колекція має бути сортована.
public List<T> GetRange(int index, int count)	Повертає частину викликаючої колекції, починаючи з індекса index, кількість елементів count. Результуюча колекція посилається на ті ж об'єкти, що і викликаюча
public int IndexOf(T item)	Повертає індекс першого входження елемента item у викликаючу колекцію. Якщо елемент не знайдено, повертає -1
public void InsertRange(int index, IEnumerable<T> collection)	Вставляє елементи колекції collection у викликаючу колекцію, починаючи з індекса index
public int LastIndexOf(T item)	Повертає індекс останнього входження елемента item у викликаючу колекцію. Якщо елемент не знайдено, повертає -1
public void RemoveRange(int index, int count)	Видаляє з викликаючої колекції починаючи з індекса index задану кількість елементів count
public void Reverse()	Розміщує елементи викликаючої колекції у зворотному порядку
public void Sort()	Сортує викликаючу колекцію за зростанням
public void Sort(IComparer<T> comparer)	Сортує викликаючу колекцію, використовуючи для цього спосіб порівняння, заданий параметром comparer
public void Sort (int index,	Сортує викликаючу колекцію, використовуючи для цього

Метод	Опис
int count, IComparer<T> comparer)	спосіб порівняння, заданий параметром comparer. Обробляє елементи починаючи з індексу index. Кількість елементів для сортування задає параметр count
public T[ ] ToArray()	Повертає масив, що містить копії елементів колекції

Колекція виду List<> має властивість Capacity цілого типу, яка містить кількість елементів колекції і дозволяє отримати чи задати її розмір.

У лістингу 9.1 приведено приклад програми, яка створює колекцію типу List<> (рядок 3) та заповнює її значеннями (рядки 4…6) за допомогою методу Add. В рядку 7 продемонстровано звертання до елемента колекції за індексом. Після цього значення з колекції виводяться на екран (рис. 9.1).

Лістинг 9.1. Робота з колекцією List<>

```
1  static void Main(string[] args)
2  {
3      List<int> lst = new List<int>();
4      lst.Add(100);
5      lst.Add(200);
6      lst.Add(300);
7      lst[1] = 222;
8
9      foreach (int i in lst)
10         Console.WriteLine("{0}", i);
11 }
```

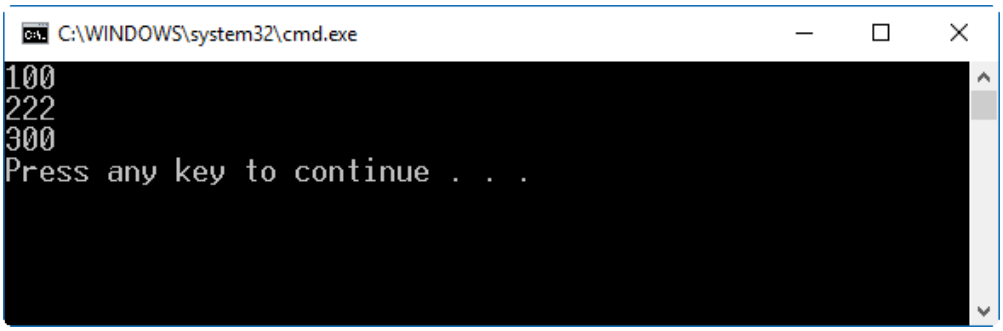


Рис. 9.1. Результат виконання програми з лістингу 9.1

Клас Dictionary<TKey, TValue>

Клас Dictionary<TKey, TValue> дозволяє зберігати пари "ключ-значення" в колекції як у словнику. У класі Dictionary<TKey, TValue> реалізовано інтерфейси IDictionary, IDictionary<TKey, TValue>, ICollection, IEnumerable. Словники є динамічними, розширюючись при необхідності.

Клас Dictionary<TKey, TValue> має кілька конструкторів. Найвживаніші з них:

```
public Dictionary()
public Dictionary(IDictionary<TKey, TValue> dictionary)
public Dictionary(int capacity)
```

Перший з перелічених конструкторів створює порожній словник з обраним за замовчуванням початковим розміром. Другий створює словник, у якому відразу міститимуться елементи зі словника dictionary. А третій конструктор створює словник з кількістю елементів capacity. Якщо розмір словника заздалегідь відомий, то, вказавши ємність колекції, можна уникнути зміни розміру словника під час виконання, що вимагає додаткових витрат обчислювальних ресурсів.

Таблиця №9.9. Найвживаніші елементи класу Dictionary<TKey, TValue>

Елемент	Опис
Методи	
public void Add(TKey key, TValue value)	Додає в колекцію пару "ключ-значення ", задану параметрами key і value. Якщо ключ вже є у словнику, його значення не змінюється і генерується виняткова ситуація ArgumentException
public bool ContainsKey(TKey key)	Повертає логічне значення true, якщо колекція містить ключ key, в іншому випадку – логічне значення false
public bool ContainsValue(TValue value)	Повертає логічне значення true, якщо колекція містить значення value, в іншому випадку – логічне значення false
public bool Remove(TKey key)	Видаляє з колекції елемент, ключ якого дорівнює key. Повертає true при успіху, і false – в протилежному випадку
Властивості	
public Dictionary <TKey,TValue>.KeyCollection Keys	Отримує колекцію ключів. Властивість тільки для читання
public Dictionary <TKey,TValue>.ValueCollection Values	Отримує колекцію значень. Властивість тільки для читання

Клас Dictionary<TKey, TValue> містить значну кількість методів, за допомогою яких керують словником. Частина з них зібрана у таблиці 9.9.

У лістингу 9.2 приведено приклад використання колекції Dictionary<TKey, TValue>. Тут продемонстровано створення словника, який містить інформацію про міста та кількості їх населення у вигляді пар "назва-кількість мешканців". При створенні словника (рядки 3…4) вказуємо, що тип ключа буде string (назва

міста), а тип значення – int (число мешканців). У рядках 6…9 додаємо до словника записи про міста, вказуючи назву як ключ та число мешканців як значення словника. Після цього отримуємо колекцію ключів (рядок 11) і в циклі виводимо весь словник на екран, звертаючись до елементів словника за їх значеннями. Результат виконання програми приведено на рис. 9.2.

Лістинг 9.2. Робота з колекцією Dictionary<TKey, TValue>

```
1  static void Main(string[] args)
2  {
3      Dictionary<string, int> dictTowns =
4          new Dictionary<string, int>();
5
6      dictTowns.Add("Тернопіль", 215000);
7      dictTowns.Add("Львів", 800000);
8      dictTowns.Add("Київ", 3800000);
9      dictTowns.Add("Хмельницький", 262000);
10
11     ICollection<string> c = dictTowns.Keys;
12     foreach (string town in c)
13         Console.WriteLine("{0}, кількість мешканців: {1}",
14             town, dictTowns[town]);
15 }
```

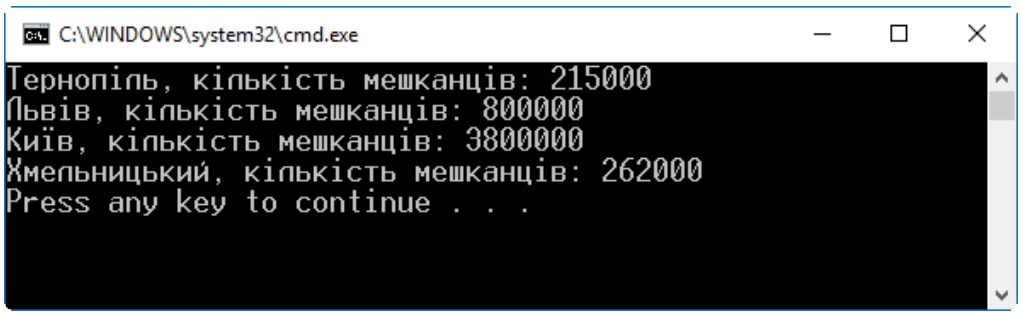


Рис. 9.2. Результат виконання програми з лістингу 9.2

Клас Stack<T>

Стек – це список, що діє за принципом "першим прийшов – останнім вийшов". Цей принцип дії стека можна наочно уявити на прикладі коробки з документами, які ставлять у коробку зверху один на інший. Перший документ, поставлений на дно коробки, буде витягнено з неї останнім.

Стек – одна з найважливіших структур даних в обчислювальній техніці. Він часто застосовується в системному програмному забезпеченні, компіляторах, а також в програмах відстеження в зворотному порядку на основі штучного інтелекту.

Клас колекції, що підтримує стек, носить назву Stack<>. Він реалізує інтерфейси ICollection, IEnumerable і IEnumerable<T>. Колекція класу Stack<T> має динамічний характер, розширюючись при необхідності, щоб вмістити всі елементи, які повинні в ній зберігатися. У класі Stack<T> визначено такі конструктори:

```
public Stack()
public Stack(int capacity)
public Stack(IEnumerable<T> collection)
```

Перший з них створює порожній стек з початковим розміром за замовчуванням, а другий – порожній стек з початковим розміром capacity. Третій вид конструктора створює стек, що містить елементи колекції, яка задана параметром collection. Його початковий розмір дорівнює кількості зазначених елементів.

У класі Stack<T> визначаено ряд власних методів (крім тих, що вже оголошені в інтерфейсах, які в ньому реалізуються). Найвживаніші з них перелічено у таблиці 9.10.

Таблиця №9.10. Найвживаніші методи класу Stack<T>

Метод	Опис
public T Peek()	Читає об'єкт з вершини стеку, але не видаляє його
public T Pop()	Витягує об'єкт зі стеку
public void Push(T item)	Розміщує у стеку об'єкт item
public T[] ToArray()	Повертає масив, який містить копії об'єктів стеку

Щоб помістити об'єкт у стек (на вершину стека), викликають метод Push. Щоб витягти і видалити об'єкт з вершини стека, викликають метод Pop. Якщо ж об'єкт потрібно тільки прочитати, але не видалити з вершини стека, то викликають метод Peek. Коли стек порожній і викликано метод Pop або Peek, то система згенерує виняткову ситуацію InvalidOperationException.

У лістингу 9.3 приведено приклад програми, яка працює зі стеком. Спочатку (рядок 3) створюємо колекцію-стек для цілих чисел (int) і посилання на неї зберігаємо у змінній st. Далі в циклі (рядки 6…10) розміщуємо у стек числа i*100, де i – цілі числа від 1 до 5. Для розміщення числа в стек використовуємо метод Push (рядок 8). Кожне число також виводимо на екран. Після цього циклом (рядки 14…18), умовою якого є непорожній стек (st.Count > 0), послідовно вибираємо зі стеку по одному значенню, і виводимо його на екран. Для отримання зі стеку значення використовуємо метод Pop (рядок 16), який повертає отримане значення. Якщо запустити програму, то з виводу у консоль (рис. 9.3) бачимо, що значення зі стеку отримуються у зворотному порядку.

Лістинг 9.3. Робота зі стеком Stack<>

```
1  static void Main(string[] args)
2  {
3      Stack<int> st = new Stack<int>();
4
5      Console.Write("Розміщуємо у стек: ");
6      for (int i = 1; i <= 5; i++)
7      {
8          st.Push(i * 100);
9          Console.Write(i * 100 + " ");
10     }
11     Console.WriteLine();
12
13     Console.Write("Отримуємо зі стеку: ");
14     while (st.Count > 0)
15     {
16         int i = st.Pop();
17         Console.Write(i + " ");
18     }
19     Console.WriteLine();
20 }
```

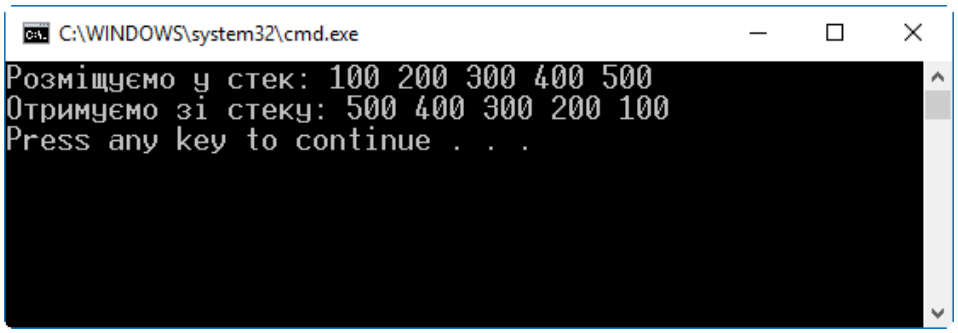


Рис. 9.3. Результат виконання програми з лістингу 9.3. Значення зі стеку отримують у зворотному порядку щодо записаного

Клас Queue<T>

Колекція типу "черга" діє за принципом: першим прийшов – першим вийшов. Це означає, що першим з черги витягується елемент, поміщений в неї першим. У програмуванні черги застосовують для зберігання таких елементів, як активні процеси, списки призупинених транзакцій в базі даних або пакети даних, отримані через мережу.

Клас Queue<T> реалізує інтерфейси ICollection, IEnumerable і IEnumerable<T>. Колекція класу Queue<T> є динамічною, розширюючись при необхідності розмістити нові дані. У класі Queue<T> визначено наступні конструктори:

```
public Queue()
public Queue(int capacity)
public Queue(IEnumerable<T> collection)
```

Перший з них створює порожню чергу з початковим розміром за замовчуванням. Другий – порожню чергу з розміром capacity. Третій створює чергу, в яку відразу розміщуються елементи колекції collection. Початковий розмір черги дорівнює кількості елементів у цій колекції. Крім методів інтерфейсів, клас Queue<T> містить ряд власних методів (таблиця 9.11).

Таблиця №9.11. Найвживаніші методи класу Queue<T>

Метод	Опис
public T Dequeue()	Вибирає об'єкт з початку черги
public void Enqueue(T item)	Додає об'єкт item в кінець черги
public T Peek()	Читає елемент з початку черги, не видаляючи його
public virtual T[] ToArray()	Повертає масив, який містить копії об'єктів черги

Щоб розмістити об'єкт у чергу, викликають метод Enqueue. Для отримання першого об'єкта з початку черги викликають метод Dequeue. Якщо потрібно прочитати, але не видаляти наступний об'єкт з черги, викликають метод Peek. Коли методи Dequeue і Peek викликати при порожній черзі, виникне виняток InvalidOperationException.

Лістинг 9.4 містить приклад роботи з колекцією виду "черга". Чергу створюємо у рядку 3. Змінна queue міститиме посилання на чергу. Далі в циклі (рядки 6…10) розміщуємо в чергу числа i*100, де i – цілі числа від 1 до 5. Для цього використовуємо метод Enqueue (рядок 8). Після цього у циклі (рядки 14…18) за допомогою методу Dequeue (рядок 16) отримуємо з черги по одному значенню і виводимо його на екран. З результату виконання програми (рис. 9.4) видно, що елементи з черги отримуються у тому ж порядку, в якому вони були записані.

Лістинг 9.4. Робота з чергою Queue<>

```
1  static void Main(string[] args)
2  {
3      Queue<int> queue = new Queue<int>();
4
5      Console.Write("Розміщуємо у черзі: ");
6      for (int i = 1; i <= 5; i++)
7      {
8          queue.Enqueue(i * 100);
9          Console.Write(i * 100 + " ");
10     }
11     Console.WriteLine();
```

```
12
13      Console.Write("Отримуємо з черги: ");
14      while (queue.Count > 0)
15      {
16          int i = queue.Dequeue();
17          Console.Write(i + " ");
18      }
19      Console.WriteLine();
20  }
```

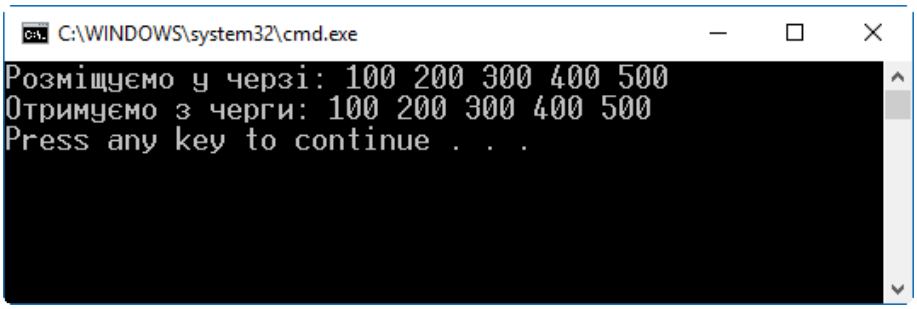
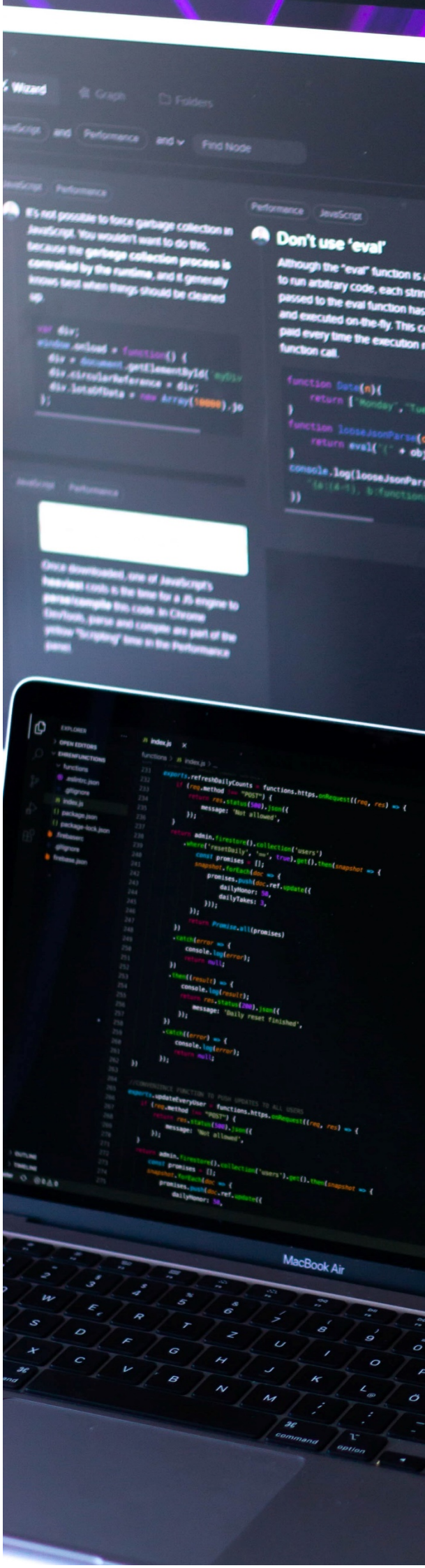


Рис. 9.4. Результат виконання програми з лістингу 9.4. Значення з черги отримують у тому ж порядку, в якому вони були записані

Розділ 10. Застосунки Windows Forms

- Windows Forms
- Структура програми Windows Forms
- Елементи керування



Windows Forms

Windows Forms (*WF*) – це частина платформи Microsoft .NET Framework, яка використовується для створення програм, які зазвичай називають клієнтськими застосунками (використовують також назви: "прикладні застосунки робочого столу", "desktop-застосунки" або просто "віконні застосунки"). Такі застосунки складаються зі звичайних виконавчих файлів (з розширенням .exe), іноді вони містять динамічні бібліотеки (з розширенням .dll) та інші необхідні файли (наприклад, файли довідкової системи).

Для графічного виводу Windows Forms використовує графічну підсистему GDI/GDI+, яка не задіює всіх можливостей сучасних відеокарт. Тому Microsoft запропонувала нову графічну підсистему для розробки вікон – Windows Presentation Foundation (WPF). WPF входить до складу .NET Framework починаючи з версії 3.0. Для графічного виводу WPF використовує DirectX – набір інтерфейсів для роботи із засобами мультимедіа, включаючи графіку та відео. Загалом WPF є складнішою за Windows Forms і надає значно більше можливостей для побудови застосунків з привабливішим інтерфейсом. Але, на даний час значна кількість застосунків для операційної системи Windows все ще використовують Windows Forms. WF дозволяє швидко та просто розробити віконний застосунок "стандартного" виду.

У Windows Forms *форма* – це частина екрану (як правило, прямокутна), на якій виводиться інформація для користувача. Форми представляють вікна програм. Зазвичай застосунок Windows Forms створюють шляхом розміщення на формі елементів керування і написання коду для реагування на дії користувача, такі як клацання миші або натискання клавіш. *Елемент керування* – це окремий елемент інтерфейсу, призначений для відображення або введення даних. До елементів керування належать кнопки, списки, комбіновані списки, меню, контекстні меню та багато інших.

При виконанні користувачем якоїсь дії з формою або одним з її елементів керування (наприклад, при натисканні клавіші клавіатури) генерується подія. Застосунок реагує на ці події за допомогою коду і виконує необхідні операції. *Подія* – це ситуація під час роботи програми, яка може представляти цінність і перехоплюватися програмою. Події – універсальний механізм, який дозволяє одному об'єктові повідомляти іншому про настання певних умов виконання шляхом виклику методу. Такий прив'язаний до події метод, що автоматично спрацьовує при її настанні, називають *оброблювачем події* (*event handler*).

Структура програми Windows Forms

Найпростіша програма Windows Forms

Почнемо розгляд принципів розробки застосунків Windows Forms з найпростішого прикладу – програми, яка відображує на екрані порожнє вікно. Щоб наочніше побачити механізм функціонування застосунку Windows Forms, створимо у Microsoft Visual Studio новий порожній проект. Для цього виберемо шаблон Visual C#\Windows\Classic Desktop\Empty Project. Буде згенеровано порожній проект без жодного файлу з кодом.

Спочатку у проект додамо посилання. Для простого застосунку Windows Forms необхідні посилання System, System.Drawing та System.Windows.Forms. Щоб додати посилання, потрібно вибрати команду меню Add Reference… Далі у вікні Reference Manager в пункті Assemblies\Framework слід відмітити вказані посилання і натиснути кнопку OK.

Щоб задати тип виводу у вікно, а не в консоль, слід задати відповідні налаштування проекту. Для цього в контекстному меню проекту у вікні Solution Explorer потрібно вибрати команду Properties. У вікні властивостей на сторінці Application в полі Output Type слід задати вид виводу Windows Application. Якщо задати тип виводу у консоль (Console Application), то програма разом з графічним вікном міститиме також вікно консолі. Зауважимо, що цей параметр сам по собі не перетворює застосунок у віконний. Він лише задає можливість виводу у консоль.

Тепер додамо у проект новий файл для коду. Для цього з контекстного меню на пункті проекту у вікні Solution Explorer потрібно вибрати команду Add\New Item… і вказати шаблон Visual C# Items\Code\Code File. Далі слід задати назву файлу і натиснути кнопку Add. До проекту буде додано новий порожній файл із вказаною назвою. Введемо в редакторі коду текст програми з лістингу 10.1.

Якщо програму запустити (натиснути F5), на екрані відобразиться порожнє вікно (рис. 10.1). Хоча клієнтська область вікна порожня, а рядок заголовку не містить тексту, його можна переміщувати, згортати і розгортати, змінювати розміри та закривати. Лістинг 10.1 представляє ядро застосунку Windows Forms, яке можна доповнювати кодом довільної складності, щоб забезпечити необхідну функціональність програми.

Розглянемо цей код детально. У рядку 1 вказано, що програма використовує простір імен System.Windows.Forms. Саме у ньому описано ряд класів, які потрібні програмі Windows Forms. Весь інший код (рядки 3…9) описує клас SimplestWinFormsApplication, який представляє програму. Клас має один метод Main. Єдина інструкція цього методу (рядок 7) містить виклик

статичного методу Run класу Application, якому через параметр передається екземпляр класу Form.

Лістинг 10.1. Найпростіший застосунок Windows Forms

```
1  using System.Windows.Forms;
2
3  class SimplestWinFormsApplication
4  {
5      public static void Main()
6      {
7          Application.Run(new Form());
8      }
9  }
```

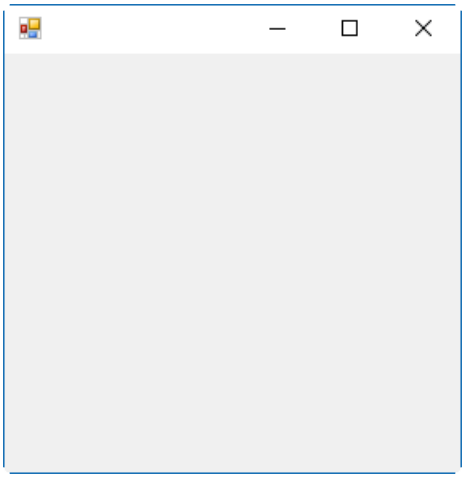


Рис. 10.1. Вікно найпростішого застосунку Windows Forms з лістингу 10.1

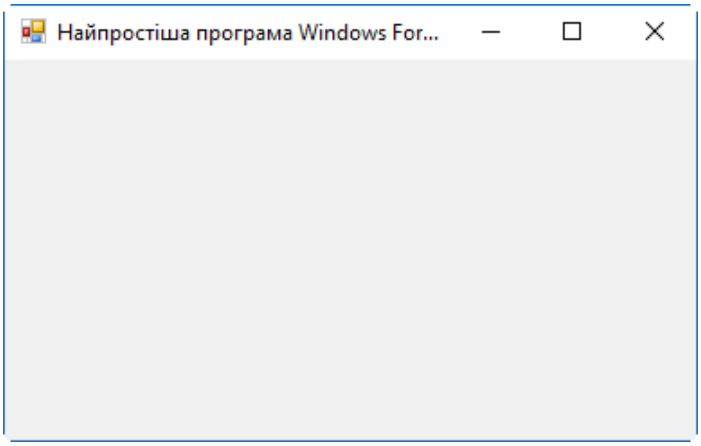


Рис. 10.2. Вікно застосунку Windows Forms з лістингу 10.2

Клас Form інкапсулює поведінку стандартного вікна застосунку Windows Forms. Методу Run класу Application передаємо об'єкт Form, який представлятиме головне вікно застосунку. Метод Run фактично "перетворює" консольний застосунок у віконний.

З точки зору операційної системи метод Run класу Application створює чергу для отримання повідомлень, які Windows відправляє застосункам. Метод Run не повертає керування програмі, поки користувач не закриє переданий цьому методу об'єкт Form. Загальний порядок виконання такий: якщо метод Run поверне керування програмі після відображення форми, метод Main (а разом з ним і програма) припиняє роботу і в процесі очищення системи вікно знищується.

Об'єкт Form, який передається у метод Run, має набір властивостей та методів, за допомогою яких можна налаштувати його вигляд та керувати

роботою вікна. Код з лістингу 10.1 має суттєвий недолік, який полягає в тому, що посилання на об'єкт форми не збережено, а безпосередньо передано методу Run. Тому до запуску метода Run вигляд форми налаштувати неможливо.

Щоб можна було задати вигляд вікна програми, потрібно спочатку створити форму, задати значення її властивостям, і тоді викликати метод Run, передавши йому форму. Це реалізовано у лістингу 10.2.

Лістинг 10.2. Створення форми і налаштування її властивостей

```
1  using System.Windows.Forms;
2
3  class SimplestWinFormsApplication
4  {
5      public static void Main()
6      {
7          Form frmMain = new Form();
8          frmMain.Text = "Найпростіша програма Windows Forms";
9          frmMain.Width = 500;
10         frmMain.Height = 250;
11         Application.Run(frmMain);
12     }
13 }
```

Таблиця №10.1. Найвживаніші елементи класу Application

Назва	Опис
Властивості	
CompanyName	Назва компанії, яка співставлена із застосунком
CurrentCulture	Дані про мову і регіональні налаштування застосунку
ExecutablePath	Повний шлях до виконавчого файлу застосунку (включаючи назву файлу)
ProductName	Назва продукту, співставленого з застосунком
ProductVersion	Версія продукту, співставленого з застосунком
StartupPath	Шлях до виконавчого файлу застосунку (без назви файлу)
Методи	
EnableVisualStyles	Вмикає візуальні стилі для застосунку
Exit	Закриває застосунок
Restart	Перезапускає застосунок (припиняє роботу і запускає новий екземпляр)
Run	Запускає цикл обробки повідомлень застосунку

Таблиця №10.2. Найвживаніші елементи класу Form

Назва	Опис
Властивості	
AcceptButton	Повертає або задає кнопку на формі, яка автоматично спрацьовує, коли користувач натискає клавішу Enter
CancelButton	Повертає або задає кнопку на формі, яка автоматично спрацьовує, коли користувач натискає клавішу Esc
ActiveControl	Повертає або задає активний елемент керування
ActiveForm	Повертає поточну активну форму для застосунку
AutoSize	Змінює розміри форми у відповідності зі значенням AutoSizeMode
AutoSizeMode	Повертає або задає режим, за допомогою якого форма може автоматично змінювати свої розміри
ContextMenu	Задає контекстне меню для форми (воно відображається при клацанні правою кнопкою миші)
Controls	Колекція елементів керування форми
Dock	Задає, які межі елемента керування прив'язані до меж батьківського елемента керування, і визначає, як компонент мінятиме розмір разом з батьківським
Focused	Показує, чи має форма фокус вводу
MaximizeBox	Отримує або задає значення, яке вказує, чи відображається кнопка "Розгорнути" в рядку заголовка форми
Menu	Задає головне меню форми
Modal	Повертає значення, яке вказує, чи відображається форма як модальна
Owner	Повертає або задає форму, яка володіє цією формою
ShowIcon	Повертає або задає значення, яке вказує, чи відображати піктограму в рядку заголовка
StartPosition	Повертає або задає початкове положення форми в режимі виконання
Text	Заголовок форми
Visible	Повертає або задає значення, яке вказує, чи відображується форма
WindowState	Повертає або задає значення, яке вказує на стан форми: розгорнутий, згорнутий або звичайний
Методи	
Close	Закриває форму
CreateGraphics	Створює об'єкт Graphics для форми
Dispose	Звільняє всі ресурси, використані об'єктом форми
Focus	Фокусує форму для вводу
Hide	Приховує форму
Show	Показує форму
ShowDialog	Показує форму як модальне вікно

У рядку 7 створюємо об'єкт frmMain – екземпляр класу Form. Далі задаємо значення для властивостей форми Text, Width та Height. Ці властивості є

елементами класу Form. Властивість Text представляє рядок заголовку вікна, а властивості Width та Height – відповідно ширину та висоту вікна у пікселях. При виклику методу Run передаємо йому посилання на попередньо створену форму. Якщо запустити цю програму, вікно прийме вигляд як на рис. 10.2.

Таким чином, для типового віконного застосунку особливу роль відіграють два класи: Application та Form. Клас Application інкапсулює загальну поведінку застосунку, а клас Form описує стандартне вікно Windows Forms. Деякі найвживаніші елементи класів Application та Form приведено у таблицях 10.1 та 10.2.

Спадкування класу Form

В обох розглянутих перед цим прикладах (лістинг 10.1 та 10.2) як головне вікно програми використано об'єкт – екземпляр класу Form. Але найчастіше використовують не клас Form, а похідний від нього клас. У програмі оголошують клас, який успадковує клас Form:

Базовий клас – Form

```
class TheForm: Form
{
    // Оголошення елементів форми
}
```

Цей клас розробляють таким чином, щоб він забезпечував необхідну поведінку та вигляд вікна. Потім створюють екземпляр оголошеного класу і передають його методу Run класу Application. Як результат – вікно матиме попередньо розроблену функціональність. Клас – нащадок Form має доступ до всіх публічних та захищених елементів класу Form. Для нього можна задати конструктор, у якому налаштувати вигляд вікна.

