МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ

Парфьонов Ю. Е.
Федорченко В. М.
Лосєв М. Ю.
Щербаков О. В.

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Конспект лекцій

Харків. Вид. ХНЕУ, 2010

УДК 004.415(042.4)
ББК 32.973я73
О-29

Рецензент – канд. техн. наук, професор, зав. кафедри інформатики та комп'ютерної техніки Харківського національного економічного університету *Степанов В. П.*

Затверджено на засіданні кафедри інформаційних систем.
Протокол № 3 від 02.12.2009 р.

Авторський колектив: Парфьонов Ю. Е., канд. техн. наук, ст. науковий співробітник – теми 3 – 5; Федорченко В. М., канд. техн. наук, доцент – теми 6, 10; Лосєв М. Ю., канд. техн. наук, доцент – теми 7 – 9; Щербаков О. В., канд. техн. наук, доцент – вступ, теми 1, 2.

О-29 Об'єктно-орієнтоване програмування : конспект лекцій для студентів напряму підготовки "Комп'ютерні науки" всіх форм навчання / Ю. Е. Парфьонов, В. М. Федорченко, М. Ю. Лосєв, О. В. Щербаков. – Харків : Вид. ХНЕУ, 2010. – 312 с. (Укр. мов.)

Викладено основи об'єктно-орієнтованого програмування на прикладі алгоритмічної мови C# та платформи Microsoft .NET. Систематизовано головні прийоми програмування мовою C# .NET: опис типів даних, оголошення змінних і констант, організацію галужень і циклів, опис й використання структурованих типів даних, підпрограм і модулів. Велику увагу приділено реалізації концепції об'єктно-орієнтованого програмування, використанню основних класів бібліотеки .Net Framework. Розглянуто реалізацію у Visual C# .NET об'єктно-орієнтованої парадигми програмування.

Рекомендовано для студентів, викладачів і користувачів, що вивчають основи об'єктно-орієнтованого програмування.

УДК 004.415(042.4)
ББК 32.973я73

Вступ

Сучасні технології програмуваня переживають період бурхливого розвитку. Причинами цього явища є зростання потужностей комп'ютерної техніки, її здешевлення, створення всесвітньої мережі Internet, виникнення все нових і нових сфер застосування комп'ютерів і потреба у різноманітному програмному забезпеченні.

Сучасне програмне забезпечення стає щораз складнішим. Зручний у використанні інтерфейс, розвинені можливості, керованість програми подіями, нові нетрадиційні сфери застосування – все це зумовлює ускладнення програмного забезпечення. Сьогодні, щоб побудувати сучасну програму, не достатньо просто об'єднати в послідовність певні машинні інструкції, оператори мови високого рівня чи навіть набори процедур та модулів. Головним стало питання розробки виразної структури програми, придатної до легкої модифікації, вільної від помилок, стійкої до змін.

Об'єктно-орієнтована технологія створення програмного забезпечення була задумана і розроблена як інструмент подолання складності. Вона успадкувала всі найкращі надбання структурного та модульного програмування, використавши їх для реалізації ряду принципово нових підходів до проектування програмного забезпечення. Головним завданням об'єктно-орієнтованого підходу є забезпечення способу структурування програми та керування складними взаємозв'язками між великою кількістю компонентів системи.

У конспекті лекцій викладаються основи об'єктно-орієнтованого програмування на прикладі алгоритмічної мови C# та платформи Microsoft .NET. Компанія Microsoft – безумовний лідер у розвитку операційних систем і технологій програмування – позиціонує .NET як базову платформу розробки програмного забезпечення найближчими роками. Технологія .NET – це технологія для розробки Web-застосувань і програмного забезпечення, орієнтованого на операційну систему Windows. За допомогою .NET можна також розробляти програмне забезпечення для портативних комп'ютерів та мобільних телефонів.

Мова програмування C# створена корпорацією Microsoft спеціально для платформи .NET. У C# представлені найважливіші аспекти розробок у галузі обчислювальних систем: об'єктно-орієнтоване програмування, обробка графічних об'єктів, компоненти графічного інтерфейсу користувача (GUI), обробка виключень, багатопотокова обробка, мультимедіа

(аудіо, зображення, анімація і відео), обробка файлів, підготовлені структури даних, робота з базами даних, розробка багатоланкових програмних додатків для Інтернету і WWW, організація мереж, Web-служби та розподілені обчислення. Дана мова підходить для створення додатків, що працюють в Інтернеті та WWW, прямо інтегрованих із програмами на базі Windows.

Отже, сучасному програмістові необхідно володіти знаннями щодо технології об’єктно-орієнтованого програмування, а також, архітектури та алгоритмічних мов платформи Microsoft.NET.

У конспекті лекцій систематизовано головні прийоми програмуваввня мовою C# .NET: опис типів даних, оголошення змінних і констант, організацію галужень та циклів, опис і використання структурованих типів даних, підпрограм і модулів. Розглянуто реалізацію у Visual C# .NET об'єктно-орієнтованої парадигми програмування.

На достатньо детальному рівні описано методи та властивості базових елементів керування, технологію розроблення проектів у середовищі Visual Studio 2008.

У результаті вивчення дисципліни "Об’єктно-орієнтоване програмування" студент набуває компетенцій з:

використання основних концепцій об’єктно-орієнтованого програмування при розробці програм алгоритмічною мовою C#;

використання головних можливостей бібліотеки .NET при розробці додатків та графічного інтерфейсу користувача;

виконання основних технологічних операцій з розробки програм у інтегрованому середовищі розробки Microsoft Visual Studio.

Модуль 1. Використання головних концепцій ООП при розробці додатків на мові C#

Тема 1. Введення до платформи Microsoft.NET та мови C#

1.1. Основні поняття платформи Microsoft .NET та мови C#

Платформа .NET

Нова технологія Microsoft. NET надає універсальні засоби створення розподілених програмних систем (продуктів), що підтримують високий ступінь сумісності пристроїв, служб і комп'ютерів. Ключовим елементом технології MS .NET є *платформа* .NET (Framework .NET) – компонентна модель програмного забезпечення, яка дає змогу спільно використовувати окремі програмні модулі, створені різними мовами програмування, у вигляді єдиної функціональної системи. Синонімом терміна *платформа* є термін *каркас*.

MS .NET Framework сформовано із *середовища виконання коду* – Common Language Runtime (CLR) і *бібліотеки класів* Framework .NET – Framework Class Library (FCL), розташованої над CLR.

У CLR забезпечено ефективну взаємодію програмних модулів, створених різними .NET-мовами. Ця взаємодія забезпечується тим, що всі .NET-мови повинні задовольняти певному *набору правил*, які обмежують типи даних і складові системи компіляції та виконання деякою групою компонентів, які надає CLR. Цей набір правил визначено в *загальномовній специфікації* – Common Language Specification (CLS), яку мають підтримувати усі компілятори для .NET-мов.

CLR – це загальне *середовище виконання* додатків для всіх .NET-мов або *віртуальна машина* – програмна система, яка завантажує програму, надає їй усі необхідні служби та виконує її.

Усі мови платформи .NET використовують ідентичну графічну оболонку або створення програм, однаковий механізм обробки виняткових ситуацій, однаковий механізм формування форм і безліч усього іншого.

Перевага такого підходу є очевидною при створенні великих колективних проектів. Наприклад, одна група програмістів пише свою частину коду мовою Visual Basic, а інша – C#. Адже обидві мови

використовують однакову CLR, і ці різні частини одного проекту можна без зусиль об'єднати. Більше того, частини коду, створеного Visual Basic, можна використати у програмах, написаних C#, і навпаки.

Ще одна перевага CLR полягає в тому, що для всіх .NET-мов використовують однакові засоби налагодження програми.

З CLR зв'язане також поняття *керованого коду* (managed code), який виконується у .NET. Програма, написана NET-мовою, і ресурси, що використовуються нею, *керуються* винятково середовищем CLR – звідси і назва. Керований код зберігається у спеціальному двійковому файлі (*складеному модулі* – assembly). Зазвичай, складений модуль та двійковий файл – синоніми, однак трапляються випадки, коли модуль містить декілька двійкових файлів.

Для візуального представлення складеного модуля створено *проміжну мову* – Intermediate Language (IL), яку ще інколи називають MSIL. IL та складений модуль знаходяться на одному рівні, але це не сам двійковий код, а лише його текстове зображення. Для порівняння можна сказати, що IL та складений модуль співвідносяться як асемблер і машинний код.

Код, який виконується безпосередньо під Windows, називають *некерованим*. Деякі мови, наприклад C++, можуть працювати без керуючого середовища CLR і не будуть керованими.

Особливістю системи .NET є використання оригінальної технології *інтеграції коду*, що забезпечує сумісність коду не на рівні виконавчого ядра (*процесора*), а на рівні самої програмної моделі.

Складений модуль якісно відрізняється від машинного коду класичної компіляції наявністю завершеного опису внутрішньої структури програми, починаючи від локальних змінних і закінчуючи класами та просторами назв.

При класичній компіляції в машинний код цілком зникає інформація про внутрішню будову програми, що, здебільшого, і не потрібно для її виконання. Зате така інформація життєво необхідна під час рішення задач міжмовної та міжпрограмної інтеграції. При компіляції у складений модуль втрати цієї інформації не відбувається і вичерпне представлення про внутрішню організацію програми зберігається у спеціальних структурах (*метаданих*).

CLR – віртуальна машина нового покоління, що компілює складений модуль у машинний код *обчислювального вузла* (апаратна

платформа + операційна система), на якій у певний момент відбувається виконання додатка (й усієї віртуальної машини).

З метою реалізації такого підходу віртуальну машину необхідно забезпечити дуже важливим компонентом – *компілятором часу виконання* (Just-In-Time Compiler), який передбачає трансляцію складеного модуля у машинний код саме під час виконання програми. Такі компілятори, зазвичай, називають *двійковими*. Для конкретної комбінації апаратної платформи (Intel, Pocket PC) та операційної системи (Windows, Linux тощо) має існувати власна реалізація двійкового компілятора – це головний недолік технології .NET

Отже, якщо програму деякою мовою програмування можна перекомпілювати в програму IL, то ця мова належатиме до групи .NET-мов. Процес компіляції програми .NET-мовою виглядає так:

1. Створюється вихідний текст програми базовою мовою.
2. Перевіряється синтаксис вихідного тексту програми.
3. Програма перекомпільовується у складений модуль.
4. За допомогою компілятора Just-In-Time програма мовою IL за потребою компілюється у машинний код процесора певної апаратної платформи.

Таким способом для програм .NET створюється закритий віртуальний світ, вийти за межі якого вони не можуть. Однак іноді необхідно звернутися до спеціальних програмних та апаратних ресурсів, які можуть не відображатися у .NET. Для розв'язання цієї проблеми розробники .NET створили технологію виконання спільного коду в рамках компілятора Manager Extension for C++ (MC++), – *керовані розширення для* C++. Цей компілятор дає змогу в одному файлі комбінувати як керований, так і некерований код. Вибір типу коду забезпечує програміст.

Зауважимо, що Visual Studio .NET надає засоби формування інсталяційного пакета програмного продукту, який може розгорнути програму для виконання поза межами .NET.

Середовище виконання коду

Середовище виконання коду **–** Common Language Runtime (CLR) абстрагує сервіси операційної системи і виконує функції середовища виконання для *керованих програм* (managed applications) – програм, кожна дія яких контролюється і підтверджується CLR. FCL надає

об’єктно-орієнтоване середовище (свого роду API), на базі якого керовані програми розробляються і виконуються. При створенні програми на основі платформи .NET Framework використовують різні технології (API, MFC, ATL, COM та інші) через бібліотеку FCL.

.NET Framework підтримує розробку відповідно до багатьох моделей: консольні програми, програми під операційну систему Windows (Windows Forms або GUI-програми), Windows- або NT-сервіси, веб-сайти на базі технології ASP.NET, веб-сервіси та ін.

CLR архітектурно стоїть над операційною системою і надає віртуальне середовище для виконання керованих програм. Коли така програма починає своє виконання, CLR завантажує в оперативну пам’ять комп’ютера модуль, який потрібно виконати, і виконує код, який він містить.

Інструкції керованого коду компілюються *за потребами* використання у машинний код під час виконання. Такий процес компіляції називають just-in-time (JIT) компіляцією. Здебільшого кожен метод компілюється лише тоді, коли його вперше активізують. Далі метод кешується в оперативній пам’яті для того, щоб його можна було використати повторно без компілювання. Отож код, який ніколи не активізується, не компілюється.

Хоча JIT-компіляція і вимагає більше ресурсів, цей недолік нівелюється тим фактом, що метод компілюється лише раз за весь час виконання програми. Отже, компілятор оптимізовано для якнайшвидшого та ефективного виконання. Теоретично JIT-компільований код має більшу швидкодію, ніж звичайний код, унаслідок того, що JIT-компілятор має механізм для оптимізації машинного коду, який він генерує, відповідно до процесора, за допомогою якого виконується компіляція.

Переваги такого коду в керованому середовищі CLR є значними. Під час компіляції IL-інструкцій у машинний код компілятор передусім виконує процес перегляду інструкцій з метою отримання *безпечного коду* з точки зору типів. Отож практично неможливо виконати інструкцію, яка має на меті доступ до ділянки пам’яті без прав доступу до неї. Перегляд коду не дає змоги написати код, який може вплинути на безпеку операційної системи.

Іншою важливою перевагою CLR є виконання багатьох програм в одному процесі, що забезпечується механізмом поділу процесу на віртуальні частини, які називають *доменами програми*.

Операційна система Windows ізолює різні програми одна від одної, використовуючи різні процеси. Недоліком такої моделі є значне використання пам’яті. Використання пам’яті не є важливим чинником для автономних систем, якими послуговується один користувач, але є визначальним чинником для серверів, які обслуговують водночас тисячі користувачів.

У деяких випадках (наприклад, при розробці веб-сайту на основі технології ASP.NET) CLR не створює новий процес для кожного користувача, а створює лише один процес з доменами програм для кожного користувача. Домени програм є безпечними з погляду доступу, як і процеси, бо вони створюють обмеження, які керовані програми не можуть порушити. Однак домени програм є дещо ефективнішими, ніж процеси. Адже один процес може містити декілька доменів програм, а бібліотеки, які використовують ці домени, можуть завантажуватись у процес лише один раз.

Зауважимо, що ресурси, які виділяє керований код, вивільняються автоматично за допомогою "збирача сміття". Це означає, що розробник виділяє пам’ять, проте не займається її вивільненням – система вивільняє її автоматично. CLR містить ефективного збирача сміття, що відстежує посилання на об’єкти, які створює код, і знищує ці об’єкти, коли пам’ять, яку вони займають, є потрібною для інших цілей.

Точні алгоритми, які використовує збирач сміття, не припускають до втрат пам’яті. Недолік такого механізму вивільнення ресурсів полягає в тому, що під час збирання сміття у деякому процесі все інше виконання у цьому процесі моментально припиняється. Проте збирання сміття трапляється порівняно рідко, отож воно не надто й впливає на швидкодію.

Ще однією перевагою .NET є те, що CLR є незалежною від мови програмування. Тому вибір мови програмування є питанням смаку. На платформі .NET мова програмування є лише синтаксичним засобом для генерування IL-коду, і все, що можна зробити за допомогою однієї мови програмування, можна зробити і за допомогою іншої. Незалежно від мови всі керовані програми використовують одну і ту ж бібліотеку класів .NET Framework. Це дає змогу створювати клас однією мовою

програмування і використовувати його або створювати його нащадка іншою.

CLR складається з трьох головних елементів: загальної системи типів – Common Type System (CTS), системи метаданих – Metadata System і системи виконання – Execution System. *Загальна система типів* – це частина середовища CLR, що визначає усі типи, які використовують програмісти. *Тип* (type) – це визначення чи "креслення", за яким створюється екземпляр значення. Середовище CLR містить множину типів, яку розділяють на *типи-значення* (value types) і *типи-посилання* (reference types). Базовим класом усіх типів є System.Object.

Типи-значення (або *розмірні* типи) походять від базового типу System.ValueType і діляться на три категорії: *вбудовані* типи, тип *перелічення* (Enum) і тип *користувача*. Тип System.ValueType є прямим нащадком System.Object. Будь-який тип, похідний від System.ValueType, є *структурою*, отож їх ще називають *структурними* типами.

Типи-посилання розділяють на *об'єктні* типи (object types), *інтерфейсні* типи (interface types), *вказівні* типи (pointer types) і тип-посилання *користувача*. Об'єктний тип аналогічний *класу* (class) у багатьох об'єктно-орієнтованих мовах програмування. Усі типи-посилання є прямими нащадками класу System.Object.

Типи можуть налічувати члени (members), які можуть бути полями (fields) чи *методами* (methods). *Властивості* (properties) і події (events) є спеціальними типами методів. Поля і методи можуть належати всьому типу (type) або якомусь *екземпляру* (instance).

Доступ до поля чи виклик методу, які належать усьому типу, можна здійснити навіть тоді, коли даний тип не має жодного екземпляра. Такі члени типу (поля/методи) іноді називають *статичними*.

Доступ до поля екземпляра можна одержати, тільки вказуючи його екземпляр, а метод екземпляра можна викликати тільки з зазначенням його екземпляра.

При оперуванні з *розмірними* типами використовують їхнє пряме зображення, яке зберігається у пам'яті комп'ютера. Типи-посилання використовують *адресні покажчики* (або *вказівники*) на осередки пам'яті комп'ютера, де розміщуються їхні значення. Тобто змінні розмірного типу – це самі дані, а змінні типу посилання – це покажчики (вказівники), які містять адреси розміщення даних.

Для розміщення значень змінних типу посилання пам’ять виділяється у *купі*, а змінних розмірних типів – у *стекові*. При присвоєнні одного *розмірного* типу іншому присвоюється не сам тип (як осередок пам’яті), а його двійкове зображення. При присвоєнні одного типу *посилання* іншому створюється ще один *покажчик*, який отримує адресу осередка пам’яті, що займає об’єкт.

Змінні типу посилання можна порівнювати за ознаками ідентичності та рівності. *Ідентичність* (identity) двох посилань означає, що вони містять покажчики (адреси) на один і той самий об’єкт, а рівність (equality) – на два різні об’єкти з однаковими даними.

Передача *аргументів* параметрам процедур *розмірного* типу здійснюється як передача *значень* (тобто процедурам передаються локальні копії значень змінних), а параметрам типу *посилання* – передача *покажчиків* (процедурам передаються адреси розміщення об’єктів у пам’яті).

Система метаданих є частиною CLR, що *описує* типи у цьому середовищі. Компілятори використовують метадані для створення типів, доступних у їхніх власних мовах, а система типів використовує метадані для керування типами під час виконання. Метадані зберігаються у *двійковому форматі*.

Система виконання є частиною середовища CLR, яка відповідає за завантаження збірок, керування потоком виконання і збирання сміття в купі.

Убудовані типи-значення

У табл. 1.1 перелічено вбудовані типи-значення CLR. У першому стовпці наведено назву типу проміжною мовою (IL). У наступному стовпці наведено назву цього типу в бібліотеці класів платформи NET (FCL). Останній стовпець табл. 1.1 містить дані про сумісність типів із *загальномовними специфікаціями* (CLS).

При створенні власних типів у багатомовному середовищі варто обмежуватися типами, сумісними з CLS. Тоді користувацькі класи, інтерфейси та структури працюватимуть без проблем з будь-якою мовою NET.

Таблиця 1.1

Убудовані типи-значення в середовищі CLR

IL-назва	Назва в FCL	Опис	Підтримка CLS
bool	System.Boolean	Логічне значення (true / false	Так
char	System.Char	Символ Unicode	Так
int8	System.SByte	8-бітове ціле число зі знаком	Ні
int16	System.Int16	16-бітове ціле число зі знаком	Так
int32	System.Int32	32-бітове ціле число зі знаком	Так
int64	System.Int64	64-бітове ціле число зі знаком	Так
unsigned int8	System.Byte	8-бітове ціле число без знака	Так
unsigned int16	System.UInt16	16-бітове ціле число без знака	Ні
unsigned int32	System.UInt32	32-бітове ціле число без знака	Ні
unsigned int64	System.UInt64	64-бітове ціле число без знака	Ні
float32	System. Single	32-бітове число з плаваючою комою	Так
float64	System.Double	64-бітове число з плаваючою комою	Так
native int	System.IntPtr	Машинне ціле число зі знаком, (32 або 64 біта залежно від процесора)	Так
native unsigned int	System.UIntPtr	Машинне ціле число без знака	Ні

Бібліотека класів платформи .NET

Бібліотека класів Framework .NET – Framework Class Library (FCL) містить понад 7000 типів (класів, структур, інтерфейсів, перелічених типів і делегатів), які поділено між "просторами назв", кожен з яких відповідає за служби з певної області (табл.1.2).

Таблиця 1.2

Основні простори назв .NET

Простір назв	Опис
System	Містить основні типи і класи
System.Collections	Кешовані таблиці, динамічні масиви та інші контейнери
System.Data	Класи ADO.NET для роботи з базами даних
System.Drawing	Класи для генерування графіки (GDI+)
System.IO	Класи для введення/виведення у файли і потоки
System.NET	Класи, що є програмною реалізацією мережевих протоколів
System.Reflection	Класи для читання/запису метаданих
System.ServiceProcess	Класи для створення NET-сервісів
System.Threading	Класи для створення і керування процесами
System.Web	Класи для підтримки Web-застосувань
System.Web.Ser-vices	Класи для розробки веб-сервісів
System.Web.Services.Protocols	Класи для розробки клієнтів веб-сервісів
System.Web.UI	Основні класи технології ASP.NET
System.Web.UI.WebControls	Серверні контролери ASP.NET
System.Windows.Forms	Класи для GUI-програм
System.Xml	Класи для читання і запису даних у форматі XML

Простори назв забезпечують ієрархію класів, отож допускають функціонування двох різних класів з однаковими назвами, які знаходяться у різних просторах назв. Отже, простір назв – це не що інше, як область дії класу.

Простір назв System – базовий простір платформи .NET. Він містить класи для обробки виняткових ситуацій, типів даних, забезпечення низькорівневих засобів введення/виведення даних, збирання сміття тощо. Програміст, що використовує вбудовані класи, отримує доступ до простору назв через директиву:

using [aliasname =] namespace.element

де:

aliasname – ідентифікатор, за допомогою якого можна посилатися усередині модуля на зазначений простір назв;

namespace – назва імпортованого простору;

element – назва елемента простору (клас, простір назв тощо).

Кожен модуль може налічувати довільне число директив Imports, однак усі вони повинні розташовуватися до будь-якого посилання на ідентифікатори. Назва простору є частиною докладної назви об’єкта, що має загалом синтаксис namespace.typename. Усі простори назв, які постачає корпорація Microsoft, розпочинаються словом System або словом Microsoft.

1.2. Основи мови C#

Основні вбудовані типи мови C#

Мова C# – це мова жорстокої типізації. Це означає, що кожний об’єкт у програмі має відноситись до одного з визначених типів. Всі типи даних, що використовуються в C#, діляться на 2 категорії: типи-значень (***value types***) та типи-посилання (***reference types***). Про відмінності між ними говорилось раніше, а зараз розглянемо основні вбудовані типи мови C#. Вони представлені у табл. 1.3.

Таблиця 1.3

Основні вбудовані типи мови C#

	Назва типу	Назва системного типу	Опис типу	Розмір у бітах	Діапазон значень
1	2	3	4	5	6
1	bool	Boolean	логічний	8	false, true
2	byte	Byte	8-розрядний цілий без знаку	8	0 255
3	sbyte	SByte	8-розрядний знаковий цілий	8	-128 +127
4	short	Int16	короткий знаковий цілий	16	-32 768 +32787
5	ushort	UInt16	короткий цілий без знаку	16	0 65535
6	int	Int32	знаковий цілий	32	-2 147 483 648 +2 147 483 647
7	uint	UInt32	цілий без знаку	32	0 +4 294 967 295

Закінчення табл. 1.3

1	2	3	4	5	6
8	long	Int64	довгий знаковий цілий	64	-9 223 372 036 854 775 808 +9 223 372 036 854 775 807
9	ulong	UInt64	довгий цілий без знаку	64	0 +18 448 744 073 709 551 615
10	float	Single	дійсний	32	1,401298E-45 3,402823E+38
11	double	Double	дійсний подвоєної точності	64	E-324 E+308
12	decimal	Decimal	числовий для фінансових розрахунків	96	29 значущих розрядів
13	char	Char	символьний	16	
14	string	String	Набір символів Unicode		

Визначення та ініціалізація змінних, область їх видимості

Для того, щоб визначити змінну одного із стандартних типів досить вказати її тип та ідентифікатор. Можлива її ініціалізація в момент визначення константним значенням або значення виразу.

```
using System;
namespace Declaration_of_variables
{
    class Program
    {
        static void Main()
        {
            float f = 1.5F;
// визначення змінної f з ініціалізацією
            char c;
// просто визначення змінної c
            int i = 0;
            bool b = true;
            decimal d = 1.55555555555555555555555M;
            double x = Math.Sin(Math.PI / 3);
```

```
    // визначення змінної x з динамічною ініціалізацією
                Console.WriteLine("f = {0} i = {1} x = {2} d =
{3}", f, i, x, d);
                Console.WriteLine(b.ToString());
            }
        }
    }
```

На екрані побачимо: f = 1,5 i = 0 x = 0,866025403784439 d = 1,55555555555555555555555

True

При виводі на консоль можливо організувати форматування виводу, тобто одержати значення на екрані у зручному для сприйнятті вигляді. Для цього у текстову константу – аргумент методу `Console.WriteLine` – треба помістити так звані плейсхолдери з номером потрібного елементу списку виводу. На місці кожного з таких плейсхолдерів на екрані з'явиться відповідне значення. Більше того, можливо використання форматів виводу. Загальний вигляд плейсхолдеру такий `{n:fk}`. Тут `n` означає порядковий номер елементу у списку виводу (нумерація починається з нуля), `f` – це символ специфікації формату, число `k` задає точність. Основні символи специфікацій формату наступні:

`F` або `f` – для виводу дійсних значень у форматі з фіксованою точністю;

`E` або `e` – для виводу дійсних значень у експоненціальному форматі;

`G` або `g` – загальний формат для виводу дійсних значень або у форматі з фіксованою точністю або у експоненціальному форматі;

`N` або `n` – формат для виводу дійсних значень з відокремленням трійок розрядів пробілами;

`C` або `c` – грошовий формат;

`X` або `x` – шістнадцятковий формат для виводу цілих типів.

Повернемось до визначення змінних. До сих пір всі змінні, які використовували автори, визначались у функції `Main`. Вони можуть використовуватись в будь-якій точці функції `Main`, починаючи з точки визначення.

Проте можна визначати змінні всередині будь-якого блоку, тобто в області програми, обмеженої парою фігурних дужок. Такі змінні створюються, коли виконання програми доходить до даного блоку, і зникають, коли блок виконаний. Це забезпечує механізм інкапсуляції, тому що звернутись до такої змінної із зовнішніх щодо даного блоку частин програми неможливо. Цей факт ілюструє наступний приклад.

```
using System;
namespace Context_of_using
{
    class Program
    {
        static void Main()
        {
            int i = 1000;            // Змінна i може
використовуватись у всій функції Main
            {
                int j = 0;           // Змінна j видима лише
в цьому блоці
                i = i + j;            // Змінна i видима в
цьому блоці
                Console.WriteLine("i = " + i.ToString());
                Console.WriteLine("j = " + j.ToString());
            }
            Console.WriteLine("j = " + j.ToString());
// Помилка - змінна j тут не існує
        }
    }
```

Приведення типів

Розберемось, як мова C# суміщає у виразах змінні різних типів, тобто як відбувається перетворення типів, адже відомо, що у всіх виразах та операціях повинні використовуватись змінні однакових типів. З цією метою розглянемо наступний приклад.

```
using System;
namespace Convert_of_variables_1
{
    class Program
    {
        static void Main()
```

```
        {
            float f = 0;
            double x = f;       // таке присвоєння припустиме
            f = x;              // а таке – ні
            f = (float)x;       // явне приведення типу
            Console.WriteLine("f = " + f.ToString());
            Console.WriteLine("x = " + x.ToString());
        }
    }
}
```

Змінній `x` типу `double` можна присвоїти значення змінної менш "потужного" типу, наприклад, типу `float`, тому що компілятор C# виконує неявне приведення типу (explicit convert), а от розраховувати, що автоматично буде виконане зворотне перетворення – неможливо. Адже при присвоєнні значень більш потужного типу змінній менш потужного типу можливі втрати інформації. Відповідальність за виконання таких операцій в разі їх потреби програміст повинен взяти на себе, необхідно виконати явне приведення типу (implicit convert). Така операція записується інструкцією:

```
(тип_до_якого_приводимо_вираз)вираз;
```

При обчисленні виразів, які містять операнди різних типів, всі вони приводяться (звісно, якщо типи сумісні між собою) до найбільш широкого типу. Таке перетворення виконується неявним чином при дотриманні низки правил "просування по сходинках типів". При звуженні потужності типу завжди потрібне явне приведення типу. Правила просування є такими:

1) якщо один із операндів має тип `decimal`, то і другий буде приводитись до такого типу (але якщо другий операнд мав тип `float` або `double` , результат буде помилковим);

2) якщо один із операндів має тип `double`, то і другий буде приводитись до такого типу `double`;

3) якщо один із операндів має тип `float`, то і другий буде приводитись до типу `float`;

4) якщо один із операндів має тип `ulong`, то і другий буде приводитись до типу `ulong` (але якщо другий операнд мав цілий знаковий тип, результат буде помилковим);

5) якщо один із операндів має тип `long`, то і другий буде приводитись до типу `long`;

6) якщо один із операндів має тип `uint`, а другий має тип `sbyte`, `short` або `int` , то обидва операнди будуть приведені до типу `long`;

7) якщо один із операндів має тип `uint`, то і другий буде приводитись до типу `uint`;

8) інакше обидва операнди перетворюються до типу `int`;

Останнє правило пояснює, чому в наступному коді виникає помилка.

```
using System;
namespace Convert_of_variables_2
{
    class Program
    {
        static void Main()
        {
            byte b1 = 16, b2 = 32;
             // нижче виникає помилка, оскільки згідно з правилом 8, результат має тип int
            byte b = b1 + b2;
            Console.WriteLine("b = " + b.ToString());

        }
    }
}
```

Щоб код успішно компілювався та працював, треба виконати явне приведення результату до типу `byte` , який має змінна `b`:

```
            byte b = (byte)(b1 + b2);   // так правильно!
```

Літерали (константи) мови C#

У мові C# ***літералом*** називається деяке фіксоване значення. Іншими словами це таке значення, яким можна ініціалізувати змінну, присвоїти константі тощо. Літерали можуть мати довільний допустимий ***тип значень***. Тип літерала визначається згідно з певними правилами. Цілий літерал не містить десяткової крапки чи знаку порядку. Автоматично відноситься до найменшого знакового цілого типу, починаючи із типу `int`, до множини значень якого входить літерал. Тобто літерали 25, -10 мають тип `int`, а літерал 3 333 333 333 має тип `long`. Якщо потрібно віднести цілий літерал до іншого типу, треба це явно

вказати, додавши один із суфіксів безпосередньо після літералу: символи U або u для літералу без знакового типу, символи L або l для довгого цілого. Таким чином, 1L – це літерал типу `long`, 1U – типу `uint`, а 1UL – типу `ulong`.

Можна визначити також шістнадцятковий літерал, він починається із префікса 0X або 0x. Тобто 0XFFFF та 0x11111 – шістнадцяткові літерали.

Літерал, який містить десяткову крапку або знак порядку відноситься до типу `double`. Якщо необхідно, щоб він мав тип `float`, додається суфікс F. Отже, літерал 1.5 має тип `double`, а літерал 1.5F – тип `float`.

Літерал типу `decimal` позначається суфіксом M, наприклад, 1.5M.

Символьний літерал задається в одинарних лапках, а рядковий (стрінговий) літерал у подвійних лапках: ‘A’ та "A" – це різні літерали, які відносяться до різних типів.

До символьних літералів відносяться і так звані esc-послідовності. Вони зображуються двома символами, перший з яких `\`, у одинарних лапках. Наприклад, `‘\n’ ‘\t’` – це символи переходу на новий рядок та табуляції. Особливу роль відіграє так званий нуль-символ `‘\0’`. Для позначення лапок подвійних та одинарних, а також самого знаку `\` використовується esc-послідовності `‘\"’` , `‘\’’` та `‘\\’` відповідно. Таким чином, якщо в програмі є інструкція `Console.WriteLine("Це \tприклад \n\"ESC-послідовності\");` на екрані побачимо:

```
Це   приклад
"ESC-послідовності"
```

Зауваження. Замість символу `\n` для переходу на новий рядок можна використовувати стрінг `Environment.NewLine`.

Цікавий ще один нюанс, який відрізняє текстові літерали мов C++ та C#. Якщо перед текстовим літералом стоїть знак @, то такий літерал називається буквальним і при зображенні на екрані виглядає дослівно так само, як у подвійних лапках. Таким чином зникає потреба у використанні знаків табуляції, нового рядку тощо. Єдиний виняток проти "буквальності" – сам знак подвійних лапок, якщо він зустрічається у літералі, має бути подвоєним. Нижче наведений приклад використання буквального (`virbatim`) літералу.

```
using System;
namespace Virbatim_Literal
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine(@"Це
буквальний літерал, а це -
"" "" - подвійні лапки у ньому;
' ' - одинарні лапки у ньому ");
        }
    }
}
```

Після запуску побачимо на екрані наступний текст:

```
Це
буквальний літерал, а це -
" " - подвійні лапки у ньому;
' '   - одинарні лапки у ньому
```

Операції мови C#

Розглянемо основні групи операцій, завдяки яким у мові C# забезпечується можливість виконання широкого спектру обчислень.

Арифметичні операції

Арифметичні операції (+) – додавання, (-) – віднімання, (*) – множення, (/) – ділення, (%) – визначення залишку від ділення є бінарними операціями і виконуються над операндами довільних цілих та дійсних чисел, формуючи результат, тип якого визначається правилами "просування по сходинках типів".

Зауваження.

1. Операція % в мові C# може виконуватись і над операндами дійсних типів.
2. При виконанні операції / над операндами цілих типів залишок буде відкинутий, тобто 5/2 дає результат 2.

Операції інкременту та декременту

Унарні операції інкременту (++) та декременту (--) викликають відповідно збільшення або зменшення операнду дійсного або цілого типу

на одиницю. Цікавості та певної загадковості цим операторам надає можливість різного часу виконання. А саме, вони мають префіксну форму (коли знак операції ++ записується **перед** змінною) та постфіксну форму (коли знак операції -- записується **після** змінної). В обох випадках змінна, до якої застосована операція інкреметну або декременту, буде збільшена або зменшена на одиницю, проте при префіксній формі ці зміни відбудуться перед використанням змінної у виразі, а при постфіксній формі – після використання змінної у виразі. Розглянемо приклад, що демонструє ці особливості.

```
using System;
namespace Increment_and_Decrement
{
    class Program
    {
        static void Main()
        {
            int x = 1, y;
            y = x++;
            Console.WriteLine("x = {0} y = {1}", x, y);
            x = 1;
            y = --x;
            Console.WriteLine("x = {0} y = {1}", x, y);
        }
    }
}
```

У результаті на екрані побачимо:

```
x = 2 y = 1
x = 0 y = 0
```

Тобто обчислення виразу `y = x++;` відбувається у два етапи: спочатку `y = x;` а потім `x = x++;`. З префіксною формою все більш очевидно. Спробуйте визначити, що буде на екрані в результаті виконання наступного фрагменту коду:

```
x = 1;
y = x++ + ++x;
Console.WriteLine("x = {0} y = {1}", x, y);
```

Операції відношення (порівняння)

Бінарні операції відношення : (==) – перевірка на рівність, (!=) – перевірка на нерівність, (>) – порівняння більше, (>=) – порівняння більше або рівне, (<) – порівняння менше, (<=) – порівняння менше або рівне. Кожна з цих операцій формує значення `true` або `false` залежно від результату порівняння. Мова C# дозволяє перевіряти на рівність або нерівність довільні об'єкти, тобто ці операції застосовні до всіх типів. Решту ж порівнянь можна виконувати лише над операндами тих типів, які підтримують операцію відношення порядку, тобто лише до числових типів. Розглянемо приклад:

```
using System;
namespace Comparing_Operators
{
    class Program
    {
        static void Main()
        {
            int x = 1, y = 2;
            bool b = (x > y);
            Console.WriteLine("x > y ? " + b.ToString());
            b = (x <= y);
            Console.WriteLine("x <= y ?" + b.ToString());
            b = (x == y);
            Console.WriteLine("x == y ?" + b.ToString());
        }
    }
}
```

Після запуску програми одержимо наступний результат:

```
x > y ? False
x <= y ? True
x == y ? False
```

Логічні операції

Логічні операції виконуються над операндами логічного (булівського) типу та визначають результат логічного типу. Логічні операції мають дві форми: звичайну та скорочену. Розглянемо спочатку набір звичайних логічних операцій: (&) – логічне множення або логічне "І", ( ) – логічне додавання або логічне "АБО", (^) – логічне виключне

"АБО", (!) – логічне заперечення "НІ". Значення логічних операцій наведені у табл. 1.4 для різних значень операндів (op1 та op2).

Таблиця 1.4

Значення логічних операцій

op1	op2	op1 & op2	op1 || op2	op1 ^ op2	!op1
true	true	true	true	false	false
true	false	false	true	true	false
false	true	false	true	true	true
false	false	false	false	false	true

Крім того, логічні множення та додавання мають ще скорочені форми, які позначаються знаками && та || відповідно. Ідея їх використання пов'язана з прискоренням логічних обчислень – обчислення логічного виразу, складеного із скорочених логічних операцій припиняється, якщо його значення стає визначеним. Наступний приклад демонструє різницю у використанні повної та скороченої форм логічних операцій.

```
using System;
namespace Logic_Operator
{
    class Program
    {
        static void Main()
        {
            int i = 10, k = 100;
            if (!(k > 10) && (++i < 100))        // Тут і не
змініться
                // Цей оператор не виконується
                Console.WriteLine("Скорочена форма &&: тут і
не зміниться" );
            Console.WriteLine("i = " + i.ToString());
            i = 10;           // Відновлюємо і
            if (!(k > 10) & (++i < 100))         // Тут і
зміниться
                // Цей оператор не виконується
```

```
                    Console.WriteLine("Звичайна форма &: тут і
змініться" );
                Console.WriteLine("i = " + i.ToString());
            }
        }
    }
```

Порозрядні (бітові) операції

Порозрядні операції виконуються над відповідними бітами внутрішнього подання лише **цілих** операндів. Результатом виконання є ціле відповідного операндам типу. Таких операцій є 6 : (`&`) – порозрядне "І", тобто кон'юнкція бітів, ( ) – порозрядне "АБО",тобто диз'юнкція бітів, (^) – порозрядне виключне "АБО", (>>) – порозрядний зсув праворуч, (<<) – порозрядний зсув ліворуч, (~) – порозрядне заперечення. Всі операції, крім останньої, є унарними. Слід зауважити, що при зсуві ліворуч (<<) на вказану правим операндом кількість позицій внутрішнє подання лівого операнду просто зсувається ліворуч, а позиції, що вивільняються, заповнюються нулями. При зсуві праворуч (>>) позиції, що вивільняються зліва, заповнюються нулями для беззнакового операнду, якщо ж зсув виконується для знакового операнду, то на вільні ліві позиції розповсюджується знаковий розряд, тобто для від'ємного числа вільні позиції ліворуч заповняться одиницями. Це обов'язково слід враховувати. У табл. 1.5 вказані результати бітових операцій для різних значень операндів (`op1` та `op2`).