У лістингу 10.3 приведено опис класу MainForm, який успадковує клас Form (рядки 6…21). Конструктор MainForm (рядки 8…14) ініціалізує вікно, присвоюючи значення деяким його властивостям. Так, властивість StartPosition задає початкове розташування вікна на екрані. Її значення FormStartPosition.CenterScreen показує, що вікно буде розміщене по центру екрана.

Форма, як і інші компоненти, має свій перелік подій. Події представляють ситуації під час виконання програми, які можуть бути цінні з точки зору реалізованого нею алгоритму. Одна з найвживаніших подій – подія клацання мишкою Click. Кожна подія, реалізована у класі Control (а клас Form успадковує його), має метод з такою ж назвою, що й подія, але з префіксом On. Код цього метода виконуватиметься завжди при виникненні його події. Такі методи мають один параметр, який передає інформацію про подію. Вони є

віртуальними (virtual), тому їх можна перевизначити у похідному класі, позначивши ключовим словом override. Оскільки ці віртуальні методи початково оголошено як захищені (protected), то перевизначені методи також треба оголосити захищеними.

Лістинг 10.3. Оголошення класу – нащадка Form

```
1  using System;
2  using System.Windows.Forms;
3
4  class WinFormsApplication
5  {
6      class MainForm : Form
7      {
8          public MainForm()
9          {
10             Text = "Головне вікно програми Windows Forms";
11             Width = 400;
12             Height = 250;
13             StartPosition = FormStartPosition.CenterScreen;
14         }
15
16         protected override void OnClick(EventArgs e)
17         {
18             base.OnClick(e);
19             MessageBox.Show("Клікнули на формі");
20         }
21     }
22
23     public static void Main()
24     {
25         MainForm frmMain = new MainForm();
26         Application.Run(frmMain);
27     }
28 }
```

У лістингу 10.3 в класі MainForm оголошено метод OnClick (рядки 16…20), який автоматично спрацьовуватиме при кожному клацанні лівою кнопкою мишки на формі. Він містить дві інструкції. Перша (рядок 18) викликає такий же метод базового класу. Друга (рядок 19) виводить на екран текстове повідомлення.

У методі Main класу WinFormsApplication (він представляє нашу програму в цілому) за раніше розглянутою схемою створюємо екземпляр класу MainForm та передаємо його методу Run (рядки 25,26). Результат виконання програми (після

клацання мишкою) приведено на рис. 10.3. Зауважимо: вікно з повідомленням з'являтиметься після кожного клацання мишкою у вікні.

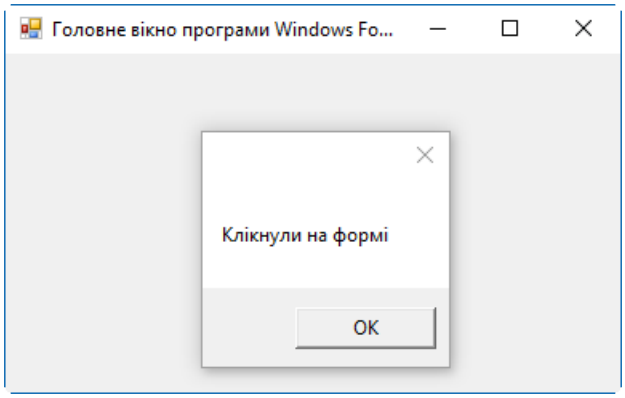


Рис. 10.3. Результат виконання програми з лістингу 10.3

Проект Windows Forms Application у Visual Studio

Для пришвидшення і спрощення розробки типових застосунків, середовище Microsoft Visual Studio містить ряд готовиш шаблонів. Серед них є шаблон для застосунків Windows Forms. Щоб створити застосунок Windows Forms, використавши типовий для Visual Studio підхід, слід під час створення проекту вибрати шаблон Visual C#\Windows\Windows Forms Application.

Visual Studio згенерує базовий, готовий для компіляції код у кількох файлах. Створений таким способом проект містить кілька файлів з кодом мовою C#:

- Program.cs
- Form1.cs
- Form1.Designer.cs

Базовий для застосунків Windows Forms код, який ми розглядали раніше, розміщено у файлі Program.cs (лістинг 10.4). В ньому описано клас Program, який входить в простір імен StudioWindowsFormsApplication, назва якого співпадає з назвою проекту. Клас містить один статичний метод Main.

Коментарі над методом Main – для XML-документації. У параметрах компіляції можна вказати файл, в якому буде зібрано всю подібну XML-документацію програми. Це зручний спосіб документування коду.

Безпосередньо перед методом Main розташовано атрибут STAThread в квадратних дужках. Він вказує виконавчому середовищу .NET, що програму слід запускати в окремому потоці, щоб забезпечити її коректну взаємодію з COM. перша інструкція методу Main (рядок 17) містить виклик методу EnableVisualStyles класу Application. Це дозволяє задати "сучасніший" зовнішній вигляд елементам керування вікна.

Лістинг 10.4. Код файлу Program.cs застосунку Windows Forms

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using System.Windows.Forms;
6
7  namespace StudioWindowsFormsApplication
8  {
9      static class Program
10     {
11         /// <summary>
12         /// The main entry point for the application.
13         /// </summary>
14         [STAThread]
15         static void Main()
16         {
17             Application.EnableVisualStyles();
18             Application.SetCompatibleTextRenderingDefault(
19                 false);
20             Application.Run(new Form1());
21         }
22     }
23 }
```

Метод SetCompatibleTextRenderingDefault, який викликається у другій інструкції методу Main, задає спосіб промальовування тексту в елементах керування вікна. Значення false, передане йому через параметр, забезпечить сумісність програми зі старими операційними системами з графічною підсистемою GDI (новіші використовують GDI+).

Третя інструкція методу Main викликає попередньо розглянутий метод Run класу Application. Методу Run передаємо посилання на об'єкт класу Form1, який створюємо безпосередньо при виклику. Сам клас Form1 визначено у файлі Form1.cs. Він описує вікно програми, вигляд якої відображено в дизайнері форм. Щоб проглянути вміст файлу Form1.cs, потрібно активувати відповідний пункт вікна Solution Explorer (буде показано дизайнер форм) і натиснути клавішу F7. Код цього файлу приведено в лістингу 10.5.

На відміну від попередньо розглянутого прикладу (лістинг 10.3), де клас форми та клас Program оголошено в одному файлі, Visual Studio при створенні застосунку Windows Forms клас кожної форми розміщує в окремий файл.

Назву класу і назву файлу, в якому його описано, можна змінити. Для цього використовують команду Rename контекстного меню у вікні Solution Explorer.

Лістинг 10.5. Код файлу Form1.cs застосунку Windows Forms

```
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel;
4  using System.Data;
5  using System.Drawing;
6  using System.Linq;
7  using System.Text;
8  using System.Threading.Tasks;
9  using System.Windows.Forms;
10
11 namespace StudioWindowsFormsApplication
12 {
13     public partial class Form1 : Form
14     {
15         public Form1()
16         {
17             InitializeComponent();
18         }
19     }
20 }
```

Розглянемо код лістингу 10.5 детальніше. У рядках 1…9 розміщено інструкції використання різних просторів імен, які найчастіше застосовують у застосунках. З усього цього переліку для нашого найпростішого випадку необхідним є лише один простір імен – System.Windows.Forms, у якому описано клас Form.

Далі у просторі імен StudioWindowsFormsApplication (його назва співпадає з назвою проекту) оголошено частковий (partial) клас Form1 (рядки 13…19). Цей клас представлятиме вікно програми і реалізовуватиме його поведінку. Модифікатор partial (частковий) використано, щоб розділити оголошення класу на кілька файлів.

За замовчуванням клас форми має один конструктор без параметрів (рядки 15…18) з викликом методу InitializeComponent. Метод потрібен для підтримки у застосунку форм, розроблених за допомогою дизайнера форм. Код цього методу середовище генерує автоматично і розміщує його в іншому файлі – Form1.Designer.cs. Щоб проглянути його код, слід активувати відповідний пункт у Solution Explorer (лістинг 10.6).

Лістинг 10.6. Код файлу Form1.Designer.cs застосунку Windows Forms

```
namespace StudioWindowsFormsApplication
{
    partial class Form1
    {
        /// <summary>
        /// Змінна потрібна дизайнеру форм.
        /// </summary>
        private System.ComponentModel.IContainer
            components = null;

        /// <summary>
        /// Очищення всіх використаних ресурсів.
        /// </summary>
        /// <param name="disposing">true якщо ресурси слід
        /// вивільнити; в іншому випадку - false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Код, згенерований Windows Form Designer

        /// <summary>
        /// Метод потрібен для підтримки дизайнера форм -
        /// не змінюйте його вміст у редакторі коду.
        /// </summary>
        private void InitializeComponent()
        {
            this.SuspendLayout();
            //
            // Form1
            //
            this.AutoScaleDimensions =
                new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode =
                System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize =
                new System.Drawing.Size(284, 261);
            this.Name = "Form1";
            this.Text = "Form1";
            this.ResumeLayout(false);
        }
```

```
47
48          #endregion
49      }
50  }
```

Таким чином, за допомогою механізму часткових класів клас форми розділено на дві частини:

- Перша описана у файлі Form1.cs. Саме тут програміст розміщує код, який забезпечує необхідну функціональність вікна, програмуючи поведінку елементів керування та реалізуючи алгоритм програми.
- Друга описана у файлі Form1.Designer.cs. Вона автоматично генерується середовищем Visual Studio. В процесі проектування вікна за допомогою дизайнера форм (зокрема, при розміщенні компонентів на формі, створенні оброблювачів їх подій тощо) ця частина оголошення класу форми також автоматично змінюватиметься. Вручну вміст цього файлу змінювати не потрібно.

Не зважаючи на суттєвий (для порожнього вікна) обсяг коду, структура файлу Form1.Designer.cs дуже проста. У рядку 1 починається простір імен проекту StudioWindowsFormsApplication (див. також рядок 7 лістингу 10.4 та рядок 11 лістингу 10.5). Таким чином, всі оголошення у лістингах 10.4, 10.5 та 10.6 розміщено в одному просторі імен StudioWindowsFormsApplication, але в різних файлах. У рядку 3 продовжено оголошення часткового класу форми Form1 (див. також рядок 13 лістингу 10.5).

Об'єкт components (рядок 8…9) представляє сукупність компонентів (елементів керування) форми. Метод Dispose вивільняє всі ресурси пам'яті, використані компонентами форми. Він використовується при знищенні форми.

Метод InitializeComponent (рядки 31…46) ініціалізує форму та її компоненти, створюючи їх та задаючи певні значення їхніх властивостей. Якщо за допомогою дизайнера форм розмістити на форму якийсь компонент, то Visual Studio автоматично додасть у цей метод код для створення цього компонента. Коли якійсь властивості компонента присвоїти значення (за допомогою вікна Properties), це також автоматично відобразиться у методі InitializeComponent. При видаленні компонента з форми середовище автоматично видаляє код, згенерований для нього. Таким чином, всі дії по розміщенню та налаштуванню компонентів форми за допомогою дизайнера відображуються у коді метода InitializeComponent. При правильному користуванні засобами Visual Studio втручатися у цей код ніколи не потрібно.

У приведеному в лістингу 10.6 коді в цьому методі задано значення деяких властивостей форми. Так, у рядку 44 заголовку форми (властивість Text) присвоєно значення Form1.

Зауважимо, що в рядку 25 директивою #region починається блок коду, який охоплює метод InitializeComponent. Закінчується цей блок директивою #endregion у рядку 48. Подібний регіон є елементом групування тексту програми в редакторі коду. Він ніяк не впливає на процес виконання програми. Наявність директиви #region спричинить до появи у редакторі коду напроти регіону мінікнопки "-", за допомогою якої можна згорнути відповідний регіон. Після цього мінікнопка буде позначена знаком "+". Натиснувши на неї ще раз, можна розгорнути код регіону. Таким чином, регіони вносять додатковий елемент у структуру коду, полегшуючи його сприйняття, особливо для великих програм. Функціонального значення з точки зору програмування регіони не мають.

Якщо запустити розглянутий щойно згенерований проект Windows Forms, то відобразиться порожнє вікно, аналогічне вікну на рис. 10.1.

Програмне створення елементів керування вікна

.NET та C# містять класи, які описують всі найвживаніші елементи керування: кнопки, списки, меню, поля текстового вводу та багато інших. Під час проектування вікна за допомогою дизайнера форм їх просто розміщують мишкою, вибираючи відповідний компонент з вікна Toolbox. Але, щоб краще зрозуміти суть елементів керування, спочатку розглянемо процес їх створення програмно.

Кожен елемент керування є екземпляром певного класу. Його створення нічим не відрізняється від створення об'єкта – екземпляра якогось класу. Наприклад, елемент керування "кнопка" описує клас Button. Щоб створити елемент керування "кнопка", потрібно створити екземпляр цього класу. Налаштувати вид цього елемента керування можна, задавши потрібні значення властивостям відповідного об'єкта. Щоб елемент керування відобразився у вікні, слід задати його властивість Parent. Ця властивість вказує на батьківський об'єкт, якому належатиме елемент керування. Дію, яка виконуватиметься при натисканні кнопки, задають через призначення її оброблювача події Click (подія клацання мишкою).

У лістингу 10.7 приведено код програми з вікном, яке містить одну кнопку. Проект створено так, як описано в пункті "Найпростіша програма Windows Forms" цього розділу (див. лістинг 10.1 та супровідний текст).

Вікно програми описує клас FormWithButton (рядки 7…25). Конструктор цього класу містить код створення та налаштування кнопки (рядки 14…18). Спочатку створюємо екземпляр btn1 класу Button. Потім задаємо текст на кнопці (рядок 15) та її координати у вікні (рядок 16).

Лістинг 10.7. Вікно з кнопкою

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace WindowWithButton
{
    class FormWithButton : Form
    {
        public FormWithButton()
        {
            StartPosition = FormStartPosition.CenterScreen;
            Text = "Просте вікно з кнопкою";

            Button btn1 = new Button();
            btn1.Text = "Кнопка";
            btn1.Location = new Point(70, 30);
            btn1.Click += ButtonOnClick;
            btn1.Parent = this;
        }

        void ButtonOnClick(object objSrc, EventArgs args)
        {
            MessageBox.Show("Кнопку натиснено!", "Button");
        }
    }

    class FormWithButtonProg
    {
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.Run(new FormWithButton());
        }
    }
}
```

Щоб при натисканні на кнопку виконувалась якась дія, слід задати для неї оброблювач події Click. Оброблювачем може бути будь-який метод, що не повертає результату та має параметри потрібного для оброблювача типу. Перелік та тип параметрів описано в документації MSDN для кожної події. У приведеному коді оброблювачем події Click для кнопки є метод ButtonOnClick. Перший його параметр вказує на об'єкт, що спричинив подію, а другий – описує саму подію.

У рядку 17 для кнопки btn1 за допомогою оператора += задаємо оброблювач події Click, вказуючи назву метода ButtonOnClick. Справа у тому, що компонент може мати кілька оброблювачів тієї ж події, які в цьому випадку спрацьовуватимуть послідовно один за другим. Метод ButtonOnClick виводить повідомлення, яке вказує, що він спрацював.

Але всі описані вище дії не приведуть до відображення створеного і налаштованого елемента керування у вікні, поки для нього не буде вказано батьківське вікно. Це здійснюють за допомогою властивості Parent. У рядку 18 властивості Parent присвоюємо посилання на поточний екземпляр класу FormWithButton (ключове слово this). Тільки після цього кнопку буде видно у вікні (рис. 10.4). Замість безпосереднього присвоєння властивості Parent можна також додати елемент керування у колекцію Controls форми. Ця колекція містить загальний перелік всіх елементів керування форми. Тому наступні інструкції у своїй дії будуть еквівалентними:

```
btn.Parent = this;
```

та

```
Controls.Add(btn);
```

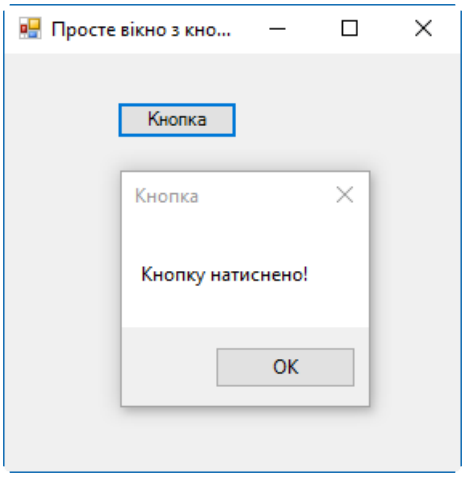


Рис. 10.4. Вікно з кнопкою (лістинг 10.7). При натисканні на кнопку з'являється повідомлення

Елементи керування з панелі Toolbox

При проектуванні вікон застосунку в середовищі Visual Studio елементи керування рідко створюють програмно. Найчастіше їх розставляють у дизайнері форм, переміщуючи потрібні компоненти з панелі Toolbox. Панель Toolbox розташована з лівої сторони вікна Visual Studio (рис. 10.5). Якщо її не видно, слід натиснути вкладку Toolbox, або вибрати команду головного меню View\Toolbox (чи натиснути комбінацію клавіш Ctrl+W,X).

Панель Toolbox містить багато компонентів, згрупованих за їх видом (елементи керування загального призначення Common Controls, діалогові вікна Dialogs тощо). Пункт All Windows Forms містить загальний перелік всіх

доступних для застосунку Windows Forms компонентів. Щоб розмістити елемент керування у дизайнері форм, слід просто перетягти його мишкою з панелі Toolbox в потрібне місце форми. Можна також вибрати компонент у панелі Toolbox і потім клацнути лівою кнопкою мишки в дизайнері. Вибраний компонент з'явиться на формі. На рис. 10.5 на формі розташовано одну кнопку. Розробку форми в дизайнері шляхом розміщення на ній компонентів називають *візуальним проектуванням*, так як воно дозволяє оцінити вигляд вікна безпосередньо під час його розробки. Це значно зручніше, ніж вручну писати код по створенню елементів керування вікна, у коді налаштовувати їх та тримати в уяві картину їх розміщення на формі.

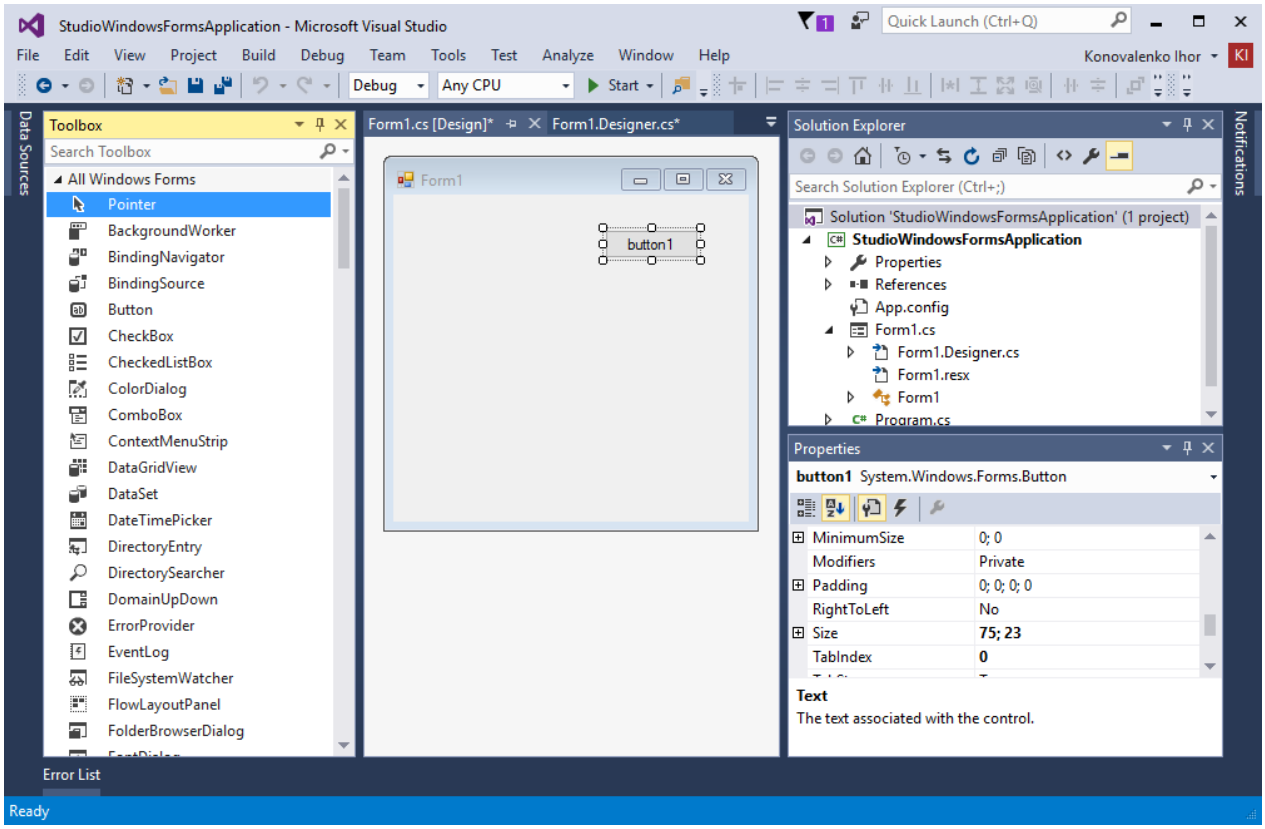


Рис. 10.5. Панель Toolbox у Visual Studio (зліва), конструктор форм (посередині), вікно Properties (справа знизу)

На форму можна помістити багато різних елементів керування. Кожен з них, залежно від типу, має свій набір властивостей. Ці властивості можна переглядати у вікні Properties, яке за замовчуванням знаходиться справа знизу (рис. 10.5). Вікно Properties відображує перелік властивостей поточного вибраного у дизайнері компонента. Щоб задати значення властивості, слід вибрати компонент, знайти потрібну властивість і ввести чи вибрати для неї потрібне значення. Зауважимо, що значення властивостям компонентів можна задавати як у процесі візуального проектування у дизайнері форм, так і програмно – під час виконання.

Крім властивостей, кожен компонент має також множину співставлених з ним подій. Їх можна проглянути, натиснувши у вікні Properties кнопку Events (події). Повернутись до перегляду властивостей можна за допомогою сусідньої кнопки Properties. Щоб задати для компонента оброблювач події, потрібно вибрати компонент у дизайнері, і у вікні Properties натиснути кнопку Events. Далі слід вибрати зі списку потрібну подію (наприклад, Click), і двічі клацнути мишкою у полі після назви події. Середовище Visual Studio у редакторі коду створить потрібний метод – оброблювач події.

Лістинг 10.8. Автоматично згенерований код для опису кнопки

```
1  private System.Windows.Forms.Button button1;
2
3  ...
4
5  private void InitializeComponent()
6  {
7      this.button1 = new System.Windows.Forms.Button();
8      ...
9      //
10     // button1
11     //
12     this.button1.Location   =   new   System.Drawing.Point(169,
13 28);
14     this.button1.Name = "button1";
15     this.button1.Size = new System.Drawing.Size(75, 23);
16     this.button1.TabIndex = 0;
17     this.button1.Text = "button1";
18     this.button1.UseVisualStyleBackColor = true;
19     this.button1.Click +=
20         new System.EventHandler(this.button1_Click);
21     ...
22     this.Controls.Add(this.button1);
   }
```

Під час візуального проектування форми в опис її класу у файлі Form1.Designer.cs (початок назви може бути інший) автоматично додається відповідний код. Наприклад, якщо на форму поставити компонент кнопки Button та створити для нього оброблювач події Click, система додасть у клас форми поле для кнопки та в методі InitializeComponent згенерує відповідний код. У лістингу 10.8 приведено фрагмент коду з файлу Form1.Designer.cs, автоматично згенерований після розміщення на формі кнопки.

В рядку 1 оголошено поле button1 типу System.Windows.Forms.Button. Назву (button1) можна змінити за допомогою вікна Properties. Клас Button з простору імен System.Windows.Forms показує, що це кнопка. Екземпляр класу Button

створюється в методі InitializeComponent (рядок 7), і посилання на нього зберігається полем button1. В рядках 12…19 налаштовуються властивості елемента керування. Якщо їх значення змінювати у вікні Properties, змінюватиметься і приведений опис. Інструкція у рядках 18…19 додає для кнопки оброблювач події Click. З коду видно, що ним буде метод button1_Click цього ж класу. У рядку 21 кнопка button1 додається у колекцію елементів керування форми Controls, що забезпечить її відображення у цій формі.

Таким чином, бачимо, що середовище Visual Studio автоматизовано генерує код того ж виду, який було розглянуто під час програмного створення елементів керування.

Елементи керування

В загальному випадку, елемент керування можна визначити як візуальний об'єкт вікна, призначений для вводу чи виводу інформації. Зазвичай елементи керування займають прямокутну область екрану (хоча можуть бути непрямокутними або навіть прихованими) і виконують команди, що надходять з клавіатури або миші. Елементи керування у дизайнері форм називають також *компонентами*.

Базовим класом для елементів керування є клас Control. Він підтримує множину подій для введення даних. До них належать, наприклад, події клавіатури KeyDown, KeyUp, KeyPress та миші – MouseDown, MouseUp, MouseMove. Крім цього, елементи керування повинні "промальовувати" себе на екрані – для цього служить подія Paint.

Елементи керування перетворюють інформацію про операції користувача в прості події і служать рівнем абстрагування між користувачем та застосунком. Наприклад, для типової роботи елемента Button достатньо задати текст для кнопки і обробник події Click.

Одна з головних властивостей класу Control – Parent. Вона вказує на інший, батьківський для даного, об'єкт типу Control. У процесі виконання програми на дисплеї відображаються лише елементи керування з коректно заданою властивістю Parent. Положення елемента завжди задається відносно батьківського елемента керування. Відображується компонент також на поверхні батьківського.

Клас Form є винятком з цього правила. Значення властивості Parent об'єкта типу Form зазвичай дорівнює null, тобто батьківським вікном для форми є робочий стіл. Однак об'єкт Form, у якого значення властивості TopLevel дорівнює false (тобто батьківським для форми є не робочий стіл), може бути нащадком будь-якого іншого елемента керування. Така методика використовується в

застосунках з багатодокументним інтерфейсом (Multiple Document Interface, MDI).

Призначити форму "предком" елемента керування можна двома способами:

- Задати у властивості Parent елемента керування посилання на форму.
- Додати елемент керування у колекцію Controls форми за допомогою її методу Add.

Щоб видалити елемент керування з форми, слід виконати протилежні дії:

- Присвоїти властивості Parent значення null.
- Видалити елемент керування з колекції Controls форми за допомогою її методу Remove.

Колекція Controls містить всі елементи керування форми. Перебравши елементи колекції в циклі, можна отримати доступ до всіх елементів керування:

```
foreach (Control Ctrl in Controls)
{
    ...
}
```

В ролі батьківського об'єкта для елементів керування, крім об'єкта Form, можуть виступати також інші елементи, найчастіше – Panel та GroupBox

За замовчуванням всі елементи керування видимі і доступні. Якщо властивості Visible присвоїти значення false, елемент керування зникне з поверхні батьківського компонента. Він і далі буде залишатися у батьківському наборі елементів керування, але у вікні не відображатиметься. Всі нащадки невидимого елемента керування також невидимі.

Але частіше змінюють властивість Enabled. Елемент з властивістю Enabled = false відображається на екрані, але виглядає знебарвленим, або вимкненим. Такий елемент керування не отримує фокус введення і не реагує на команди клавіатури або миші.

Кожен елемент керування має розмір та положення, яке задають відносно лівого верхнього кута його батьківського компонента. Положення задають у пікселях за допомогою властивості Location типу Point, який має дві властивості: X (відступ від лівого краю батьківського компонента) та Y (відступ від верхнього краю). Розмір задають через властивість Size – об'єкт Size з властивостями Width (ширина) та Height (висота).

Текст для елемента керування задає властивість Text.

Розглянемо найвживаніші елементи керування дещо детальніше. Зауважимо, що нижче приведено лише деякі головні характеристики елементів керування. За їх повним описом слід звертатися до MSDN.

Кнопка: Button

Поведінку одного з найвживаніших елементів керування – кнопки – інкапсулює клас Button. Кнопки використовують для ініціювання користувачем якоїсь дії. На кнопці, як правило, присутній текст, але вони можуть містити зображення, з текстом або без нього. Для задання зображення використовують властивість Image. Якщо кнопка повинна відображувати і текст, і зображення, то слід задати їх взаємне розміщення властивістю TextImageRelation.

Задати положення та розмір компонента button1 можна так:

```
button1.Location = new Point(100, 50);
button1.Size = new Size(90, 30);
```

Розмір також можна задати без створення об'єкта Size:

```
button1.Width = 90;
button1.Height = 30;
```

Текст, який відображується у кнопці, задають властивістю Text. При цьому властивість Font задає шрифт, яким виводитиметься текст, а BackColor та ForeColor – кольори фону та напису. Нижче приведено приклад налаштування виду кнопки:

```
button1.Text = "Кнопка";
button1.Location = new Point(60, 50);
button1.Size = new Size(170, 55);
button1.ForeColor = Color.DarkSlateBlue;
button1.Font = new Font("Times New Roman", 24, FontStyle.Italic);
```

На рис. 10.6 приведено вигляд вікна з такою кнопкою.

Група елементів керування: GroupBox

Група елементів керування представлена компонентом класу GroupBox. Група по периметру виділена лінією і вгорі містить текст, заданий властивістю Text. Рамка групи темніша від кольору фону. Компонент використовують для візуального об'єднання логічно пов'язаних між собою елементів керування. Приклад групи показано на рис. 10.7.

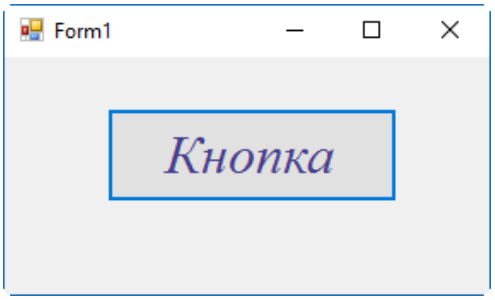


Рис. 10.6. Кнопка – об'єкт класу Button

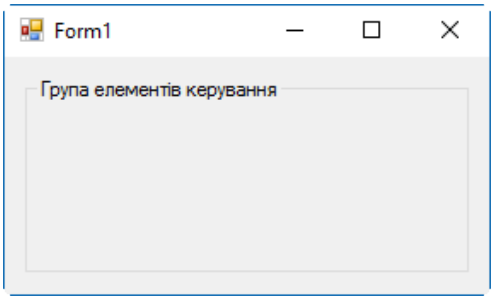


Рис. 10.7. Група елементів керування – об'єкт класу GroupBox

Мітка: Label

Мітка Label – це елемент керування, який відображає нередагований текст. Сам текст задають у властивості Text. Хоча мітка може відображати текст в декількох рядках, вона не виводить смуг прокрутки, якщо текст не вміщується в елементі керування.