Таблиця 1.5

Результати бітових операцій для різних значень операндів

op1	op2	op1 & op2	op1 op2	op1 ^ op2	~op1
1	1	1	1	0	0
1	0	0	1	1	0
0	1	0	1	1	1
0	0	0	0	0	1

Наступний приклад демонструє можливості операцій над бітами.

```
using System;
namespace Bits_Operators
```

```
    {
        class Program
        {
            static void Main()
            {
                int x = 5, y = 6, z;
                z = x & y;                  // Логічне множення "І"
                Console.WriteLine("x = {0} y = {1}  x & y = {2}",
x, y, z);
                z = x   y;                 // Логічне додавання "АБО"
                Console.WriteLine("x = {0} y = {1}  x   y = {2}",
x, y, z);
                z = x ^ y;                 // Логічне виключне "АБО"
                Console.WriteLine("x = {0} y = {1}  x ^ y = {2}",
x, y, z);
                z = x << 1;              // Зсув на 1 позицію ліворуч
                Console.WriteLine("x = {0}  x << 1 = {1}", x, z);
                z = x >> 1;             // Зсув на 1 позицію праворуч
                Console.WriteLine("x = {0}  x >> 1 = {1}", x, z);
                z = ~y;                  // Логічне заперечення "НІ"
                Console.WriteLine("y = {0}  ~y = {1}", y, z);
                z = y   0X7;             // Встановлюємо одиниці в
останні 3 розряди
                Console.WriteLine("y = {0}  y 0X7 = {1}", y, z);
                z = x & ~0X7;                  // Встановлюємо нулі
в останні 3 розряди
                Console.WriteLine("x = {0}  x&~0X7 = {1}", x, z);
            }
        }
    }
```

Виконання цієї програми приведе до наступних результатів:

```
x = 5 y = 6  x & y = 4
x = 5 y = 6  x   y = 7
x = 5 y = 6  x ^ y = 3
x = 5  x << 1 = 10
x = 5  x >> 1 = 2
y = 6  ~y = -7
```

Умовна (тернарна) операція

Умовна операція має таку синтаксичну форму:

```
<вираз_1> ? < вираз_2> : < вираз_3>
```

Назву "тернарна" ця операція має тому, що її визначають 3 вирази. Перший вираз повинен мати тип `bool`. Якщо його значення рівне `true`, то обчислюється значення другого виразу, яке і визначає всю тернарну операцію. Якщо ж значення першого виразу `false`, то обчислюється значення третього виразу, і його значення стає значенням всього тернарного виразу. Наприклад, у наступному фрагменті коду обчислюється максимум дві цілих змінних:

```
int x = 3, y = 4;
int z = (x > y) ? x : y;
Console.WriteLine("максимум із {0} та {1} є {2}",
x, y, z);
```

Після виконання цього фрагменту на екрані побачите повідомлення :

```
максимум із 3 та 4 є 4
```

Операції присвоєння

Крім звичайної операції присвоєння (=) в мові C# виконуються ще 10 скорочених операцій присвоєння, які позначаються як `<op>=` , де знак `<op>` може бути замінений на знак однієї з операцій : `+ - * / % & ^ >> <<` . При цьому значення конструкції `<змінна> <op>= <вираз>` ; обчислюється наступним чином:

`<змінна> = <змінна> <op> <вираз>` ;

Наприклад, інструкція `x += 1;` еквівалента наступній інструкції: `x = x + 1;` .

Оскільки виконання операції присвоєння генерує результат, рівний значенню своєї правої частини, допускається виконання ланцюжку присвоєнь. Так, наприклад, в наступному фрагменті коду ініціалізуються нулями відразу 3 змінні:

```
int i, j, k;
i = j = k = 0;
```

Пріоритет операцій

При складанні виразів слід враховувати, що порядок виконання операції операцій у виразі визначається пріоритетом операцій. Проте правилом хорошого тону слід вважати використання дужок, які підкреслюють порядок обчислень та роблять вираз більш зрозумілим.

Табл. 1.6 вказує основні операції мови C# в порядку спадання їх пріоритетів від найвищого до найнижчого.

Таблиця 1.6

Основні операції мови C# в порядку спадання їх пріоритетів

( ) [ ] ++(постфіксний) --(постфіксний) new sizeof
! ~ +(унарний) -(унарний) ++(префіксний) --(префіксний)
*** / %**
+ -
<< >>
< <= > >= is
== !=
&
|
^
&&
||
? :
= <op>=

Зауваження. Різниця у пріоритетах префіксних та постфіксних операціях інкременту-декременту можна побачити у наступному прикладі.

```
using System;
namespace Prioritet
{
    class Program
    {
        static void Main()
        {
            int x = 1, y = 2, z;
            z = x++ + y;    // Вираз мав вигляд: z = x+++y;
            Console.WriteLine("x = {0} y = {1} z = {2}", x,
y, z);
            x = 1; y = 2;   // Встановлюємо початкові
значення змінних
            z = x + ++y;    // Визначаємо пріоритет пробілами
та дужками
            Console.WriteLine("x = {0} y = {1} z = {2}", x,
y, z);
```

```
        }
    }
}
```

Тобто, вираз записаний у вигляді `z = x+++y;` був проінтерпретований згідно з таблицею пріоритетів як:

`z = x++ + y;` , а не як `z = x + ++y;` . Якщо хочемо визначити порядок операцій іншим, використовуємо дужки та пробіли.

Основні інструкції керування мови C# – розгалуження та цикли. Розгалуження у мові C#

Для реалізації галужень (одного із трьох основних елементів блок-схем програм) у мові C# використовуються конструкції `if` та `switch`.

Конструкція `if` може бути використана у двох формах. Повна форма має наступний синтаксис.

```
if (<умова>)     інструкція_1;
then     інструкція_2;
```

`<Умовою>` є вираз з результатом типу `bool`. Порядок виконання цієї конструкції очевидний – якщо результатом умови є "Істина" `(true)`, то виконується перша з інструкцій, якщо ж результатом умови є "Хибність" `(false)`, – то друга. Коли необхідно виконати не одну інструкцію, а більше, їх необхідно об’єднати у блок:

```
if (<умова>)
{ // інструкції
}
then
{ // інструкції
}
```

Частина `then` не є обов’язковою і може бути пропущена. Тоді при істинності умови єдина інструкція `if` виконується, а при хибності – управління виконанням програми передається наступній конструкції програми.

Розглянемо приклади.

У першому з них шукаємо максимум з двох чисел, використовуючи спочатку коротку, а потім – повну форму конструкції `if`.

```
using System;
namespace Construct_if
{
```

```
        class Program
        {
            static void Main()
            {                       // Знаходження максимуму із
двох чисел
                float x, y, max;
                Console.Write("Введіть значення x = ");
                x = float.Parse(Console.ReadLine());
                Console.Write("Введіть значення y = ");
                y = float.Parse(Console.ReadLine());
                max = x;          // Призначаємо x максимумом
                if (y > max)      // Можливо, максимумом є y?
                    max = y;
                Console.WriteLine("Максимум із {0} та {1}
дорівнює {2}", x, y, max);
                Console.WriteLine("");
                // Шукаємо максимум іншим способом
                if (y > x) max = y;
                else max = x;
                Console.WriteLine("Максимум із {0} та {1}
дорівнює {2}", x, y, max);
            }
        }
    }
```

У наступному прикладі для деякого вкладника можливе одержання бонусу в разі, коли він вказує правильний пароль та сума на його рахунку перевищує контрольну константу. Зверніть увагу на умову (parol == RIGHT_PAROL). Поширеною помилкою, навіяною синтаксисом аналогічної конструкції в Паскалі, є використання одного знаку = замість двох == при перевірці на рівність двох об'єктів. Крім того, тут конструкції `if-else` вкладені одна в одну.

```
    using System;
    namespace Construct_if_else
    {
        class Program
        {
            static void Main()
            {
                const decimal MINSUM = 10000;  // Мінімальна сума
для бонусу
                const string RIGHT_PAROL = "myparol"; // Значення
справжнього паролю
```

```
                string parol;
                decimal sum;
                Console.WriteLine("Введіть пароль");
                parol = Console.ReadLine();
                                // Перевірка правильності паролю
                if (parol == RIGHT_PAROL)
                {                               // Пароль правильний
                    Console.WriteLine("Введіть суму внеску");
                    sum = int.Parse(Console.ReadLine());
                    if (sum > MINSUM)  // Перевіряємо суму внеску
                        Console.WriteLine("Вітаємо! Ви одержуєте
бонус!");
                    else
                        Console.WriteLine("Накопичуйте ще!");
                }
                else                          // Пароль неправильний
                    Console.WriteLine("Ви ввели неправильний
пароль");
            }
        }
    }
```

Використання вкладених умовних конструкцій може привести до випадку, коли на дві або більше частини `if` доводиться лише одна частина `else`. Тоді вважається, що вона відноситься до найближчої частини `if`, що не має частини `else`.

```
    using System;
    namespace Nested_if
    {
        class Program
        {
            static void Main()
            {           // Розбираємось із вкладеними if-else
                int i = 1, j = 2, k = 5;
                if ((i != 0) & (j > 1))
                    if (k == 10)
                        Console.WriteLine("Перевірка k == 10 дає
true");
                    else
                        Console.WriteLine("Перевірка k == 10 дає
false");
            }
        }
    }
```

Тому при виконанні даного прикладу на екрані з'являється повідомлення:

Перевірка k == 10 дає false

Поширеною практикою при "багатошарових перевірках" є використання вкладених конструкцій `if-else-if`. У наступному прикладі завжди виконується одна і тільки одна інструкція.

```
using System;
namespace Construct_if_else_if
{
    class Program
    {
        static void Main()
        {           // Виконуємо операції з цілими числами
            char op;
            int x = 10, y = 20;
            Console.WriteLine("Введіть знак для операції з
числами {0} {1}", x, y);
            op = (char)Console.Read();
            if (op == '+')              // Додаємо?
               Console.WriteLine("{0} " + op + " {1} = {2}" ,
x, y, x + y);
            Else                        // Ні!
               if (op == '-')           // Віднімаємо?
                  Console.WriteLine("{0} " + op + " {1} =
{2}", x, y, x - y);
               Else                 // Ні!
                  if (op == '*')        // Множимо?
                     Console.WriteLine("{0} " + op + " {1} =
{2}", x, y, x * y);
                  Else                  // Ні!
                     if (op == '/')          // Ділимо?
                        Console.WriteLine("{0} " + op + " {1}
= {2}", x, y, x / y);
                     Else               // Ні!
                        if (op == '%')  // Шукаємо залишок?
                          Console.WriteLine("{0} " + op + "
{1} = {2}", x, y, x % y);
                        Else            // Ні!
                           Console.WriteLine("Неприпустимий
знак операції");

        }
    }
}
```

Для більш зручної та компактної реалізації подібних багатоальтернативних галужень в мові C# використовується конструкція `switch`. Вона має наступний синтаксис.

```
switch (<вираз>)
{
   case <константа_1> : <інструкції>
   break;
    case <константа_2> : <інструкції>
   break;
…
   case <константа_n> : <інструкції>
   break;
   default            : <інструкції>
   break;
}
```

Тут `<вираз>` повинен мати цілий тип (або тип char), кожна з констант є унікальною і сумісною за типом із `<виразом>`. Після обчислення `<вираз>` виконується та гілка конструкції `switch`, яка містить константу рівну значенню `<виразу>`. Якщо жодна з констант не співпадає із значенням `<виразу>`, виконуються інструкції гілки `default`. Втім, остання не є обов'язковою і в разі її відсутності у випадку, коли жодна з констант не співпадає із значенням `<виразу>`, жодна інструкція конструкції `switch` не виконується взагалі.

Перепишемо попередню програму з допомогою конструкції `switch`.

```
using System;
namespace Construct_switch
{
    class Program
    {
        static void Main()
        {
            char op;
            int x = 10, y = 20;
            Console.WriteLine("Введіть знак для операції з
числами {0} {1}", x, y);
            op = (char)Console.Read();
            switch (op)
            {
                case '+':      // Додаємо?
```

```
                        Console.WriteLine("{0} " + op + " {1} =
{2}", x, y, x + y);
                        break;
                    case '-':       // Віднімаємо?
                        Console.WriteLine("{0} " + op + " {1} =
{2}", x, y, x - y);
                        break;
                    case '*':       // Множимо?
                        Console.WriteLine("{0} " + op + " {1} =
{2}", x, y, x * y);
                        break;
                    case '/':       // Ділимо?
                        Console.WriteLine("{0} " + op + " {1} =
{2}", x, y, x / y);
                        break;
                    case '%':       // Беремо залишок?
                        Console.WriteLine("{0} " + op + " {1} =
{2}", x, y, x % y);
                        break;
                    default:        // Помилка
                        Console.WriteLine("Неприпустимий знак
операції");
                        break;
                }
            }
        }
    }
```

Зауваження.

1. У даному прикладі конструкція `switch` керувалась змінною символьного типу – це цілком припустимо, адже символьний тип є підмножиною цілого, оскільки містить коди символів.

2. Однією (або декількома) із інструкцій `switch` може бути вкладений `switch`.

3. Якщо одну й ту саму групу інструкцій потрібно виконати для деякої множини значень констант, ці константи потрібно послідовно перелічити із службовими словами `case`. Наприклад, як у наступному прикладі програми:

```
using System;
namespace Construct_case_case
{
    class Program
```

```
        {
            static void Main()
            {
                Console.WriteLine("Введіть ціле число");
                int i = int.Parse(Console.ReadLine());
                switch( i*i % 4)
                {
                    case 0: // Однаковий набір інструкцій для i*i
% 4 == 0 або 2
                    case 2:
                        Console.WriteLine("Число було парним");
                        break;
                    case 1: // Однаковий набір інструкцій для i*i
% 4 == 1 або 3
                    case 3:
                        Console.WriteLine("Число було непарним");
                        break;
                }
            }
        }
    }
```

Цикли у мові C#

Для реалізації ітерацій деякої інструкції або групи інструкцій у мові C# передбачено 4 види циклів. Зараз познайомимось з трьома з них.

1. Цикл `for`.

Синтаксис цього циклу такий.

```
    for  (<вираз-ініціалізація>;<вираз-умова>;<вираз-
ітерація>) <інструкція циклу>;
```

або

```
    for  (<вираз-ініціалізація>;<вираз-умова>;<вираз-
ітерація>)
    {
    <група інструкцій>;
    }
```

Виконання циклу `for` відбувається наступним чином.

1. Обчислюється `<вираз-ініціалізація>`.

2. Обчислюється `<вираз-умова>`. Якщо він має значення `false`, то дія циклу закінчується. Інакше виконується `<інструкція циклу>`.

3. Обчислюється <вираз-ітерація>.

4. Відбувається перехід до кроку 2.

Найчастіше у <виразі-ініціалізації> встановлюється початкове значення деякої змінної, яка відіграє роль змінної циклу. <Вираз-ітерація> змінює значення цієї змінної, а у <виразі-умові> її значення порівнюється з деяким граничним значенням для прийняття рішення щодо продовження чи завершення циклу. Розглянемо приклади.

У першому прикладі обчислюється сума $s = \sum_{i=1}^{10} \frac{\sin i}{i}$.

```
using System;
namespace Construct_for_sum
{
    class Program
    {
        static void Main()
        {               // Обчислюємо суму
            double s = 0;
            for (int i = 1; i <= 10; i++)
                s += Math.Sin(i) / i;
            Console.WriteLine("Сума sin(i)/i від 1 до 10
рівна {0}", s);
        }
    }
}
```

Можливості мови дозволяють записати цикл, еквівалентний попередньому "порожнім", тобто таким, що містить лише порожню інструкцію ; (зауважимо, що за синтаксисом "тіло" циклу for повинно містити принаймні одну інструкцію, хоч і порожню).

```
using System;
namespace Construct_for_sum
{
    class Program
    {
        static void Main()
        {                   // Обчислюємо суму
            double s = 0;   // Тіло циклу – порожнє, все
"заховано" в ітерації
```

```
                for (int i = 1; i <= 10; s += Math.Sin(i)/i++) ;
                Console.WriteLine("Сума sin(i)/i  від 1 до 10
рівна {0}", s);
            }
        }
    }
```

У наступному прикладі розглядається ціле число типу byte (тип byte може бути замінений з відповідними виправленнями у програмі на довільний цілий ***беззнаковий*** тип – беззнаковий тому, що використовується операція зсуву праворуч, про її особливості автори говорили раніше). За результатом порозрядного множення числа на маску 0X1, яка у внутрішньому поданні є послідовністю нулів з одиницею лише в останньому розряді, встановлюємо зміст останнього розряду числа. А оскільки треба "переглянути" всі розряди, то просто по черзі використовують порозрядний зсув числа праворуч, починаючи з найстаршого розряду, а отже, з максимального зсуву на `sizeof(byte)*8-1` позицій.

```
    using System;
    namespace Construct_for_Bits
    {
        class Program
        {
            static void Main()
            {
                 // друкуємо біти внутрішнього подання цілого
числа
                byte ui;
                Console.WriteLine("Введіть ціле число");
                ui = byte.Parse(Console.ReadLine());
                byte size = sizeof(byte) * 8;
                for (int i = size - 1; i >= 0; i--)
                { // використовуємо бітові операції; 0X1 =
00000…001
                    if (((ui >> i) & 0X1) != 0)
Console.Write("1");
                    else Console.Write("0");
                    if (i % 4 == 0) Console.Write(" "); // пробіл
між четвірками бітів
                }
                Console.WriteLine();
            }
        }
    }
```

Зауваження.

1. Усі вирази у заголовку циклу не є обов'язковими і можуть бути пропущені. В разі відсутності <виразу-умови> його значенням вважається true. Таким чином можна записати "нескінченний" цикл:

```
for (;;)
{ // інструкції нескінченного циклу
  // вихід з циклу має бути передбаченим якимось чином
}
```

2. Можливе використання відразу кількох змінних циклу. Тоді вирази, пов'язані з їх ініціалізаціями та ітераціями, відокремлюються комами. Як наприклад у циклі

```
int i, j;
for (i = 0, j = 10; i < j; i++, j--)  // тут 2 змінні циклу: i та j
    Console.WriteLine("i = {0} j = {1}", i, j);
```

2. Цикл `while`.

Синтаксис цього циклу наступний.

```
while (<вираз-умова >) <інструкція циклу>;
```

Порядок виконання циклу наступний:

1. Обчислюється `<вираз-умова>` , значення якого має бути типу `bool`. Якщо значенням є `false`, то дія циклу закінчується. Інакше виконується `<інструкція циклу>`.

2. Після цього повторюється перший крок.

Зрозуміло, що в інструкції циклу (або в блоці інструкцій) слід передбачити якийсь вплив на умову циклу, інакше він, розпочавшись, не зможе завершитись.

У наступній програмі обчислюється та сама сума, що була запрограмована з допомогою циклу for у першому прикладі. Переконайтесь у незмінності результату.

```
using System;
namespace Construct_WhileSum
{
    class Program
```

```
        {                   // Обчислюємо ту саму суму, що й у
прикладі з циклом for
            static void Main()
            {
                int i = 1;
                double sum = 0;
                while (i <= 10)
                {
                    sum += Math.Sin(i) / i;
                    i++;
                }
                Console.WriteLine("sum = {0}", sum);
            }
        }
    }
```

У прикладі, наведеному нижче, знаходимо кількість цифр у цілому числі. В циклі `while` фіксується кількість кроків, за яке число шляхом цілочисельного ділення на 10 перетворюється на 0.

```
    using System;
    namespace Construct_whileDigits
    {
        class Program
        {                       // знаходимо кількість цифр у
цілому числі
            static void Main()
            {
                Console.WriteLine("Введіть ціле число");
                long num = long.Parse(Console.ReadLine());
                if (num < 0) num = Math.Abs(num);
                byte count = 0;
                while (num != 0)
                {
                    num /= 10;    // Зменшуємо число на один
розряд
                    count++;
                }
                Console.WriteLine("Кількість цифр = {0}", count);
            }
        }
    }
```

3. Цикл do-while.

Цей цикл на відміну від попереднього називають циклом з післяумовою, оскільки умова виходу з циклу аналізується вже після його виконання. Він має наступний синтаксис.

```
do
{
<інструкції циклу>
}   while (<вираз-умова>)
```

Порядок виконання циклу наступний:

1. Виконуються всі інструкції в тілі циклу.
2. Обчислюється <вираз-умова>, значення якого має бути типу bool. Якщо його значенням є false, то дія циклу закінчується. Інакше повторюється перший крок.

Цикл do-while відрізняється від циклів while та for тим, що його інструкції виконуються ***завжди*** принаймні один раз.

Розглянемо приклад, який забезпечить повторні запуски для тестування вашої програми. Сигналом для виходу стане натискання клавіші ESC . Цей контроль відбувається в умові циклу do-while.

```
using System;
namespace Construct_do_while
{
    class Program
    {                          // Забезпечуємо повторний запуск
програми
                               // до натискання клавіши ESC
        static void Main()
        {
            ConsoleKeyInfo conKey; // Змінна для збереження
значень натиснутих клавіш
            do
            {
                Console.WriteLine("Тут виконуються інструкції
вашої програми");
                Console.WriteLine("Для виходу натисніть
ESC");
                conKey = Console.ReadKey(true);        //
Одержуємо значення клавіші
            } while (conKey.Key != ConsoleKey.Escape);//
Порівнюємо його з ESC
        }
    }
}
```

Нижче – інший варіант реалізації тієї ж ідеї, програма виконується повторно до тих пір, поки не буде натиснутий символ ‘1’.

```
using System;
namespace Construct_do_while
{
    class Program
    {                          // Запит на повторний запуск
програми
                               // Погодження – натискання символу
1
        static void Main()
        {
            string answer;
            do
            {
                Console.WriteLine("Тут виконуються інструкції
вашої програми");
                Console.WriteLine("Продовжувати виконання?
Для підтвердження
                                        натисніть 1");
                answer = Console.ReadLine();
            } while (answer == "1");
        }
    }
}
```

Зауваження.

Будь-який з означених циклів може містити інший цикл – так виникають вкладені цикли. Наприклад, для обчислення подвійної суми виду $s = \sum_{i=1}^{N} \sum_{j=1}^{M} a(i, j)$ можна використати вкладений цикл:

```
for (int i = 1; i <= N; i++)
    for (int j = 1; i <= M; j++)
s += a (i, j);
```

Керування виходом із циклів C#

Інколи виникає необхідність термінового переривання циклів. Таку логіку програми забезпечує інструкція `break;`, з якою було ознайомлено

у конструкції switch. Якщо всередині циклу знаходиться інструкція break;, то відбувається вихід із циклу, а управління передається інструкції, що безпосередньо слідує за даним циклом. Таким чином можна, наприклад, перервати виконання "нескінченного" циклу.

```
bool condition;
…
      for (;;)
      {
      …
            if (condition) break;
      …
      }
```

Слід зауважити, що у випадку вкладених циклів інструкція break; викликає вихід лише із того циклу, де вона знаходиться, у зовнішній, не впливаючи на останній.

Крім інструкції break; для керування циклами використовується інструкція continue. У циклі for вона викликає перехід до обчислення <виразу-ітерації> (тобто до кроку 3), а у циклах while та do-while – перехід до обчислення <виразу-умови> циклу.

Радикальну дію викликає інструкція безумовного переходу goto. Її вживання у програмах вкрай небажане і може бути зумовлене лише крайньою необхідністю, наприклад, дострокового виходу із кількох вкладених циклів у разі виникнення помилки. Синтаксис цієї інструкції наступний:goto <мітка>;

Тут <мітка> – це звичайний ідентифікатор, який помічає інструкцію, на яку передбачено передачу управління програмою. Наприклад,

```
int  counter = 10;
label :   counter--;
                  Console.WriteLine("counter = " +
counter.ToString());
                  if (counter > 0) goto label;           //
Реалізований цикл із 10 кроків
```

Масиви в мові C#. Визначення та ініціалізація масиву

Масив – це тип даних (***reference-type***) для збереження елементів однакового типу, доступ до яких здійснюється за індексом. Першим значенням індексу є 0. Головна особливість масивів у мові C# полягає у тому, що вони реалізовані як ***об'єкти***, що ***спадкують*** свої властивості та методи від класу `System.Array`.

Синтаксис визначення масиву наступний:

```
<тип_елементів> [] <ідентифікатор_масиву>;
<ідентифікатор_масиву> = new <тип_елементів> [кількість_елементів];
```

У першому рядку масив ***декларується*** (описується), точніше кажучи, визначається адресна змінна, яка буде вказувати на масив. У другому рядку масив власне ***створюється*** (операція `new`) в області Heap і адреса його місця розташування пов'язується із ідентифікатором масиву. Ті самі дії можна здійснити в одному рядку:

```
<тип_елементів> [] <ідентифікатор_масиву> = new <тип_елементів> [кількість_елементів];
```

Індексами елементів масиву будуть значення `0, 1, …, кількість_елементів-1`. Для ініціалізації масиву можна використати фігурні дужки, в яких перелічити значення його елементів. У цьому разі `кількість_елементів` при визначенні масиву можна не вказувати – компілятор визначить її по кількості ініціалізованих елементів. До елементів масиву звертаються, вказуючи ідентифікатор масиву та індекс потрібного елементу у квадратних дужках. Розглянемо приклади визначення масивів.

```
using System;
namespace Array_1
{
    class Program
    {
        static void Main()
        {           // Робота з масивом
            const int SIZE = 10;
            int[] iArray = new int[SIZE];      // Визначаємо масив
            for (int i = 0; i < SIZE; i++) // Елементи масиву ініціалізуються нулями!
```

```
                    Console.WriteLine("iArray [{0}] = {1}", i,
iArray [i]);
            }
        }
    }
```

У даному прикладі масив був створений, але не проініціалізований. Тим не менше на екрані побачимо, що всі елементи масиву мають нульові значення. Проте, хорошим стилем у програмуванні вважаються явні ініціалізації. Зробимо це у наступному прикладі.

```
    using System;
    namespace Array_2
    {
        class Program
        {
            static void Main()
            {
                // Визначаємо (без new) та ініціалізуємо масив –
                // його розмір визначається компілятором
                int[] iArray = {1, 2, 3, 4, 5};
                Console.WriteLine("Масив цілий iArray");
                for (int i = 0; i < iArray.Length; i++)
                    Console.WriteLine("iArray [{0}] = {1}", i,
iArray[i]);
                // Визначаємо та ініціалізуємо масив
                double[] dArray = new double  { 1.1, 2.2, 3.3,
4.4, 5.5 };
                Console.WriteLine("Масив дійсний dArray");
                for (int i = 0; i < dArray.Length; i++)
                    Console.WriteLine("dArray [{0}] = {1}", i,
dArray[i]);
                // Інший масив з тим самим ідентифікатором
                dArray = new double  { 1.2, 2.3, 3.4, 4.5 };
                Console.WriteLine("Ще один дійсний масив
dArray");
                for (int i = 0; i < dArray.Length; i++)
                    Console.WriteLine("dArray [{0}] = {1}", i,
dArray[i]);
                double[] dArray_new;
                dArray_new = dArray; // Той самий масив з іншим
ідентифікатором
                Console.WriteLine("Той самий дійсний масив
dArray_new");
                for (int i = 0; i < dArray_new.Length; i++)
```

```
                    Console.WriteLine("dArray [{0}] = {1}", i,
dArray_new[i]);
                }
            }
        }
```

Зауваження.

1. У даному прикладі у рядку `int[] iArray = {1, 2, 3, 4, 5};` створюється масив цілих чисел `iArray`, який ініціалізується п'ятьма значеннями. Зверніть увагу, в цьому випадку службове слово `new` та вказання кількості елементів не потрібно – компілятор автоматично визначає розмір масиву за кількістю ініціалізованих елементів.

2. У рядку `double[] dArray = new double { 1.1, 2.2, 3.3, 4.4, 5.5 };` створюється та ініціалізується дійсний масив із 5 елементів, про що явно повідомляється компілятору. Ця вказівка в принципі є надмірною, хоча й синтаксично правильною. Але в разі явного визначення розміру масиву службове слово `new` є необхідним.

3. Інструкцією `dArray = new double { 1.2, 2.3, 3.4, 4.5 };` для уже визначеного ідентифікатора `dArray` створюється та ініціалізується новий масив із чотирьох елементів.

4. Зручною формою для умови продовження циклу `for` є використання властивості `Length`, яка для будь-якого масиву визначає кількість елементі в ньому.

5. Зверніть увагу на інструкції :

```
double[] dArray_new;
dArray_new = dArray;
```

У першій з них описується ідентифікатор `dArray_new` дійсного масиву. А у другій – цьому ідентифікатору присвоюється значення ідентифікатора уже визначеного масиву `dArray`. У результаті обидва ідентифікатори вказують на один і той самий масив, точніше на одне й те саме місце у пам'яті, де записані 4 дійсних числа.

Щоб підкреслити "об'єктну природу" масивів, розглянемо ще один приклад, що стосується зауваження 5. В ньому створюються, ініціалізуються та виводяться на екран 2 різних масиви. Потім ідентифікатору першого масиву присвоюється ідентифікатор другого. Фізично масиви знаходяться на своїх місцях у пам'яті, проте їх ідентифікатори посилаються тепер на одну й ту саму область, що і бачимо при виводі першого масиву. Посилання на перший масив тепер втрачене і в певний

час він буде знищений з пам’яті спеціальною системою збору "сміття" GC – garbage collector.

```
using System;
namespace Array_3
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] iArray1 = {1, 2, 3, 4, 5}; // Ініціалізуємо
перший масив
            Console.WriteLine("Перший масив");
            for (int i = 0; i < iArray1.Length; i++)
                Console.WriteLine("Елемент масиву [{0}] =
{1}", i, iArray1[i]);
            Console.WriteLine("Другий масив");
            int[] iArray2 = { 6, 7, 8, 9, 10 };
// Ініціалізуємо другий масив
            for (int i = 0; i < iArray2.Length; i++)
                Console.WriteLine("Елемент масиву [{0}] =
{1}", i, iArray2[i]);
            iArray1 = iArray2;   // Присвоєння
ідентифікаторів масиву -
                                 // тепер перший
ідентифікатор вказує на другий масив
            Console.WriteLine("Другий масив");
            Console.WriteLine("Ще раз перший масив");
            for (int i = 0; i < iArray2.Length; i++)
// Друкуємо перший масив ще раз
                Console.WriteLine("Елемент масиву [{0}] =
{1}", i, iArray2[i]);
        }
    }
}
```

Важливо наголосити також, що C# суворо контролює значення індексів елементів масиву. Якщо значення індексу виводить за межі виділеної для масиву області пам’яті – виникає помилка типу "index out of range".

Цикл foreach

Цикл `foreach` використовується для опитування елементів ***колекцій***. Під колекцією в мові C# розуміють певний набір об'єктів. Однією із форм колекцій є масив. Отже, цикл `foreach` має наступний синтаксис.

`foreach (<тип> <змінна> in <колекція>) <інструкція>;`

У результаті виконання даного циклу `<змінна>` пробігає значення всіх елементів колекції, при цьому щоразу виконується `<інструкція>`. Очевидно, що в разі, коли колекцією є масив, `<тип>` у циклі `foreach` має збігатись із типом елементів масиву. Ітераційна `<змінна>` циклу не може бути змінена, вона доступна лише для читання.

У наступному прикладі масив із `SIZE` цілих `iArray` заповнюється випадовими значеннями (використовуємо об'єкт класу `Random`, який ініціалізуємо значенням 1 – це значення, яке встановлює базу для псевдовипадкової послідовності чисел; далі використовується метод `Next` цього класу, він визначає чергове ціле невід'ємне псевдовипадкове число). Далі у циклі `foreach` обчислюється сума всіх елементів цього масиву та знаходиться мінімальне та максимальне значення.

```
using System;
namespace Array_foreach
{
    class Program
    {
        static void Main(string[] args)
        {
            const int SIZE = 5;
            int[] iArray = new int [SIZE];   // Декларуємо
масив
            // Заповнюємо масив випадковими значеннями
            Random r = new Random(1);        // Декларуємо
об'єкт Random з базою 1
            for (int i = 0; i < SIZE; i++)
            {
                iArray[i] = r.Next(100);    // Одержуємо
випадкове число від 0 до 100
                Console.WriteLine("Елемент масиву [{0}] =
{1}", i, iArray[i]);
```

```
                }
            // Сумуємо елементи та шукаємо мінімальний та
максимальний елементи масиву
                int sum = 0;
                int min = iArray[0];
                int max = iArray[0];
                foreach (int elem in iArray)    // Переглядаються
всі елементи масиву
                {
                    sum += elem;
                    if (elem < min) min = elem;
                    if (elem > max) max = elem;
                }
                Console.WriteLine("Сума = {0}", sum);
                Console.WriteLine("Min = {0} Max = {1}", min,
max);
            }
        }
    }
```

Багатовимірні масиви

Крім одновимірних масивів можна використовувати також масиви більшої вимірності. Елементи двовимірного масиву індексуються двома індексами, трьовимірного – трьома, і т. д. Для того, щоб продекларувати та створити двовимірний масив, необхідно записати:

```
<тип_елементів> [,] <ідентифікатор_масиву>;
<ідентифікатор_масиву>  =  new  <тип_елементів>
[кількість_1, кількість_2];
```

У результаті буде створено масив, який можна уявляти у вигляді таблиці із `кількістю_1` рядків та `кількістю_2` стовпчиків. Розглянемо приклад використання двовимірних масивів.

```
using System;
namespace Array_two_dimensional_1
{
    class Program
    {
        static void Main(string[] args)
        {
            float[,] f_arr1;     // Декларація двовимірного
масиву
            // Ініціалізація двовимірного масиву розміру 2*3
```

```
                f_arr1 = new float[,] {
                                        { 1, 2, 3 },
                                        { 4, 5, 6 }
                                        };
                foreach (float elem in f_arr1)    // Вивід
першого масиву
                {
                    Console.WriteLine(elem);
                }
                Random r = new Random();           // Створюємо
об'єкт Random
                float[,] f_arr2 = new float[3, 4]; // Створюємо
інший двовимірний масив
                for (int i = 0; i < f_arr2.GetLength(0); i++)
                    for (int j = 0; j < f_arr2.GetLength(1); j++)
                        f_arr2[i, j] = (float)r.NextDouble(); //
Елементи-випадкові числа
                // Вивід першого масиву у вигляді таблиці
                Console.WriteLine("Перший масив");
                for (int i = 0; i < f_arr1.GetLength(0); i++)
                {
                    for (int j = 0; j < f_arr1.GetLength(1); j++)
                    {
                        Console.Write(f_arr1[i, j]);
                        Console.Write("\t");
                    }
                    Console.WriteLine();
                }
                // Вивід другого масиву у вигляді таблиці
                Console.WriteLine("Другий масив");
                for (int i = 0; i < f_arr2.GetLength(0); i++)
    // Цикл по рядках
                {
                    for (int j = 0; j < f_arr2.GetLength(1); j++)
    // Цикл по стовпчиках
                    {
                        Console.Write(f_arr2[i, j]);
                        Console.Write("\t");
                    }
                    Console.WriteLine();
                }
            }
        }
    }
```

Зауваження.

1. Масив `f_arr1` ініціалізується двома наборами значень – для першого та другого рядків. Кожен з них поміщається в окремі фігурні дужки і відповідає окремим рядкам масиву.

2. Зверніть увагу, як оформлюється вивід масивів у табличному вигляді – використовується метод `GetLength(dim)`, який повертає кількість елементів масиву по вказаній вимірності `dim`. Тобто при `dim = 0` одержуємо кількість рядків, при `dim = 1` одержуємо кількість стовпчиків і т. д.

Використання деяких методів класу `System.Array`

Клас `System.Array` містить низку властивостей та методів, які зручно використовувати при роботі з масивами. До деяких з них автори уже звертались у прикладах. Так властивість `Length` вказує кількість елементів масиву (для багатовимірних масивів – загальну кількість елементів), метод `GetLength()` – повертає кількість елементів масиву по вказаному виміру. Серед інших можна відзначити деякі наступні. Властивість `Rank` дає кількість вимірів даного масиву, Метод `Array.Sort()` дозволяє відсортувати одновимірний масив (за умовчанням – у порядку зростання, або обираючи певний ключ з допомогою інтерфейсу `IComparable`), метод `Array.Reverse()` переставляє елементи одновимірного масиву у зворотному порядку, метод `Array.Clone()` створює копію масиву, метод `Array.Clear()` заповнює нулями певні елементи масиву. Розглянемо простий приклад використання цих методів.

```
using System;
namespace Array_4
{
    class Program
    {
        static void Main(string[] args)
        {
            const int SIZE = 10;        // Розмір масивів
            int[] iArray = new int[SIZE];
            Console.WriteLine("Введіть {0} цілих чисел",
SIZE);
            for (int i = 0; i < SIZE; i++)
```

```
                {
                    Console.Write ("[{0}] = ", i);
                    iArray [i] = int.Parse(Console.ReadLine());
                }
                Console.WriteLine("Створюємо копію масиву:");
                int[] iCloneArray = (int[])iArray.Clone();
                Console.WriteLine("Сортуємо цю копію по
зростанню");
                Array.Sort(iCloneArray);            // Сортування
                for (int i = 0; i < SIZE; i++)
                    Console.WriteLine("iCloneArray [{0}] = {1}",
i, iCloneArray[i]);
                Console.WriteLine("Переставляємо елементи");
                Array.Reverse(iCloneArray);       // Перестановка
                for (int i = 0; i < SIZE; i++)
                    Console.WriteLine("iCloneArray [{0}] = {1}",
i, iCloneArray[i]);
                Console.WriteLine("Зануляємо 5 елементів,
починаючи з індекса 3");
                Array.Clear(iCloneArray, 3, 5);      // Очистка
деяких елементів
                for (int i = 0; i < SIZE; i++)
                    Console.WriteLine("iCloneArray [{0}] = {1}",
i, iCloneArray[i]);

            }
        }
    }
```

Тема 2. Реалізація головних концепцій об’єктно-орієнтованого програмування у мові C#

2.1. Основні положення об’єктно-орієнтованого підходу

Очевидно, не всі програмні системи складні. Є багато програм, задуманих, розроблених, супроводжуваних і використовуваних однією людиною. Але вони, як правило, мають обмежену галузь застосування і коротку тривалість використання. Системи банківських розрахунків, резервування залізничних квитків, операційні системи та інші належать до індустріально розробленого програмного забезпечення (ПЗ). Його створює і вдосконалює колектив авторів, використовує тривалий час

багато користувачів. Суттєвою рисою такого ПЗ є його велика складність: практично неможливо одному розробникові охопити всі тонкощі системи. Таким чином, можна зробити висновок – складність притаманна програмному забезпеченню.

Можна виділити чотири основні причини складності ПЗ:

Складність проблеми. Прикладні проблеми часто містять складні елементи, до яких ставиться багато різних, часто суперечливих вимог. Ситуація погіршується тим, що розробники ПЗ можуть мати недостатню кваліфікацію в галузі проблеми, а замовники часто не до кінця уявляють, що власне їм потрібно. Додаткова складність зумовлюється зміною вимог до програми в процесі розробки, оскільки вже наявність проекту системи змінює саму проблему і ступінь її розуміння. На жаль, розробку не можна щоразу починати з самого початку, тому великі системи мають тенденцію до еволюції.

Складність керування процесом розробки. Головне завдання розробників полягає у створенні ілюзії простоти, яка захищатиме коритувачів від складності процесу, який описують. Однак складність проблеми і велика кількість вимог до ПЗ вимагає колективної праці розробників. У цій ситуації важливим завданням є координація робіт і підтримання цілісності основної ідеї.