Як правило, мітки використовують для розташування у вікні незмінного тексту, що пояснює призначення інших елементів керування. Вигляд мітки видно з рис. 10.8.

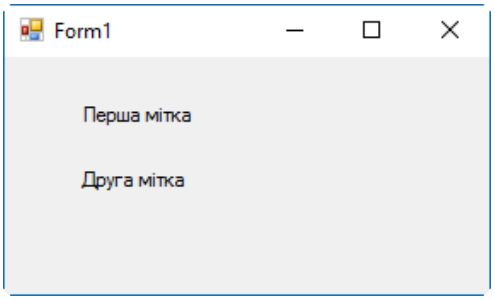


Рис. 10.8. Мітки – об'єкти класу Label

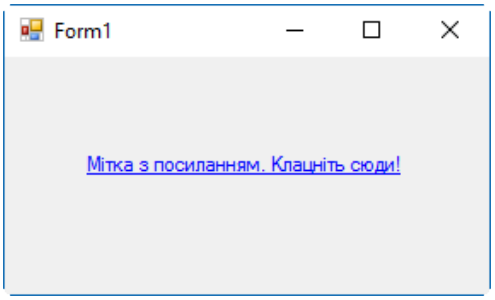


Рис. 10.9. Мітка-посилання – об'єкт класу LinkLabel

Мітка-посилання: LinkLabel

Цей елемент керування відображає виділений текстовий рядок, щоб показати, що клацання мишкою по ньому викликає якусь дію (наприклад запуск програми або відкриття Web-сторінки). Клацання на посилання LinkLabel ініціює подію LinkClicked. Вигляд мітки видно з рис. 10.9.

Графічний блок: PictureBox

Елемент керування PictureBox служить для розміщення у вікні зображення, яке може бути растровим (у форматі bmp, jpg, png чи gif) або векторним (формату wmf). Зображення задається властивістю Image. Властивість SizeMode класу PictureBox регулює масштабування зображення в межах батьківського елементу керування. Вона може приймати такі значення:

- Normal – зображення розміщується починаючи з верхнього лівого кута графічного блоку. Може не вміститися в межах поточних розмірів елемента керування.
- CenterImage – зображення розміщується по центру елемента керування. Також може не вміститися в межах поточних розмірів.

- AutoSize – графічний блок підганяється під розмір зображення. Якщо зображення велике, воно разом з графічним блоком може не вміститися у вікні.
- StretchImage – зображення підганяється під розмір елемента керування (і може спотворюватися, якщо співвідношення сторін компонента не співпадає зі співвідношенням сторін самого зображення).
- Zoom – зображення масштабується без спотворень.

Елементи керування PictureBox часто використовують для надання вікну програми привабливішого вигляду. Приклад зображення, виведеного у вікно за допомогою цього компонента, приведено на рис. 10.10.

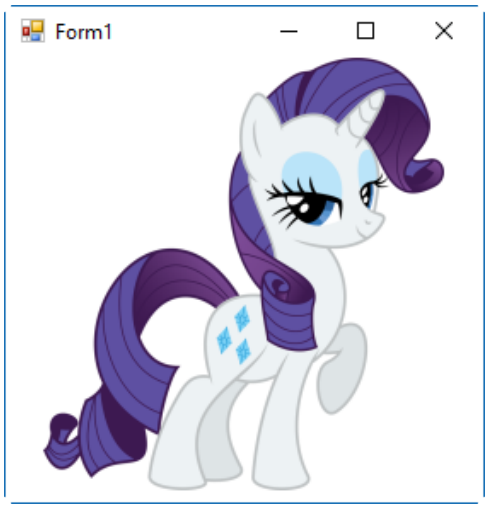


Рис. 10.10. Графічний блок – об'єкт класу PictureBox

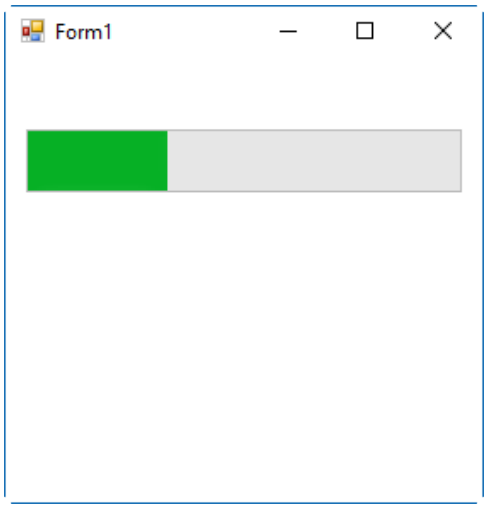


Рис. 10.11. Індикатор процесу – об'єкт класу ProgressBar

Індикатор процесу: ProgressBar

Цей елемент керування використовують для індикації перебігу якогось достатньо тривалого процесу. Основні властивості ProgressBar – це цілочисельні властивості Minimum, Maximum і Value. По мірі виконання процесу властивості Value присвоюють зростаючі значення в діапазоні від Minimum до Maximum.

Властивість Style задає стиль елемента керування і може приймати значення: Blocks (за замовчуванням), Continuous або Marquee. При виборі Marquee відображається не ступінь виконання, а анімація, хід процесу. Це використовують, коли під час виконання визначити ступінь виконання неможливо, але потрібно показати, що програма працює. На рис. 10.11 показано загальний вигляд елемента керування ProgressBar.

Прапорець: CheckBox

Зазвичай елемент керування CheckBox виглядає як невеликий квадратик зліва від тексту, в якому можна ставити маркер – "галочку". Однак якщо властивості Appearance задати значення Button (за замовчуванням – Normal), елемент керування буде схожий на кнопку, яка прийматиме два положення: "натиснено" та "відтиснено".

Логічна властивість Checked вказує поточний стан прапорця – "встановлений" (true) або "скинутий" (false). Подія CheckedChanged інформує про зміну стану перемикача.

Іноді не достатньо тільки двох станів прапорця – встановлений або скинутий. Для використання третього стану потрібно властивості ThreeState логічного типу присвоїти значення true. Далі замість властивості Checked потрібно використовувати CheckState. Це значення з переліку CheckState, що складається з елементів Unchecked, Checked і Indeterminate, Крім цього, замість події CheckedChange, використовують подію CheckStateChanged.

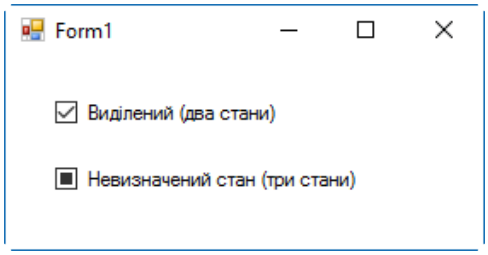


Рис. 10.12. Прапорець – об'єкт класу CheckBox

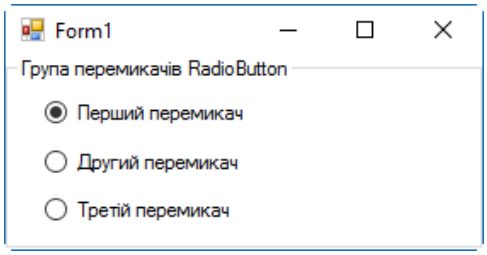


Рис. 10.13. Перемикач – об'єкт класу RadioButton

На рис. 10.12 показано обидва види компонента CheckBox. Зверху розміщено прапорець, який може приймати два стани. Він виділений (властивість Checked = true). Знизу розташовано прапорець, що може приймати три стани (на рисунку його властивість CheckState = Indeterminate).

Перемикач: RadioButton

Цей елемент керування зазвичай містить розташований зліва від тексту кружечок, який можна відмічати. Як і для CheckBox, у класу RadioButton є властивість Appearance, якій можна задати значення Button – тоді елемент керування виглядає як кнопка. Властивість Checked логічного типу показує, чи перемикач відмічено. При його зміні виникає подія CheckedChanged.

Зазвичай кілька елементів керування RadioButton розташовують на одному батьківському елементі (найчастіше ним виступає компонент GroupBox). Тоді стандартна поведінка групи перемикачів така: при виділенні одного з

перемикачів всі інші перемикачі групи стають невиділеними. Таким чином, в кожен момент часу серед групи є тільки один виділений елемент керування RadioButton. Саме такий випадок ілюструє рис. 10.13. На групі елементів (яку представляє компонент GroupBox) розташовано три елементи RadioButton.

Повзунок TrackBar

Цей елемент керування виглядає як шкала з поділками. Властивість TickStyle визначає стиль поділок, і може приймати значення: None, TopLeft, BottomRight і Both. Клас TrackBar має цілі властивості: Minimum, Maximum, SmallChange, LargeChange і Value (поточне значення). Minimum і Maximum визначають діапазон значень Value.

Повзунок може приймати горизонтальне та вертикальне положення. Положення задають властивістю Orientation, яка може мати два значення: Horizontal та Vertical.

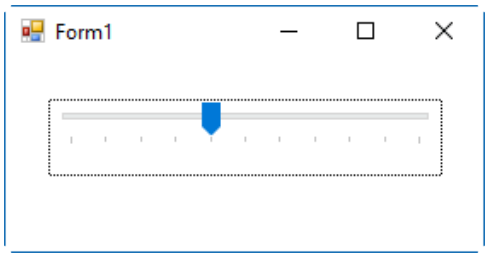


Рис. 10.14. Повзунок – об'єкт класу TrackBar

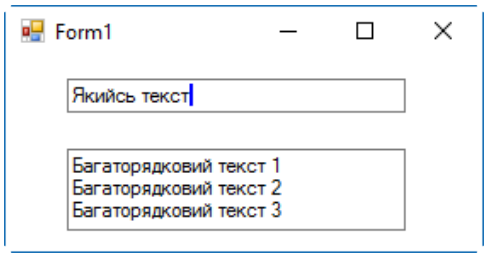


Рис. 10.15. Текстовий блок – об'єкт класу TextBox

Повзунки часто використовують для вибору значення з дискретного діапазону. На рис. 10.14 показано вигляд горизонтального елемента керування TrackBar.

Текстовий блок: TextBox

Текстовий блок TextBox служить для вводу та редагування тексту. Найважливіша властивість, яка дозволяє вводити та змінювати текст – Text. Вона містить текст, що відображається в елементі керування.

Властивість Multiline визначає, чи може елемент керування містити багаторядковий текст. За замовчуванням вона дорівнює false, і текст вводять лише в один рядок. Якщо задати Multiline = true, то компонент стає багаторядковим.

Елемент керування TextBox використовують і для вводу пароля – тоді слід задати властивість PasswordChar символьного типу. Найчастіше вказують

символ "*". Тоді при введенні в компонент тексту відображуватимуться не введені символи, а задані властивістю PasswordChar.

На рис. 10.15 показано дві форми елемента керування TextBox – однорядковий та багаторядковий.

Текстовий блок з форматуванням: RichTextBox

Елемент керування RichTextBox містить текст у форматі RTF (Microsoft Rich Text Format). За замовчуванням, цей компонент багаторядковий. Якщо обсяг тексту в компоненті значний, властивістю ScrollBars задають відображення полос прокрутки.

Компонент дозволяє програмно проводити форматування тексту, задаючи шрифт, його розмір, колір, тип тощо. На рис. 10.16. показано елемент керування RichTextBox з введеним у нього масивом тексту.

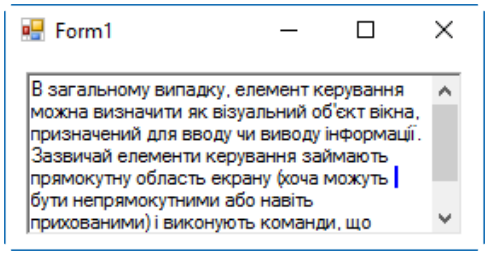


Рис. 10.16. Текстовий блок з форматуванням – об'єкт RichTextBox

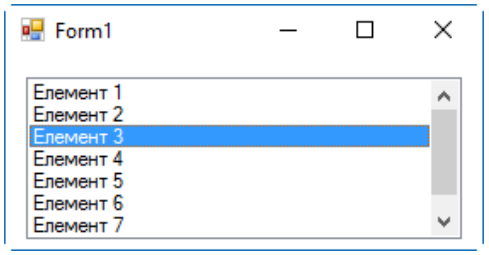


Рис. 10.17. Список – об'єкт класу ListBox

Список: ListBox

Елемент керування виду "список" (його представляє клас ListBox) містить перелік об'єктів. Користувач може вибрати один (або кілька) пунктів списку, використовуючи клавіатуру або мишу. Вибрані об'єкти виділяються. Відображувані елементи списку задають властивістю Items. Якщо властивості Sorted логічного типу задати значення true, елементи списку автоматично сортуватимуться за зростанням.

Властивості SelectedIndex та SelectedItem містять індекс виділеного елемента та сам виділений елемент списку. Якщо у списку не виділено жодного елемента, SelectedIndex приймає значення -1, а SelectedItem – значення null.

Загальний вигляд типового елемента керування ListBox показано на рис. 10.17.

Список з прапорцями: CheckListBox

При необхідності вибору кількох елементів списку замість ListBox краще використовувати елемент керування CheckedListBox. У ньому зліва від кожного пункту розташовані прапорці з можливістю відмічування маркерами. Способи взаємодії з цим компонентом за допомогою клавіатури і миші простіші і зрозуміліші для користувача, ніж у списках з можливістю вибору кількох елементів.

Властивості тільки для читання CheckedIndices та CheckedItems надають набори цілочисельних індексів або об'єкти, вибрані в даний момент.

Вигляд елемента керування CheckListBox видно з рис. 10.18.

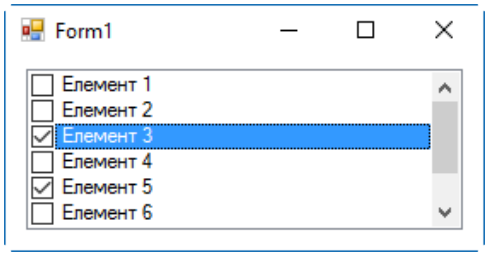


Рис. 10.18. Список з прапорцями – об'єкт класу CheckListBox

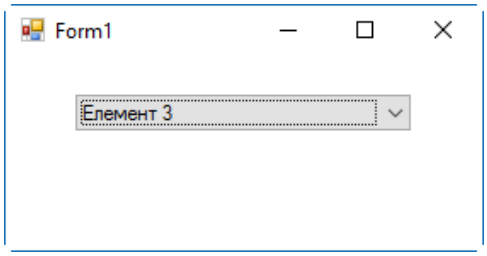


Рис. 10.19. Комбінований список – об'єкт класу ComboBox

Комбінований список: ComboBox

Компонент ComboBox є комбінацією списку та текстового блоку. В нормальному стані відображується тільки текстове вікно зі стрілкою "вниз" праворуч від нього. При клацанні цієї стрілки розкривається список, з якого можна вибрати елемент. Властивість DropDownStyle задає стиль списку і може мати такі значення:

- Simple – текст можна редагувати, список постійно відкритий.
- DropDown – текст можна редагувати, список розгортається при натисканні на стрілку.
- DropDownList – текст не можна редагувати, список розгортається при натисканні на стрілку.

Властивості SelectedIndex та SelectedItem містять індекс виділеного елемента та сам виділений елемент списку.

На рис. 10.19 показано вигляд елемента керування ComboBox в стилі DropDownList.

Головне меню: MenuStrip

Клас MenuStrip дозволяє створити меню, яке прив'язане до верхнього краю вікна. Кожен елемент меню може мати різний вид і описується класом ToolStripMenuItem. Головна подія кожного елемента меню – Click.

Як правило, кожен елемент меню позначають коротким текстом, який описує суть операції, що виконуватиметься при виборі цього елемента. Кожен елемент може мати своє підменю, що розгортається при активуванні батьківського елемента. Для групування команд часто використовують розділювачі – горизонтальні лінії, які дозволяють відокремити різні групи команд в межах меню. Меню використовують для побудови розгалужених ієрархічних наборів команд програми.

Елементи меню містяться у властивості Items елемента керування MenuStrip. Але у Microsoft Visual Studio найчастіше меню проектують за допомогою візуального дизайнера, який з'являється при розміщенні на формі цього компонента.

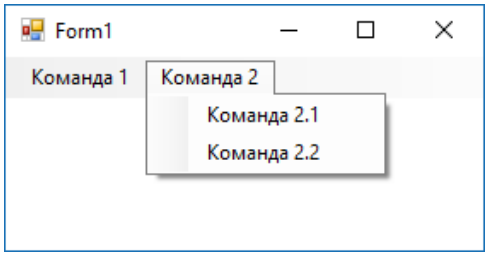


Рис. 10.20. Головне меню – об'єкт класу MenuStrip

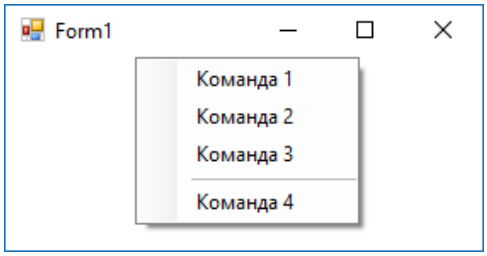


Рис. 10.21. Контекстне меню – об'єкт класу ContextMenuStrip

Кожен елемент можна налаштовувати окремо. Для цього слід його вибрати в дизайнері форм. Далі за допомогою вікна Properties для елемента задають текст, піктограму (зображення), клавіші швидкого виклику та багато інших параметрів.

На рис. 10.20 показано головне меню вікна з розгорнутим другим підменю.

Контекстне меню: ContextMenuStrip

Клас ContextMenuStrip дозволяє створити контекстне меню, яке викликається клацанням правої кнопки миші на компоненті. Кожен елемент меню може мати різний вид і описується (як і для головного меню) класом ToolStripMenuItem. Головна подія кожного елемента меню – Click.

Конструюють контекстне меню так само, як і головне – за допомогою дизайнера меню, який викликається активуванням компонента контекстного меню в дизайнері форм.

Візуальні елементи керування вікна мають властивість з назвою ContextMenuStrip, якій присвоюють посилання на елемент керування ContextMenuStrip. Тоді при клацанні правою кнопкою миші на елементі керування з'являтиметься вказане контекстне меню (рис. 10.21).

Панель інструментівToolStrip

Клас ToolStrip дозволяє створити панель інструментів, яка містить набір елементів керування (як правило, кнопок) для швидкого виклику команд. Кожен елемент панелі інструментів може налаштовуватися окремо. Кнопкам можна задавати малюнки та написи.

Вміст панелі керування конструюють за допомогою дизайнера, який активується при виділенні компонента ToolStrip у дизайнері форм. На панелі можна розміщувати елементи керування таких видів: кнопка (Button), мітка (Label), кнопка з випадаючим списком (DropDownButton), розділювач (Separator), комбінований список (ComboBox), текстовий блок (TextBox) та індикатор процесу (ProgressBar). Кожен розміщений на панелі інструментів елемент налаштовують окремо, задаючи за допомогою вікна Properties значення його властивостям. На рис. 10.22 показано вікно з панеллю інструментів, яка містить чотири кнопки Button, попарно розділені елементом Separator, мітку Label та текстовий блок TextBox.

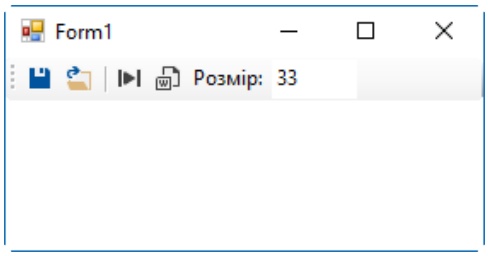
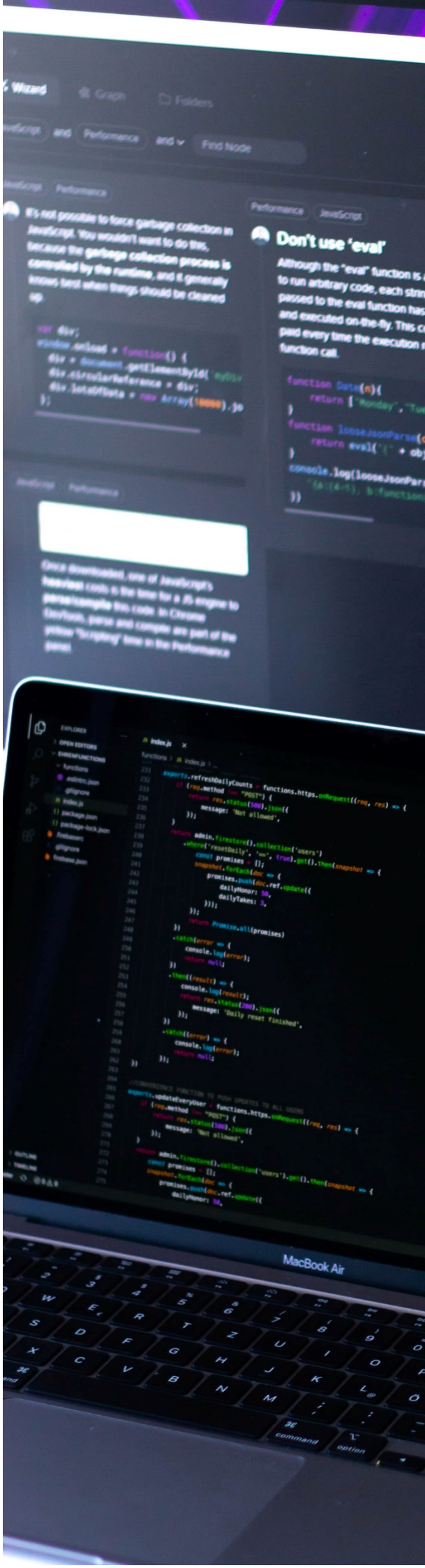


Рис. 10.22. Панель інструментів – об'єкт класу ToolStrip

Розділ 11. Основи Windows Presentation Foundation

- Огляд технології WPF
- Найпростіший WPF-застосунок
- XAML
- Макет вікна
- Панелі
- Елементи керування
- Приклад простої програми WPF



Огляд технології WPF

Windows Presentation Foundation (*WPF*) – це платформа для розробки та виконання .NET застосунків з покращеним графічним інтерфейсом користувача [7]. Відомо, що зображення на екрані монітора формується за допомогою відеокарти. Майже всі сучасні комп'ютери оснащені відеокартами, які дозволяють відтворювати якісну графіку. Проте, за винятком ігор, більшість застосунків не використовують цих графічних можливостей. Частково це пов'язано з тим, що їх залучення потребує занурення у доволі складну сферу програмування DirectX – API для високопродуктивного використання графіки та засобів мультимедіа.

До WPF для відтворення елементів інтерфейсу у Windows використовувалась бібілотека GDI (і вдосконалена версія GDI+). Але GDI для промальовування елементів інтерфейсу використовує програмні алгоритми, і дуже незначно залучає апаратні можливості відеокарти. Технологія Windows Forms працює саме таким чином.

Недоліками такого підходу є:

- Мала ефективність. Не використовуються апаратні можливості сучасних відеокарт.
- Обмеженість. Оскільки функціонал відтворення графіки вмонтований в операційну систему, програміст має малий вплив на вид і поведінку елементів керування у вікнах.

На відміну від Windows Forms, технологія WPF для відтворення графіки використує бібліотеку Direct3D (яка є підсистемою DirectX для роботи з тривимірною графікою). Це не тільки суттєво підвищує продуктивність роботи графічної підсистеми, а й дозволяє застосувати принципово нові підходи при генерації візуального інтерфейсу користувача.

Для опису вікон та елементів керування в них WPF- застосунки зазвичай використовують мову розмітки XAML (*eXtensible Application Markup Language*), що в певній мірі нагадує застосування HTML для формування web-вмісту. Це, порівняно з технологією Windows Forms, значно розширює можливості формування візуального контенту.

WPF реалізується через сукупність збірок, які розподілені між двома шарами:

- Керований шар WPF містить набір класів і типів, які використовують .NET Framework. Цей шар містить три збірки, які утворюють API WPF:
  - збірка WindowsBase містить фундаментальні типи, які лежать в основі застосунку WPF, у тому числі класи Application та Window;

- збірка PresentationFramework містить класи і типи, які формують API розробника;
- збірка PresentationCore містить низькорівневі класи і типи, які виступають будівельними компонентами для PresentationFramework.

► Шар інтеграції з медіа містить некерований код і проектує WPF-конструкції на DirectX, який у свою чергу використовує апаратні ресурси відеокарти. Некерований код робить цей шар максимально продуктивним.

Найпростіший WPF-застосунок

Microsoft Visual Studio містить шаблон WPF App, який дозволяє швидко створити порожній проект WPF-застосунку. Для його використання слід вибрати команду меню File/ New/ Project та вказати шаблон WPF App. Visual Studio згенерує кілька файлів та автоматично створить посилання (references) на необхідні для роботи WPF-застосунку бібліотеки. Зокрема, буде згенеровано два класи: App, похідний від класу Application, та MainWindow, похідний від класу Window. Кожен з цих класів реалізовано відразу в двох вихідних файлах: .cs (файл з кодом C#) та .xaml (файл з розміткою XAML).

XAML – це основана на XML мова розмітки, яку використовують для налаштування класів. Більшість WPF- застосунків використовують XAML для задання статичних та візуальних аспектів графічного інтерфейсу користувача через налаштування властивостей об'єктів. Мову C# використовують для задання активної частини програми та реалізації її поведінки (зокрема, й реакції на зміну стану описаних за допомогою XAML елементів інтерфейсу).

Наприклад, за допомогою XAML можемо розмістити у вікні кнопку та описати її вигляд. Але дії, які виконуватимуться після її натискання, розміщують у файлі C#.

Головне вікно застосунку описане парою файлів: MainWindow.xaml та MainWindow.xaml.cs. У файлі MainWindow.xaml.cs оголошено частковий клас MainWindow, який інкапсулює поведінковий механізм вікна.

Запишемо у конструкторі MainWindow після виклику методу InitializeComponent наступні рядки коду:

```
Title = "My Simple Window";
Content = "Hi there.";
```

Перший рядок задає заголовок вікна, а другий встановлює його вміст. Якщо запустити цей застосунок, на екрані буде відображено головне вікно (рис. 11.1).

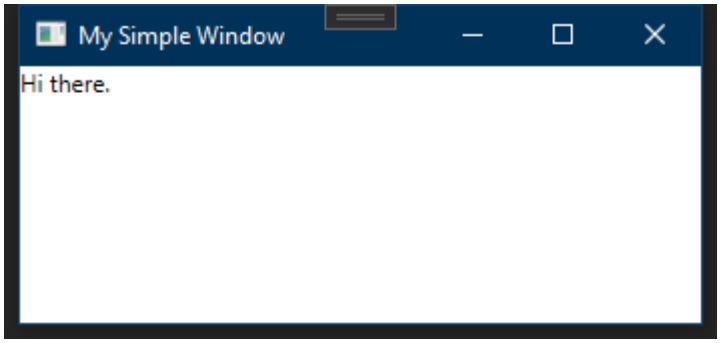


Рис. 11.1. Головне вікно простого WPF-застосунку.

Клас Application інкапсулює механізм, який дозволяє застосунку виконуватися. Кожен застосунок WPF має єдиний екземпляр цього класу. Клас Application оголошено у просторі імен System.Windows (на відміну від іншого класу з такою ж назвою – Application, який оголошено у просторі імен System.Windows.Forms і використовується для застосунків Windows Forms).

Клас Application містить події, які генеруються у певні моменти життєвого циклу застосунку: Startup (виникає під час запуску), Exit (під час закінчення роботи), Activated та Deactivated (коли застосунок відповідно отримує та втрачає фокус вводу), SessionEnding (коли сесія Windows завершується).

Компіляція програм, які використовують як XAML, так і код C#, є двохетапною. Спочатку компілятор опрацьовує XAML- файли та генерує тимчасовий проміжний код, після чого на основі C#-файлів та результатів першого етапу формується виконавчий файл.

XAML

Одна з найважливіших концепцій WPF полягає в тому, що графічний інтерфейс користувача є деревом об'єктів – екземплярів класів WPF. Це дерево на вершині має єдиний вузол, і кожен вузол дерева може містити інші вузли.

Мова XAML чутлива до регістру символів, і відповідає більшості синтаксичних правил XML. У застосунках WPF XAML зазвичай використовують для опису графічного інтерфейсу, але може бути використана для опису інших об'єктів (наприклад, Application). Слід зауважити, що застосування XAML не є обов'язковим – графічний інтерфейс можна описати виключно засобами C#, але застосування XAML дозволяє розділити інтерфейс користувача та код, а також робить застосунок гнучкішим, а код компактнішим.

При цьому область використання XAML не обмежується тільки WPF. Наприклад, платформа Microsoft Silverlight використовує XAML для побудови інтерфейсу web- застосунків.

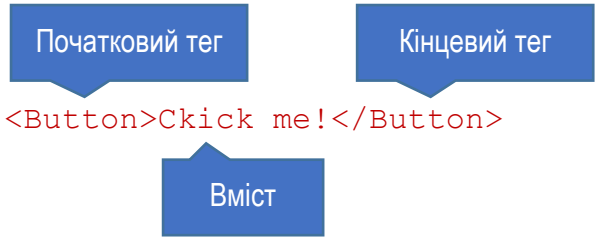
Кожен документ XAML на верхньому рівні містить єдиний елемент верхнього рівня. Всі інші елементи є вкладеними. WPF-застосунок у ролі елементів верхнього рівня може використовувати класи Window, Application, і Page.

Елементи XAML

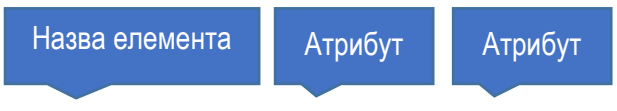
У файлі XAML об'єкти інтерфейсу користувача представлено елементами розмітки. Зазвичай синтакс елемента містить три частини: початковий тег, область вмісту і кінцевий тег. При цьому:

- Початковий тег оточують трикутними дужками (< , >). Тег повинен містити назву класу для об'єкта інтерфейсу. Початковий тег позначає початок опису елемента.
- Кінцевий тег відкривають двосимвольним рядком (</), а закривають трикутною дужкою (>). Кінцевий тег позначає кінець опису елемента.
- Область вмісту – це ділянка між початковим та кінцевим тегами. Тут можна розмістити текст, інші елементи, або нічого. Все, що описане у області вмісту, формуватиме вміст об'єкта інтерфейсу за замовчуванням.

Наприклад, нижче приведено найпростіший XAML-опис елемента – кнопки (клас Button).



Якщо для об'єкта інтерфейсу слід задати вміст за замовчуванням, то наведений вище синтаксис є достатнім. Проте часто потрібно задати значення для інших властивостей об'єкта. Це роблять за допомогою атрибутів елемента XAML. Наприклад, кнопка має цілий ряд властивостей (довжину, ширину, колір тексту тощо), за допомогою яких можна надати їй потрібного вигляду. Щоб задати розміри кнопки, використовують її властивості: Width та Height. Тоді елемент розмітки XAML матиме вигляд:




```
<Button Width="100" Height="50">
    Ckick me!
</Button>
```

Атрибути завжди розміщують всередині початкового тегу, відразу після назви елемента. Вони не можуть бути в області вмісту. Синтаксис атрибута містить назву властивості, знак рівності (=) та рядок зі значенням атрибута у

подвійних лапках. Елемент може містити довільне число атрибутів, які розділяють пробілом (пробілами, або пишуть з нового рядка).

Інколи об'єкт інтерфейсу не потребує вмісту. Наприклад, кнопка може бути без напису. Такий елемент можна задати двома способами:

- Кінцевий тег можна розмістити відразу після початкового (або між ними може бути тільки порожній простір):

```
<Button Width="100" Height="50"> </Button>
```

- Використати синтаксис порожнього елемента. При цьому замість пари початкового та кінцевого тега застосовують єдиний тег, який повністю ідентичний початковому, але закінчується не символом ">", а парою символів "/>":

```
<Button Width="100" Height="50"/>
```

Коли на основі розмітки XAML створюється об'єкт, який представляє компонент графічного інтерфейсу користувача, підсистемою .NET виконуються такі кроки:

- Створюється об'єкт вказаного у тезі класу. Для цього викликається конструктор без параметрів. Навіть якщо клас має інші конструктори, їх задати не можна.
- Присвоюється вміст об'єкта, який описаний в області вмісту розмітки.
- Встановлюються значення інших властивостей об'єкта на основі вказаних у розмітці атрибутів.

Якщо властивості прості, то значення для них зручно задавати за допомогою атрибутів. Але інколи зручніше використовувати синтаксис елемента-властивості у розмітці. При цьому властивість вказують не як атрибут у початковому тезі, а як окремий елемент розмітки, вкладений в область вмісту базового елемента. Назву для цього вкладеного елемента складають з назви класу та назви властивості, з'єднавши їх крапкою:


```
<Button>
    <Button.Background>
        LightGray
    </Button.Background>
    Ckick me!
</Button>
```

Цей фрагмент аналогічний до наступного:

```
<Button Background="Lightgray">Ckick me!</Button>
```

Тег елемента-властивості не може містити атрибутів. Зауважимо, що хоча такий синтакс має вигляд елемента, але він не описує створення об'єкта, а лише

задає значення його властивості. Синтакс елемента-властивості зручно використовувати, якщо значення для властивості є складним і багаторядковим.

Назви об'єктів

При виконанні застосунку на основі елементів розмітки XAML створюються об'єкти – екземпляри заданих у розмітці класів. За замовчуванням такі об'єкти не мають назв, тому до них неможливо звертатися з коду програми. У простих випадках цього й не потрібно. Але якщо програма повинна маніпулювати таким об'єктом, йому слід явно задати назву. Багато класів мають властивість Name, якій можна присвоїти значення, використавши x як атрибут:

```
<Button Name="btnClickMe">Click Me!</Button>
```

Проте, деякі класи не мають властивості Name. У цьому випадку можна використати спеціальну конструкцію XAML:

```
<Button x:Name="btnClickMe">Click Me!</Button>
```

За допомогою такого синтаксу в коді також можна буде звертатися до об'єкта із заданою назвою.

Простори імен

Кожен елемент, який відповідає компоненту графічного інтерфейсу, посилається на певний клас .NET. Щоб створити об'єкти цих типів, парсер XAML повинен знати визначення потрібних класів. Цю інформацію вказують у просторах імен розмітки. Простір імен вказують як атрибут xmlns кореневого елемента документа XAML. Два стандартних простори імен представляють класи WPF та класи XAML. Їхні імена схожі на веб-адреси:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Синтакс використання просторів імен у XAML не такий, як у C#. У XAML тільки один простір імен може бути простором імен за замовчуванням. Його оголошують за допомогою ключового слова xmlns. Назви типів з цього простору вказують безпосередньо. Саме так вказано назву класу Button у прикладах з попереднього пункту. Простором імен за замовчуванням може бути будь-який, але для застосунку WPF зручно, щоб ним був простір імен, що містить типи WPF (http://schemas.microsoft.com/winfx/2006/xaml/presentation).

Всім іншим просторам імен потрібно давати префікс. Префікс вказують після ключового слова xmlns через двокрапку. Якщо у розмітці слід використати тип (клас) з такого простору імен, перед його назвою вказують префікс, після якого ставлять крапку і далі розміщують назву типу (класу).

Макет вікна

До WPF графічний інтерфейс настільного застосунку складався з вікон, у яких статично розміщували елементи керування (це ж стосуєтсья і технології Windows Forms). Тільки деякі компоненти можна було прив'язувати до країв вікна. На відміну від цього, макет вікна у WPF є динамічним (і, як наслідок, дещо складнішим). При зміні розміру вікна графічні елементи у ньому можуть змінювати свій розмір та положення.

У WPF кожна зміна розміру вікна спричиняє перерахунок його макету. Це здійснюється за два кроки:

- Вимірювання. Вікно, панель чи інший контейнер запитує свої дочірні елементи, який розмір для них є бажаним. Ця операція є рекурсивною вздовж дерева елементів.
- Вирівнювання. Коли батьківський елемент (наприклад, вікно) отримує відповіді від усіх дочірніх, він обчислює, скільки простору він може виділити для кожного елемента насправді, і розташовує їх на своїй поверхні.

При створенні інтерфейсу за допомогою WPF зазвичай не задають точних розмірів для візуальних елементів. Натомість важливо правильно підібрати сукупність контейнерів, у яких буде розміщено графічні елементи.

Якщо ж все-таки необхідно явно задати розміри компонентів, використовують їх властивості Width та Height. Властивості MinWidth, MaxWidth, MinHeight, та MaxHeight дозволяють вказати діапазон розмірів. При зміні розмірів вікна розмір елемента ніколи не буде меншим за вказане мінімальне значення, та більшим за вказане максимальне.

Коли простір комірки макету, у якому розміщено графічний елемент, є більшим від нього, використовують атрибути вирівнювання: HorizontalAlignment та VerticalAlignment. Властивість HorizontalAlignment може набувати значень Left, Center, Right, чи Stretch; а VerticalAlignment – значень Top, Center, Bottom, чи Stretch. У свою чергу, графічний елемент може мати вміст, який розміщується на його поверхні відповідно до атрибутів вирівнювання вмісту: HorizontalContentAlignment та VerticalContentAlignment.

На розміщення вмісту графічного елемента впливають також властивості Padding та Margin (рис. 11.2). Padding задає відступи для вмісту всередині елемента, а Margin – навколо нього (подібний підхід використовують також у розмітці web-сторінок за допомогою HTML та CSS).

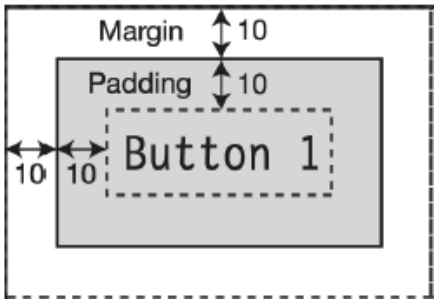


Рис. 11.2. Ілюстрація властивостей Padding та Margin. Розмітка такої кнопки виглядала б так:

```
<Button Padding="10" Margin="10">
  Button 1
</Button>
```

Існує три способи задання властивостей Padding та Margin:

- 1 значення. Якщо властивості присвоєно єдине значення, воно використовується для всіх чотирьох сторін (зліва, справа, зверху, знизу):

```
<Button Padding="10">Click Me!</Button>
```

- 2 значення. Якщо властивості присвоєно два значення, то перше з них використовується для відступів зліва та справа, а друге – для відступів зверху та знизу:

```
<Button Padding="10,5">Click Me!</Button>
```

- 4 значення. Якщо властивість має чотири значення, вони відповідають відступам зліва, зверху, справа та знизу відповідно:

```
<Button Padding="0,5,10,10">Click Me!</Button>
```

Панелі

Панелі є контейнерами для розміщення у них візуальних елементів інтерфейсу. Існує кілька класів панелей, кожен з яких розташовує дочірні елементи певним чином. Найважливішими видами панелей є:

- StackPanel – розміщує елементи у виді стеку;
- WrapPanel – розміщує елементи один за одним, і коли ряд елементів досягає краю панелі, переносить наступні елементи на наступний рядок (чи стовпчик);
- DockPanel – елементи прив'язуються до одного з чотирьох країв панелі;
- Grid є однією з універсальних та найскладніших панелей. Для панелі визначають рядки та стовпчики, а елементи розміщують у комірках сформованої таким чином сітки;
- Canvas – панель, на якій елементи розміщують відносно до певного кута панелі. Використання цієї панелі найбільше схоже на принципи побудови інтерфейсу користувача у Windows Forms;
- UniformGrid – найпростіший вид сітки, в якій всі комірки однакового розміру. Елементи розміщують у цих комірках.

На рис. 11.3 приведено приклад розташування п'яти елементів керування – кнопок на панелях різних видів. Зауважимо, що на панелях можна розташувати не лише кнопки, а будь-які елементи керування.

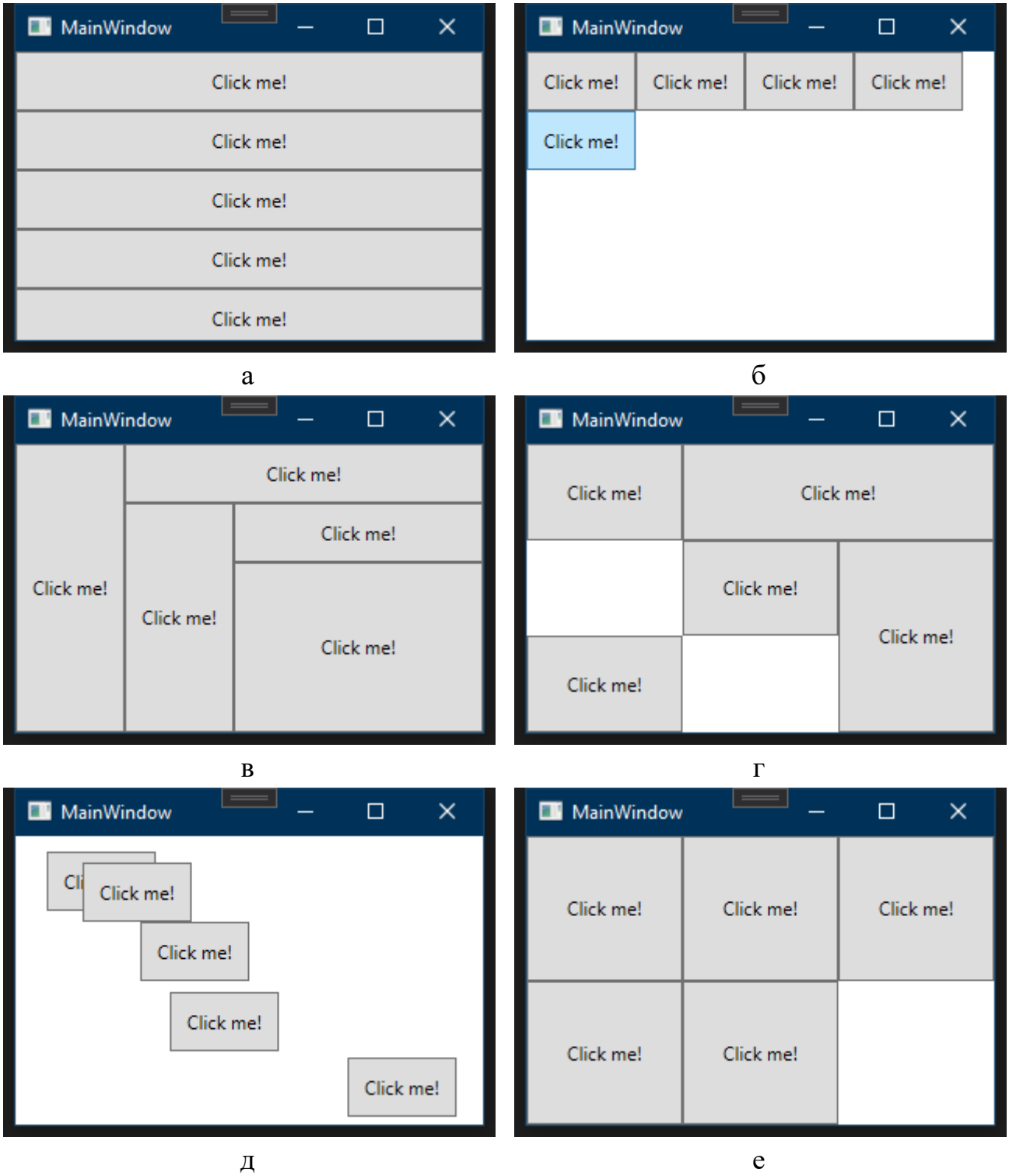


Рис. 11.3. Приклади розташування п'яти кнопок на панелях різних видів: StackPanel (а), WrapPanel (б), DockPanel (в), Grid (г), Canvas (д) і UniformGrid (е)

Найскладнішою панеллю є Grid, тому розглянемо її детальніше. Фрагмент розмітки XAML, який описує розташування кнопок відповідно до рис. 11.3, г, приведено у лістингу 11.1.

Лістинг 11.1. Приклад розмітки XAML для розміщення кнопок на панелі Grid

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Button Padding="10" Grid.Row="0" Grid.Column="0">
        Click me!
    </Button>
    <Button Padding="10" Grid.Row="0" Grid.Column="1"
        Grid.ColumnSpan="2">
        Click me!
    </Button>
    <Button Padding="10" Grid.Row="1" Grid.Column="1">
        Click me!
    </Button>
    <Button Padding="10" Grid.Row="1" Grid.Column="2"
        Grid.RowSpan="2">
        Click me!
    </Button>
    <Button Padding="10" Grid.Row="2" Grid.Column="0">
        Click me!
    </Button>
</Grid>
```

Означення рядків та стовпців панелі роблять відповідно за допомогою елементів <Grid.RowDefinitions> та <Grid.ColumnDefinitions>. Відповідно до приведеного коду, панель міститиме три рядки (є три елементи RowDefinition) і три стовпці (три елементи ColumnDefinition). Індексація рядків та стовпців починається з 0.

Щоб вказати, у якій саме комірці слід розмістити елемент графічного інтерфейсу, для нього вказують значення приєднаних властивостей Grid.Row та Grid.Column. Приєднаною властивістю є властивість, доступна тільки в контексті батьківського елемента. Розмістити кнопку у верхній лівій комірці можна,

вказавши для неї властивості Grid.Row="0" та Grid.Column="0" (перша кнопка у лістингу 11.1).

Елемент у панелі Grid може поширюватись на кілька суміжних комірок (як по горизонталі, так і по вертикалі). Зокрема, щоб розтягнути кнопку на 2 комірки сітки вправо, для неї слід вказати властивість Grid.RowSpan="2" (четверта кнопка на лістингу 11.1).

Зауважимо, що в кожну комірку сітки можна розмістити тільки один елемент. Але він, у свою чергу, також може бути панеллю, яка міститиме свій набір елементів графічного інтерфейсу.

Елементи керування

У платформах, які передували WPF (наприклад, MFC, Windows Forms), кожен елемент керування графічного інтерфейсу був призначений для однозначно визначених цілей і його візуальна репрезентація була зазвичай більш-менш одинаковою у різних застосунках. На відміну від цього, елементи візуального інтерфейсу WPF є значно гнучкішими і мають значно більше можливостей їх налаштування.

Елементи інтерфейсу мають пов'язані з ними події, для яких можна створювати методи – оброблювачі. У середовищі Visual Studio процедура створення такого оброблювача дуже схожа на аналогічну процедуру для застосунку Windows Forms.

Розглянемо деякі з найвживаніших елементів візуального інтерфейсу застосунків WPF.

Зображення: Image

Елемент дозволяє відобразити у вікні зображення таких типів: .jpg, .bmp, .png, .tiff, .wdp, та .ico. Файл з малюнком спочатку потрібно додати до проекту. Це можна зробити так: у Solution Explorer вибрати команду Add/New Folder, задати їй назву (наприклад, Images), і перетягнути туди файл з малюнком. Тоді наступний фрагмент розмітки відобразить файл у вікні (рис. 11.4):

```
<Image VerticalAlignment="Top" Source="Images/c.jpg"/>
```

Атрибут VerticalAlignment показує, що у контейнері, де розташоване зображення (наприклад Grid) воно по вертикалі буде вирівняне по верхньому краю.

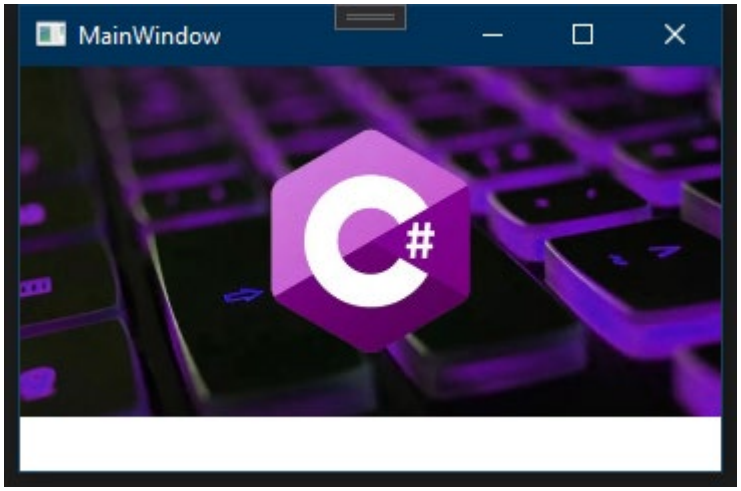


Рис. 11.4. Елемент Image

Мітка: Label

Мітка Label є простим елементом, який зазвичай використовують для відображення статичного тексту. Але вмістом мітки може бути будь-який інший похідний від класу UIElement елемент.

Ось приклад простої текстової мітки (вона за видом буде ідентичною до звичайної мітки Windows Forms):

```
<Label Content="Текст мітки" FontWeight="Bold"/>
```

Проте в мітку можна розмістити, наприклад, зображення:

```
<Label>
    <Image Source="Images/OceanPerch.jpg"></Image>
</Label>
```

При відтворенні такої мітки у ній не буде тексту, але буде малюнок. Можна створити також "комплексну мітку", у якій буде і текст, і малюнок. Для цього як вміст мітки потрібно задати панель (наприклад, StackPanel), а у ній вже розмістити і потрібний текст, і зображення (лістинг 11.2).

Лістинг 11.2. Мітка, яка складається з іншої мітки та зображення

Зовнішня мітка

Вміст зовнішньої мітки: панель-стек

Внутрішня мітка з текстом

```
1  <Label>
2      <StackPanel>
3          <Label HorizontalAlignment="Center" FontWeight="Bold"
4              FontSize="20">
5                  Приклад зображення у мітці
6          </Label>
7          <Image VerticalAlignment="Top"
8              Source="Images/c.jpg"/>
9      </StackPanel>
10 </Label>
```

Зображення у зовнішній мітці

Результат такої "комплексної" мітки показано на рис. 11.5. Цей приклад ілюструє гнучкість WPF- підходу до проектування графічного інтерфейсу. У певній мірі застосування XAML до проектування вікон нагадує використання HTML при створенні web-сторінок.

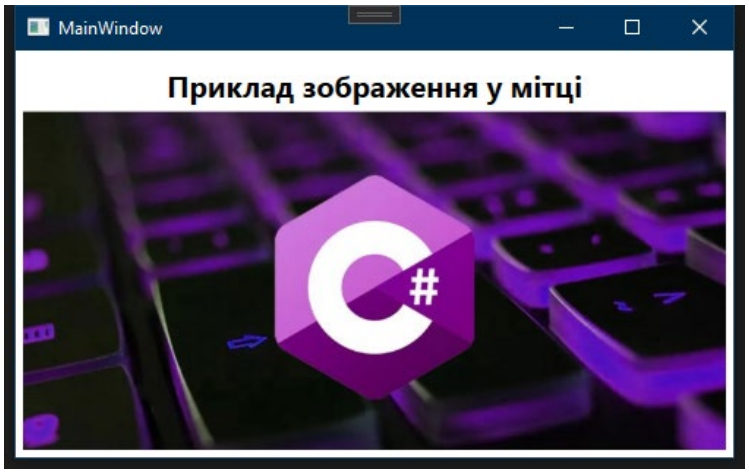


Рис. 11.5. Елемент Label зі складеним вмістом: текст та зображення

Кнопка: Button

Кнопку, як і у випадку з Windows Forms, використовують для того, щоб "дати знати" програмі про необхідність виконання певної дії. При натисканні кнопки виникає пов'язана з нею подія, з якою можна асоціювати метод – оброблювач. У ньому розміщують код мовою C#, який реалізує потрібну поведінку.

Назву метода – оброблювача події натискання вказують в атрибуті Click розмітки:

```
<Button Click="Button_Click">Button example</Button>
```

Якщо файл з розміткою XAML має назву MainWindow.xaml, то у файлі з кодом C# та назвою MainWindow.xaml.cs слід описати метод Button_Click:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Button just clicked");
}
```

У даному випадку оброблювач події Click просто відображує текстове повідомлення.

Як і у випадку з міткою, вміст кнопки може бути складеним. Якщо у лістингу 11.2 зовнішню мітку замінити кнопкою (замість тегу Label у рядках 1 та 10 вставити тег Button), то отримаємо велику кнопку, яка складається з мітки за зображенням під ним. Вигляд цієї кнопки буде таким самим, як вигляд мітки

на рис. 11.5, але елемент вестиме себе як кнопка (зокрема, матиме можливітсь обробляти подію Click).

Елементи CheckBox та RadioButton

Звичайна кнопка не зберігає інформації про те, чи її перед цим натискали. Таку можливість мають елементи прапорець CheckBox та перемикач RadioButton.

При кожному натисканні на прапорець CheckBox його стан змінюється шляхом зміни його властивості IsChecked. Елемент RadioButton схожий на CheckBox, за винятком того, що він належить групі, в межах якої у кожен момент часу може бути виділеним тільки один елемент RadioButton.

Ось приклад означення елементів CheckBox та RadioButton через XAML:

```
<StackPanel>
    <CheckBox Name="cb1" Margin="5,10,0,0">Прапорець 1</CheckBox>
    <CheckBox Name="cb2" Margin="5,0,0,0" IsChecked="True">
        Прапорець 2
    </CheckBox>
    <RadioButton Name="rb1" Margin="5,10,0,0">
        Радіокнопка 1
    </RadioButton>
    <RadioButton Name="rb2" Margin="5,0,0,0">
        Радіокнопка 2
    </RadioButton>
    <RadioButton Name="rb3" Margin="5,0,0,0" IsChecked="True">
        Радіокнопка 3
    </RadioButton>
</StackPanel>
```

На рис. 11.6 показано вікно, згенероване з такої розмітки.

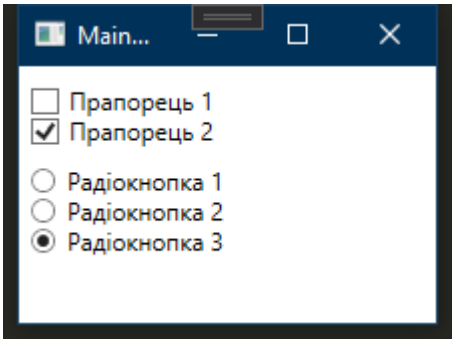


Рис. 11.6. Елементи CheckBox та RadioButton

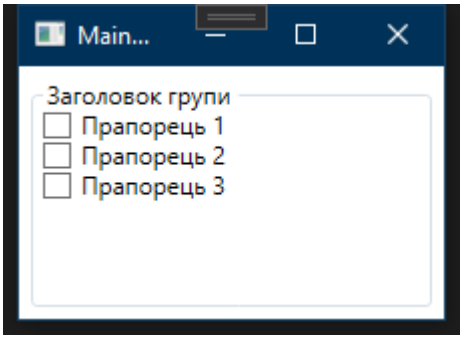


Рис. 11.7. Групування елементів керування за допомогою GroupBox

Група елементів: GroupBox та Expander

GroupBox призначений для оформлення групи елемнтів. Він має заголовок та оточує елементи, розташовані у ньому, рамкою (поведінка, аналогічна до

однойменного компонента Windows Forms). Ось приклад розмітки з використанням GroupBox (візуалізація розмітки – на рис. 11.7):

```
<GroupBox Header="Заголовок групи" Margin="5">
    <StackPanel>
        <CheckBox Name="cb1">Прапорець 1</CheckBox>
        <CheckBox Name="cb2">Прапорець 2</CheckBox>
    </StackPanel>
</GroupBox>
```

За допомогою коду XAML у заголовок групи легко розмістити не тільки текст, а й інший елемент. Наприклад, щоб зробити заголовок у вигляді прапорця CheckBox, слід в опис групи (перед <StackPanel> у попередньому прикладі) додати код:

```
<GroupBox.Header>
    <CheckBox Margin="7">Заголовок панелі</CheckBox>
</GroupBox.Header>
```

Вікно з такою групою показано на рис. 11.8.

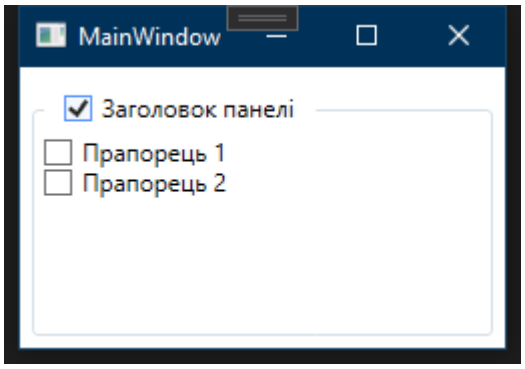
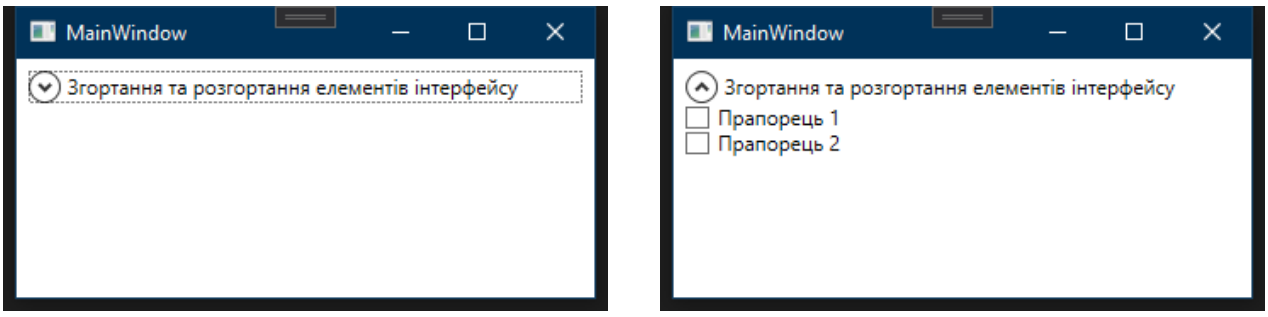


Рис. 11.8. Група з елементом CheckBox у заголовку

Елемент Expander, подібно до GroupBox, створює візуально пов'язану групу елементів інтерфейсу, але ця група може бути прихована від користувача (рис. 11.9,а), або, після натискання на кнопку зі стрілкою, розгорнута та відображена у вікні (рис. 11.9,б).



а б

Рис. 11.9. Елемент групування Expander у згорнутому (а) та розгорнутому (б) стані

Список: ListBox

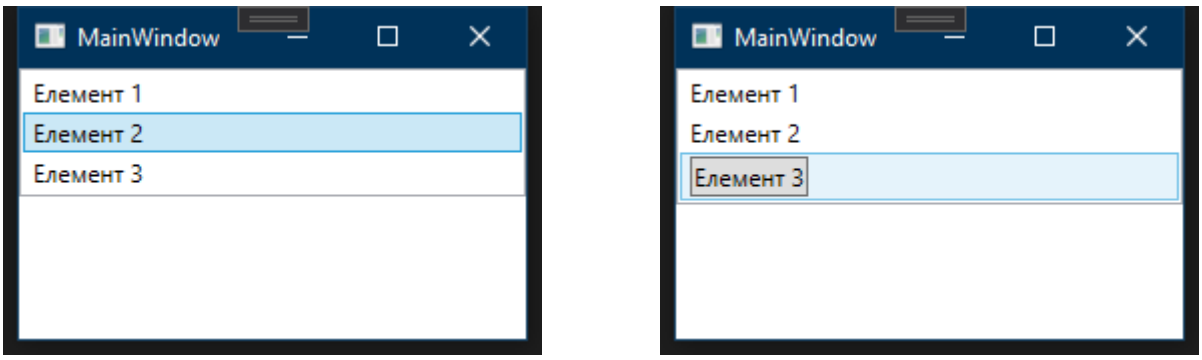
ListBox містить колекцію Items елементів, з якої їх можна вибрати. Елементи такої колекції можуть бути будь-які, похідні від класу UIElement. За замовчуванням ширина елементів встановлюється відповідно до ширини максимального елемента. Елементи можна помістити у колекцію Items двома способами.

По-перше, можна явно задати кожен елемент за допомогою тегів ListBoxItem. Наступний приклад описує ListBox з трьома елементами-рядками (рис. 11.10,а):

```
<ListBox Name="lstbx">
    <ListBoxItem>Елемент 1</ListBoxItem>
    <ListBoxItem>Елемент 2</ListBoxItem>
    <ListBoxItem>Елемент 3</ListBoxItem>
</ListBox>
```

Крім цього, можна розмістити елементи безпосередньо в області вмісту (рис. 11.10,б):

```
<ListBox Name="lstbx">
    <TextBlock>Елемент 1</TextBlock>
    <TextBlock>Елемент 2</TextBlock>
    <TextBlock>Елемент 3</TextBlock>
</ListBox>
```



а б

Рис. 11.10. Приклади списку з текстовими елементами (а) та списку з кнопкою (б)

Отримати доступ до виділеного елемента списку можна за допомогою властивості SelectedItem:

```
object obj = lstbxCats.SelectedItem;
string selectedText = (obj == null) ? "Unselected" :
  (string)((ListBoxItem)obj).Content;
```

У цьому прикладі текстовій змінній selectedText присвоюється рядкове значення "Unselected", якщо в списку нічого не вибрано, або значення властивості Content у протилежному випадку.

При кожному виборі нового елемента списку виникає подія SelectionChanged. В оброблювачі цієї події записують код, який має виконуватися при зміні виділення у списку.

Комбінований список: ComboBox

Комбінований список ComboBox, подібно до LictBox, містить колекцію Items, але у звичному режимі відображує тільки один вибраний елемент цієї колекції. Повний список розгортається при натисканні на кнопку зі стрілкою. Ось приклад розмітки комбінованого списку, який містить текстові елементи:

```
<ComboBox Name="cbox" SelectedIndex="0" Width="200">
    <ComboBoxItem>First Item</ComboBoxItem>
    <ComboBoxItem>Second Item</ComboBoxItem>
    <ComboBoxItem>Third Item</ComboBoxItem>
</ComboBox>
```

Приклад простої програми WPF

Розробимо простий застосунок WPF, який розраховуватиме арифметичний вираз. Вікно застосунку показане на рис. 11.11.