Гнучкість програмного забезпечення. Програміст може сам забезпечити себе всіма необхідними елементами конструкції ПЗ, які належать до будь-якого рівня абстракції, починаючи з найнижчих. Ця спокуслива властивість та ще й відсутність єдиних стандартів на такі елементи часто призводить до того, що програми починають писати "з нуля". Тому розробка ПЗ залишається дуже копіткою справою.

Складність опису поведінки окремих підсистем. Програма сучасного (цифрового) комп’ютера є системою зі скінченною кількістю дискретних станів, причому їх може бути дуже і дуже багато. Переходи системи з одного стану в інший не можна моделювати неперервними фунціями, тому іноді, у випадку несприятливого збігу обставин, зміна стану однієї частини системи може привести до катастрофічних змін у інших частинах. Адже всеохоплююче тестування великої програмної системи провести просто неможливо.

Прикладом складної системи може бути персональний комп’ютер (ПК). Основними складовими ПК є системний блок, монітор та клавіатура. Відповідно кожну з цих частин теж можна розділити на

складові: наприклад, системний блок містить процесор, оперативну пам'ять, накопичувачі на магнітних дисках, блок живлення. Далі можна розглянути влаштування процесора і т. д. Так одержимо *структурну ієрархію* ПК. Комп'ютер працює добре, коли злагоджено функціонують усі його частини. Кожна з частин є відносно незалежною від інших.

Складні системи є не просто ієрархічні: рівні ієрархії відображають різні рівні абстракції, що випливають один з одного, будучи деякою мірою автономними. Для кожної конкретної задачі розглядаємо відповідний рівень. Наприклад, щоб підготувати за допомогою ПК деякий текст, достатньо лише уявляти загальну будову комп'ютера (найвищий рівень абстракції). Проте таких знань буде замало, щоб написати ефективну програму мовою асемблера.

Можна також навести приклад іншої ієрархії. Різні ПК бувають оснащені різними типами процесорів. Маючи спільну властивість виконувати певний набір команд, процесори бувають різної потужності, різних фірм-виробників тощо. У цьому випадку говорять про *ієрархію типів* процесорів.

Сучасне програмне забезпечення стає щораз складнішим. Зручний у використанні інтерфейс, розвинені можливості, керованість програми подіями, нові нетрадиційні сфери застосування – все це зумовлює ускладнення ПЗ. Можна сказати, що виготовлення ПЗ, зручного у використанні, робить це ПЗ складним. Сьогодні, щоб побудувати сучасну програму, не достатньо просто об'єднати в послідовність певні машинні інструкції, оператори мови високого рівня чи навіть набори процедур та модулів. Головним стало питання розробки виразної структури програми, придатної до легкої модифікації, вільної від помилок, стійкої до змін. Як сказав Алан Кей, розробник Smalltalk'у, зі зростанням складності архітектура домінує над матеріалами.

Об'єктно-орієнтована технологія створення ПЗ була задумана і розроблена як інструмент подолання складності. Вона успадкувала всі найкращі надбання структурного та модульного програмування, використавши їх для реалізації ряду принципово нових підходів до проектування ПЗ. Головним завданням об'єктно-орієнтованого підходу є забезпечити спосіб структурування програми та керування складними взаємозв'язками між великою кількістю компонентів системи.

Об'єктно-орієнтоване структурування зменшує число зв'язків між компонентами. Воно заставляє об'єкти взаємодіяти через вузькі

інтерфейси, що дає змогу легко ізолювати помилки та визначати, котрий з методів є відповідальним за помилку, що виникла.

Об’єкти захищають свої дані від несанкціонованого доступу. Оголошені протоколи взаємодії дозволяють компіляторові чи інтерпретаторові попереджувати користувача про несанкціонований доступ до даних і навіть не допустити його. Добре спроектовані інтерфейси класів дають змогу будувати добре модульовані, легко переміщувані компоненти програми.

Найбільшою зручністю технології є безпосереднє проектування об’єктів прикладної галузі в об’єкти програми. Легше моделювати реальний світ у вигляді об’єктів, ніж проектувати його у вигляді процедур.

2.2. Класи та об’єкти, співвідношення між ними

Що ж є об'єктом, а що ні? Здатністю до розпізнання об'єктів фізичного світу людина володіє із самого раннього віку. З точки зору сприйняття людиною об'єктом може бути:

відчутний і (або) видимий предмет;

щось, сприймане мисленням;

щось, на що спрямована думка або дія.

Таким чином, об'єкт моделює частину навколишньої дійсності й у такий спосіб існує в часі та просторі. Термін *об'єкт* у програмному забезпеченні вперше був уведений у мові Simula і застосовувався для моделювання реальності.

Об'єктами реального світу не вичерпуються типи об'єктів, цікаві при проектуванні програмних систем. Інші важливі типи об'єктів уводяться на етапі проектування, і їхня взаємодія один з одним служить механізмом відображення поводження більш високого рівня.

Існують такі об'єкти, для яких визначені явні концептуальні границі, але самі об'єкти становлять невловимі події або процеси. Наприклад, хімічний процес на заводі можна трактувати як об'єкт, тому що він має чітку концептуальну границю, взаємодіє з іншими об'єктами за допомогою впорядкованого й розподіленого в часі набору операцій і проявляє добре визначене поводження. Розглянемо систему просторового проектування CAD/CAM. Два тіла, наприклад, сфера й куб, мають як правило нерегулярне перетинання. Хоча ця лінія перетинання не існує

окремо від сфери й куба, вона все-таки є самостійним об'єктом із чітко певними концептуальними границями.

Подібно тому, як людина, що взяла у руки молоток, починає бачити у навколишньому світі тільки цвяхи, проектувальник з об’єктно-орієнтованим мисленням починає сприймати увесь світ у вигляді об'єктів. Зрозуміло, такий погляд трохи спрощений, тому що існують поняття, що явно не є об'єктами. До їхнього числа відносяться атрибути, такі, як час, краса, колір, емоції (наприклад, любов або гнів). Однак, потенційно все перераховане – це властивості, властиві об'єктам. Можна, наприклад, стверджувати, що деяка людина (об'єкт) любить свою дружину (інший об'єкт), або що конкретний кіт (ще один об'єкт) – сірий.

Корисно розуміти, що об'єкт – це щось, що має чітко певні границі, але цього недостатньо, щоб відокремити один об'єкт від іншого або дати оцінку якості абстракції. На основі наявного досвіду можна дати наступне визначення: *Об'єкт має стан, поводження і ідентичність; структура й поводження схожих об'єктів визначає загальний для них клас; терміни "екземпляр класу" і "об'єкт" взаємозамінні.*

Стан

Розглянемо торговельний автомат, що продає напої. Поводження такого об'єкта полягає в тому, що після опускання в нього монети й натискання кнопки автомат видає обраний напій. Що відбудеться, якщо спочатку буде натиснута кнопка вибору напою, а потім уже опущена монета? Більшість автоматів при цьому просто нічого не зроблять, тому що користувач порушив їхні основні правила.

Інакше кажучи, автомат відігравав роль (очікування монети), що користувач ігнорував, нажавши спочатку кнопку. Або припустимо, що користувач автомата не звернув увагу на попереджуючий сигнал "Киньте стільки дрібних грошей, скільки коштує напій" і опустив в автомат зайву монету. У більшості випадків автомати не дружні до користувача й радісно "заковтують" всі гроші.

У кожній з таких ситуацій бачимо, що поводження об'єкта визначається його історією: важлива послідовність чинених над об'єктом дій. Така залежність поводження від подій і від часу пояснюється тим, що в об'єкта є внутрішній стан. Для торговельного автомата, наприклад, стан визначається сумою грошей, опущених до натискання кнопки вибору. Інша важлива інформація – це набір сприйманих монет і запас напоїв.

На основі цього прикладу можна дати наступне визначення: с*тан об'єкта характеризується переліком (звичайно статичним) всіх властивостей даного об'єкта й поточними (звичайно динамічними) значеннями кожного із цих властивостей.*

Однією із властивостей торговельного автомата є здатність приймати монети. Це статична (фіксована) властивість, у тому розумінні, що вона – істотна характеристика торговельного автомата. З іншого боку, цій властивості відповідає динамічне значення, що характеризує кількість прийнятих монет. Сума збільшується в міру опускання монет в автомат і зменшується, коли продавець забирає гроші з автомата. У деяких випадках значення властивостей об'єкта можуть бути статичними (наприклад, заводський номер автомата), тому в даному визначенні використаний термін "звичайно динамічними". Всі властивості мають деякі значення. Ці значення можуть бути простими кількісними характеристиками, а можуть посилатися на інший об'єкт.

Поводження

Об'єкти не існують ізольовано, а піддаються впливу або самі впливають на інші об'єкти. *Поводження – це те, як об'єкт діє й реагує; поводження виражається в термінах стану об'єкта й передачі повідомлень.*

Іншими словами, поводження об'єкта – це його діяльність, що спостерігається й ззовні перевіряється. Операцією називається певний вплив одного об'єкта на іншій з метою викликати відповідну реакцію.

Природа класів

Що таке клас?

Поняття класу й об'єкта настільки тісно зв'язані, що неможливо говорити про об'єкт безвідносно до його класу. Однак існує важливе розходження цих двох понять. У той час як об'єкт позначає конкретну сутність, певну в часі й у просторі, клас визначає лише абстракцію істотного в об'єкті. Таким чином, можна говорити про клас "Ссавці", що включає характеристики, загальні для всіх ссавців. Для вказівки на конкретного представника ссавців необхідно сказати "це – ссавець" або "то – ссавець".

У контексті об’єктно-орієнтованого аналізу дамо наступне визначення класу: *Клас – це якась безліч об'єктів, що мають загальну структуру й загальне поводження.*

Будь-який конкретний об'єкт є просто екземпляром класу. Що ж не є класом? Об'єкт не є класом, хоча надалі побачимо, що клас може бути об'єктом. Об'єкти, не зв'язані спільністю структури й поводження, не можна об'єднати в клас, тому що по визначенню вони не зв'язані між собою нічим, крім того, що всі вони об'єкти.

Класи – серце кожної об’єктно-орієнтованої мови.

Визначення класів

Синтаксис визначення класів на C#, простий. Помістивши перед ім'ям вашого класу ключове слово *class,* ви вставляєте члени класу, укладені у фігурні дужки, наприклад:

```
class Employee {
private long employeeId; }
```

Як бачите, цей найпростіший клас із ім'ям *Employee* містить єдиний член — *employeeId.* Помітьте: ім'я члена передує ключове слово *private* — це *модифікатор доступу* (access modifier).

Члени класу

Члени класу C# бувають наступних видів.

Поле. Так називається член-змінна, яка має деяке значення. В ООП поля іноді називають даними об'єкта. До поля можна застосовувати кілька модифікаторів залежно від того, як ви збираєтеся це поле використовувати. У число модифікаторів входять *static, readonly* і *const.* Нижче познайомимося з їхнім призначенням і способами їхнього застосування.

Метод. Це реальний код, що впливає на дані об'єкта (або поля). Фактично, методи – це функції, які визначають певні дії з даними.

Властивості. Їх іноді називають *"розумними" полями* (smart fields), тому що вони насправді є методами, які клієнти класу сприймають як поля. Це забезпечує клієнтам більший ступінь абстрагування за рахунок того, що їм не потрібно знати, чи звертаються вони до поля прямо або через виклик метода-аксессора.

Константи. Як можна припустити, виходячи з ім'я, константа — це поле, значення якого змінити не можна. Нижче обговоримо константи й зрівняємо їх із сутністю за назвою *незмінні* (readonly) поля.

Індексатори. Якщо властивості — це "розумні" поля, то індексатори – це "розумні" масиви, тому що вони дозволяють індексувати об'єкти методами-аксессорами *get* і *set.* За допомогою індексатора легко проіндексувати об'єкт для установки або одержання значень.

Події. Подія викликає виконання деякого фрагмента коду. Події – невід'ємна частина програмування для Microsoft Windows. Наприклад, події виникають при русі миші, клацанні або зміні розмірів вікна.

Оператори. Використовуючи перевантаження операторів C#, можна додавати до класу стандартні математичні оператори, які дозволяють писати більш інтуїтивно зрозумілий код.

Модифікатори доступу

Познайомимося з модифікаторами, що використовуються для завдання ступеня "видимості", або доступності даного члена для коду, що лежить за межами його власного класу. Вони називаються *модифікаторами доступу* (access modifiers) і наведені в табл. 2.1.

Таблиця 2.1

Модифікатори доступу

Модифікатор доступу	Опис
public	Член доступний поза визначенням класу й ієрархії похідних класів
protected	Член недоступний за межами класу, до нього можуть звертатися тільки похідні класи
private	Член недоступний за межами області видимості класу, в якому його визначили. Тому доступу до цих членів немає навіть у похідних класів
internal	Член доступний тільки в межах поточної одиниці компіляції. Модифікатор доступу *internals* в плані обмеження доступу є гібридом *public* і *protected,* залежним від місця розташування коду

Конструктори

Одна з найбільших переваг мов ООП, таких, як C#, полягає в тому, що ви можете визначати спеціальні методи, які викликаються щораз при створенні екземпляра класу. Ці методи називаються *конструкторами* (constructors). C# вводить в ужиток новий тип конструкторів – *статичні* (static constructors).

Гарантія ініціалізації об'єкта належним чином, перш ніж він буде використаний, – ключова вигода від конструктора. Коли користувач створює екземпляр об'єкта, викликається його конструктор, що повинен повернути керування, перш ніж користувач зможе виконати над об'єктом іншу дію. Саме це допомагає забезпечувати цілісність об'єкта й зробити написання додатків на об’єктно-орієнтованих мовах набагато надійніше.

Як і в мові C++, у конструкторів у мові C# повинне бути те ж ім'я, що і у самого класу. Ось простий клас із таким же простим конструктором:

```
class ConstructorApp {
ConstructorApp()
{
Console.WriteLine("Я конструктор.");
}
public static void Main() {
ConstructorApp app = new ConstructorApp(); }
```

Значень конструктори не повертають. При спробі використовувати з конструктором як префікс ім'я типу, компілятор повідомить про помилку.

Варто звернути увагу і на спосіб створення екземпляру об'єкта у C#. Це робиться за допомогою ключового слова *new:*

<клас> <об'єкт> = new *<клас> (аргументи конструктора)*

Ініціалізатори конструкторів

У всіх конструкторах C#, крім *System.Object,* конструктори базового класу викликаються прямо перед виконанням першого рядка конструктора. Ці ініціалізатори конструкторів дозволяють задавати клас і підлягаючий виклику конструктор. Вони бувають двох видів.

Ініціалізатор у вигляді *base(...)* активізує конструктор базового класу поточного класу.

Ініціалізатор у вигляді *this(...)* дозволяє поточному базовому класу викликати інший конструктор, певний у ньому самому. Це корисно, коли ви перевантажили трохи конструкторів і хочете бути впевненими, що завжди буде викликаний конструктор за замовчуванням. Перевантаженими називаються два й більше методи з однаковим ім'ям, але з різними списками аргументів.

Щоб побачити порядок подій у дії, зверніть увагу на наступний код: він спочатку виконає конструктор класу A, а потім конструктор класу B:

```
class A
    {
        public A()
        {
            Console.WriteLine("A");
        }
    }
class B : A
    {
        public B()
        {
            Console.WriteLine("B");
        }
    }
class DefaultlnitializerApp
    {
        public static void Main()
        {
            B b = new B();
        }
    }
```

Цей код – функціональний еквівалент наступного, де є явний виклик конструктора базового класу:

```
class A
    {
        public A()
        {
            Console.WriteLine("A");
        }
    }
class B : A
    {
        public B()
            : base()
        {
            Console.WriteLine("B");
        }
    }
class BaseDefaultlnitializerApp
    {
        public static void Main()
        {
            B b = new B();
        }
    }
```

А тепер розглянемо більш вдалий приклад ситуації, коли вигідно використовувати ініціалізатори конструкторів. Розглянемо знову два класи: A і B. Цього разу в класі A два конструктори, один з них не вимагає аргументів, а інший приймає аргумент типу *int.* У класі B один конструктор, що приймає аргумент типу *int.* При створенні класу B виникає проблема. Якщо запустити наступний код, буде викликаний конструктор класу A, який не приймає аргументів:

```
class A
    {
        public A()
        {
            Console.WriteLine("A");
        }
        public A(int foo)
        {
            Console.WriteLine("A = {0}", foo);
        }
    }

class B : A
    {
        public B(int foo)
        {
            Console.WriteLine("B = {0}", foo);
        }
    }
class DerivedInitializer2App
    {
        public static void Main()
        {
            B b = new B(42);
        }
    }
```

Як же гарантувати, що буде викликаний саме потрібний конструктор класу A? Потрібно явно вказати компіляторові, який конструктор в ініціалізаторі повинен бути викликаний першим, наприклад, так:

```
class A
    {
        public A()
        {
            Console.WriteLine("A");
        }
        public A(int foo)
        {
```

```
            Console.WriteLine("A = {0}", foo);
        }
    }

class B : A
    {
        public B(int foo): base(foo)
        {
            Console.WriteLine("B = {0}", foo);
        }
    }
class DerivedInitializer2App
    {
        public static void Main()
        {
            B b = new B(42);
        }
    }
```

Константи й незмінні поля

Можна із упевненістю сказати, що виникнуть ситуації, коли зміна деяких полів при виконанні додатка буде небажана. Наприклад, це можуть бути файли даних, від яких залежить ваш додаток, значення *pi* для математичного класу або будь-яке інше використовуване в додатку значення, про яке ви знаєте, що воно ніколи не зміниться. У цій ситуації C# дозволяє визначати члени двох тіснопов'язаних типів: константи і незмінні поля.

Константи

З назви легко здогадатися, що *константи* (constants), представлені ключовим словом *const,* – це поля, що залишаються постійними протягом усього часу життя додатка. Визначаючи що-небудь як *const,* досить пам'ятати два правила. По-перше, константа – це член, значення якого встановлюється в період компіляції програмістом або компілятором (в останньому випадку це значення за умовчанням). По-друге, значення члена-константи повинне бути записане у вигляді літерала.

Щоб визначити поле як константу, укажіть перед обумовленим членом ключове слово *const:*

```
using System;
class MagicNumbers {
public const double pi = 3.1415;
public const int answerToAllLifesQuestions = 42; }
```

Glass ConstApp {
public static void Main () {
Console.WriteLine("pi = {0}, все інше = {1}",
MagicNumbers.pi, MagicNumbers.answerToAllLifesQuestions); } }

Зверніть увагу на один важливий момент, пов'язаний із цим кодом. Клієнтові немає потреби створювати екземпляр класу *MagicNumbers,* оскільки за замовчуванням члени *const* є статичними. Щоб одержати більш чітке подання про предмет, гляньте на MSIL-Код, згенерований для цих двох членів:

answerToAHLifesQuestions : public static literal int32
=int32(Ox0000002A) pi :public static literal float64
=float64(3.14150000000000002)

Незмінні поля

Поле, визначене як *const,* ясно вказує, що програміст має намір помістити в нього постійне значення. Але воно працює, тільки якщо відомо значення подібного поля в період компіляції. А що ж робити, коли виникає потреба в полі, чиє значення не відомо до періоду виконання, але після ініціалізації не повинне мінятися? Ця проблема вирішена розроблювачами мови C# за допомогою *незмінного поля* (read-only field).