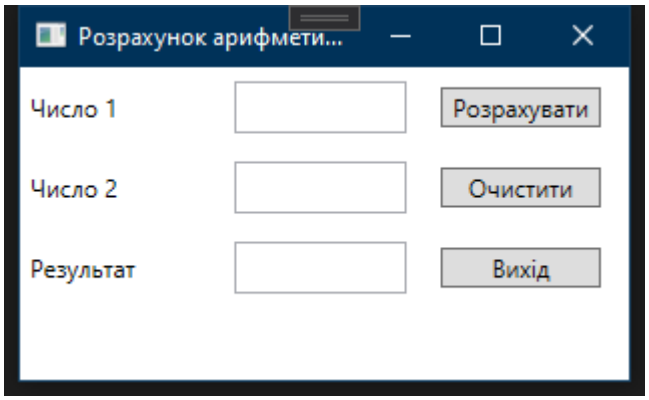


Рис. 11.11. Вікно простого застосунку WPF

Застосунок має два поля ("Число 1" та "Число 2"), у які слід ввести числові значення для змінних *x1* та *x2*. При натисканні на кнопку "Розрахувати" виконується розрахунок арифметичного виразу $y = \sin(x1 \cdot \pi) + \cos(x2/\pi)$. Результат обчислення виводиться у поле "Результат". Очищення текстових полів у вікні здійснюється за натисканням кнопки "Очистити". Для припинення роботи застосунку призначена кнопка "Вихід".

Спочатку створимо новий проект WPF. Для цього слід вибрати команду меню File/New/Project та вибрати шаблон WPF App (.NET Framework) для мови програмування C#.

Розробка інтерфейсу користувача

Головне вікно застосунку описане у файлі MainWindow.xaml. На даний момент розмітка XAML містить один кореневий елемент (тег) Window, який і описує порожнє вікно. Задамо для елемента Window атрибут Title (заголовок вікна), присвоївши йому значення "Розрахунок арифметичного виразу", та атрибут WindowStartupLocation="CenterScreen". Другий атрибут спричиняє показ вікна застосунку після його запуску по центру екрану. Можна задати також більш прийнятний розмір вікна.

На даному етапі код розмітки XAML головного вікна має виглядати як у лістингу 11.3. У коді виділено додані рядки.

Лістинг 11.3. Розмітка вікна на початку

```
1  <Window x:Class="WpfApp1.MainWindow"
2      xmlns=
3    "http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5      xmlns:d=
6          "http://schemas.microsoft.com/expression/blend/2008"
7      xmlns:mc=
8  "http://schemas.openxmlformats.org/markup-compatibility/2006"
9      xmlns:local="clr-namespace:WpfApp1"
10     mc:Ignorable="d"
11     Title="Розрахунок арифметичного виразу"
12     Height="195" Width="320"
13     WindowStartupLocation="CenterScreen">
14
15 </Window>
```

Додані рядки

Всю подальшу розмітку XAML додаватимемо між тегами Window (рядок 14). Спочатку слід продумати макет розмітки. Елементи керування у вікні розмістимо у трьох рядках та трьох стовпчиках (як на рис. 11.11). Перший стовпчик міститиме текстові написи, другий – поля для вводу чисел, а третій – кнопки. Для створення такого макету використаємо панель Grid. Для цього введемо між тегами Window (на місці рядка 14 лістингу 11.3) код розмітки відповідно до лістингу 11.4.

Макет містить три рядки (висота кожного з них не може перевищувати 40 пікселів) та три стовпці (їх ширина не перевищуватиме 100 пікселів).

Лістинг 11.4. Розмітка макету сітки для розміщення в ній елемнтів графічного інтерфейсу

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition MaxHeight="40"></RowDefinition>
        <RowDefinition MaxHeight="40"></RowDefinition>
        <RowDefinition MaxHeight="40"></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition MaxWidth="100"></ColumnDefinition>
        <ColumnDefinition MaxWidth="100"></ColumnDefinition>
        <ColumnDefinition MaxWidth="100"></ColumnDefinition>
    </Grid.ColumnDefinitions>

</Grid>
```

Тепер у макет сітки Grid слід додати елементи графічного інтерфейсу користувача: мітки Label, поля текстового вводу TextBox та кнопки Button. Для цього у місці рядка 12 лістингу 11.4 слід додати XAML-код відповідно до лістингу 11.5. Зауважимо, що код для кожного елемента інтерфейсу зручніше вводити одним рядком (код в лістингу 11.5 розбито на рядки із-за обмежень розміру друкованої сторінки).

Лістинг 11.5. Розмітка XAML елементів графічного інтерфейсу

```
<Label Grid.Row="0" Grid.Column="0"
    VerticalAlignment="Center">
        Число 1
</Label>
<Label Grid.Row="1" Grid.Column="0"
    VerticalAlignment="Center">
        Число 2
</Label>
<Label Grid.Row="2" Grid.Column="0"
    VerticalAlignment="Center">
        Результат
</Label>

<TextBox Name="txtX1" Grid.Row="0" Grid.Column="1"
    Margin="7" Padding="4">
</TextBox>
<TextBox Name="txtX2" Grid.Row="1" Grid.Column="1"
    Margin="7" Padding="4">
</TextBox>
<TextBox Name="txtY" Grid.Row="2" Grid.Column="1"
    Margin="7" Padding="4" IsReadOnly="True">
```

```
</TextBox>

<Button Name="btnCalc" Margin="10"
    Grid.Row="0" Grid.Column="2">
        Розрахувати
</Button>
<Button Name="btnClear" Margin="10"
    Grid.Row="1" Grid.Column="2">
        Очистити
</Button>
<Button Name="btnExit" Margin="10"
    Grid.Row="2" Grid.Column="2">
        Вихід
</Button>
```

Якщо на цьому етапі запустити застосунок до виконання, його вікно виглядатиме як на рис. 11.11, але поки-що ніяких дій для кнопок не запрограмовано.

Програмування функціоналу застосунку

Спочатку запрограмуємо поведінку кнопки "Розрахувати". Щоб створити метод – оброблювач події Click, кнопку потрібно виділити у дизайнері, далі перейти у вкладку "Event handlrs for selected element" вікна Properties і двічі клікнути лівою кнопкою мишки у порожньому полі напроти події Click. Середовище Visual Studio створить порожній метод btnCalc_Click. У цей метод слід внести код відповідно до лістингу 11.6 (зауважимо, що рядки 1, 2 та 15 вводити не потрібно, так як вони вже автоматично згенеровані).

Лістинг 11.6. Код оброблювача події Click кнопки "Розрахувати"

```
1  private void btnCalc_Click(object sender, RoutedEventArgs e)
2  {
3      bool x1Correct =
4          double.TryParse(txtX1.Text, out double x1);
5      bool x2Correct =
6          double.TryParse(txtX1.Text, out double x2);
7
8      double y;
9      if (x1Correct && x2Correct)
10     {
11         y = Sin(x1 * PI) + Cos(x2 / PI);
12         txtY.Text = y.ToString("0.0000");
13     }
14     else txtY.Text = "Невірні дані";
15 }
```

Спочатку за допомогою методу TryParse типу double текстовий рядок з поля txtX1 (або txtX2) перетворюється у число з плаваючою комою, яке зберігається у відповідній змінній (x1 чи x2). Якщо число в рядок введено правильно, метод TryParse поверне значення true, в іншому випадку – false. Результат про успіх перетворення зберігається у змінних x1Correct та x2Correct. Після цього, якщо обидва числа введено коректно (змінні x1Correct та x2Correct дорівнюють true), відбувається розрахунок та вивід результату у текстове поле txtY з форматом "чотири знаки після коми".

Зауважимо, що математичні методи Sin та Cos викликаються без звзнвчення класу, якому вони належать. Тому на початку файлу слід вказати інструкцію

```
using static System.Math;
```

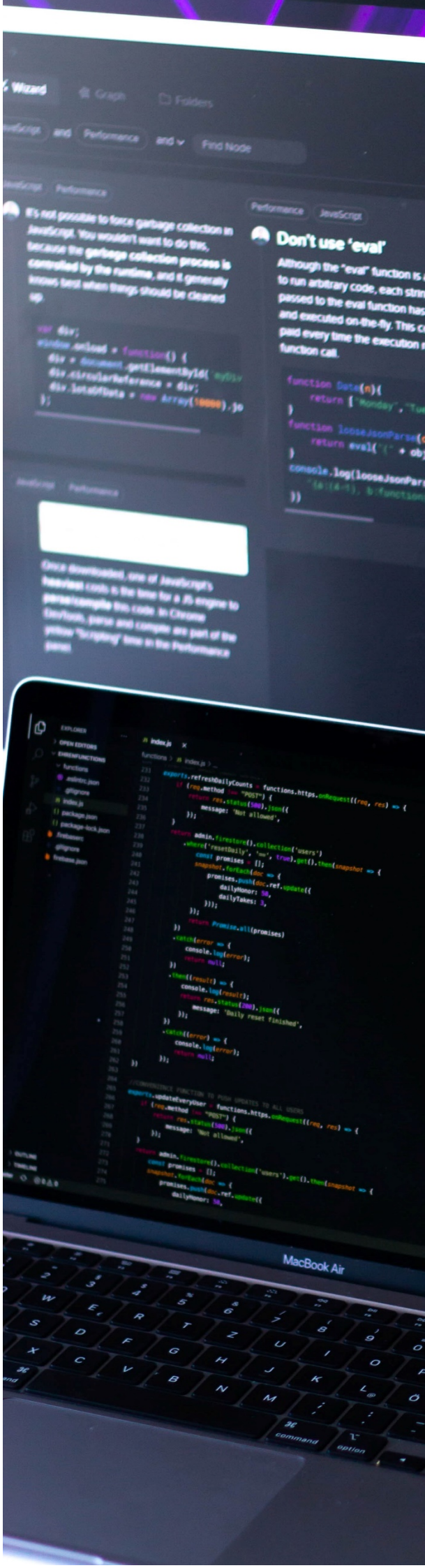
Код оброблювачів події Click для кнопок "Очистити" та "Вихід" приведено в лістингу 11.7. Самі оброблювачі створюють так само, як і для кнопки "Розрахувати": спочатку слід виділити потрібну кнопку в дизайнері, а потім у вікні Properties двічі клікнути напроти події Click.

Лістинг 11.7. Код оброблювачів події Click кнопок "Очистити" та "Вихід"

```
1  private void btnClear_Click(object sender, RoutedEventArgs e)
2  {
3      txtX1.Text = txtX2.Text = txtY.Text = "";
4  }
5
6  private void btnExit_Click(object sender, RoutedEventArgs e)
7  {
8      if (MessageBox.Show("Вийти з програми?",
9          "Справді вийти?",
10         MessageBoxButton.YesNo,
11         MessageBoxImage.Question, MessageBoxResult.No) ==
12         MessageBoxResult.Yes)
13           Application.Current.Shutdown();
14 }
```

Розділ 12. Ввід-вивід

- Потоки вводу-виводу
- Класи потоків
- Консольний ввід-вивід
- Байтовий ввід-вивід у файл
- Символьний ввід-вивід
- Читання та запис двійкових даних
- Довільний доступ до даних файлу
- Потоки вводу-виводу у пам'яті
- Використання потоків для роботи з рядками тексту
- Класи для роботи з файлами та папками
- Перетворення числових рядків при вводі-виводі



Потоки вводу-виводу

Програми C# виконують ввід-вивід за допомогою потоків. *Потік* (*stream*) – це абстракція, яка на загальному рівні описує виникнення або споживання інформації [6]. Потік поєднується з відповідним фізичним пристроєм за допомогою підсистеми вводу-виводу операційної системи. Всі потоки діють однаково, навіть якщо вони з'єднані з різними апаратними пристроями. Тому одні і ті ж класи та методи вводу-виводу можуть бути застосовані до багатьох різних пристроїв.

Байтові та символьні потоки

На найнижчому рівні всі операції вводу-виводу є байтовими, оскільки більшість пристроїв орієнтовані на обмін інформацією за допомогою байтів. Але людині при роботі з текстом зручніше оперувати символами. У C# символ представлений 16-бітним типом char, а байт – 8-бітним типом byte. Тому байтові потоки не дуже зручні для обробки символьно-орієнтованих даних. Для вирішення цього завдання у .NET Framework оголошено кілька класів, які перетворюють байтовий потік у символьний і навпаки.

Попередньо визначені потоки

Існує три попередньо визначені потоки, доступні через властивості Console.In, Console.Out та Console.Error класу Console, оголошеного у просторі імен System. Потік Console.Out представляє стандартний потік виводу (за замовчуванням – консоль). Так, коли ми викликаємо метод Console.WriteLine, відбувається вивід у потік Console.Out. Console.In представляє стандартний потік вводу, яким за замовчуванням є клавіатура. Console.Error представляє стандартний потік для виводу повідомлень про помилки, яким за замовчуванням також є консоль.

Проте, кожен з попередньо визначених потоків може бути перенаправлений на інший сумісний пристрій вводу-виводу. Стандартні потоки є символьними, тому у них записують і з них читають символи.

Класи потоків

У.NET Framework визначено класи як для байтових, так і для символьних потоків. Але насправді класи символьних потоків є лише обгортками для перетворення укладеного в них байтового потоку в потрібний символьний,

автоматично виконуючи перетворення типів даних. Таким чином, символьні потоки ґрунтуються на байтових, хоча вони і розділені логічно.

Основні класи потоків визначено в просторі імен System.IO. Щоб використати їх у програмі, слід додати інструкцію:

```
using System.IO;
```

Простір імен System.IO не вказують для консольного вводу-виводу, бо для нього визначено клас Console у просторі імен System.

Серцевиною механізму для роботи з потоками вводу-виводу є клас Stream. Він представляє байтовий потік і є базовим класом для інших класів потоків. Цей клас є абстрактним, тому неможливо створити екземпляр цього класу. Клас Stream оголошує множину стандартних операцій з потоком. Головні методи класу Stream описано у таблиці 12.1.

Таблиця №12.1. Головні методи класу Stream

Метод	Опис
void Close()	Закриває потік
void Flush()	Скидає вміст потоку, який ще знаходиться в тимчасовому буфері, на фізичний носій
int ReadByte()	Читає наступний байт з потоку і повертає його подання у форматі int. При досягненні кінця файлу повертає -1
int Read(byte[] buffer, int offset, int count)	Читає count байтів з потоку в буфер buffer, починаючи з елемента buffer[offset]
long Seek(long offset, SeekOrigin origin)	Зміщує поточне положення потоку на величину offset відносно заданого положення origin. Повертає нове положення
void WriteByte (byte value)	Виводить байт у потік
void Write(byte[] buffer, int offset, int count)	Виводить count байтів з буфера buffer у потік, починаючи з елемента buffer[offset]

Похідними від класу Stream є кілька класів байтових потоків, призначених для виконання конкретних завдань вводу-виводу. До них, зокрема, належать:

- BufferedStream. Буферизований байтовий потік. Буферизація часто підвищує продуктивність роботи.
- FileStream. Байтовий потік, розроблений для файлового вводу-виводу.
- MemoryStream. Байтовий потік для роботи з пам'яттю.
- UnmanagedMemoryStream. Байтовий потік для роботи з некерованою пам'яттю.

Щоб створити символьний потік, слід байтовий потік "загорнути" в один з класів-оболонок символьних потоків. На вершині ієрархії класів символьних потоків знаходяться абстрактні класи TextReader (для читання) та TextWriter (для

запису). Ці класи успадковано кількома символьно-орієнтованими класами потоків:

- StreamReader. Читає символи з байтового потоку. Клас є обгорткою над байтовим потоком вводу.
- StreamWriter. Записує символи у байтовий потік. Клас є обгорткою над байтовим потоком виводу.
- StringReader. Читає символи з рядка.
- StringWriter. Записує символи у рядок.

Крім класів байтових та символьних потоків, у C# є ще два класи двійкових потоків, які дозволяють працювати безпосередньо з двійковими даними. Це класи BinaryReader (для читання) та BinaryWriter (для запису).

Консольний ввід-вивід

Консольні потоки представлені властивостями Console.In, Console.Out та Console.Error класу Console. Властивість In реалізує клас TextReader, а властивість Out – клас TextWriter. Зразки консольного вводу-виводу ми часто розглядали у попередніх розділах, тому що такі програми дуже прості і добре підходять як навчальні приклади. Але більшість реальних застосунків C# орієнтовані не на консольний ввід-вивід, а на графічний віконний інтерфейс користувача. Тому частина системи вводу-виводу, орієнтована на роботу з консоллю, не має широкого практичного використання.

Для вводу клас Console має три методи: Read, ReadLine та ReadKey.

Щоб прочитати один символ, використовують метод Read:

```
static int Read()
```

Він чекає, доки користувач натисне клавішу, і повертає його ціле подання, яке слід привести до типу char. Якщо станеться помилка, метод поверне значення -1 та згенерує виняток IOException. При використанні методів консольного вводу слід пам'ятати, що вхідні дані буферизуються порядково, тому для завершення вводу користувач має натиснути клавішу Enter. Тільки після цього програма отримає введені дані.

Для зчитування рядка символів використовують метод ReadLine:

```
static string ReadLine()
```

Клас Console має також метод ReadKey, який дозволяє безпосередньо отримати дані про натиснену клавішу клавіатури без порядкової буферизації (тобто, користувач не повинен натискати Enter, щоб програма отримала дані). Метод має дві форми:

```
static ConsoleKeyInfo ReadKey()
```

```
static ConsoleKeyInfo ReadKey(bool intercept)
```

Метод ReadKey без параметрів очікує на натискання клавіші, повертає її дані і відображує відповідний символ на екрані. Друга форма цього методу має параметр intercept. Якщо він дорівнюватиме true, то символ у вікні консолі не відображуватиметься.

Метод повертає дані про натиснену клавішу в структурі типу ConsoleKeyInfo. Її властивість KeyChar типу char містить символьний еквівалент клавіші. Властивість Key містить значення переліку ConsoleKey, який має елементи для всіх клавіш клавіатури. Властивість Modifiers описує використані при натисканні клавіші модифікатори Ctrl, Alt чи Shift. Модифікатори представлено переліком ConsoleModifiers, який має три значення: Control, Shift, та Alt.

Лістинг 12.1. Отримання символів з клавіатури

```
1  static void Main()
2  {
3      ConsoleKeyInfo key;
4      Console.WriteLine("Натисніть клавішу. Для виходу-
5  Alt+x");
6      do
7      {
8          // Читаємо клавішу
9          key = Console.ReadKey(true);
10         // Перевіряємо клавіші-модифікатори
11         if ((ConsoleModifiers.Alt & key.Modifiers) != 0)
12             Console.WriteLine("Натиснено Alt");
13         if ((ConsoleModifiers.Control & key.Modifiers) != 0)
14             Console.WriteLine("Натиснено Control");
15         if ((ConsoleModifiers.Shift & key.Modifiers) != 0)
16             Console.WriteLine("Натиснено Shift");
17         Console.WriteLine("Натиснено клавішу: " +
18             key.KeyChar);
19     } while (!((key.KeyChar == 'x') &&
20                ((ConsoleModifiers.Alt & key.Modifiers) !=
   0)));
   }
```

У лістингу 12.1 подано програму, яка у циклі зчитує натиснені на клавіатурі клавіші та відображує їх на екрані. Дані про натиснену клавішу зберігаються у змінній key (вона оголошена у рядку 3). Цикл виконуватиметься, доки не буде натиснено комбінацію клавіш Alt+x (відповідну умову прописано у рядках 18…19). Приклад виводу програми приведено на рис. 12.1.

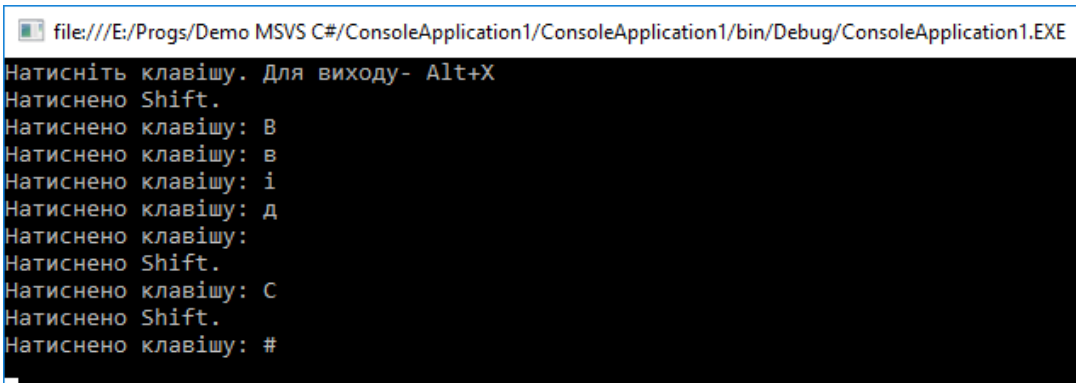


Рис. 12.1. Приклад консольного вводу-виводу програми з лістингу 12.1

Байтовий ввід-вивід у файл

Для створення байтового потоку, прив'язаного до файлу, використовують клас FileStream, який є похідним від класу Stream. Зауважимо, що класи потоків вводу-виводу, у тому числі FileStream, оголошено у просторі імен System.IO.

Щоб створити потік, створюють екземпляр класу FileStream. Один з найвживаніших його конструкторів:

```
FileStream(string path, FileMode mode)
```

Параметр path задає повну назву файлу, а mode – режим доступу до файлу. Режим задають одним із значень, які містить перелік FileMode (див. таблицю 12.2).

Таблиця №12.2. Режими відкривання файлу

Значення	Опис
Append	Додає дані, які виводяться, в кінець існуючого файлу
Create	Створює новий вихідний файл. Існуючий файл з такою ж назвою буде знищено
CreateNew	Створює новий вихідний файл. Файл з таким же ім'ям не повинен існувати
Open	Відкриває існуючий файл
OpenOrCreate	Відкриває файл, якщо він існує. В іншому випадку створює новий файл
Truncate	Відкриває існуючий файл, але скорочує його довжину до нуля

Розглянутий конструктор відкриває файл для читання-запису. Якщо слід обмежити доступ до файлу тільки читанням чи тільки записом, використовують інший конструктор:

```
FileStream(string path, FileMode mode, FileAccess access)
```

На відміну від попереднього, він має параметр access, яким задають спосіб доступу до файлу. Він може приймати одне із значень, визначених переліком FileAccess (таблиця 12.3).

Таблиця №12.3. Способи доступу до файлу

Значення	Опис
Read	Файл відкривають для читання
ReadWrite	Файл відкривають для читання та запису
Write	Файл відкривають для запису

Наприклад, файл text.txt у режимі "тільки для читання" можна відкрити так:

```
FileStream fin = new FileStream("text.txt", FileMode.Open,
    FileAccess.Read);
```

Після закінчення роботи з файлом, його слід закрити. Для цього використовують метод Close:

```
void Close()
```

При закриванні файлу вивільняються системні ресурси, виділені для роботи з файлом.

Клас FileStream містить два методи для читання з файлового потоку вводу-виводу: ReadByte та Read. Щоб прочитати єдиний байт із файлу, використовують метод ReadByte, загальна форма якого така:

```
int ReadByte()
```

При кожному виклику метод читає один байт з файлу і повертає його цілочисельне подання. При досягненні кінця файлу метод повертає -1.

Щоб прочитати блок байтів, використовують метод Read, загальна форма якого:

```
int Read(byte[] array, int offset, int count)
```

Метод намагається прочитати count байтів у масив array, починаючи з байту offset.

При роботі з потоками вводу-виводу можуть виникати помилки. У цьому випадку генерується виняткова ситуація класу IOException.

В лістингу 12.2 приведено фрагмент програми, який побайтово читає вміст текстового файлу text.txt за допомогою потоку вводу-виводу FileStream та виводить вміст цього файлу у консоль [6].

Змінна fin типу FileStream оголошена у рядку 2. Екземпляр класу FileStream створюється у рядку 6. При цьому програма намагається відкрити файл text.txt для читання (вказано режим FileMode.Open, див. табл №12.2). Зауважимо, що вказана операція розташована всередині блоку try, який дозволяє відслідкувати виняткові ситуації. У секції catch (рядки 8…13) перехоплюємо виняток IOException, який описує загальну помилку вводу-виводу. Якщо така помилка трапляється при відкриванні файлу, то вказує на те, що файл не вдалося

відкрити. Тому в секції catch виводимо відповідне повідомлення з описом помилки та припиняємо роботу програми.

Лістинг 12.2. Читання вмісту файлу text.txt у потоці FileStream

```
1  int theByte;
2  FileStream fin;
3
4  try
5  {
6      fin = new FileStream("text.txt", FileMode.Open);
7  }
8  catch (IOException exc)
9  {
10     Console.WriteLine("Не вдалося відкрити файл!");
11     Console.WriteLine(exc.Message);
12     return; // Припиняємо виконання програми
13 }
14 // Читаємо в циклі по одному байту до кінця файлу
15 try
16 {
17     do
18     {
19         theByte = fin.ReadByte();
20         if (theByte != -1) Console.Write((char)theByte);
21     } while (theByte != -1);
22 }
23 catch (IOException exc)
24 {
25     Console.WriteLine("Помилка читання файлу!");
26     Console.WriteLine(exc.Message);
27 }
28 finally
29 {
30     fin.Close();
31 }
```

Коли файл відкрито успішно, то у циклі (рядки 17…21) за допомогою методу ReadByte потоку fin побайтово читаємо вміст файлу. Кожен прочитаний байт зберігається у змінній theByte цілого типу (рядок 19). У рядку 20 перевіряємо, чи байт з файлу було прочитано. Значення -1 змінної theByte вказує на те, що нових даних не отримано, оскільки досягнуто кінець файлу. Далі символ, який відповідає прочитаному байту, виводимо на екран консолі. При цьому прочитане значення приводимо до типу char. Зауважимо: розглянутий приклад передбачає, що файл text.txt записаний у виді звичайного тексту у форматі ANSI.

Цикл для читання даних з файлу розташований всередині іншого блоку try для відслідковування виняткових ситуацій. Якщо станеться виняткова ситуація IOException, то вона буде опрацьована відповідною секцією catch (рядки 23…27). Крім цього, згаданий блок try містить секцію finally (рядки 28…31), за допомогою якої забезпечується гарантоване закривання файлу. Якщо виклик метода Close об'єкта fin розташувати за межами секції finally, то при виникненні винятку файл не буде закрито. Це може спричинити помилки у роботі інших застосунків при звертанні до цього файлу.

Щоб записати байт у файл, використовують метод WriteByte класу FileStream. Найпростіша його форма така:

```
void WriteByte(byte value)
```

Метод записує байт value у файл, асоційований з потоком. У файл можна також записати відразу цілий масив:

```
void Write(byte[] array, int offset, int count)
```

Цей метод записує у потік count байтів з масиву array, починаючи з елемента array[offset].

Слід пам'ятати, що часто при записі у файл дані не відразу записуються на носій, а накопичуються у спеціальному дисковому буфері. Це дозволяє прискорити виконання серії дрібних операцій по запису в файл.

Для того щоб бути певним, що дані насправді записано на диск, слід викликати метод Flush потоку вводу-виводу:

```
void Flush()
```

Після закінчення запису в файл його слід закрити. Для цього використовують вже розглянутий метод Close. При цьому всі файлові дані з системного буфера також переносяться на носій. Тому викликати метод Flush перед закриванням файлу не потрібно.

У лістингу 12.3 приведено фрагмент програми, яка записує у файл test.txt літери англійського алфавіту [6].

Змінну fout, яка представляє файловий потік, оголошено у рядку 1 (ця змінна також відразу ініціалізована значенням null). У рядку 5 відкриваємо файл test.txt. Далі в циклі (рядки 7…8) у файл записуємо великі літери англійського алфавіту. Для відслідковування виняткових ситуацій операції файлового вводу-виводу розташовано у блоці try. Винятки типу IOException обробляє відповідна секція catch. Закриття файлу виконано у секції finally. При цьому перевіряємо, чи змінна fout не дорівнює null. Якщо операція відкриття файлу завершиться помилкою, то змінна fout далі матиме значення null, і закривати файл не потрібно.

Лістинг 12.3. Запис у файл test.txt

```
FileStream fout = null;
try
{
    // Відкриваємо файл
    fout = new FileStream("test.txt", FileMode.CreateNew);
    // Записуємо у файл англійський алфавіт
    for (char c = 'A'; c <= 'Z'; c++)
        fout.WriteByte((byte)c);
}
catch (IOException exc)
{
    Console.WriteLine("Помилка вводу-виводу:\n" +
       exc.Message);
}
finally
{
    if (fout != null) fout.Close();
}
```

Символьний ввід-вивід

Хоча на базовому рівні всі операції вводу-виводу є байтовими, при роботі з текстом зручніше оперувати безпосередньо символами. Перевагою символьних потоків є те, що для них базовою одиницею інформації є символ у форматі Unicode. Тому при вводі-виводі тексту Unicode використання символьних потоків часто є найкращим варіантом. У загальному випадку для виконання символьно-орієнтованих файлових операцій байтовий потік (наприклад, FileStream) потрібно "загорнути" у символьний потік StreamReader чи StreamWriter.

Щоб створити символьний потік виводу, використовують конструктор класу StreamWriter:

```
StreamWriter(Stream stream)
```

Тут stream – назва відкритого байтового потоку. Створений таким чином екземпляр класу StreamWriter автоматично перетворює символи у байти. Після закінчення роботи з потоком StreamWriter, його слід закрити за допомогою методу Close. Закривання потоку приводить до закриття відповідного йому байтового потоку.

У лістингу 12.4 приведено код, який введені з клавіатури рядки записує в текстовий файл у форматі Unicode з назвою test.txt [6]. Текст записується у файл, доки користувач не введе рядок "stop" (рис. 12.2).

Лістинг 12.4. Символьний запис у файл

```
string str;
FileStream fout;
// Відкриваємо потік файлового вводу-виводу
try
{
    fout = new FileStream("test.txt", FileMode.Create);
}
catch (IOException exc)
{
    Console.WriteLine("Помилка відкривання файлу:\n" +
        exc.Message);
    return;
}
//   Загортаємо   потік   файлового   вводу-виводу   у   клас
StreamWriter
StreamWriter fstr_out = new StreamWriter(fout);
try
{
    Console.WriteLine(
        "Введіть текст, або для виходу з програми — 'stop'");
    do
    {
        Console.Write(": ");
        str = Console.ReadLine();
        if (str != "stop")
        {
            str = str + "\r\n"; // Додаємо новий рядок
            fstr_out.Write(str);
        }
    } while (str != "stop");
}
catch (IOException exc)
{
    Console.WriteLine("Помилка вводу-виводу:\n" +
        exc.Message);
}
finally
{
    fstr_out.Close();
}
```

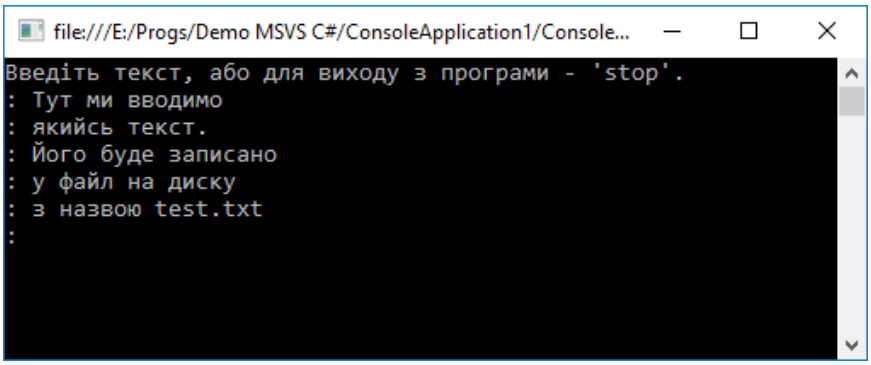


Рис. 12.2. Робота програми для символьного виводу

Інколи при роботі з текстовими файлами зручніше відкривати файл безпосередньо при створенні екземпляра класу StreamWriter, без попереднього створення байтового потоку. Для цього використовують такі конструктори:

```
StreamWriter(string path)
StreamWriter(string path, bool append)
```

Тут path задає шлях до текстового файлу, який слід відкрити. У другому конструкторі, якщо параметр append дорівнює true, то нововведений текст додається в кінець вже наявного тексту. В іншому випадку попередній вміст файлу буде замінено новим текстом.

Щоб створити символьний потік вводу, слід попередньо створений байтовий потік загорнути в екземпляр класу StreamReader. Найчастіше для цього використовують такий конструктор:

```
StreamReader(Stream stream)
```

Тут stream – назва відкритого потоку.

Після закінчення роботи з потоком StreamReader, його слід закрити за допомогою методу Close. Закривання потоку приводить до закриття відповідного йому байтового потоку.

У лістингу 12.5 приведено код, який читає текст з файлу test.txt (створеного у попередньому прикладі), і виводить його на екран [6]. Зауважимо, що індикатором досягнення кінця файлу є те, що метод ReadLine потоку після читання повертає значення null (рядок 16).

Лістинг 12.5. Читання з файлу за допомогою символьного потоку

```
1  FileStream fin;
2  string s;
3  try
4  {
5      fin = new FileStream("test.txt", FileMode.Open);
6  }
```

```
 7  catch (IOException exc)
 8  {
 9      Console.WriteLine("Помилка при відкриванні файлу:\n" +
10          exc.Message);
11      return;
12  }
13  StreamReader fstr_in = new StreamReader(fin);
14  try
15  {
16      while ((s = fstr_in.ReadLine()) != null)
17      {
18          Console.WriteLine(s);
19      }
20  }
21  catch (IOException exc)
22  {
23      Console.WriteLine("Помилка вводу-виводу:\n" +
24          exc.Message);
25  }
26  finally
27  {
28      fstr_in.Close();
29  }
```

Крім цього, клас StreamReader має властивість EndOfStream. Ця властивість логічного типу містить значення true, якщо досягнуто кінець файлу, і false – у протилежному випадку.

Таким чином, перевірити досягнення кінця файлу після читання з нього даних можна і так:

Якщо властивість EndOfStream = true, то досягнуто кінець потоку

```
while(!fstr_in.EndOfStream)   // Поки не кінець потоку (файлу)
{
    s = fstr_in.ReadLine();   // Читаємо рядок
    Console.WriteLine(s);     // Виводимо рядок у консоль
}
```

Символьний потік можна також створити без попереднього створення байтового потоку. Для цього можна використати такий конструктор:

```
StreamReader(string path)
```

Тут path задає шлях до текстового файлу, який слід відкрити.

Читання та запис двійкових даних

Перед цим ми розглянули ввід та вивід байтів і символів, але можна оперувати також іншими типами даних, наприклад, int, double, або short. Щоб читати і записувати дані попередньо визначених типів C#, використовують відповідно класи BinaryReader та BinaryWriter.

BinaryWriter є обгорткою навколо байтового потоку, яка керує записом двійкових даних. Найчастіше для створення екземпляру цього класу використовують конструктор

```
BinaryWriter(Stream output)
```

Тут параметр output задає потік, у який записуватимуться двійкові дані. Цей потік повинен бути відкритим для запису.

Клас BinaryWriter містить ряд методів з назвою Write, які дозволяють записувати дані попередньо визначених C# типів. У ролі параметра методи Write отримують значення одного з типів C# (int, double тощо), яке буде записане у потік. Наприклад, для запису значення типу double використовують такий метод:

```
void Write(double value)
```

Клас BinaryReader є обгорткою над байтовим потоком, який керує вводом двійкових даних. При роботі з ним найчастіше використовують конструктор

```
BinaryReader(Stream input)
```

Тут параметр input є потоком, з якого читаються дані. Потік має бути відкритим для читання. В ролі цього потоку може бути використано потік FileStream.

Для читання даних попередньо визначених типів клас BinaryReader містить методи з назвами типу ReadNNN, де NNN представляє назву відповідного типу .NET. Наприклад, для читання цілого 64-бітного значення використовують метод

```
long ReadInt64()
```

Метод повертає прочитане значення відповідного типу.

У лістингу 12.6 приведено програмний код, який спочатку записує у файл data.bin кілька значень різних типів, а потім читає їх і відображує на екрані. Зауважимо, що для скорочення тексту код не відслідковує виняткові ситуації, але у прикладних застосунках слід обов'язково перехоплювати винятки. Без цього при виникненні помилки вводу-виводу (що трапляється не так вже й рідко) програма просто припинить роботу. Лістинг 12.6 містить коментарі, які пояснюють призначення відповідних ділянок коду. Результат роботи програми приведено на рис. 12.3.

При роботі з бінарними даними слід пам'ятати, що вони мають бути зчитані точно у тому порядку, в якому вони були записані.

Лістинг 12.6. Запис та читання бінарних даних

```
BinaryWriter outStream;
BinaryReader inStream;

// Значення, які записуватимемо у файл
bool b = true;
uint ui = 10;
int i = -100;
long l = 9876543210;
double d = Math.PI;
string s = "Зразок тексту";

// Відкриваємо файл для запису
outStream = new BinaryWriter(new FileStream("data.bin",
    FileMode.Create));

//Записуємо у файл
Console.WriteLine("Запис: " + b);
outStream.Write(b);
Console.WriteLine("Запис: " + ui);
outStream.Write(ui);
Console.WriteLine("Запис: " + i);
outStream.Write(i);
Console.WriteLine("Запис: " + l);
outStream.Write(l);
Console.WriteLine("Запис: " + d);
outStream.Write(d);
Console.WriteLine("Запис: " + s);
outStream.Write(s);

// Закриваємо потік
outStream.Close();

Console.WriteLine("--------------------------");

// Відкриваємо файл для читання
inStream = new BinaryReader(new FileStream("data.bin",
    FileMode.Open));

// Читаємо дані у такому порядку, в якому вони були записані
b = inStream.ReadBoolean();
Console.WriteLine("Reading " + b);
ui = inStream.ReadUInt32();
Console.WriteLine("Reading " + ui);
```

```
44  i = inStream.ReadInt32();
45  Console.WriteLine("Reading " + i);
46  l = inStream.ReadInt64();
47  Console.WriteLine("Reading " + l);
48  d = inStream.ReadDouble();
49  Console.WriteLine("Reading " + d);
50  s = inStream.ReadString();
51  Console.WriteLine("Reading " + s);
52
53  // Закриваємо потік
54  inStream.Close();
```

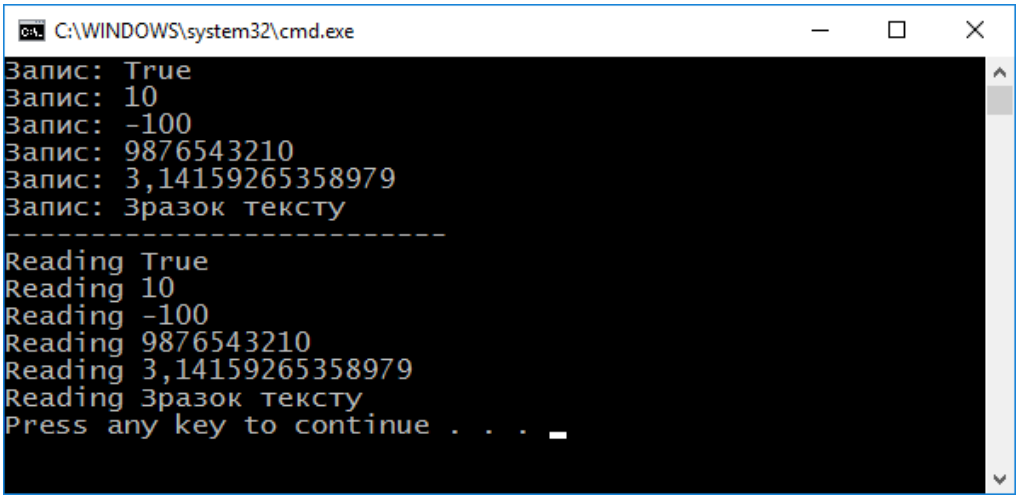


Рис. 12.3. Робота програми для символьного виводу

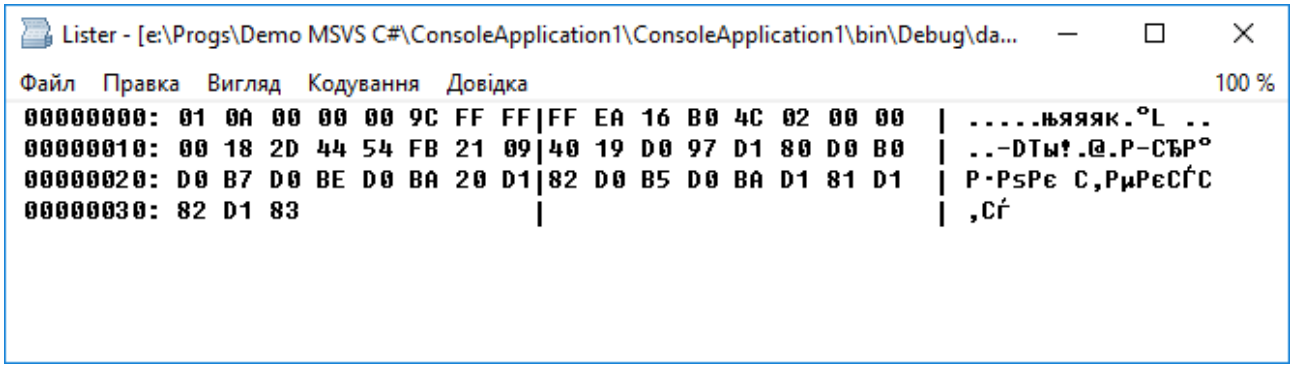


Рис. 12.4. Бінарні дані у файлі

Якщо відкрити файл data.bin, створений розглянутою програмою, то можна переконатися, що дані у ньому зберігаються у двійковому форматі і не придатні для читання людиною (рис. 12.4).

Довільний доступ до даних файлу

У попередніх прикладах було розглянуто *послідовний доступ* до файлу, який полягає у читанні даних в лінійній послідовності. При цьому кожна наступна порція зчитаних даних розміщена у файлі безпосередньо після попередньої.

Під час операцій файлового вводу-виводу використовується *вказівник положення у файлі* (який ще називають *файловим вказівником*). Цей вказівник представляє собою номер байта у файлі. Місце, позначене вказівником, називають *поточною позицією* у файлі. Коли з файлу зчитуються (чи записуються) дані, то вони зчитуються (записуються) починаючи з місця, позначеного цим вказівником. Кожна операція читання (запису) супроводжується переміщенням файлового вказівника на величину, яка відповідає обсягу прочитаних чи записаних даних.

Безпосередньо після відкривання файлу вказівник положення розташований перед першим байтом файлу. Тому операція читання, наприклад, двобайтового значення, приведе до читання перших двох байтів з файлу. Після цього файловий вказівник переміститься і вказуватиме на третій (ще не прочитаний) байт. Під час наступного читання буде отримано байти, починаючи з третього і т.д. Саме це ілюстрував приклад з лістингу 12.6. І саме це є причиною того, що в цьому лістингу дані зчитуються строго у тій послідовності, у якій вони були записані.

Але доступ до даних файлу може бути і довільним. При цьому можна точно задати потрібне положення файлового вказівника. Для цього використовують метод Seek класу FileStream. Загальна його форма така:

```
long Seek(long offset, SeekOrigin origin)
```

Тут параметр offset задає нову позицію файлового вказівника у байтах відносно положення, заданого параметром origin. origin може приймати одне із трьох значень, заданих переліком SeekOrigin (таблиця №12.4).

Метод Seek встановлює і повертає нову позицію файлового вказівника. Після виклику Seek наступна операція читання чи запису відбуватиметься у цій новій позиції. Якщо протягом пошуку заданого положення трапиться помилка, виникне виняткова ситуація (наприклад, коли пов'язаний потік не підтримує позиціювання).

Таблиця №12.4. Елементи переліку SeekOrigin

Значення	Опис
SeekOrigin.Begin	Пошук від початку файлу
SeekOrigin.Current	Пошук від поточної позиції файлового вказівника
SeekOrigin.End	Пошук з кінця файлу

У лістингу 12.7 приведено код, який читає дані з файлу data.bin, створеного після роботи коду в лістингу 12.6. На відміну від попереднього прикладу, тут дані читаємо у довільному порядку.

Лістинг 12.7. Читання з довільної позиції файлу

```
1  // Відкриваємо файл для читання
2  BinaryReader inStream;
3  inStream = new BinaryReader(new FileStream("data.bin",
4      FileMode.Open));
5
6  // Читаємо дані
7  inStream.BaseStream.Seek(1, SeekOrigin.Begin);
8  uint ui = inStream.ReadUInt32();
9  Console.WriteLine("Прочитано uint: " + ui);
10
11
12 inStream.BaseStream.Seek(25, SeekOrigin.Begin);
13 string s = inStream.ReadString();
14 Console.WriteLine("Прочитано string: " + s);
15
16 inStream.BaseStream.Seek(9, SeekOrigin.Begin);
17 long l = inStream.ReadInt64();
18 Console.WriteLine("Прочитано long: " + l);
19
20 inStream.BaseStream.Seek(0, SeekOrigin.Begin);
21 bool b = inStream.ReadBoolean();
22 Console.WriteLine("Прочитано bool: " + b);
23
24 inStream.BaseStream.Seek(17, SeekOrigin.Begin);
25 double d = inStream.ReadDouble();
26 Console.WriteLine("Прочитано double: " + d);
27
28 inStream.BaseStream.Seek(5, SeekOrigin.Begin);
29 long i = inStream.ReadInt32();
30 Console.WriteLine("Прочитано int: " + i);
31
32 // Закриваємо потік
33 inStream.Close();
```

Пропускаємо 1 байт

Від початку файлу

Пропускаємо 25 байт

Щоб правильно прочитати значення, використовуючи точне позиціювання файлового вказівника, слід знати побайтову структуру даних у файлі. Неточність у позиціюванні приведе до неправильно зчитаних даних. Для розуміння коду з лістингу 12.7 розглянемо структуру файлу data.bin з попереднього прикладу (див. лістинг 12.6). У файл дані записано в такій послідовності: bool, uint, int, long, double та string (рядки 17…28 лістингу 11.6).

Щоб прочитати, наприклад, третє значення (int), потрібно перемістити файловий вказівник у позицію, яка дорівнює сумі байтів, зайнятих першими двома значеннями (bool та uint). Звичайно, при цьому слід знати розміри (у байтах) відповідних типів даних. У таблиці 12.5 приведено побайтову структуру даних у файлі data.bin. Таким чином, щоб прочитати третє число типу int, слід пропустити 5 байт (1 + 4), зайнятих попередніми двома значеннями – bool та uint.

Таблиця №12.5. Структура даних у файлі data.bin

Порядок	Тип	Кількість байт	Опис та значення, попередньо записане у файл
1	**bool**	1	Булеве значення (відповідно до лістингу 11.6 – true)
2	**uint**	4	Беззнакове ціле число (10)
3	**int**	4	Знакове ціле число (-100)
4	**long**	8	Довге знакове ціле число (9876543210)
5	**double**	8	Число з плаваючою комою подвійної точності (Math.PI)
6	**string**	Змінна	Рядок тексту ("Зразок тексту")

Розглянемо тепер код у лістингу 12.7. У рядках 3…4 створено потік BinaryReader та відкрито файл, з якого читатимуться дані. Приклад передбачає, що записані дані слід прочитати в такому порядку: uint, string, long, bool, double та int.

У рядках 7…8 програма читає значення другої змінної (типу uint). З цією метою викликом методу Seek пропущено 1 байт, зайнятий попередніми даними (bool). Після цього за допомогою методу ReadUInt32 потоку прочитано потрібне значення uint. Зауважимо: при читанні слід використовувати метод, який прочитає саме ту кількість байт, яку займають відповідні дані (у даному випадку – uint).

У рядках 12…13 читаємо значення останньої змінної типу string. Для цього файловий вказівник встановлено у позицію 25, яка дорівнює кількості байт, зайнятих попередніми даними (1 + 4 + 4 + 8 + 8 = 25, див. таблицю 12.5). Сам рядок прочитано за допомогою методу ReadString потоку.

Дані інших типів з файлу читаємо аналогічно. Кожне прочитане значення виводимо у консоль. На рис. 12.5 приведено результат роботи розглянутої програми. Порівнюючи його з рис. 12.3 бачимо, що всі значення відповідають попередньо записаним.

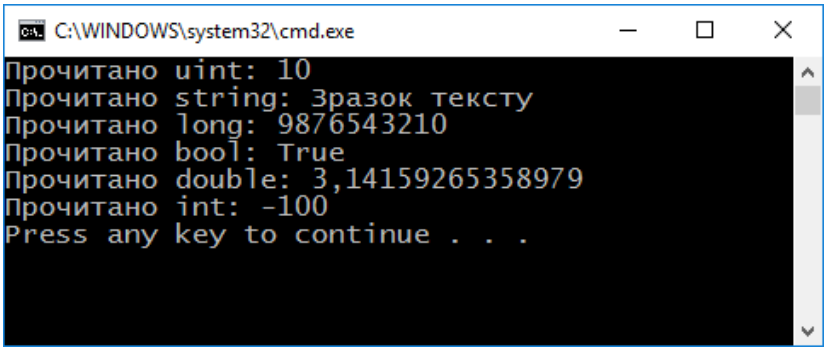


Рис. 12.5. Читання з довільної позиції файлу

Потоки вводу-виводу у пам'яті

Інколи дані зручніше зберігати не у дисковому файлі, а в пам'яті. Для реалізації такого потоку вводу-виводу використовують клас MemoryStream. Він є нащадком класу Stream, який для вводу-виводу використовує масив байтів. Клас MemoryStream має кілька конструкторів. Один з них має такий вигляд:

```
MemoryStream(byte[] buffer)
```

Тут параметр buffer – це масив байтів, який буде використано як джерело/приймач для операцій вводу-виводу. Потік, створений таким конструктором, підтримує читання, запис та позиціювання за допомогою методу Seek. Перед створенням потоку слід бути впевненим, що розмір масиву buffer достатній для зберігання даних, що будуть записані у ньому.

У лістингу 12.8 приведено код, який зберігає дані у потік, що використовує як сховище даних пам'ять. Самі дані представлено кількома змінними (рядки 2…7). Для зберігання даних оголошено масив buffer розміром 200 байт (рядок 9). Спочатку створено потік MemoryStream (рядок 12), а потім його "загорнуто" у потоки BinaryWriter і BinaryReader для запису та читання (рядки 15…16). Дані в потік записують інструкції у рядках 19…24. Виклик методу Flush гарантує, що записані значення будуть перенесені з буфера вводу-виводу у сховище потоку (тобто у масив buffer). Далі викликом методу Seek вказівник потоку встановлюємо на початок (після операцій запису він знаходиться у самому кінці потоку). Після цього щойно записані дані читає потік mrd типу BinaryReader (рядки 33…38). Отримані дані відразу виводяться у вікно консолі (рис. 12.6).

Лістинг 12.8. Використання класу MemoryStream для вводу-виводу у пам'ять

```
// Значення, які записуватимемо в потік
bool b = true;
uint ui = 10;
int i = -100;
long l = 9876543210;
double d = Math.PI;
string s = "Зразок тексту";

byte[] buffer = new byte[200];

// Створюємо потік для вводу-виводу через пам'ять
MemoryStream ms = new MemoryStream(buffer);

// Потоки для запису та читання
BinaryWriter mwr = new BinaryWriter(ms);
BinaryReader mrd = new BinaryReader(ms);

// Записуємо дані у потік
mwr.Write(b);
mwr.Write(ui);
mwr.Write(i);
mwr.Write(l);
mwr.Write(d);
mwr.Write(s);

mwr.Flush();

// Переходимо до початку потоку
ms.Seek(0, SeekOrigin.Begin);

// Читаємо дані з потоку та виводимо їх
Console.WriteLine("Читаємо з потоку у пам'яті");
Console.WriteLine("  Прочитано bool: " + mrd.ReadBoolean());
Console.WriteLine("  Прочитано uint: " + mrd.ReadUInt32());
Console.WriteLine("  Прочитано int: " + mrd.ReadInt32());
Console.WriteLine("  Прочитано long: " + mrd.ReadInt64());
Console.WriteLine("  Прочитано double: " + mrd.ReadDouble());
Console.WriteLine("  Прочитано string: " + mrd.ReadString());

// Закриваємо потоки
mwr.Close();
mrd.Close();
```

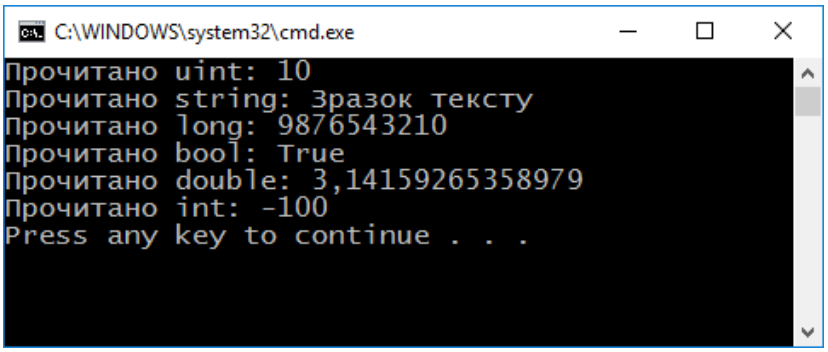


Рис. 12.6. Робота з потоком, що використовує пам'ять як сховище даних

Якщо потік повинен зберігати не бінарні дані, а дані іншого типу, то в ролі "обгорток" для читання та запису використовують інші класи (наприклад, StreamWriter та StreamReader для символьних потоків).

Використання потоків для роботи з рядками тексту

Інколи для зберігання даних зручніше використовувати рядок, а не байтовий масив. Для цього використовують класи StringReader і StringWriter. Клас StringReader є нащадком класу TextReader, а StringWriter – класу TextWriter. Для створення екземпляра класу StringReader використовують конструктор

```
StringReader(string s)
```

Тут параметр s – рядок, з якого потік читатиме дані.

Щоб створити екземпляр класу StringWriter, можна використати такий його конструктор:

```
StringWriter()
```

Він створює потік виводу, який записує дані у рядок типу string. Цей рядок (у вигляді StringBuilder) створюється автоматично. Його можна отримати викликом методу ToString.

Лістинг 12.9 демонструє використання потоків StringReader та StringWriter. Спочатку за допомогою об'єкта StringWriter у потік записано 10 рядків тексту виду "Рядок номер N" (рядки 8…9 лістингу). Далі щойно записаний текст прочитано з використанням StringReader і виведено на екран (рис. 12.7).

Лістинг 12.9. Використання рядкового потоку

```
1  StringWriter strwtr = null;
2  StringReader strrdr = null;
3  try
4  {
5      // Створюємо StringWriter (для запису)
```

```
6       strwtr = new StringWriter();
7       // Записуємо у StringWriter 10 рядків тексту
8       for (int i = 0; i < 10; i++)
9           strwtr.WriteLine("Рядок номер : " + i);
10
11      // Створюємо StringReader (для читання)
12      strrdr = new StringReader(strwtr.ToString());
13
14      string str;
15      // Читаємо з потоку StringReader
16      while ((str = strrdr.ReadLine()) != null)
17          Console.WriteLine(str);
18  }
19  catch (IOException exc)
20  {
21      Console.WriteLine("Помилка        вводу-виводу\n"        +
22  exc.Message);
23  }
24  finally
25  {
26      // Вивільняємо ресурси, зайняті потоками
27      if (strrdr != null) strrdr.Close();
28      if (strwtr != null) strwtr.Close();
    }
```

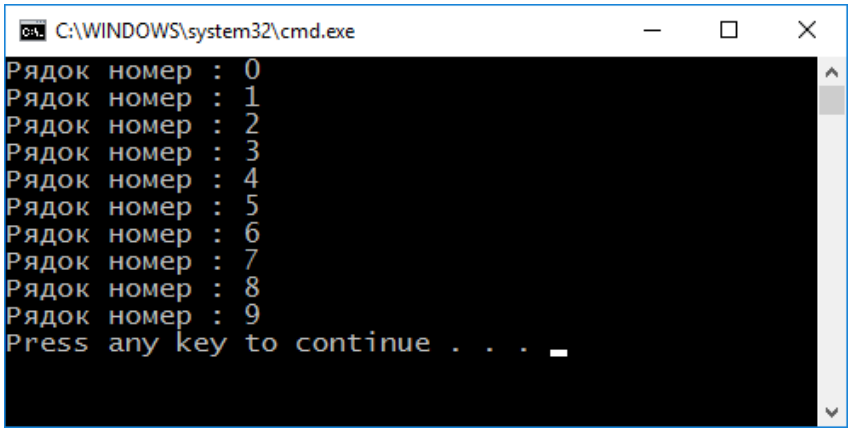


Рис. 12.7. Використання потоків для роботи з рядками тексту

Класи для роботи з файлами та папками

Для виконання типових операцій з файлами використовують клас File, визначений у .NET Framework. Він містить кілька статичних методів, які дозволяють виконувати часто вживані операції: копіювання файлів, їх переміщення, видалення, шифрування тощо. Крім цього, клас File містить

методи для зручного відкриття файлу та отримання посилання на потік FileStream для виконання операцій вводу-виводу.

Для копіювання файлів використовують методи Copy цього класу:

```
static void Copy(string sourceFileName, string destFileName)
static void Copy(string sourceFileName, string destFileName,
    boolean overwrite)
```

Методи Copy копіюють файл, заданий параметром sourceFileName, у файл, заданий параметром destFileName. Перша форма методу копіює файл лише якщо файл destFileName не існує. У другій формі методу, якщо параметр overwrite встановлено у true, то вже існуючий файл destFileName буде переписано.

Для переміщення файлу використовують метод Move:

```
static void Move(string sourceFileName, string destFileName)
```

Він переміщує (перейменовує) файл, заданий параметром sourceFileName, у файл, заданий параметром destFileName.

Метод Delete видаляє файл, заданий параметром path:

```
static void Delete(string path)
```

Метод Exists класу File перевіряє існування заданого параметром path файлу:

```
static bool Exists(string path)
```

Якщо файл існує, метод повертає true, якщо ні – false.

Клас File містить також ряд методів, які дозволяють отримати інформацію про файл. Так, метод GetLastAccessTime повертає час останнього доступу до файлу:

```
static DateTime GetLastAccessTime(string path)
```

Щоб отримати дату та час створення файлу, використовують метод GetCreationTime. Час останньої зміни файлу повертає метод GetLastWriteTime.

За допомогою метода Open можна відкрити потік FileStream для виконання операцій вводу-виводу:

```
static FileStream Open(string path, FileMode mode)
static FileStream Open(string path, FileMode mode,
    FileAccess access)
```

Параметром FileMode задають режим відкривання файлу (див. таблицю 12.2), а параметром FileAccess – спосіб доступу до файлу (див. таблицю 12.3). Метод Open повертає посилання на об'єкт потоку вводу-виводу FileStream, створений при відкриванні файлу.

Крім розглянутих, клас File містить багато інших методів, які дозволяють виконувати операції з файлом.

.NET Framework містить інший клас, який також дозволяє працювати з файлами та виконувати часто вживані файлові операції – FileInfo. Цей клас відрізняється від класу File тим, що він містить не статичні методи, а методи екземпляра. Тому, якщо слід виконувати кілька файлових операцій з тим самим файлом, то клас FileInfo забезпечить кращу ефективність. Клас FileInfo має ряд властивостей, за допомогою яких зручно отримувати інформацію про файл (таблиця 12.6).

Таблиця №12.6. Властивості класу FileInfo

Властивість	Опис
Attributes	Отримує або задає атрибути для файлу
CreationTime	Отримує або задає час створення файлу
DirectoryName	Повертає рядок, який містить повний шлях до папки файлу
Exists	Повертає логічне значення, яке показує, чи існує файл
Extension	Повертає рядок, який містить розширення файлу
FullName	Повертає рядок, який містить повний шлях до файлу
LastAccessTime	Отримує або задає час останнього доступу до файлу
LastWriteTime	Отримує або задає час останньої зміни файлу
Length	Повертає розмір файлу в байтах
Name	Повертає рядок, який містить назву файлу

Платформа .NET Framework містить також класи DirectoryInfo та DriveInfo, за допомогою яких можна отримати дані відповідно про папку та диск.

У лістингу 12.10 приведено код, який виводить у вікно інформацію про файл, а також папку та диск його розташування. Для цього використано класи FileInfo, DirectoryInfo та DriveInfo. Вивід програми показано на рис. 12.8. Змінна path задає повний шлях до файлу, про який буде отримано дані. Екземпляр класу FileInfo створено у рядку 4. При цьому конструктору передано назву файлу через змінну path. У рядках 5…16 через звертання до властивостей об'єкта fi отримано інформацію про файл і виведено її на екран.

Лістинг 12.10. Отримання інформації про файл, папку та диск

```
1  string path = @"e:\Utils\cpu-z_1.78-en\cpuz_x64.exe";
2
3  Console.WriteLine("    Інформація про файл");
4  FileInfo fi = new FileInfo(path);
5  Console.WriteLine("Назва файлу: " + fi.Name);
6  Console.WriteLine("Повний шлях: " + fi.FullName);
7  Console.WriteLine("Папка розташування: " + fi.DirectoryName);
8  Console.WriteLine("Розширення: " + fi.Extension);
9  Console.WriteLine("Розмір у байтах: " + fi.Length);
10 Console.WriteLine("Атрибути: " + fi.Attributes);
```