Визначаючи поле за допомогою ключового поля *readonly,* ви можете встановити значення поля лише в одному місці – у конструкторі. Після цього поле не можуть змінити ні сам клас, ні його клієнти. Допустимо, для графічного додатка потрібно відслідковувати роздільну здатність екрана. Вирішити цю проблему за допомогою *const* не можна, тому що до періоду виконання додаток не може визначити роздільну здатність екрана у користувача. Тому для цієї мети найкраще використовувати такий код:

```
class GraphicsPackage
    {
        public readonly int ScreenWidth;
        public readonly int ScreenHeight;
        public GraphicsPackage()
        {
            this.ScreenWidth = 1024;
            this.ScreenHeight = 768;
        }
    }
    class ReadOnlyApp
```

```
    {
        public static void Main()
        {
            GraphicsPackage graphics = new GraphicsPackage();
            Console.WriteLine("Ширина = {0}, Висота = {1}",
     graphics.ScreenWidth, graphics.ScreenHeight);
        }
     }
```

На перший погляд здається, що це те, що потрібно. Але тут одна маленька проблема: визначені авторами незмінні поля є полями екземпляра, а виходить, щоб задіяти ці поля, користувачеві потрібно створювати екземпляри класу. Може, це й не проблема, і цей код може навіть використовуватися, коли значення незмінного поля визначається способом створення екземпляра класу. Але якщо вам потрібна константа, по визначенню статична, але та, що ініціалізується в період виконання? Тоді потрібно визначити поле з обома модифікаторами – *static* і *readonly,* а потім створити особливий – статичний – тип конструктора. *Статичні конструктори* (static constructor) використовуються для ініціалізації статичних, незмінних і інших полів. Змінимо попередній приклад так, щоб зробити поля, що визначають дозвіл екрана, статичними й незмінними, а також додамо статичний конструктор. Зверніть увагу на ключове слово *static,* додане до визначення конструктора:

```
class GraphicsPackage
    {
        public static readonly int ScreenWidth;
        public static readonly int ScreenHeight;
        static GraphicsPackage()
        {
            // Тут буде код для розрахунку роздільної здатності
екрана.
            ScreenWidth = 1024;
            ScreenHeight = 768;
        }
    }

    class ReadOnlyApp
    {
        public static void Main()
        {
            Console.WriteLine("Ширина = {0}, Висота = {1}",
GraphicsPackage.ScreenWidth, GraphicsPackage.ScreenHeight);

        }
    }
```

Таким чином, поняття про класи і їхні взаємини з об'єктами – основа ідеї об'єктного програмування. Об'єкт має стан, поводження і ідентичність; структура й поводження схожих об'єктів визначає загальний для них клас; терміни "екземпляр класу" і "об'єкт" взаємозамінні. Стан об'єкта характеризується переліком (звичайно статичним) всіх властивостей даного об'єкта й поточними (звичайно динамічними) значеннями кожного із цих властивостей. Клас – це якась безліч об'єктів, що мають загальну структуру й загальне поводження.

C# додає деякі нові засоби, не передбачені в класичній моделі об’єктно-орієнтованого програмування: статичні конструктори, які надають можливість ініціалізації статичних полів.

2.3. Створення та руйнування об’єктів

У сучасному програмуванні на C# велику роль відіграє використання збирання сміття в пам'яті і її звільнення, що забезпечує раціональніше використання ресурсів представленої пам'яті для програмного коду і додатку.

Для початку необхідно з'ясувати, як середовище CLR управляє вже розміщеними об'єктами за допомогою процесу, який називається *збиранням сміття.* Програмістам, які використовують C#, не доводиться видаляти об'єкти з пам'яті "уручну".

Об'єкти .NET розміщуються в області пам'яті, яка називається *керованою динамічною пам'яттю,* де ці об'єкти "в деякий відповідний момент" будуть автоматично знищені складальником сміття.

Клас – це своєрідний "шаблон" з описом того, як екземпляр даного типу повинен виглядати і поводитися в пам'яті. Розглянемо простий клас Car (автомобіль).

```
public class Car
    {
        private int currSp;
        private string petName;
        public Car() { }
        public Car(String name, int speed)
        {
            petName = name;
            currSp = speed;
        }
        public override string ToString()
```

```
        {
            return string.Format("{0} має швидкість {1} км/ч",
petName, currSp);
        }
     }
```

Визначивши клас, ви можете розмістити в пам'яті будь-яке число відповідних об'єктів, використовуючи ключове слово C# new. При цьому, проте, слід розуміти, що ключове слово new повертає посилання на об'єкт у динамічній пам'яті, а не сам реальний об'єкт. Ця змінна з посиланням запам'ятовується в стеку для використання в додатку надалі. Для виклику членів об'єкта слід застосувати до збереженого посилання операцію C#, що позначається крапкою.

```
class Program
    {
        static void Main(string[] args)
        {
            Car refToMyCar = new Car("Lanos", 50);
            Console.WriteLine(refToMyCar.ToString());
            Console.ReadLine();
        }
    }
```

Основні відомості про існування об'єктів

При побудові C#-додатків ви маєте право припускати, що керована динамічна пам'ять оброблятиметься без вашого прямого втручання. Правило управління пам'яттю .NET є дуже простим – слід помістити об'єкт в керовану динамічну пам'ять за допомогою ключового слова new і забути про це.

Складальник сміття знищить створений об'єкт, коли цей об'єкт більше не буде потрібний. Очевидне питання: "Як складальник сміття визначає, що об'єкт більше не потрібний?" Коротка, тобто спрощена відповідь полягає в тому, що складальник сміття видаляє об'єкт з динамічної пам'яті тоді, коли об'єкт стає недоступним для всіх частин програмного коду. Припустимо, що ви маєте метод, що розміщує локальний об'єкт Car.

```
public static Void MakeACar ()
{
```

```
// Якщо myCar є єдиним посиланням на об'єкт Car,
//то об'єкт може бути знищений після повернення з методу.
Car myCar = new Car () ;
}
```

Посилання на об'єкт (myCar) було створене безпосередньо в методі MakeACar () і не передавалася за межі області видимості, що визначає це посилання об'єкта ні у вигляді повертаного значення, ні у вигляді параметрів ref/out. Тому після завершення роботи викликаного методу посилання myCar стає недоступним і відповідний об'єкт Car виявляється кандидатом для видалення в "сміття".

Проте слід відмітити, що не можна гарантувати негайне видалення цього об'єкта з пам'яті відразу ж після закінчення роботи MakeACar (). У цей момент можна гарантувати тільки те, що при наступній збірці сміття в загальномовному середовищі виконання (CLR) об'єкт myCar може бути без побоювань знищений.

Програмування в оточенні, що забезпечує автоматичну збірку сміття, значно спрощує завдання розробки додатків. Програмісти, які використовують C++, знають про те, що якщо в C++ забути уручну видалити розміщені в динамічній пам'яті об'єкти, може відбутися "витік пам'яті".

Насправді ліквідація витоків пам'яті є одним із найбільш трудомістких аспектів програмування мовами, які не є керованими. Доручивши складальникові сміття знищення об'єктів, ви знімаєте з себе вантаж відповідальності за управління пам'яттю і перекладаєте його на CLR.

CIL-*код для new*

Коли компілятор C# виявляє ключове слово new він генерує CIL інструкцію newobj в рамках реалізації відповідного методу. Якщо виконати компіляцію програмного коду поточного прикладу і за допомогою ildasm.exe розглянути отриманий компонувальний блок, то в рамках методу MakeACar () ви побачите такі CIL оператори

```
.method public hidebusig static void MakeACar () cil managed
{
// Code size   7 (0x7)
.maxstack   1
```

```
.locals init ([0] class SimpleFinalize.Car c)
IL_0000 IL_0005 IL 0006
newobj instance void   SimpleFinalize.Car:
stloCpOret }
} // end of method Program:.MakeACar
```

Перш ніж обговорити точні правила, що визначають момент видалення об'єкта з керованої динамічної пам'яті, давайте з'ясуємо роль CIL-інструкції newobj.

Керована динамічна пам'ять є не просто випадковим фрагментом пам'яті, доступної для середовища виконання. Складальник сміття .NET є виключно акуратним "двірником" у динамічній пам'яті — він (за необхідності) навіть стискає порожні блоки пам'яті з метою оптимізації. Щоб спростити завдання збірки сміття, керована динамічна пам'ять має покажчик (зазвичай званий *покажчиком на наступний об'єкт,* або *покажчиком на новий об'єкт),* який ідентифікує точне місце розміщення наступного об'єкта.

Інструкція newobj інформує середовище CLR про те, що необхідно виконати наступні головні завдання.

Обчислити загальний об'єм пам'яті, необхідної для розміщення об'єкта (включаючи пам'ять, необхідну для членів-змінних і базових класів типу).

Перевірити керовану динамічну пам'ять, щоб гарантувати достатній об'єм пам'яті для розміщуваного об'єкта. Якщо пам'яті досить, викликається конструктор типу, і стороні, яка викликала конструктор, повертається посилання на новий об'єкт у пам'яті, причому адреса цього об'єкта відповідатиме останній позиції покажчика на наступний об'єкт.

Нарешті, перед поверненням посилання стороні, яка викликала, необхідно змінити значення покажчика на наступний об'єкт, щоб покажчик відповідав початку вільної області керованої динамічної пам'яті.

Генерації об'єктів

Коли середовище CLR намагається знайти недоступні об'єкти, це не означає, що буде розглянутий буквально кожен об'єкт, розміщений у керованій динамічній пам'яті. Очевидно, що це вимагало б дуже багато часу, особливо в реальних (тобто великих) додатках.

Щоб оптимізувати процес, кожен об'єкт у динамічній пам'яті приписується певній "генерації". Ідея достатньо проста: чим довше об'єкт існує в динамічній пам'яті, тим більше вірогідно те, що він винен там і залишатися. Наприклад, об'єкт, реалізовуючий Main (), знаходитиметься в пам'яті до тих пір, поки програма не закінчиться. З іншого боку, об'єкти, які недавно розміщені в динамічній пам'яті, найімовірніше, стануть незабаром недосяжними (наприклад, об'єкти, створені в рамках області видимості методу). При цих припущеннях кожен об'єкт у динамічній пам'яті можна віднести до однієї з наступних категорій.

Генерація 0. Нові, тільки що розміщені об'єкти, які ще ніколи не призначалися для використання в процесі збірки сміття.

Генерація 1. Об'єкти, які "пережили" одну збірку сміття (тобто були позначені для використання в процесі збірки сміття, але не були видалені з тієї причини, що в динамічній пам'яті опинилося достатньо місця).

Генерація 2. Об'єкти, які "пережили" декілька збірок сміття.

Складальник сміття спочатку розглядає об'єкти генерації 0. Якщо в результаті виявлення непотрібних об'єктів і відповідного чищення вільної пам'яті виявляється достатньо, об'єкти, що ще залишилися, відносяться до генерації 1.

Якщо всі об'єкти генерації 0 вже розглянуті, але пам'яті все одно ще не достатньо, то розглядається "досяжність" об'єктів генерації 1 і виконується збірка сміття серед цих об'єктів. Об'єкти генерації 1, що "вижили", переходять до генерації 2. Якщо складальник сміття *все ще* вимагає додаткової пам'яті, тоді оцінюються об'єкти генерації 2. Тут, якщо об'єкт генерації 2 "виживає" в процесі збірки сміття, то такий об'єкт зберігає приналежність до генерації 2, оскільки це межа для генерацій об'єктів.

Отже, за допомогою призначення ознаки генерації об'єктам у динамічній пам'яті новіші об'єкти (наприклад, локальні змінні) віддалятимуться швидше, тоді як старі об'єкти (такі, як, наприклад, об'єкт додатку програми) "турбуватимуться" значно рідше.

Тип System.GC

Бібліотеки базових класів пропонують тип класу System.GC, який дозволяє програмно взаємодіяти з складальником сміття, використовуючи безліч статичних членів вказаного класу. Слід відмітити, що безпосередньо

використовувати цей тип у програмному коді доводиться дуже рідко (якщо доводиться взагалі). Найчастіше члени типу System.GC використовуються тоді, коли створюються типи, що використовують некеровані ресурси.

Collect () – Змушує GC виконати збірку сміття.

AddMemoryPressure (), RemoveMemoryPressure () – Дозволяють вказати числове значення, що характеризує "терміновість" виклику процесу збірки сміття.

CollectionCount() – Повертає числове значення, вказуюче, скільки разів "виживала" дана генерація при збірці сміття.

GetGeneration() – Повертає інформацію про генерацію, до якої зараз відноситься об'єкт.

GetTotalMemory() – Повертає оцінку об'єму пам'яті (у байтах), виділеної для керованої динамічної пам'яті зараз. Логічний параметр указує чи повинен виклик чекати початку складання сміття, щоб повернути результат.

MaxGeneration – Повертає максимум для числа генерацій, підтримуваних у системі. У Microsoft .NET 2.0, передбачається існування трьох генерацій (0, 1 і 2).

SuppressFinalize() – Встановлює індикатор того, що даний об'єкт не повинен викликати свій метод Finalize ().

WaitForPendingFinalizers() – Припиняє виконання поточного потоку, поки не будуть відпрацьовані всі об'єкти, які передбачають фіналізацію. Цей метод зазвичай викликається безпосередньо після виклику GC.Collect().

Розглянемо наступний метод Main(), у якому ілюструється використання вказаних членів System.GC.

```
static void Main(string[] args)
        {
            // Виведення оцінки (у байтах) для динамічної пам'яті.
            Console.WriteLine("Оцінка об'єму пам'яті (у байтах) ;
{0}", GC.GetTotalMemory(false));
            // Відлік для MaxGeneration починається з нуля, //
тому для зручності додаємо 1.
            Console.WriteLine("Число генерацій для даної ОС: {0}",
GC.MaxGeneration + 1);
            Car refToMyCar = new Car("Lanos", 100);
            Console.WriteLine(refToMyCar.ToString());
```

```
            // Виведення інформації про генерацію для об'єкта
refToMyCar.
            Console.WriteLine("Генерація RefToMyCar: {0}",
GC.GetGeneration(refToMyCar));
        }
```

Активізація збірки сміття

Отже, складальник сміття в .NET покликаний управляти пам'яттю за вас. Проте в дуже окремих випадках, перерахованих нижче, буває вигідно програмно активізувати початок збірки сміття, використовуючи для цього GC.Collect():

перед входом додатка в блок програмного коду, для якого небажано, щоб його виконання уривалося можливою збіркою сміття;

після закінчення розміщення дуже великого числа об'єктів, коли ви бажаєте звільнити якомога більше пам'яті.

Якщо ви визнаєте, що буде вигідно виконати збірку сміття, то можете явно почати процес збірки сміття так, як показано нижче.

```
static void Main (string[] args) {
   // Активізація збірки сміття і
   // очікування завершення фіналізації об'єктів.
   GC.Collect ();
   GC. WaitForPendingFinalizers () ;

...}
```

При безпосередній активізації збірки сміття ви повинні викликати GC.WaitForPendingFinalizers(). У рамках цього підходу ви можете бути упевнені, що всі передбачаючі фіналізацію об'єкти обов'язково дістануть можливість виконати всі необхідні завершуючі дії, перш ніж ваша програма продовжить свою роботу. Методу GC.Collect () можна передати числове значення, яке вказує стару генерацію, для якої повинна бути виконана збірка сміття. Наприклад, якщо ви бажаєте повідомити CLR, що слід розглянути тільки об'єкти генерації 0, ви повинні надрукувати наступне.

```
static void Main(string [] args)
{
    // Розглянути тільки об'єкти генерації 0.
```

```
        GC.Collect(0);
        GC.WaitForPendingFinalizers() ;

    }
```

Подібно до будь-якої збірки сміття, виклик GC.Collect () підвищить статус генерацій, що вижили.

2.4. Реалізація поліморфізму в C#

Поліморфізм (polymorphism) (від грецького polymorphos) – це властивість, яка дозволяє одне і те ж ім'я використовувати для вирішення двох або більше схожих, але технічно різних завдань. Метою поліморфізму, стосовно об'єктно-орієнтованого програмування, є використання одного імені для завдання загальних для класу дій. Виконання кожної конкретної дії визначатиметься типом даних.

У більш загальному сенсі, концепцією поліморфізму є ідея "один інтерфейс, безліч методів". Це означає, що можна створити загальний інтерфейс для групи близьких по сенсу дій. Перевагою поліморфізму є те, що він допомагає усунути складність програм, використовуючи один і той же інтерфейс для завдання єдиного класу дій. Вибір же конкретної дії, залежно від ситуації, покладається на компілятор. Програмістові при цьому не потрібно робити цей вибір самому. Потрібно тільки пам'ятати і використовувати загальний інтерфейс.

Поліморфізм може застосовуватися також і до операторів. Фактично у всіх мовах програмування обмежено застосовується поліморфізм, наприклад, в арифметичних операторах.

Ключовим у розумінні поліморфізму є те, що він дозволяє вам маніпулювати об'єктами різного ступеня складності шляхом створення загального для них стандартного інтерфейсу для реалізації схожих дій.

Основні поняття. Поняття поліморфізму

Поліморфізм є найістотнішою властивістю об'єктно-орієнтованого програмування.

Поліморфізм, згідно з Г. Бучу, полягає в тому, що імена можуть відповідати різним класам об'єктів, що входять в один суперклас. Отже,

один об'єкт, відмічений таким ім'ям, може по-різному реагувати на деяку безліч дій.

Проілюструємо це на такому прикладі:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Sample1_1
{
   class Starter
   {
      [STAThread]
      static void Main(string[] args)
      {
         OneWordWriter w1 = new OneWordWriter();
         TwoWordsWriter w2 = new TwoWordsWriter();
         ThreeWordsWriter w3 = new ThreeWordsWriter();
         w1.Write();
         ((OneWordWriter)w3).Write();
         // вывод Three words written
         ((OneWordWriter)w2).Write();
         // вывод Two words
      }
   }
   class OneWordWriter
   {
      public virtual void Write()
      {
         Console.WriteLine("One");
      }
   }

   class TwoWordsWriter : OneWordWriter
   {
      public override void Write()
      {
         Console.WriteLine("Two words");
      }
   }

   class ThreeWordsWriter : TwoWordsWriter
   {
```

```
        public override void Write()
        {
            Console.WriteLine("Three words written");
        }
    }
}
```

У даному прикладі створюємо три класи, "суперклас" OneWordWriter і підкласи TwoWordsWriter і ThreeWordsWriter, кожний з яких має метод Write(). Навіть після приведення відповідних об'єктів в методі Main до "суперкласу" OneWordWriter, кожний з них виконує виклик свого методу Write().

Статичне і динамічне зв'язування

Під поняттям "поліморфізм" розуміють здатність програми виконувати методи об'єктів у точній відповідності з конкретним типом об'єкта. Наприклад, метод "Їхати" об'єкта-екземпляра класу "Запорожець" напевно відрізняється від методу "Їхати" об'єкта класу "Мерседес". Але при цьому імена таких методів співпадають, що нерідко і заплутує розробників.

Крім того допускається зберігання в змінній об'єктного типу (класу) реального об'єкта іншого типу. Типи ці позв'язані деяким чином за ієрархією спадкоємства. Наприклад, класи Car (Автомобіль) і Truck (Вантажівка), спадкоємець класу Car:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Sample2
{
    class Program
    {
        class Car
        {
        public double Benzin; // бензин
        void Go( float dist )
        {
         Benzin -= dist / 10; // 1 літр бензину на 10 км
         Rasst += dist; // проїхали
```

```
      }
      protected double Rasst; // пройдена відстань
     }

  class Truck : Car
  {
    public void Go( float dist )
    {
      Benzin -= dist / 5; // 1 літр бензину на 5 км
      Rasst += dist; // проїхали
    }
  }

      static void Main(string[] args)
      {
         Truck truct = new Truck();
         Car car2 = new Car();
         //Car car1 = new Truck();
         truct.Go(100);
       }
   }
}
```

Стандартні такі об’яви об’єктів:

```
Truck truct = new Truck();
Car car2 = new Car();
```

Однак принцип поліморфізму допускає і такий запис:

```
Car car1 = new Truck();
```

Які конкретно версії методу Go() будуть викликані – класу Car або Truck? Це і визначає принцип поліморфізму. Відповідно до нього в мові C# визначені так звані статичне зв’язування і динамічне зв’язування.

Статичне зв’язування – це зв’язування методу з батьківським об'єктом на етапі компіляції. Оскільки компілятор не знає, як конкретно працюватиме програма, він завжди виходить з відомого і явно заданого типу об'єкта-власника методу. Якщо використовується змінна car класу Car, то буде викликаний метод Go() класу Car, хоча фізично в змінній car зберігається екземпляр класу Truck. Так само і у випадку із змінною truck – хоча автори примусово записали в неї посилання на об'єкт класу Car, компілятор про це не знає і виходить з типу змінної truck – викликає метод Go() класу Truck.