```
11  Console.WriteLine("Час створення: " +
12      fi.CreationTime.ToString());
13  Console.WriteLine("Час останнього доступу: " +
14      fi.LastAccessTime.ToString());
15  Console.WriteLine("Час останньої зміни файлу: " +
16      fi.LastWriteTime.ToString());
17
18  Console.WriteLine();
19
20  Console.WriteLine("    Інформація про папку");
21  DirectoryInfo di = new DirectoryInfo(fi.DirectoryName);
22  Console.WriteLine("Назва папки: " + di.Name);
23  Console.WriteLine("Повний шлях: " + di.FullName);
24  Console.WriteLine("Батьківський каталог: " + di.Parent);
25  Console.WriteLine("Кореневий елемент: " + di.Root);
26  Console.WriteLine("Час створення: " +
27      di.CreationTime.ToString());
28
29  Console.WriteLine();
30
31  Console.WriteLine("    Інформація про диск");
32  DriveInfo dri = new DriveInfo(di.Root.ToString());
33  Console.WriteLine("Назва диска: " + dri.Name);
34  Console.WriteLine("Мітка тому: " + dri.VolumeLabel);
35  Console.WriteLine("Розмір диска, байт: " + dri.TotalSize);
36  Console.WriteLine("Вільно, байт: " + dri.AvailableFreeSpace);
37  Console.WriteLine("Файлова ситема: " + dri.DriveFormat);
38  Console.WriteLine("Тип диска: " + dri.DriveType);
```

У рядку 21 створено екземпляр класу DirectoryInfo для отримання інформації про папку. При цьому конструктору передано значення властивості fi.DirectoryName попередньо створеного об'єкта з даними про файл, яка містить шлях до файлу, вказаного зміною path. Таким чином, у рядках 22…27 отримано та виведено у консоль інформацію про папку, у якій розташовано файл.

Далі створено екземпляр класу DriveInfo (рядок 32), за допомогою якого отримано дані про диск, на якому розташовано файл. Для цього конструктору передано рядкову інтерпретацію властивості di.Root об'єкта, який зчитував інформацію про папку розташування файлу.

Зауважимо, що розглянутий приклад використовує лише частину можливостей, наданих класами FileInfo, DirectoryInfo та DriveInfo.

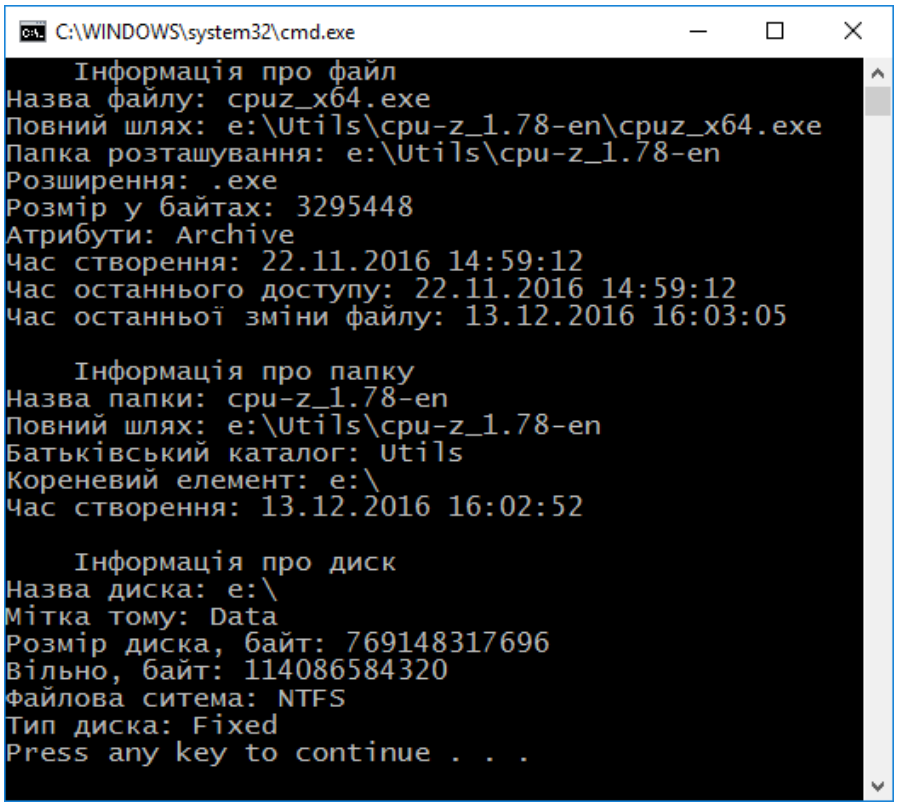


Рис. 12.8. Отримання інформації про файл, папку та диск

Перетворення числових рядків при вводі-виводі

Під час вводу-виводу доводиться часто перетворювати нетекстові дані у текст і навпаки. Наприклад, для використання числа у математичних розрахунках воно має бути подане в одному з числових форматів (int, uint, double тощо). Але якщо дані було прочитано з текстового файлу, то вони будуть у форматі string. Інший приклад: при вводі з консолі за допомогою методу Read отримуємо рядок, навіть якщо користувач вводив число. Якщо програмі потрібно числове подання введеного рядка, його слід перетворити у числовий формат.

Для перетворення числових даних з рядка (string) в один із числових форматів використовують метод Parse. Його мають всі попередньо визначені числові типи C#, такі як int, double та інші. Щоб правильно розуміти структуру типів даних C#, слід пам'ятати, що насправді всі попередньо визначені числові типи є псевдонімами для структур, визначених у .NET Framework. У таблиці 12.7 приведено відповідність назв структур .NET та їх псевдонімів у C# для числових типів даних.

Таблиця №12.7. Відповідність типів .NET Framework та C#

Назва структури .NET Framework	Назва типу C#
Decimal	decimal
Double	double
Single	float
Int16	short
Int32	int
Int64	long
UInt16	ushort
UInt32	uint
UInt64	ulong
Byte	byte
SByte	sbyte

Числові структури оголошено у просторі імен System. Таким чином, повною назвою для Double є System.Double. Ці структури мають широкий набір методів, які дозволяють виконувати найвживаніші операції над даними відповідних типів. Серед них є метод Parse, який перетворює числовий рядок у відповідний числовий тип. Існує кілька перевантажених форм методу Parse. Найпростіша з них містить один параметр – рядок, який потрібно перетворити у число. Інші форми дозволяють задавати додаткові умови перетворення, наприклад, національний формат.

Якщо рядок, що перетворюють у число, містить недопустимі для даного формату символи, метод Parse згенерує виняток FormatException. Якщо рядок містить значення null, виникне виняток ArgumentNullException. Коли значення, задане рядком, перевищує межі типу, виникне виняток OverflowException.

Нижче приведено приклади використання методу Parse для деяких типів:

```
decimal d = Decimal.Parse("3,3");
int i = int.Parse("100");
uint ui = UInt32.Parse("123");
DateTime dt = DateTime.Parse("01.01.2017");
```

У першому та третьому випадку використано назви типів .NET Framework (Decimal та UInt32), а в другому – назву типу C# (int замість Int32). Оскільки типи C# є псевдонімами відповідних типів платформи .NET, так можна робити. Але рекомендується дотримуватися одного підходу. При використанні назв .NET середовище Microsoft Visual Studio пропонує виправити їх на назви C#.

Зауважимо, що при перетворенні розділювач цілої та десяткової частини дійсного числа, а також розділювачі дати і часу мають відповідати діючим

налаштуванням операційної системи. В іншому випадку виникне виняткова ситуація.

Щоб уникнути генерування винятків при перетворенні рядка, використовують метод TryParse, який існує для всіх числових типів. Ось його визначення для типу Int32:

```
static bool TryParse(string s, out int result)
```

Рядок, що містить число, передають параметром s. Результат перетворення – ціле число – отримують з методу через вихідний параметр result. Після успішного перетворення метод поверне значення true. Якщо під час перетворення виникне виняткова ситуація (рядок містить недопустимі символи), метод поверне false. Таким чином, перевіривши результат роботи методу, можна дізнатися, чи перетворення було успішним.

До цього ми розглядали перетворення рядкових даних у відповідний двійковий тип. Для зворотного перетворення використовують метод ToString, який є у всіх попередньо визначених типів C#. Найпростіша форма цього методу не має параметрів і повертає результат перетворення у виді рядка типу string. Існують також інші форми методу ToString, які дозволяють задати формат перетворення (наприклад, для дійсних чисел – кількість знаків після коми).

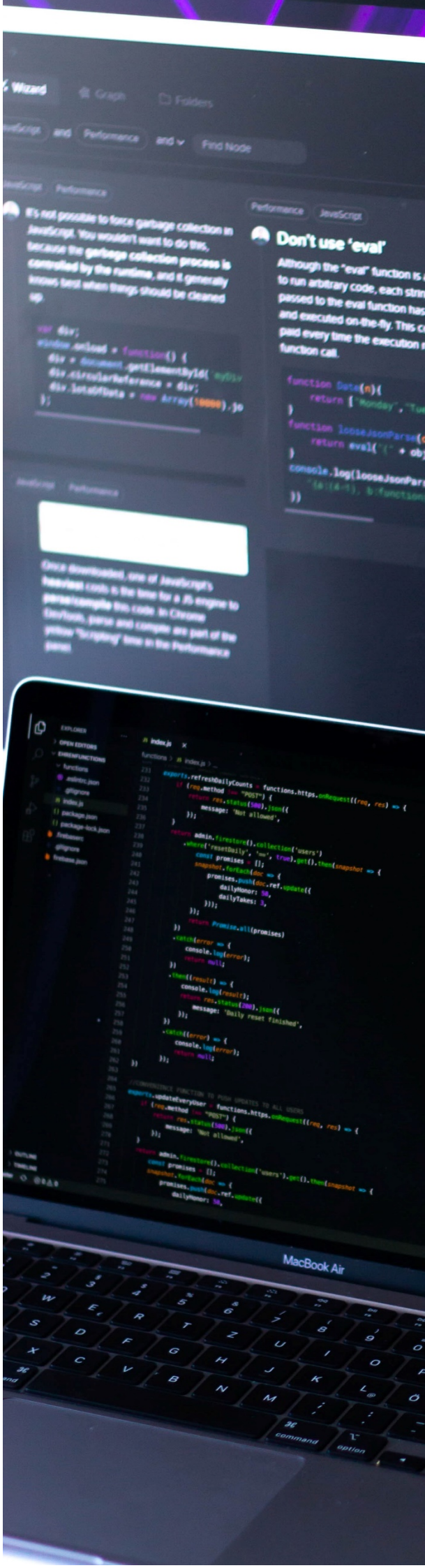
Ось кілька прикладів перетворення нерядкових типів у рядок для кількох змінних з попереднього прикладу:

```
string s1 = d.ToString();
string s2 = d.ToString("0.00");
string s3 = ui.ToString();
string s4 = dt.ToString();
```

У другому випадку задано формат для перетворення змінної типу double.

Розділ 13. Багатопотокове програмування

- Процеси та потоки виконання
- Клас Thread
- Виявлення моменту закінчення роботи потоку
- Передача параметра у потік
- Припинення виконання потоку
- Види потоків та їх пріоритет
- Синхронізація потоків через блокування
- Інші засоби синхронізації



Процеси та потоки виконання

Для реалізації механізму багатозадачності операційна система Microsoft Windows використовує механізм *процесів* та *потоків виконання*. Коли запускається прикладна програма, система створює спеціальний об'єкт – *процес*, – який призначений для підтримки її виконання. Процес є контейнером ресурсів, які надаються системою для виконанні екземпляра програми [8]. Переглянути перелік працюючих процесів можна за допомогою утиліти "Диспетчер завдань" (рис. 13.1). Сам по собі процес не виконує ніяких дій, він лише надає ресурси для їх виконання.

Потік виконання (thread) – це структурний елемент процесу, який містить виконавчий код та отримує процесорний час для його виконання. Потік, на відміну від процесу, може здійснювати якісь дії. Без потоку програма не може виконуватися. Розподіляючи процесорний час між потоками, операційна система забезпечує багатозадачність – одночасне виконання кількох застосунків.

> Зауважимо, що у програмуванні існує кілька різних термінів, які українською мовою зазвичай перекладають як "потік". " У розділі "Ввід-вивід" було розглянуто *потоки вводу-виводу* (оригінальний англійський термін – *stream*), що описували передачу даних (потік даних) від (до) джерела або приймача інформації. Зараз ми розглядаємо *потоки виконання* (вихідний англійський термін – *thread*), які представляють код, що виконується процесором. Термін "потік виконання" українською також інколи перекладають як "нитка" або "тред". У цьому посібнику використано частіше вживаний термін "потік".
>
> Слід пам'ятати, що потоки вводу-виводу (*stream*) та потоки виконання (*thread*) – це зовсім різні речі і спільність українського терміну не означає спорідненості відповідних механізмів. Про який саме вид "потоку" йдеться, в більшості випадків можна зрозуміти з контексту. При вивченні матеріалу цього розділу потрібно пам'ятати, що мова йде про потоки виконання.

У кожного процесу (застосунку) є потік, з якого починається виконання. Його називають *головним потоком*. З головного потоку можна створити інший потік (потоки). Процес, який містить кілька потоків, називають *багатопотоковим*. У цьому випадку він одночасно виконуватиме кілька різних завдань. Багатопотокові програми дозволяють ефективніше використовувати

системні ресурси. Наприклад, поки один потік виконуватиме тривалі розрахунки, інший може реалізовувати повільний мережевий обмін даними.

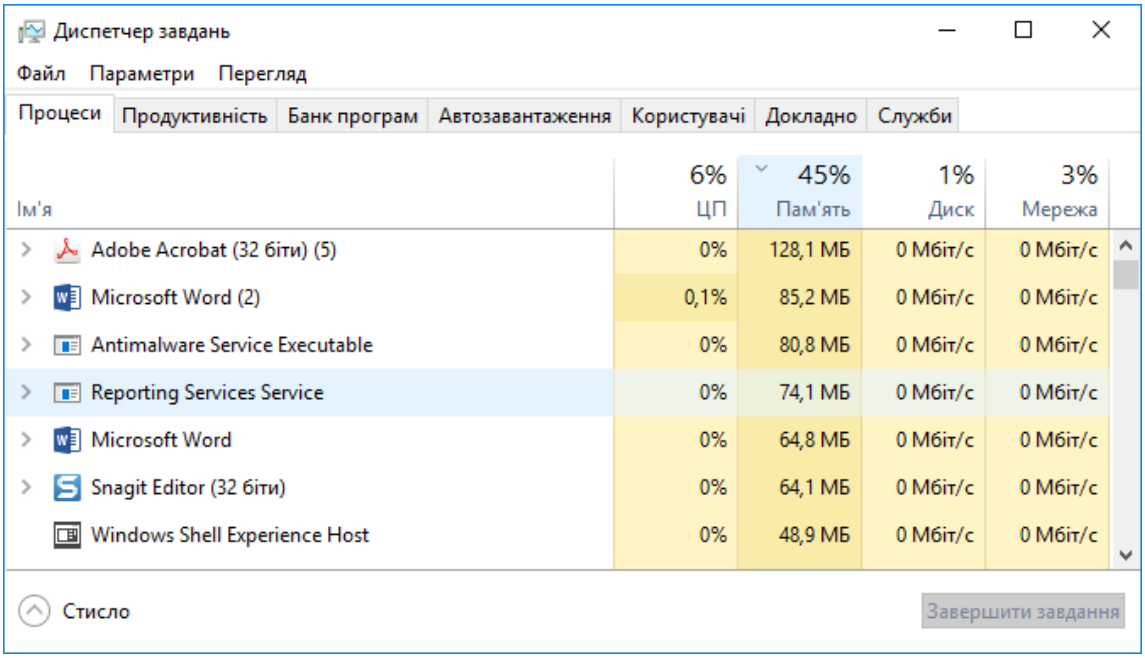


Рис. 13.1. Список процесів у диспетчері завдань Windows

На комп'ютерах із одним одноядерним процесором одночасно може виконуватись лише один потік, тому багатозадачність реалізовується через розподіл процесорного часу між різними програмами. Справжня багатозадачність можлива тільки в багатопроцесорних (багатоядерних) системах, або в розподілених обчислювальних системах.

У .NET Framework розрізняють потоки двох видів: важливі (foreground) та фонові (background). За замовчуванням всі потоки створюються важливими, але їх можна зробити фоновими. Єдина відмінність важливого потоку від фонового у тому, що фоновий потік автоматично припиняє роботу, якщо всі важливі потоки процесу зупинені [6].

Потоки можуть мати різний пріоритет, який впливає на те, скільки часу система виділятиме на виконання потоку. У Windows визначено 7 рівнів пріоритету потоків [7].

Оскільки різні потоки можуть працювати зі спільними даними, то їх роботу часто потрібно узгоджувати між собою. Дії по координації роботи потоків називають *синхронізацією потоків*.

Платформа .NET Framework та мова C# мають зручні засоби для побудови багатопотокових програм. Класи, які підтримують багатопотокове програмування, описано у просторі імен System.Threading.

Клас Thread

Основою системи багатопотокової обробки є клас Thread, який інкапсулює потік виконання. Клас Thread запечатаний (sealed), тому не може бути успадкований іншим класом. У ньому визначено ряд методів і властивостей, призначених для управління потоками.

Щоб створити потік, слід створити екземпляр класу Thread. Найпростіший його конструктор такий:

```
public Thread(ThreadStart start)
```

Тут start – це назва методу, який містить код для потоку, а ThreadStart – прототип цього методу (точніше – делегат), визначений у середовищі .NET Framework. Метод start є точкою входу в потік. Потік починається і закінчується виконанням цього методу, але з нього можна викликати й інші методи. Цей метод називають також *функцією потоку*.

Функція потоку має бути оголошена як void і не повинна мати параметрів. Створення об'єкта для потоку має такий же вид, як і створення будь-якого іншого об'єкта:

Новий потік не почне виконуватися, поки не буде викликано його метод Start. Одна з форм цього метода:

```
public void Start()
```

Потік буде виконуватися, поки не відбудеться повернення з методу, на який вказує параметр start, вказаний при створення потоку. Після припинення цього методу потік автоматично припиняється. Якщо ж спробувати викликати метод Start для потоку, який вже виконується, то виникне виняток ThreadStateException.

Створення і запуск потоку може виглядати так:

```
Thread newThrd = new Thread(threadFunc);
newThrd.Start();
```

Розглянемо приклад програми, яка з головного потоку (він представлений функцією Main) створює додатковий потік. При цьому головний потік в циклі виводить на екран консолі літеру "M", а додатковий потік – цифру "1" (див рис. 13.2). Так як обидва потоки виконуються паралельно, то на екрані символи "M" та "1", які є індикатором виконання того чи іншого потоку, будуть

перемішані. Код програми, яка реалізує таку поведінку, приведено у лістингу 13.1.

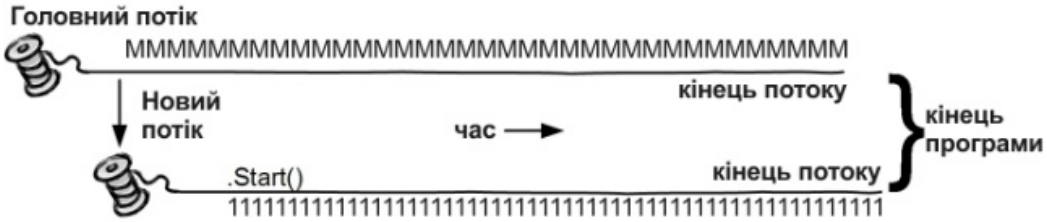


Рис. 13.2. Схематична структура простої двопотокової програми

Лістинг 13.1. Проста двопотокова консольна програма

```
1  static void ThreadFunc()
2  {
3      for (int i = 1; i <= 40; i++)
4      {
5          Console.Write("1");
6          Thread.Sleep(100);
7      }
8  }
9
10 static void Main()
11 {
12     Console.WriteLine("Головний потік (M) виконується...");
13
14     Console.WriteLine("Запускаємо потік (1)...");
15
16     Thread thread1 = new Thread(ThreadFunc);
17     thread1.Start();
18
19     for (int i = 1; i <= 30; i++)
20     {
21         Console.Write("M");
22         Thread.Sleep(50);
23     }
24 }
```

Функція додаткового потоку представлена методом ThreadFunc (рядки 1…8). Він містить цикл з 40 ітерацій, який при кожному проходженні виводить у консоль цифру "1" (потік №1). Щоб трохи "розтягнути" у часі виконання коду потоку, в цикл введено виклик методу Sleep класу Thread, який призупиняє виконання на 100 мілісекунд (тривалість паузи задана параметром).

Головний потік завжди представлено методом Main. Об'єкт додаткового потоку thread1 створено у рядку 16. При цьому у конструктор передано

посилання на метод ThreadFunc, який містить код для потоку. Виконання додаткового потоку розпочато у рядку 17 викликом методу Start його об'єкта thread1.

Після цього головний потік у циклі (рядки 19…23) виводить літеру "M" (Main – головний потік). Оскільки на даний момент додатковий потік вже виконується, то у консоль писатимуть обидва потоки – головний та додатковий. Тому на екрані будуть присутні і літери "M" (вивід головного потоку програми), і цифри "1" (вивід додаткового потоку), що видно з рис. 13.3.

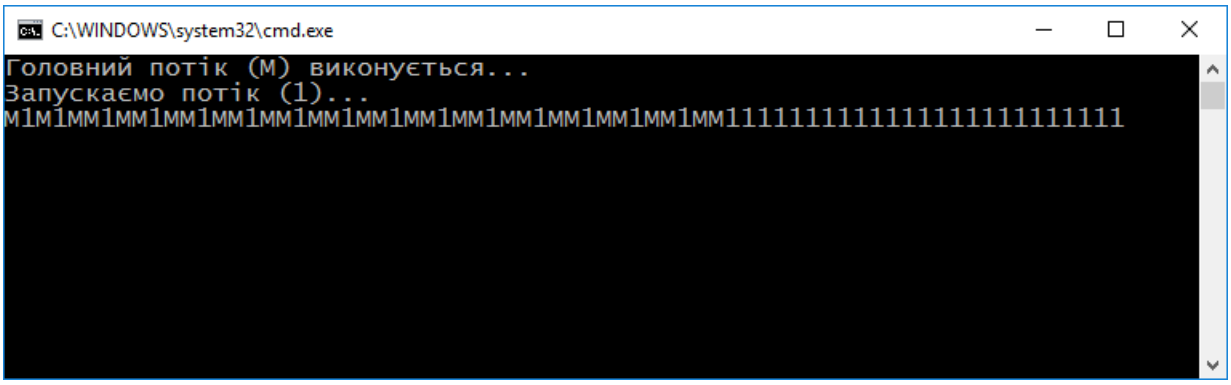


Рис. 13.3. Результат виконання двопотокової програми з лістингу 13.1. Літеру "M" виводить головний потік, а цифру "1" – додатковий

З поданому на рис. 13.3 виводі двох потоків можна помітити такі особливості:

- У отриманому в консолі рядку є ділянка, де вивід обох потоків ("M" та "1") перемішано. Це вказує на одночасну роботу обох потоків.
- На проміжку, де символи "M" та "1" виведено вперемішку, літери "M" трапляються приблизно вдвічі частіше, ніж цифри "1". Так і має бути: часова затримка у циклі головного потоку (50 мс) вдвічі менша за часову затримку в циклі додаткового потоку (100 мс).
- Ділянка, де вивід обох потоків перемішаний, одного моменту закінчується, і до кінця виводяться тільки цифри "1". Це пов'язано з тим, що цикл головного потоку коротший за цикл додаткового (відповідно 30 та 40 ітерацій, та й часова затримка у головному потоці менша).

Звичайно, програма може створювати таку кількість додаткових потоків, яка потрібна для реалізації її алгоритму. Програма у лістингу 13.2 є доповненнням попереднього прикладу і створює два додаткових потоки. Перший потік виводить у консоль цифру "1", а другий – цифру "2". Функції першого та другого потоків – відповідно ThreadFunc1 та ThreadFunc2. Структура та робота програми принципово не відрізняється від раніше розглянутої.

Вивід трьохпотокової програми показано на рис. 13.4.

Лістинг 13.2. Проста трьохпотокова консольна програма

```
1  static void ThreadFunc1()
2  {
3      for (int i = 1; i <= 40; i++)
4      {
5          Console.Write("1");
6          Thread.Sleep(100);
7      }
8  }
9  static void ThreadFunc2()
10 {
11     for (int i = 1; i <= 30; i++)
12     {
13         Console.Write("2");
14         Thread.Sleep(300);
15     }
16 }
17 static void Main()
18 {
19     Console.WriteLine("Головний потік (M) виконується...");
20     Console.WriteLine("Запускаємо потоки (1) та (2)...");
21     Thread thread1 = new Thread(ThreadFunc1);
22     Thread thread2 = new Thread(ThreadFunc2);
23     thread1.Start();
24     thread2.Start();
25     for (int i = 1; i <= 15; i++)
26     {
27         Console.Write("M");
28         Thread.Sleep(50);
29     }
30     Console.ReadKey();
31 }
```

Виявлення моменту закінчення роботи потоку

Часто у програмі необхідно знати, коли новостворений потік закінчує свою роботу. Часове узгодження роботи потоків називають *синхронізацією потоків*. Для синхронізації клас Thread має два засоби: властивість IsAlive та метод Join.

Властивість IsAlive доступна лише для читання і має логічний тип:

```
public bool IsAlive { get; }
```

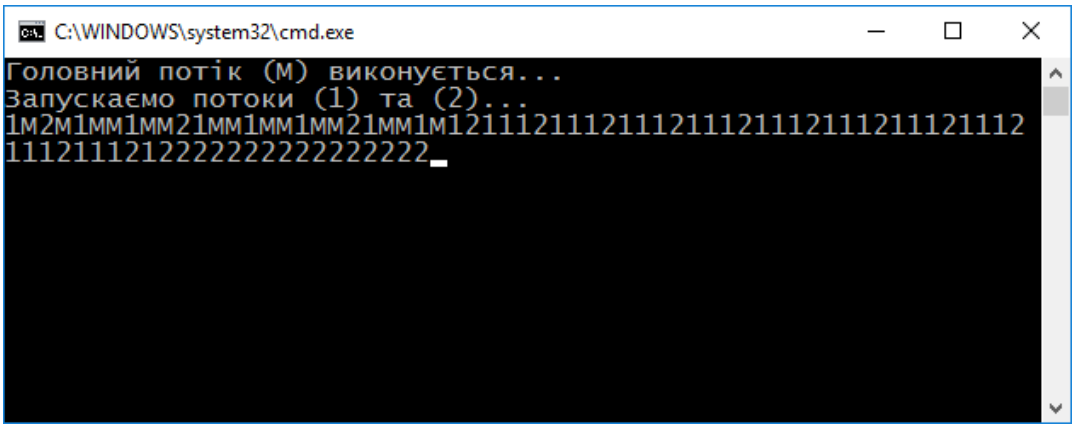


Рис. 13.4. Результат виконання трьохпотокової програми з лістингу 13.2

Властивість IsAlive повертає значення true, якщо її потік все ще працює, і false – якщо він вже завершив роботу. Таким чином, періодично читаючи значення властивості IsAlive, можна пересвідчитися, що потрібний потік вже відпрацьовано.

Метод Join припиняє виконання викликаючого потоку до завершення роботи потоку, представленого екземпляром, для якого викликано метод. Найпростіша форма методу Join:

```
public void Join()
```

Інші форми цього методу дозволяють задати час очікування, і повертають інформацію про те, чи завершено роботу потоку протягом часу очікування.

Щоб продемонструвати важливість синхронізації потоків, розглянемо таке завдання. Програма має дві змінних: A та B. Змінну A розраховує перший потік, а змінну B – другий потік. На основі значень змінних A і B у головному потоці розраховуємо значення третьої змінної – C (C=A+B). Якщо розраховувати значення змінної C до того, як закінчать роботу потоки, то її значення буде неправильним.

У лістингу 13.3 приведено програму, яка намагається виконати це завдання, не очікуючи на завершення потоків, що розраховують змінні A та B. Зауважимо, що несинхронізованість потоків приведе до неправильного (!) значення змінної C.

Розглянемо код лістингу 13.3 детальніше. У класі Program оголошено три поля цілого типу: A, B та C (рядок 3). Метод ThreadFuncA "розраховує" (насправді, імітує розрахунок) змінної A, а метод ThreadFuncB – змінної B. Змінна A "підраховує" кількість проходжень циклу у рядках 7…11 і після його завершення дорівнюватиме 40. Кожна ітерація циклу супроводжується затримкою 100 мілісекунд (рядок 10). Останньою інструкцією потоку є вивід у консоль повідомлення про припинення роботи потоку (рядок 12). Після цього потік закінчує роботу.

Лістинг 13.3. Розрахунок значення змінної C без синхронізації з потоками A і B

```
class Program
{
    static int A, B, C;

    static void ThreadFuncA()
    {
        for (int i = 1; i <= 40; i++)
        {
            A++;
            Thread.Sleep(100);
        }
        Console.WriteLine("\tПотік 1 припинено. A = {0}", A);
    }

    static void ThreadFuncB()
    {
        for (int i = 1; i <= 30; i++)
        {
            B++;
            Thread.Sleep(200);
        }
        Console.WriteLine("\tПотік 2 припинено. B = {0}", B);
    }

    static void Main()
    {
        Console.WriteLine(
            "Головний потік (M) виконується...");
        Console.WriteLine("Запускаємо потоки (1) та (2)...");

        Thread thread1 = new Thread(ThreadFuncA);
        Thread thread2 = new Thread(ThreadFuncB);

        thread1.Start();
        thread2.Start();

        C = A + B;
        Console.WriteLine("A = {0}, B = {1},  C = {2}",
           A, B, C);
    }
}
```

Аналогічно "розраховано" значення змінної B (див. метод ThreadFuncB). Коли потік, який її розраховує, припинить роботу, значення B дорівнюватиме 30. Структура методу ThreadFuncB аналогічна до структури методу ThreadFuncA.

У головному потоці (метод Main) створено та запущено обидва додаткових потоки (рядки 31…35). Відразу після цього розраховано значення змінної C (рядок 37) і всі три змінні виведено у вікно. Якщо виконати цю програму, то можна переконатися, що буде виведено проміжні значення змінних. Більше того, отримані значення будуть непередбачувані і після різних запусків можуть відрізнятися (рис. 13.5). Конкретні отримані значення залежатимуть від швидкодії системи, її завантаженості та інших факторів. Причиною такої поведінки програми є те, що використано значення змінних до закінчення роботи потоків.

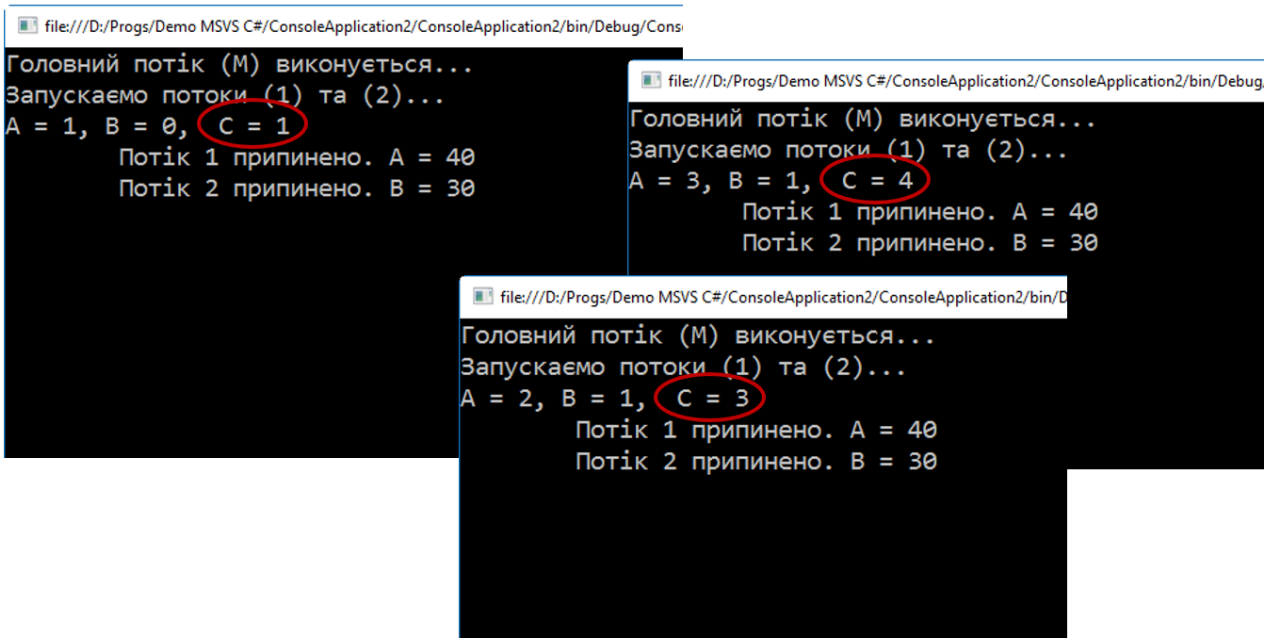


Рис. 13.5. Результат виконання несинхронізованих потоків непередбачуваний

У лістингу 13.4 приведено код методу Main з попереднього прикладу, у який додано два рядки, що забезпечують синхронізацію головного потоку з потоками A та B. Код методу повністю ідентичний попередньо розглянутому, за винятком очікування на закінчення роботи потоків для розрахунку параметрів A та B (рядки 12…13). Тепер результати розрахунку цілком очікувані: змінна A=40, B=30, а C=70 (рис. 12.6).

Лістинг 13.4. Правильний розрахунок значення змінної C