Динамічне зв’язування – це зв’язування методу з об'єктом, яке відбувається під час роботи програми. При цьому потрібний метод визначається вже не типом змінної-власника, а реальним типом об'єкта, посилання на який вона зберігає. Але таке зв’язування можливо тільки для так званих віртуальних методів, – розділення типів методів потрібне, щоб підказати компілятору, де яке зв’язування реалізовувати.

Віртуальні функції

Базові класи можуть визначати і реалізовувати віртуальні методи, а похідні класи можуть перевизначати їх. Це означає, що вони надають свої власні визначення і реалізацію. Під час виконання, коли клієнтський код викликає метод, середовище CLR шукає тип часу виконання об'єкта і викликає це перевизначення віртуального методу. Таким чином, у початковому коді можна викликати метод у базовому класі і викликати виконання методу з версією похідного класу.

Віртуальні методи дозволяють єдиним чином працювати з групами зв'язаних об'єктів. Наприклад, припустимо, є додаток малювання, яке дає можливість користувачу створювати на поверхні малювання різні форми. Під час компіляції невідомо, які конкретні форми створюватиме користувач. Але додаток повинен враховувати всі різні типи створюваних форм, і воно повинне оновлювати їх у відповідь на дії миші користувача. Поліморфізм можна використовувати для вирішення цієї проблеми в два основних етапи.

1. Створення ієрархії класів, в якій клас кожної конкретної форми проводиться від загального базового класу.
2. Використання віртуального методу для виклику відповідного методу в якому-небудь похідному класі одним викликом методу базового класу.

Приклад:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Sample3
{
   public class Shape
```

```
    {
        // Virtual method
        public virtual void Draw()
        {
            Console.WriteLine("Performing base class drawing tasks");
        }
    }

    class Circle : Shape
    {
        public override void Draw()
        {
            // Code to draw a circle...
            Console.WriteLine("Drawing a circle");
            base.Draw();
        }
    }
    class Rectangle : Shape
    {
        public override void Draw()
        {
            // Code to draw a rectangle...
            Console.WriteLine("Drawing a rectangle");
            base.Draw();
        }
    }
    class Triangle : Shape
    {
        public override void Draw()
        {
            // Code to draw a triangle...
            Console.WriteLine("Drawing a triangle");
            base.Draw();
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            System.Collections.Generic.List<Shape> shapes = new
System.Collections.Generic.List<Shape>();
            shapes.Add(new Rectangle());
```

```
            shapes.Add(new Triangle());
            shapes.Add(new Circle());

            foreach (Shape s in shapes)
            {
               s.Draw();
            }
        }
    }
}
```

У даному прикладі видно: для оновлення поверхні малювання використовується цикл foreach для ітерації списку і виклику методу Draw на кожному об'єкті Shape в списку. Хоча кожен об'єкт в списку має оголошений тип Shape, викликатиметься саме тип часу виконання (перевизначена версія методу в кожному похідному класі).

Запобігання перевизначенню віртуальних членів похідними класами

Віртуальні члени залишаються віртуальними без будь-яких обмежень відносно кількості класів, оголошених між віртуальним членом і класом, у якому він був спочатку оголошений. Якщо в класі A оголошується віртуальний член і клас B є похідним від класу A, а клас C є похідним від класу B, то клас C успадковує віртуальний член і забезпечує можливість його перевизначення незалежно від того, чи було в класі B оголошено перевизначення цього члена.

Похідний клас може припинити спадкоємство, оголосивши перевизначення як sealed. Для цього потрібно додати ключове слово sealed перед ключовим словом override в оголошенні члена класу.

Наприклад:

```
public class C : B
{
   public sealed override void DoWork() { }
}
```

Тобто метод DoWork більше не є віртуальним для будь-якого похідного від класу C класу. Він залишається віртуальним для екземплярів класу C, навіть якщо вони приведені до типу B або A.

Закриті методи можна заміщати в похідних класах за допомогою ключового слова new, як показано в наступному прикладі:

```
public class D : C
{
   public new void DoWork() { }
}
```

У цьому випадку при виклику методу DoWork для типу D з використанням змінної типу D викликається новий метод DoWork. Якщо для доступу до екземпляра типу D використовується змінна типу C, B або A, то виклик методу DoWork підкорятиметься правилам віртуального спадкоємства, направляючи виклики реалізації методу DoWork в класі C.

Абстрактні класи

Із спадкоємством тісно пов'язаний ще один важливий механізм проектування сімейства класів – механізм абстрактних класів.

Клас називається абстрактним, якщо він має хоч би один абстрактний метод.

Метод називається абстрактним, якщо при визначенні методу задана його сигнатура, але не задана реалізація методу.

Абстрактні класи мають ряд властивостей:

Такі класи не можуть бути sealed.

Вони не можуть бути інстанційовані оператором new.

При оголошенні методу класу абстрактним мається на увазі що він віртуальний, тому метод не може бути оголошений як virtual.

Усі абстрактні члени повинні бути визначені в класі, що успадковує, за виключення випадку, коли клас, що успадковує так само абстрактний.

Абстрактні методи класу не можуть бути оголошені як static.

При реалізації абстрактним класом інтерфейсу, всі члени інтерфейсу повинні бути або реалізованими, або описані їх сигнатури і оголошені як abstract.

Абстрактний клас може мати як абстрактних, так і не абстрактних членів, взагалі кажучи обмежень на кількість абстрактних членів немає,

тобто їх може не бути зовсім, але тоді сенс оголошення класу абстрактним втрачається.

Оголошені абстрактними можуть бути методи, властивості, індексатори і події.

Абстрактний клас може реалізовувати декілька інтерфейсів і може бути спадкоємцем класу, при цьому він може перевизначати віртуальні методи неабстрактного класу.

Клас, що успадковує при перевизначенні абстрактної властивості зобов'язаний реалізовувати одного з аксесорів. Якщо перевизначається абстрактна властивість, що має тільки один аксесор, то перевизначатися може тільки він. Якщо абстрактну властивість мають обидва аксесори, то перевизначати може як один, так і обидва.

Оголошення абстрактних методів і абстрактних класів повинне супроводжуватися модифікатором abstract. Оскільки абстрактні класи не є повністю певними класами, то не можна створювати об'єкти абстрактних класів. Абстрактні класи можуть мати нащадків, що частково або повністю реалізовують абстрактні методи батьківського класу. Абстрактний метод частіше за все розглядається як віртуальний метод, що перевизначається нащадком, тому до нього застосовується стратегія динамічного скріплення.

Абстрактні класи є одним з найважливіших інструментів об'єктно-орієнтованого проектування класів. У основі будь-якого класу лежить абстракція даних. Абстрактний клас описує цю абстракцію, не входивши в деталі реалізації, обмежуючись описом тих операцій, які можна виконувати над даними класу. Так, проектування абстрактного класу Stack, що описує стек, може складатися з розгляду основних операцій над стеком і не визначати, як буде реалізований стек – списком або масивом. Два нащадки абстрактного класу – ArrayStack і ListStack – можуть бути вже конкретними класами, заснованими на різних представленнях стека.

Ось опис повністю абстрактного класу Stack:

```
public abstract class Stack
{
  public Stack()
  { }
  public abstract void put(int item);
  public abstract void remove();
```

```
    public abstract int item();
    public abstract bool IsEmpty();
}
```

Опис класу містить тільки сигнатури методів класу і їх специфікацію, задану тегами <summary>. Побудуємо тепер одного з нащадків цього класу, реалізація якого заснована на списковому уявленні. Клас ListStack буде нащадком абстрактного класу Stack і клієнтом класу Linkable, що задає елементи списку.

Клас Linkable виглядає таким чином:

```
public class Linkable
{
    public Linkable()
    { }
    public int info;
    public Linkable next;
}
```

У ньому – два поля і конструктор за умовчанням. Побудуємо тепер клас ListStack:

```
public class ListStack : Stack
{
    public ListStack()
    {
        top = new Linkable();
    }
    Linkable top;
    public override void put(int item)
    {
        Linkable newitem = new Linkable();
        newitem.info = item;
        newitem.next = top;
        top = newitem;
    }
    public override void remove()
    {
        top = top.next;
    }
    public override int item()
    {
```

```
        return (top.info);
    }
    public override bool IsEmpty()
    {
        return (top.next == null);
    }
     }
```

Клас має одне поле top класу Linkable і методи, успадковані від абстрактного класу Stack.

Приклад роботи із стеком:

```
static void Main(string[] args)
    {
        ListStack stack = new ListStack();
        stack.put(7); stack.put(9);
        Console.WriteLine(stack.item());
        stack.remove(); Console.WriteLine(stack.item());
        stack.put(11); stack.put(13);
        Console.WriteLine(stack.item());
        stack.remove(); Console.WriteLine(stack.item());
        if (!stack.IsEmpty()) stack.remove();
        Console.WriteLine(stack.item());

    }
```

Інтерфейси

Поліморфізм може бути реалізований за допомогою інтерфейсів.

Властивості інтерфейсів:

клас може реалізовувати більше одного інтерфейсу;

інтерфейс не містить реалізації методів;

метод класу, що реалізує метод інтерфейсу, повинен бути не статичним, оголошеним із специфікатором доступу public, мати ту ж сигнатуру;

реалізовуючий інтерфейс клас може не дотримуватися політики реалізації аксесорів властивостей інтерфейсу, за винятком ситуації, коли інтерфейс оголошений як explicit;

інтерфейс не може інстанціюватися оператором new;

інтерфейси можуть містити методи, властивості, індексатори і події.

Приклад реалізації:

```
using System;
using System.Collections.Generic;
using System.Text;

    namespace Sample5
    {
       class Program
       {
          interface IFileTypesProvider
          {
             void Open();
             void Save();
          }

          class CFileNewtype : IFileTypesProvider
          {
             public void Open()
             {
                Console.WriteLine("Open method for Newtype File");
             }

             public void Save()
             {
                Console.WriteLine("Save method for Newtype File");
             }
          }

          class CFileOldtype : IFileTypesProvider
          {
             public void Open()
             {
                Console.WriteLine("Open method for Oldtype File");
             }

             public void Save()
             {
                Console.WriteLine("Save method for Oldtype File");
             }
          }
          static void Main(string[] args)
          {
```

```
        CFileNewtype type = new CFileNewtype();
        type.Open();
        type.Save();
      }
   }
}
```

Так, коли варто застосовувати реалізацію поліморфізму через абстрактні класи, а коли через інтерфейси?

Враховуючи відсутність множинного спадкоємства, реалізація декількох інтерфейсів рятує положення.

Абстрактні класи не підходять для value types. Таким чином, поліморфізм для структур тільки через інтерфейси.

Абстрактні класи можуть містити реалізацію внутрішніх, не абстрактних методів без порушення правил спадкоємства, проте додавання нового абстрактного методу порушить всі зв'язки із спадкоємцями, в цьому випадку, допоможе додавання нового інтерфейсу.

Інтерфейс – це чергова синтаксична конструкція, яка здатна спростити вихідний код, зробивши його більш зрозумілим і логічним. Напевно, багато хто з вас вже знайомі з поняттям інтерфейсу. Простий приклад: розетка та вилка. У розетки є свій інтерфейс – це два отвори певного діаметру, розташовані на певній відстані один від одного. Якщо ви хочете, щоб ваш пристрій живився від мережі, вам необхідно реалізувати цей інтерфейс. Результатом реалізації цього інтерфейсу є вилка з двома ніжками. Якщо інтерфейс буде реалізований неправильно (наприклад, відстань між ніжками менша, ніж потрібно), то це призведе до помилки (ваш пристрій не зможе живитися від мережі).

Автори вважають, що вже зрозуміло, навіщо потрібні інтерфейси. Інтерфейси відповідають на питання "Як це повинно працювати?" або "Як це повинно виглядати?". Причому, зверніть увагу, що це не рекомендації, а жорсткі правила, недотримання яких призведе до неправильної роботи. Інтерфейси дуже схожі на абстрактні класи, але вони мають відмінності і служать для інших цілей. Інтерфейс не відповідає на питання "Як робити?". Давайте на практиці розглянемо будь-який інтерфейс:

```
      public interface Car
   {
      int Speed{get; set;}
      void GetInfo();
   }
```

Створено інтерфейс Car (машина). Зверніть увагу на ключове слово interface. Даний інтерфейс "говорить" про те, що в класі, який буде реалізовувати цей інтерфейс, має бути властивість Speed (швидкість) і метод GetInfo (одержати інформацію). Як бачите, інтерфейс не містить інформації, яка б демонструвала те, як будуть працювати даний метод і властивість. Зверніть увагу і на те, що властивість Speed доступна як для запису, так і для читання. Якщо необхідно зробити властивість тільки для читання, потрібно прибрати метод set. Для того, щоб зробити властивість тільки для запису потрібно прибрати метод get.

Тепер у нас є інтерфейс машини (грубо кажучи). Зараз необхідно реалізувати цей інтерфейс і відповісти на питання "Як це працює?". Давайте створимо клас ferrari (для тих, хто не знає, це марка італійського автомобіля). Даний клас повинен реалізовувати інтерфейс Car (все-таки Феррарі теж машина). Зробити це можна так:

```
public class ferrari: Car
{
    private int spd; //швидкість
    public int Speed
    {
        get
        {
            return spd;
        }
        set
        {
            spd = value;
        }
    }
    public void GetInfo()
    {
        Console.WriteLine("Це суперкар Ferrari".);
    }
}
```

Тепер у нас є клас, який реалізує інтерфейс Car. Реалізація інтерфейсу за синтаксисом дуже схожа на спадкування, якщо не знати, що Car – це інтерфейс, то можна і переплутати. Обов'язково зверніть увагу на той факт, що клас повинен реалізовувати всі без виключення властивості та методи інтерфейсу. Якщо ви забудете реалізувати будь-який метод або властивість, то при компіляції вам буде показана

помилка, наприклад така: 'ConsoleApplication7.ferrari' does not implement interface member 'ConsoleApplication7.Car.GetInfo ()'.

Один клас може реалізовувати декілька інтерфейсів. Це цілком логічно. Така машина, як Феррарі може бути одночасно як розкішшю, так і засобом пересування. Тому давайте створимо ще один інтерфейс Luxury (розкіш): public interface Luxury

```
        {
    int Price{get; set;}
    void GetInfo();
}
```

А тепер перепишіть наш клас Ferrari таким чином, щоб він реалізовував обидва інтерфейси:

```
public class ferrari: Car, Luxury
  {
    private int spd; //швидкість
    private int prc; //ціна

    public int Speed
    {
       get
       {
          return spd;
       }
       set
       {
          spd = value;
       }
    }

    public void GetInfo()
    {
       Console.WriteLine("Це суперкар Ferrari".);
    }

    public int Price
    {
       get
       {
          return prc;
       }
       set
       {
```

```
            prc = value;
        }
    }
}
```

Тепер клас Ferrari реалізує обидва інтерфейси. Інтерфейсів може бути як завгодно багато, їх необхідно перераховувати через кому, як це показано вище. У даному прикладі є одна дуже цікава річ. Ви маєте звернути на неї увагу. Вся справа в тому, що у обох інтерфейсів є метод GetInfo (). Реалізували ж його всього лише один раз. З синтаксичної точки зору тут все правильно. Але найімовірніше розробник інтерфейсів Car і Luxor очікував іншої поведінки. Як же зробити так, щоб клас Ferrari по-різному реалізовував метод GetInfo () для кожного інтерфейсу? Вихід з даної ситуації, звичайно ж, є. Досить чітко вказати, який інтерфейс реалізує даний метод: void Car.GetInfo()

```
{
    Console.WriteLine("Це суперкар Ferrari ".);
}

void Luxury.GetInfo()
{
    Console.WriteLine("Це Ferrari коштує багато грошей!".);
}
```

Зверніть увагу на відсутність модифікатора доступу (public). У даному випадку його використовувати не можна.

Як же виконати потрібний метод? Перед тим, як розповісти, як це робиться, розглянемо оператор is. Даний оператор дозволяє нам у процесі роботи програми перевірити, чи реалізує даний об'єкт той чи інший інтерфейс. Ця інформація дуже корисна, тому що знаючи, що клас реалізує той чи інший інтерфейс, можемо з впевненістю сказати, що він реалізує і методи, які цей інтерфейс надає. Отже, наведемо невеликий приклад роботи з нашим класом Ferrari:

```
    static void Main(string[] args)
{
    Ferrari c = new Ferrari ();
    c.Price = 1000000;
    c.Speed = 450;
    if (c is Car)
    {
        Car tmp = (Car)c;
        tmp.GetInfo();
```

```
        }
        if (c is Luxury)
        {
            Luxury tmp2 = (Luxury)c;
            tmp2.GetInfo();
        }
        Console.ReadLine();
    }
```

У першу чергу створюємо екземпляр класу Ferrari. Після чого присвоює значення властивостями щойно створеного об'єкта. Далі починається найцікавіше. Перевіряємо, чи реалізує об'єкт с інтерфейс Car. Оскільки інтерфейс Car реалізується, то ця умова поверне позитивне значення, тобто виконати. Наступний рядок є дуже важливим. Створюємо посилання на інтерфейс Car. За допомогою круглих дужок наводимо об'єкт с до об'єкта Car (тобто перетворюємо). При цьому з об'єктом с нічого не відбувається, він живий-здоровий, зате тепер у нас з'явилося посилання на інтерфейс Car, яке автори назвали tmp. Ось зараз, за допомогою цього посилання, і можемо звернутися до потрібного нам методу, тобто до того, що виводить рядок "Це суперкар Ferrari". Далі все аналогічно. До речі, дізнатися чи реалізує об'єкт той чи інший інтерфейс можна не тільки за допомогою оператора is, але і за допомогою оператора as і за допомогою звичайної конструкції if (), але про це ви вже почитайте самі (якщо is вам не достатньо).

Крім того, інтерфейси можуть узагальнити кілька об'єктів. Адже якщо кілька об'єктів (маються на увазі примірники різних класів) реалізують один і той же інтерфейс, то у них є загальні методи і властивості, правда реалізовані вони можуть бути по-різному. З огляду на цей факт, можна використовувати інтерфейс як параметр, який передається функції, а функція вже сама буде вирішувати, яку версію методу запускати.

Як і класи, інтерфейси можна наслідувати, при цьому успадкування інтерфейсів практично нічим не відрізняється від успадкування класів. Але на відміну від класів інтерфейси підтримують множинне успадкування, тобто один інтерфейс може наслідувати відразу від двох і більше не пов'язаних між собою інтерфейсів. Робиться це точно так само, як і реалізація класом декількох інтерфейсів, тобто успадковані інтерфейси пишуться через кому.

Класи в мові C # зазнали досить серйозних змін порівняно з мовою програмування Си + +, який і був взятий за основу. Перше – це неможливість множинного спадкоємства. По-друге, в C #, дозволено успадкування від декількох інтерфейсів.

Інтерфейсом у C # є тип посилань, що містить тільки абстрактні елементи, які не мають реалізації. Безпосередньо реалізація цих елементів повинна міститися в класі, похідне від цього інтерфейсу (ви не можете безпосередньо створювати екземпляри інтерфейсів). Інтерфейси C # можуть містити методи, властивості та індексатори, але на відміну, наприклад, від Java, вони не можуть містити константні значення. Розглянемо найпростіший приклад використання інтерфейсів:

```
using System;
class CShape
{
bool IsShape() {return true;}
}
interface IShape
{
double Square();
}
class CRectangle: CShape, IShape
{
double width;
double height;
public CRectangle(double width, double height)
{
this.width = width;
this.height = height;
}
public double Square()
{
return (width * height);
}
}
class CCircle: CShape, IShape
{
double radius;
public CCircle(double radius)
{
this.radius = radius;
}
```