```
1  static void Main()
2  {
3      Console.WriteLine("Головний потік (M) виконується...");
4      Console.WriteLine("Запускаємо потоки (1) та (2)...");
```

```
 5
 6      Thread thread1 = new Thread(ThreadFuncA);
 7      Thread thread2 = new Thread(ThreadFuncB);
 8
 9      thread1.Start();
10      thread2.Start();
11
12      thread1.Join();
13      thread2.Join();
14
15      C = A + B;
16      Console.WriteLine("A = {0}, B = {1},  C = {2}", A, B, C);
17  }
```

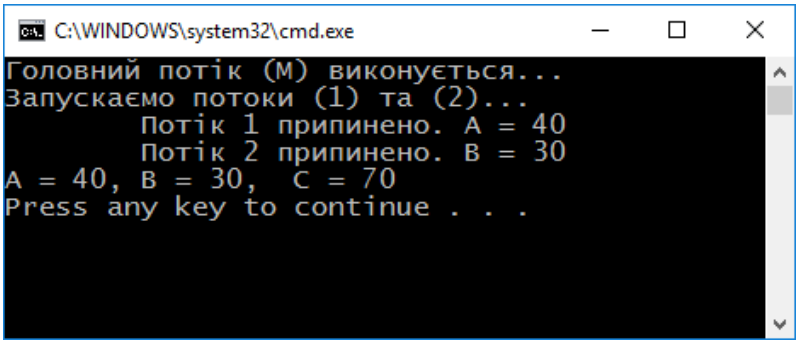


Рис. 13.6. Результат виконання синхронізованих потоків

Для очікування на завершення роботи потоків A та B можна також використати властивість IsAlive. Для цього у код з лістингу 13.4 замість рядків 12…13 можна записати такий код очікування:

```
while (thread1.IsAlive || thread2.IsAlive)
{
    Thread.Sleep(50);
}
```

Програма у циклі буде перевіряти значення властивостей IsAlive обох потоків. Цикл закінчить свою роботу тоді, коли припинять роботу обидва потоки (обидві властивості дорівнюватимуть false).

Передача параметра у потік

Потоку при його запуску можна передати параметр з даними. Для цього слід використовувати такий варіант метода Start:

```
public void Start(object param)
```

Об'єкт param буде передано у функцію потоку. Таким чином, щоб передати у потік параметр, слід передати його методу Start.

Відповідно, і метод для функції потоку повинен мати один параметр типу object, який представлятиме дані для передачі у потік.

Наприклад, функція потоку може мати такий вигляд:

```
void Run(object num)
{
    ...
    if ((int)num > 10) ...
}
```

Тоді створення і запуск потоку з одночасною передачею йому певного значення виглядатиме так:

```
Thrd = new Thread(Run);
Thrd.Start(num);
```

Припинення виконання потоку

Для переривання потоку до його нормального завершення служить метод Abort класу Thread. Найпростіша форма цього методу така:

```
public void Abort()
```

Метод Abort не завжди зупиняє потік негайно, тому, якщо потік потрібно зупинити перед тим, як продовжити виконання програми, то після методу Abort слід відразу ж викликати метод Join.

Види потоків та їх пріоритет

В середовищі .NET Framework є два різновиди потоків: важливі (foreground) та фонові (background.). Програма не завершиться, поки не закінчать роботу її важливі потоки, тоді як фонові потоки завершуються автоматично після закінчення всіх важливих.

За замовчуванням новостворений потік стає важливим. Але його можна зробити фоновим, використовуючи логічну властивість IsBackground. Щоб зробити потік фоновим, досить присвоїти значення true цій властивості. Логічне значення false вказує на те, що потік є важливим.

У кожного потоку є свій пріоритет, який задає, наскільки часто потік отримуватиме доступ до мікропроцесора. Загалом, низькопріоритетні потоки отримують доступ до процесора рідше, ніж високопріоритетні. Таким чином,

протягом певного проміжку часу низькопріоритетному потоку буде доступно менше процесорних ресурсів, ніж високопріоритетному.

Крім пріоритету, на частоту доступу потоку до процесора впливають і інші фактори. Так, якщо високопріоритетний потік очікує на доступ до певного ресурсу (наприклад, для введення з клавіатури), він призупиняє роботу, а замість нього виконується низькопріоритетний потік. У подібній ситуації низькопріоритетний потік може отримати більше процесорних ресурсів, ніж високопріоритетний. Крім цього, планування завдань на рівні операційної системи також впливає на обсяг процесорного часу, який виділяється для потоку.

Коли потік створено, він за замовчуванням має звичайний пріоритет. В подальшому пріоритет потоку можна змінити, задавши відповідне значення властивості Priority (вона є елементом класу Thread):

```
public ThreadPriority Priority{ get; set; }
```

Тут ThreadPriority позначає перелік, який містить значення пріоритетів потоку (таблиця 13.1). Відразу після створення потік має значення за замовчуванням – ThreadPriority.Normal.

Таблиця №13.1. Значення пріоритету потоку

Пріоритет	Опис
ThreadPriority.Highest	Найвищий пріоритет
ThreadPriority.AboveNormal	Пріоритет вищий за звичайний
ThreadPriority.Normal	Звичайний пріоритет (значення за замовчуванням)
ThreadPriority.BelowNormal	Пріоритет нижчий за звичайний
ThreadPriority.Lowest	Найнижчий пріоритет

Задати пріоритет потоку у програмі можна, наприклад, так:

```
thread1.Priority = ThreadPriority.Highest;
thread2.Priority = ThreadPriority.Lowest;
```

де thread1 та thread2 – екземпляри класу Thread.

Багатопотоковий код може поводитися по-різному на різному обладнанні, тому ніколи не слід покладатися на результати його виконання тільки в одній системі. Так, було б помилкою вважати, що низькопріоритетний потік завжди буде виконуватися лише протягом невеликого періоду часу до тих пір, поки не завершиться високопріоритетний потік. Інколи високопріоритетний потік може, наприклад, завершитися ще до того, як низькопріоритетний потік почне виконуватися. В деяких випадках високопріоритетний потік може виконуватися навіть довше, ніж низькопріоритетний (навіть якщо вони виконують одинакові за обсягом завдання).

Розглянемо приклад, який ілюструє вплив пріоритетів потоків на час їх виконання (лістинг 13.5). Ідея програми така: є дві властивості – A та B (рядки 3,4) і два потоки, представлені методами ThreadFuncA та ThreadFuncB (рядки 7…13 та 15…21). Перший потік матиме високий пріоритет, а другий – низький. Кожен з потоків у циклі неперервно збільшуватиме "свою" властивість (A чи B) на 1. Цикли обох потоків припинять виконання, коли властивість Stop матиме значення true. Функції обох потоків однакові, за винятком змінної-лічильника: перший потік збільшує значення змінної A, а другий – змінної B. Потоки виконуватимуться паралельно протягом одного проміжку часу – 5 секунд. Таким чином, значення змінної буде індикатором часу, виділеного для потоку. Чим більше значення матиме змінна, тим більше її потоку було виділено часу.

Лістинг 13.5. Дослідження впливу пріоритету потоку на час його виконання

```
1  class Program
2  {
3      static long A { get; set; }
4      static long B { get; set; }
5      static bool Stop { get; set; }
6
7      static void ThreadFuncA()
8      {
9          while (!Stop)
10         {
11             A++;
12         }
13     }
14
15     static void ThreadFuncB()
16     {
17         while (!Stop)
18         {
19             B++;
20         }
21     }
22
23     static void Main()
24     {
25         StartAgain:
26
27         A = B = 0;
28         Console.WriteLine();
29         Console.WriteLine("Запускаємо потоки...");
30
31         Stop = false;
```

```
32
33          Thread threadA = new Thread(ThreadFuncA);
34          Thread threadB = new Thread(ThreadFuncB);
35
36          threadA.Priority = ThreadPriority.Highest;
37          threadB.Priority = ThreadPriority.Lowest;
38
39          threadA.Start();
40          threadB.Start();
41
42          Thread.Sleep(5000);
43
44          Stop = true;
45          threadA.Join();
46          threadB.Join();
47
48          Console.WriteLine("A = {0:0,0}, B = {1:0,0}", A, B);
49          Console.Write("Знову? (y - так) : ");
50          if (Console.ReadKey().Key == ConsoleKey.Y)
51              goto StartAgain;
52      }
53  }
```

Розглянемо метод Main. Спочатку, перед створенням потоків, властивостям-лічильникам присвоєно значення 0 (рядок 27), а властивості Stop – значення false (рядок 31). Властивість Stop є прапорцем, встановлення якого у true припинить роботу потоків.

Далі створено два потоки (рядки 33,34). Об'єкт першого потоку має назву threadA, а другого – threadB. Після цього першому потоку присвоєно найвищий пріоритет Highest, а другому – найнижчий пріоритет Lowest (рядки 36,37). Обидва потоки відразу запущено до виконання, а головний потік призупинено на 5000 мілісекунд викликом метода Thread.Sleep (рядок 42). Протягом цього часу головний потік не виконуватиметься, але два додаткових потоки продовжуватимуть роботу, постійно відпрацьовуючи свої цикли та збільшуючи значення властивостей-лічильників.

По закінченню відведеного часу очікування властивості Stop присвоєно значення true (рядок 44). Після цього потоки припинять роботу, оскільки цикли у методах ThreadFuncA та ThreadFuncB перестануть працювати. Але наступна інструкція головного потоку може почати виконання ще до того, як обидва потоки зупиняться. Тому викликом методу Join відповідного екземпляра чекаємо на припинення роботи потоків (рядки 45,46). Після цього виводимо на екран значення змінних-лічильників A та B (рядок 48).

Нагадаємо: більше значення змінної означатиме, що потік, який з нею працював, отримав більше процесорного часу. Оскільки перший потік має високий пріоритет, а другий – низький, то логічно очікувати, що значення властивості A буде помітно більшим за значення властивості B.

Програму було протестовано на двох комп'ютерах різної конфігурації. Для "старої" та повільної системи на основі одноядерного CPU Athlon 64 з частотою 2.0 ГГц (1 Гб RAM) результат очікуваний: "обраховане" значення змінної A постійно суттєво вище за B (рис. 13.7). Зі скріншоту видно, що приблизно 98% процесорного часу надавалося першому потоку (з високим пріоритетом).

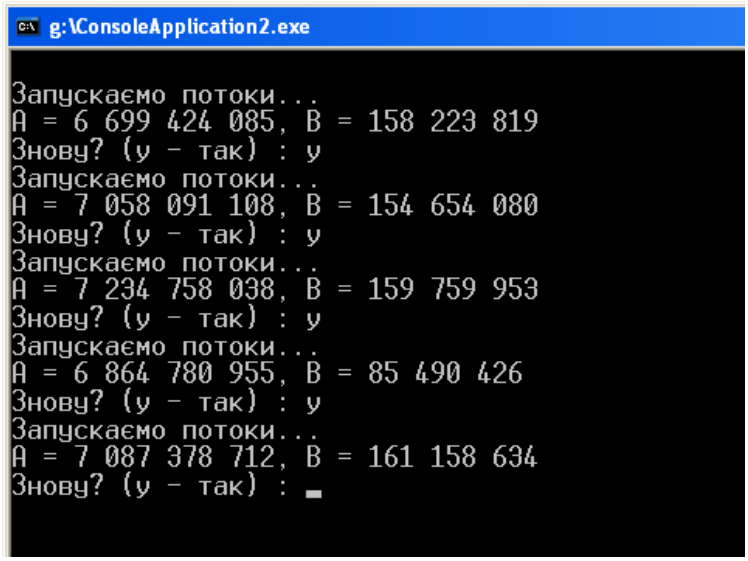


Рис. 13.7. Перший потік виконувався значно більше часу, ніж другий

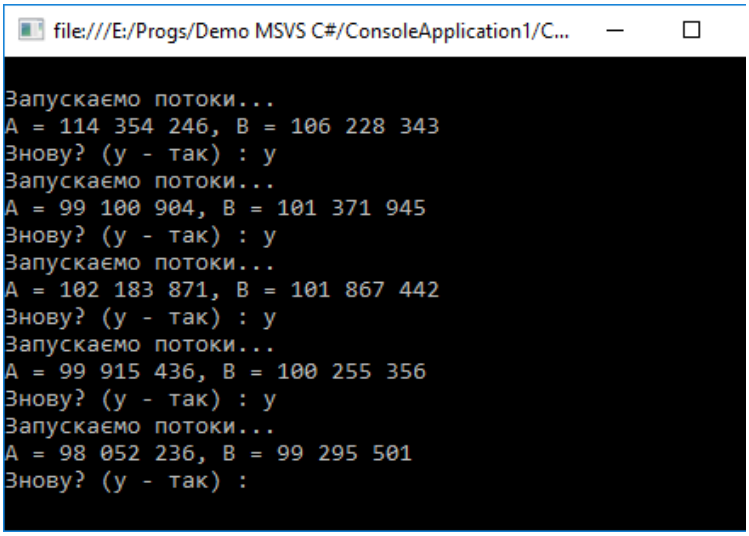


Рис. 13.8. Обидва потоки виконувалися приблизно однакові проміжки часу

Але для "нової" багатоядерної системи на основі CPU Intel Core i7 (4 ядра, 8 потоків виконання) з частотою ядра 2.4 ГГц (8 Гб RAM) результат дещо інший: A приблизно дорівнює B, до того ж, A може бути навіть трохи менше за B (рис. 13.8).

Це пояснюється тим, що в багатоядерній (або багатопроцесорній) системі обидва потоки виконуються одночасно на двох ядрах протягом однакового проміжку часу. Тобто, при наявності потоку з низьким пріоритетом, система спеціально не "гальмує" його виконання. Пріоритет є суттєвим при розподілі обмежених системних ресурсів: якщо їх не вистачає на два потоки, то в першу чергу вони надаються потоку з більшим пріоритетом. Якщо ж системних ресурсів достатньо, обидва потоки отримуватимуть їх у повній мірі.

До цього увагу було зосереджено на додаткових потоках, але з головним потоком можна виконувати ті ж дії, що і з усіма іншими потоками. Для доступу до головного потоку необхідно отримати об'єкт типу Thread, який посилається на головний потік. Це роблять за допомогою властивості CurrentThread, що є елементом класу Thread. Наприклад:

```
Thread Thrd = Thread.CurrentThread;
```

Синхронізація потоків через блокування

Вище було наголошено, що при роботі кількох потоків виникає завдання їх синхронізації. Розглянутий випадок очікування на завершення роботи потоку є лише одним з випадків синхронізації. Часто потрібно синхронізувати потоки, які виконуються і повинні далі виконуватися.

Найпоширенішою причиною синхронізації служить необхідність розподіляти серед кількох потоків загальний ресурс, який одночасно може бути доступний тільки одному потоку. Наприклад, якщо один потік записує дані у змінну чи файл, то другому потоку має бути заборонено робити це в той же самий момент часу. Синхронізація необхідна і тоді, коли один потік очікує на подію, яку спричиняє інший потік. У подібних ситуаціях потрібні засоби, що дозволяють призупинити один з потоків, поки не відбудеться подія в іншому потоці. Після цього очікуючий потік може відновити своє виконання.

Якщо відсутня синхронізація роботи потоків, які використовують спільні ресурси, то дані, сформовані цими потоками, можуть не відповідати логіці ні одного з потоків, що працював із ними. Порушення логіки алгоритму програми, викликане неузгодженістю роботи потоків, називають *конфліктом потоків*.

При синхронізації ключовою концепцією є *блокування* потоком виконання ділянки коду [6]. Коли така ділянка коду (чи об'єкт) заблокована потоком, ніякий інший потік не може отримати доступ до заблокованого коду. Цей код буде доступний іншим потокам тільки після його розблокування потоком, який його блокував.

Щоб позначити ділянку коду, яка має бути заблокована, коли потік досягне її, використовують ключове слово lock. В загальному блокування ділянки коду здійснюють так:

```
lock(lockObj) {
    // Інструкці, які потрібно синхронізувати
}
```

Тут lockObj – посилання на об'єкт синхронізації. Він використовується для позначення ділянки коду, виконання якої необхідно синхронізувати з іншими потоками.

Інструкція lock гарантує, що фрагмент коду, захищений блокуванням, буде використано тільки в потоці, який ініціює блокування. Всі інші потоки, які також хочуть отримати доступ до даного фрагмента коду, очікуватимуть, поки блокування не буде зняте. Блокування автоматично знімається по завершенню захищеного ним фрагмента коду.

Лістинг 13.6. Синхронізація потоків через блокування ділянки коду

```
1  class Program
2  {
3      static SumArray sa;
4      static int answer;
5
6      class SumArray
7      {
8          int sum;
9          object lockOn = new object();
10         public int SumIt(int[] nums)
11         {
12             lock (lockOn)
13             {
14                 sum = 0;
15                 for (int i = 0; i < nums.Length; i++)
16                 {
17                     sum += nums[i];
18                     Console.WriteLine(
19                         Thread.CurrentThread.Name +
20                         " sum = " + sum);
21                     Thread.Sleep(100);
22                 }
23                 return sum;
24             }
25         }
26     }
27
28     static void ThreadFunc(Object num)
29     {
```

Об'єкт синхронізації

Початок ділянки коду, захищеної блокуванням

```
30          Console.WriteLine(Thread.CurrentThread.Name +
31             " стартував.");
32
33          answer = sa.SumIt((int[])num);
34
35          Console.WriteLine("Сума для '" +
36              Thread.CurrentThread.Name + "' : " + answer);
37          Console.WriteLine(Thread.CurrentThread.Name +
38              " закінчено.");
39      }
40
41      static void Main()
42      {
43          sa = new SumArray();
44
45          int[] a = { 1, 2, 3, 4, 5 };
46
47          Thread thread1 = new Thread(ThreadFunc);
48          thread1.Name = "Потік 1";
49          Thread thread2 = new Thread(ThreadFunc);
50          thread2.Name = "Потік 2";
51          thread1.Start(a);
52          thread2.Start(a);
53          thread1.Join();
54          thread2.Join();
55      }
56  }
```

Розглянемо приклад, у якому два різних потоки обчислюють суму масиву і записують результат в одну і ту ж змінну. У лістингу 13.6 описано клас SumArray, який містить засоби для підрахунку суми значень масиву [6]. Поле sum (рядок 8) зберігає проміжні і кінцеве значення суми елементів масиву. Метод SumIt (рядки 10…25) розраховує суму елементів масиву, який передається йому через параметр. Для наочності на екран виведено проміжні значення суми і вказано, у якому потоці воно розраховане (рядки 18…20). Щоб примусити програму перемикатися на інший потік, у цикл розрахунку суми введено метод Sleep (рядок 21), який призупиняє виконання потоку і спонукає систему перемкнутися на виконання іншого потоку. Зауважимо, що для розрахунку суми виклик методу Sleep не потрібний.

Поле sa класу Program представляє екземпляр класу SumArray, який використовуватиметься у програмі. Сам об'єкт створено на початку виконання методу Main (рядок 43). Масив a, сума якого розраховуватиметься, оголошений (і відразу ініціалізований значеннями) у рядку 45. Масив має такий вигляд:

1	2	3	4	5

Сума елементів масиву: 15

Сума масиву обчислюватиметься паралельно у двох потоках, які представлені функцією потоку ThreadFunc (рядки 28…39). Потік спочатку виводить повідомлення про початок роботи (рядки 30…31), потім викликає вже розглянутий метод SumIt (рядок 33), і в кінці виводить результат розрахунку суми (рядки 35…36). Саме при виконанні методу SumIt різними потоками виникатиме конфлікт потоків, бо цей метод використовує спільне для різних потоків поле sum класу SumArray.

Потоки, які розраховуватимуть суму масиву, створено у рядках 47,49. При цьому як функцію потоку вказано метод ThreadFunc. Для зручності розпізнавання дій, виконаних тим чи іншим потоком, їм дано назви. З цією метою властивостям Name екземплярам потоків присвоєно відповідні рядкові ідентифікатори (рядки 48, 50). Далі програма запускає потоки і очікує на припинення їх роботи (рядки 51…54).

Щоб наочніше продемонструвати наслідки конфлікту несинхронізованих потоків, спочатку у розглянутій програмі "відключимо" синхронізацію. Для цього закоментуємо рядок 12 (поставивши на його початку символи //).

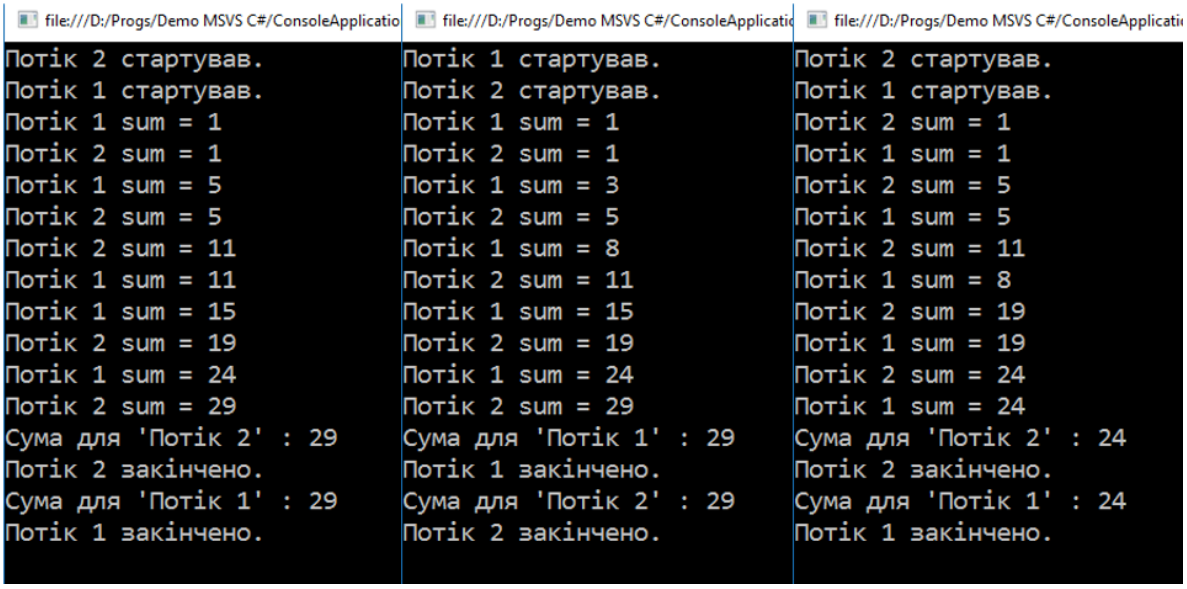


Рис. 13.9. Результат конфлікту потоків: неправильно розрахована сума елементів масиву

Після різних запусків програми несинхронізовані потоки дають різні і неправильні результати підрахунку суми одного і того ж масиву. На рис. 13.9 показано результати для трьох випадків. Серед розрахованих значень – 29, 24, але немає правильної суми (15). Послідовність відпрацювання потоків також різна.

Тепер розкоментуємо рядок 12, який задіює механізм синхронізації через блокування ділянки коду, і ще раз запустимо програму. Тепер результат завжди буде правильним (рис. 13.10).

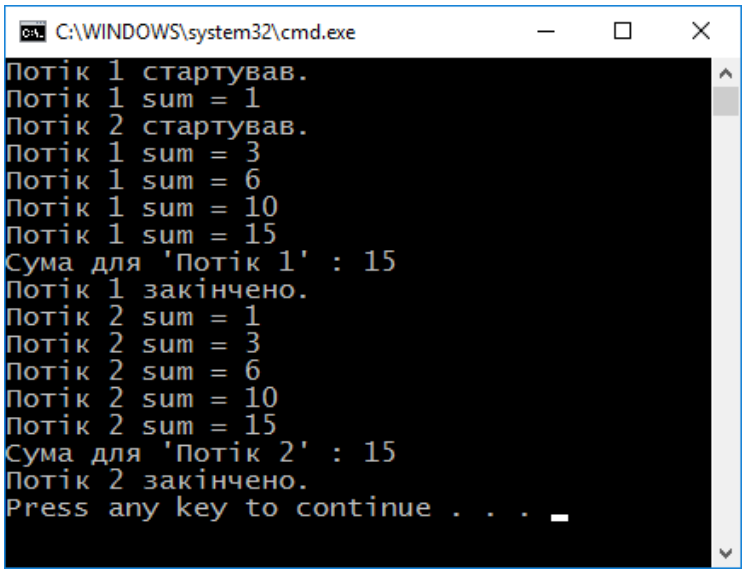


Рис. 13.10. Результат роботи синхронізованих потоків

Механізм, який задіюється при використанні інструкції lock, насправді у C# реалізовано у класі Monitor з простору імен System.Threading. Якщо потік входить у ділянку коду, позначену інструкцією lock, то для ініціювання блокування автоматично викликається метод Enter. При виході з блокованої ділянки для розблокування об'єкта синхронізації викликається метод Exit. Обидва ці методи через параметр отримують посилання на об'єкт синхронізації.

Клас Monitor містить також метод TryEnter, за допомогою якого можна перевірити спробу захоплення потоком керування над блокованою ділянкою коду:

```
public static bool TryEnter(object obj)
```

Метод повертає значення true, якщо спроба була успішною, і false – у протилежному випадку.

Інші засоби синхронізації

Інструкція lock реалізує механізм синхронізації, який у термінах операційної системи Windows відомий як *критична секція* (critical section).

Операційна система Microsoft Windows забезпечує ряд об'єктів, які також використовують для синхронізації роботи окремих потоків та процесів. До них належать:

- М'ютекс.

- Семафор.
- Подія.

М'ютекс – об'єкт взаємного виключення (mutual exclusion) – подібно до критичної секції, призначений для захисту певного об'єкта у потоці від доступу інших потоків. Потоки можуть володіти м'ютексом тільки по черзі. У C# підтримка м'ютексів реалізована у класі Mutex з простору імен System.Threading.

Семафор – це лічильник, який містить ціле число в діапазоні від 0 до заданого при його створенні максимального значення. Семафор надає одночасний доступ до загального ресурсу не одному, а кільком потокам. Семафор управляє доступом до загального ресурсу, використовуючи для цієї мети лічильник. Якщо значення лічильника більше нуля, то доступ до ресурсу дозволений. А якщо це значення дорівнює нулю, то доступ до ресурсу заборонений. За допомогою лічильника ведеться підрахунок кількості дозволів. Отже, для доступу до ресурсу потік повинен отримати дозвіл від семафора. Механізм семафорів реалізує клас Semaphore з простору імен System.Threading.

Об'єкти *подій* використовують для сигналізації очікуючим потокам про досягнення певних умов виконання програми. Події є двох видів: з ручним скиданням та з автоматичним скиданням. Вони реалізовані у двох класах: ManualResetEvent та AutoResetEvent.

Посилання

1. Кравець П.О. Об'єктно-орієнтоване програмування. – Видавництво Львівської політехніки, 2012. – 624 с.
2. Гради Буч, Роберт А. Максимчук, Майкл У. Энгл, Бобби Дж. Янг, Джим Коналлен, Келли А. Хьюстон. Объектно-ориентированный анализ и проектирование с примерами приложений. М.: Вильямс, 2008. – 720 с.
3. Daniel Solis. Illustrated C# 7. The C# Language Presented Clearly, Concisely, and Visually. – APress, 2018. – 799 p.
4. Standard ECMA-334. C# Language Specification. 4th Edition. – Ecma International. – June 2006.
5. International standard ISO/IEC 23270:2006. Information technology – Programming languages – C#. Second edition. – ISO/IEC. – 2006.
6. Herbert Schildt. C# 4.0: The Complete Reference. – McGraw-Hill/Osborne Media, 2010. – 949 p.
7. Daniel Solis. Illustrated WPF. Windows Presentation Foundation presented visually and concisely. – APress, 2009.– 507 p.
8. Коноваленко І.В., Федорів П.С. Системне програмування у Windows з прикладами на Delphi. Навчальний посібник для технічних спеціальностей вищих навчальних закладів. – Тернопіль: ТНТУ. – 2012. – 320 с.

Зміст

ДЛЯ НОТАТОК

Науково-методичне видання

КОНОВАЛЕНКО Ігор Володимирович,
МАРУЩАК Павло Орестович

Платформа .NET та мова програмування C# 8.0

Навчальний посібник

Комп’ютерне макетування та верстка *Ігор Коноваленко.*

К64 Коноваленко І.В. Платформа .NET та мова програмування C# 8.0: навчальний посібник / Коноваленко І.В., Марущак П.О. – Тернопіль: ФОП Паляниця В. А., 2020 – 320 с.

ISBN 978-617-7875-08-5

Підписано до друку 29.10.2020. Формат 60×90, 1/16.
Друк лазерний. Папір офсетний. Гарнітура TimesNewRoman.
Умовно–друк. арк. 11,9. Наклад – 300 прим.
Замовлення № 29102020

*****

Друк ФОП Паляниця В. А.
Свідоцтво ДК №4870 від 20.03.2015 р.
м. Тернопіль, вул. Б. Хмельницького, 9а, оф.38.
тел. (0352) 528–777.

Архітектура платформи .NET

Мова програмування C# 8.0

Проектування класів

Обробка виняткових ситуацій

Інтерфейси

Масиви

Колекції

Платформа Windows Forms

Платформа WPF

Потоки вводу-виводу

Багатопотоковість