```
public double Square()
{
return (Math.PI * radius * radius);
}
}
class Example_3
{
public static void Main()
{
CRectangle rect = new CRectangle(3, 4);
CCircle circ = new CCircle(5);
Console.WriteLine(rect.Square());
Console.WriteLine(circ.Square());
}
}
```

Обидва об'єкти, rect і circ, є похідними від базового класу CShape і тим самим вони успадковують єдиний метод IsShape (). Задавши ім'я інтерфейсу IShape в оголошеннях CRectangle і CCircle, вказуємо на те, що в даних класах міститься реалізація всіх методів інтерфейсу IShape. Крім того, члени інтерфейсів не мають модифікаторів доступу. Їх область видимості визначається безпосередньо реалізуємим класом.

Властивості

Розглядаючи класи мови C #, просто не можна обійти таке "нововведення", як властивості (properties). Треба сказати, що тут відчувається вплив мов Object Pascal і Java, в яких властивості завжди були невід'ємною частиною класів. Що ж становлять ці самі властивості? З точки зору користувача, властивості виглядають практично так само, як і звичайні поля класу. Їм можна присвоювати деякі значення і отримувати їх назад. У той же час властивості мають більшу функціональність, так як читання і зміна їх значень виконується за допомогою спеціальних методів класу. Такий підхід дозволяє ізолювати налаштовувану модель класу від її реалізації. Пояснимо дане визначення на конкретному прикладі:

```
using System;
using System.Runtime.InteropServices;
class Screen
{
[DllImport("kernel32.dll")]
static extern bool SetConsoleTextAttribute(
```

```
int hConsoleOutput, ushort wAttributes
);
[DllImport("kernel32.dll")]
static extern int GetStdHandle(
uint nStdHandle
);
const uint STD_OUTPUT_HANDLE = 0x0FFFFFFF5;
static Screen()
{
output_handle = GetStdHandle(STD_OUTPUT_HANDLE);
m_attributes  = 7;
}
public static void PrintString(string str)
{
Console.Write(str);
}
public static ushort Attributes
{
get
{
return m_attributes;
}
set
{
m_attributes = value;
SetConsoleTextAttribute(output_handle, value);
}
}
private static ushort m_attributes;
private static int    output_handle;
}
class Example_4
{
public static void Main()
{
for (ushort i = 1; i < 8; i++)
{
Screen.Attributes = i;
Screen.PrintString("Property Demo\n");
}
}
}
```

Програма виводить повідомлення "Property Demo", використовуючи різні кольори символів (від темно-синього до білого). Давайте спробуємо розібратися в тому, як вона працює. Отже, спочатку ми імпортуємо важливі для нас функції API-інтерфейсу Windows: SetConsoleTextAttribute і GetStdHandle. На жаль, стандартний клас середовища .NET під назвою Console не має засобів управління кольором виводу текстової інформації. Треба думати, що корпорація Microsoft в майбутньому все-таки вирішить цю проблему. Поки ж для цих цілей доведеться скористатися службою виклику платформи PInvoke (зверніть увагу на використання атрибуту DllImport). Далі, в конструкторі класу Screen ми отримуємо стандартний дескриптор потоку виводу консольної програми та вміщуємо його значення в закриту змінну output_handle для подальшого використання функцією SetConsoleTextAttribute. Крім цього, присвоюємо іншій змінній m_attributes початкове значення атрибутів екрана (7 відповідає білому кольору символів на чорному тлі). Зауважимо, що в реальних умовах варто було б отримати поточні атрибути екрана за допомогою функції GetConsoleScreenBufferInfo з набору API-інтерфейсу Windows.

У класі Screen автори оголосили властивість Attributes, для якого визначили функцію читання (getter) і функцію запису (setter). Функція читання не виконує будь-яких специфічних дій і просто повертає значення поля m_attributes (в реальній програмі вона мала б повертати значення атрибутів, отримане за допомогою тієї ж GetConsoleScreenBufferInfo). Функція запису трохи складніше, тому що крім тривіального оновлення значення m_attributes вона викликає функцію SetConsoleTextAttribute, встановлюючи задані атрибути функцій виводу тексту. Значення установлюваних атрибутів передається спеціальною змінною value. Зверніть увагу на те, що поле m_attributes є закритим, а мабуть, воно не може бути доступним поза класом Screen. Єдиним способом читання та / або зміни цього методу є властивість Attributes.

Тема 3. Основи використання мови XML під час розробки додатків для .NET

3.1. Основи використання мови XML під час розробки додатків для .NET

Розширювана мова розмітки (Extensible Markup Language – XML) становить технологію, яка охоплює широке коло застосувань – від опису конфігурації додатків до передачі інформації між Web-службами. XML – це спосіб зберігання даних у простому текстовому форматі, який означає, що він може бути прочитаний майже на будь-якому комп'ютері. Це робить його неперевершеним форматом для передачі даних через Internet. Його навіть нескладно читати і людині.

XML надзвичайно важлива в світі .NET, оскільки використовується як формат за умовчанням для передачі даних, а тому важливо розуміти хоч би його основи.

Повний набір даних в XML відомий як документ XML. Документом XML може бути фізичний файл на комп'ютері або просто рядок у пам'яті. Проте він винен бути завершений сам по собі, і повинен слідувати певним правилам (які будуть описані нижче). Документ XML складається з безлічі різних частин. Найбільш важливі з них – елементи XML, які містять самі дані документа.

Елементи XML складаються з відкриваючого дескриптора (імені елементу, що міститься в кутових дужках, наприклад, <myElement>), даних усередині елементу і закриваючого дескриптора (такого ж, як відкриваючий дескриптор, але з ведучим слешем після відкриваючої дужки: </myElement>).

Наприклад, визначити елемент для зберігання заголовка книги можна наступним чином:

<book>Tristram Shandy</book>

Синтаксис мови XML дуже схожий на синтаксис HTML. Головна відмінність в тому, що XML не має зумовлених елементів – користувач сам вибирає імена для своїх елементів, так що ніщо не обмежує їх кількість. Найбільш важливий момент, про який потрібно пам'ятати, полягає в тому, що XML, не дивлячись на її ім'я – це насправді зовсім не мова. Швидше за все це стандарт для визначення мов (відомих як XML-додатки). Кожна мова має свій власний відмінний від інших словник –

певний набір елементів, які можуть застосовуватися в документі, і структуру, яку можуть приймати ці елементи.

При цьому, можна явно обмежувати допустимі елементи в документі XML. Альтернативно можна дозволити будь-які елементи, і дозволити самій програмі, що використовує документ, вирішувати, яка має бути структура.

Імена елементів залежать від регістра, тому <book> і <BOOK> трактуються як різні елементи. Це означає, що якщо ви спробуєте закрити елемент <book> використовуючи закриваючий дескриптор, записаний в іншому регістрі (наприклад, </BOOK>), то XML-документ не буде правильним. Програми, що читають XML-документи і що аналізують їх індивідуальні елементи, відомі як XML-аналізатори (XML parsers), відхиляють будь-який документ, що містить неправильний XML.

Елементи також можуть містити в собі інші елементи, так що ви можете модифікувати елемент <book> так, щоб включити інформацію про автора в заголовку книги в двох піделементах:

```
<book>
<title>Tristram Shandy</title>
<author>Lawrence Sterne</author>
</book>
```

Перекриття елементів не допускається, тому піделементи повинні закриватися перед появою закриваючого дескриптора батьківського елементу. Це означає наприклад, що не можна зробити так:

```
<book>
<title>TristramShandy
<author>LawrenceSterne
</title></author>
</book>
```

Це некоректно, оскільки елемент <author> відкритий усередині елементу <title>, але закриваючий дескриптор </title> знаходиться перед закриваючим дескриптором </autho>.

Існує одне виключення з правила, що вимагає, щоб всі елементи мали закриваючий дескриптор. Допускається мати "порожні" елементи, які не мають вкладених даних або тексту. В цьому випадку можна просто додавати дескриптор, що закриває, відразу ж після того, що відкриває, як показано в попередньому коді, або ж використовувати скорочений

синтаксис, додаючи слеш закриваючого дескриптора в кінець того, який відкриває:

<book />

Це ідентично такому синтаксису:

<book></book>

Разом із зберіганням даних у тілі елементу ви можете також зберігати їх усередині атрибутів, які додаються до дескриптора, що відкриває елемент. Атрибути мають наступну форму:

name="value"

Тут значення атрибуту має бути поміщене в одиночні або подвійні лапки. Наприклад:

<booktitle="TristramShandy"></book>

або

<booktitle='TristramShandy'></book>

Обидва наведених варіанти правильні, а наступний — ні:

<book title=Tristram Shandy></book>

Навіщо потрібно два способи зберігання даних у XML? У чому різниця між наступними двома способами запису?

```
<book>
<title>Tristram Shandy</title>
</book>
```

і

```
<book title="Tristram Shandy"></book>
```

Фактично між ними і немає якоїсь фундаментальної відмінності. Ні один, ні інший спосіб не дають великої переваги перед другим. Елементи – більш вдалий вибір, якщо є шанс, що знадобиться додавати інформацію про цю частину даних пізніше – ви завжди можете додати піделемент або атрибут до елементу, але не можете робити те ж саме з атрибутами. Можливо, елементи більш читабельні і елегантніші (але це – справа особистого смаку). На противагу цьому атрибути споживають менше смуги пропускання, коли документ пересилається по мережі без стискування (при стискуванні різниця не велика), і зручні для зберігання інформації, яка не істотна кожному користувачеві документа.

Оголошення XML

На додаток до елементів і атрибутів XML-документи можуть містити множину інших складових частин. Ці індивідуальні частини документа

XML відомі як вузли (nodes); елементи, текст усередині елементів і атрибути – все це вузли документа XML.

Деякі елементи використовуються досить рідко, проте один тип вузла з'являється майже в кожному документі XML. Це оголошення XML, і якщо ви його включаєте, воно повинне знаходитися в першому вузлі документа.

Оголошення XML за своїм форматом подібне до елементу, але має усередині себе знаки питань і кутові дужки. Воно завжди має ім'я xml і завжди забезпечено атрибутом на ім'я version; в даний час єдине допустиме значення для нього – 1.0. Проста можлива форма XML-оголошення така:

<?xml version="1.0"?>

Додаткове оголошення XML також включає атрибути encoding (із значенням, що вказує символьний набір, який має бути використаний для прочитання документа, такий, як "UTF-16", щоб показати, що документ використовує 16-бітовий символьний набір Unicode) і standalone (із значеннями "yes" або "no" щоб вказати, чи залежить XML-документ від будь-яких інших файлів). Проте ці атрибути не обов'язкові, і ви, ймовірно включите тільки атрибут version у власні XML-файли.

Структура документа XML

Одна з найбільш важливих характеристик XML полягає в тому, що він надає спосіб структуризації даних, який істотно відрізняється від реляційних баз даних. Більшість сучасних систем управління базами даних зберігають інформацію в таблицях, які зв'язані один з одним через значення певних стовпців. Таблиці зберігають дані в рядках і стовпцях: кожен рядок представляє окремий запис, а кожен стовпець в ній – певний елемент даних у цьому рядку. На відміну від цього, дані XML структуровані ієрархічно подібно до організації папок і файлів в Windows Explorer. Кожен документ повинен мати єдиний кореневий елемент, усередині якого містяться всі елементи і текстові дані. Якщо на вершині документа знаходиться більш ніж один елемент, то такий документ не вважається за правильний документ XML. Проте ви можете включити інші вузли XML у верхній рівень – особливе оголошення XML. Таким чином нижче представлений правильний документ XML:

```
<?xml version=1.0"?>
<books>
```

<book>Tristram Shandy</book>
<book>Moby Dick</book>
<book>Ulysses</book>
</books>
А цей фрагмент не є правильним документом XML:
<?xml version="1.0"?>
<book>Tristram Shandy</book>
<book>Moby Dick</book>
<book>Ulysses</book>

Під кореневим елементом надана значна свобода відносно структури ваших даних. На відміну від реляційних даних, в яких кожний рядок має однакову кількість стовпців, тут немає обмежень на число піделементів, які може містити елемент. Додатково, хоча документи XML часто структуровані подібно до реляційних даних, з елементом для кожного запису документи XML не потребують ніяких зумовлених структур взагалі. Це – одна з основних відмінностей між традиційними реляційними базами даних і XML. Тоді як реляційні бази даних завжди визначають структуру інформації перед додаванням яких-небудь даних, інформація може бути збережена в XML без початкових накладних витрат, що дуже зручно для зберігання невеликих блоків даних. Як ви незабаром переконаєтеся, цілком можливо представити структуру вашого XML, але на відміну від реляційних баз даних, ніхто не вимагає від вас вказівки цієї структури явно.

Простори імен XML

Як відомо, будь-хто може визначати власні класи C#, і будь-хто може визначати свої власні елементи XML, що приводить до виникнення очевидної проблеми: як дізнатися, які елементи до якого словника відносяться? Відповідь – за допомогою просторів імен. Точно так, як визначаються простори імен для організації ваших типів C#, можна використовувати простори імен XML для визначення власних словників XML. Це дозволяє включати елементи безліч різних словників усередині єдиного документа XML, не ризикуючи неправильно інтерпретувати їх, тому що (наприклад) два різних словники визначають елемент <customer>.

Простори імен XML можуть бути достатньо складними, проте базовий синтаксис простий. Певні елементи або атрибути асоційовані з певним простором імен за допомогою префікса, за яким слідує

двокрапка. Наприклад, <wrox:book> представляє елемент <book>, що знаходиться в просторі імен wrox. Як ви дізнаєтеся, який простір імен представляє wrox? Щоб цей підхід працював, ви повинні гарантувати унікальність кожного простору імен. Простий спосіб домогтися цього – відобразити префікси на щось, унікальність чого відома, і саме так і робиться. Десь у вашому документі XML ви повинні асоціювати будь-які префікси простору імен з універсальним ідентифікатором ресурсів (Uniform Resource Identifier – URI). URI існують в декількох іпостасях, але найбільш поширений вигляд – проста адреса Web, наприклад, www. wrox. com.

Щоб ідентифікувати префікс з певним простором імен використовуйте атрибут xmlns :prefix усередині елементу, встановивши це значення в унікальний URI, який ідентифікує простір імен. Префікс може застосовуватися в будь-якому місці елементу, включаючи будь-які вкладені дочірні елементи:

```
<?xml version=1.0"?>
<books>
<book xmlns:wrox="http://www.wrox.com">
<wrox: title>Beginnmg C#</wrox: title>
<wrox:author>Karli Watson</wrox:author>
</book>
</books>
```

Тут можна використовувати префікс wrox: з елементами <title> і <author>, тому що вони знаходяться усередині елементу <book>, де визначений префікс. Проте якщо ви спробуєте додати цей префікс до елементу <books>, то XML стане неправильним, оскільки префікс для цього елементу не визначений.

Можна також визначити простір імен за умовчанням для елементу використовуючи атрибут xmlns:

```
<?xml version=1.0"?>
<books>
<book xmlns="http://www.wrox.com">
<title>Beginning Visual C# </title>
<author>Karli Watson </author>
<html:img src="begvcsharp.gif"
xmlns:html="http://www.w3.org/1999/xhtml" />
</book>
</books>
```

Тут простір імен за умовчанням для елементу <book> визначений як "http://www.wrox.com". Тому все, що знаходиться усередині цього елементу, буде відноситися до цього простору імен, якщо тільки ви явно не специфікуєте протилежне, додавши префікс іншого простору імен, як робите це для елементу <img> (коли ви встановлюєте простір імен, що використовується XML-сумісними документами HTML).

Правильно оформлений і дійсний XML

До цих пір говорилося про правильний (legal) XML. Фактично XML робить відмінність між двома формами правильності. Документи, відповідні всім правилам стандарту XML, вважаються як правильно оформлені (well-formed). Якщо документ XML не є правильно оформленим, то аналізатори не зможуть інтерпретувати його коректно, і відхилять такий документ. Щоб бути правильно оформленим, документ може відповідати перерахованим нижче вимогам:

мати один, і лише один кореневий елемент;

мати закриваючі дескриптори для кожного елементу (за виключенням скороченого синтаксису, згаданого раніше);

не мати елементів, що перекривають, – всі дочірні елементи мають бути повністю поміщені усередині батьківського;

мати всі атрибути, розміщені в дужки.

Це не повний список вимог, але він висвічує найбільш поширені помилки, що допускаються програмістами – новачками в XML. Проте документи XML можуть відповідати всім цим правилам і, проте, не бути дійсними (valid). Пригадаєте, що раніше було згадано, що XML сам по собі не є мовою, а швидше стандартом для визначення додатків XML. Правильно оформлені документи XML просто відповідають стандарту XML щоб бути дійсними, вони також повинні відповідати всім правилам, специфікованим для додатків XML. Не всі аналізатори перевіряють дійсність документів; ті, що роблять це, називаються перевіряючими аналізаторами. Щоб перевірити, чи відповідає документ правилам додатку, спочатку потрібний спосіб визначення цих правил.

XML підтримує два шляхи визначення того, які елементи і атрибути можуть бути поміщені в документ, і в якому порядку визначення типів документів (Document Type Definitions — DTD) і схеми. DTD використовують відмінний від XML синтаксис, успадкований від батька XML, і поступово замінюються схемами. DTD не дозволяють специфікувати

типи даних елементів і атрибути, а тому відносно не гнучкі і не особливо широко використовуються в контексті .NET Framework. Напротивагу цьому, схеми використовуються часто; вони дозволяють специфікувати типи даних, і написані в XML-сумісному синтаксисі. На жаль, схеми дуже складні, і існують різні форми для їх визначення, навіть усередині технології .NET.

Існують два формати схем, що підтримуються .NET – мова визначення схем XML (XML Schema Definition – XSD) і скорочені схеми XML-даних (XML-Data Reduced – XDR). Визначення схем XDR – старіший стандарт, який належить Microsoft; зазвичай він не використовується і не розпізнається аналізаторами, не належними Microsoft. XSD – відкритий стандарт, рекомендований W3C, і тому його визначення присутнє тут. Схеми можуть як включатися всередину XML-документа, так і зберігатися в окремому файлі. Ви маєте бути добре знайомі з XML, перш ніж приступати до написання схем, та варто уміти розпізнавати головні елементи схеми, тому нижче будуть викладені базові принципи. Щоб досягти розуміння, розглянемо просту схему XSD для простого документа XML, що містить базові деталі про пару книг Wrox на тему C#.

```
<?xml version=1.0"?>
<books>
<book>
<title>Beginning Visual C#</title>
<author>Karli Watson</author>
<code>7582</code>
</book>
<book>
<title>Professional C# 2nd Edition</title>
<author>Simon Robinson</author>
<code>7043</code>
</book>
</books>
```

Схеми XSD

Елементи в схемах XSD повинні відноситися до простору імен http://www. w3.org/2001/XMLSchema. Якщо цей простір імен не буде включений, елементи схеми не будуть розпізнані.

Щоб асоціювати документ XMLз схемою XSD в іншому файлі, необхідно додати елемент schemalocation до кореневого елементу:

```
<?xml version=1.0"?>
<books schemalocation="file://C:\Chapter 25\books.xsd">
```

```
</books>
```
Розглянемо коротко приклад схеми XSD:
```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
<element name="books">
<complexType>
<choice maxOccurs="unbounded">
<element name="book">
<complexType>
<sequence>
<element name="title" />
<element name="author" />
<element name="code" />
</sequence>
</complexType>
</element>
</choice>
<attribute name="schemalocation" />
</complexType>
</element>
</schema>
```

Перше, що треба відмітити – це те, що простір імен за умовчанням встановлено в простір. Це повідомляє аналізатор, що елементи документа відносяться до схеми. Якщо ви не специфікуєте цей простір імен, то аналізатор вважатиме, що це просто нормальні елементи XML, і не зрозуміє, що він повинен застосовувати їх для перевірки.

Уся схема міститься усередині елементу під назвою <schema>. Кожен елемент, який може з'явитися в документі, має бути представлений елементом <element>. Цей елемент має атрибут name, вказуючий ім'я елементу. Якщо елемент повинен містити в собі дочірні елементи, то ви повинні передбачити для них дескриптори <element> усередині елементу <contextType>. Усередині цього ви специфікуєте, як саме дочірні елементи повинні з'являтися.

Наприклад, ви використовуєте елемент <choice>, щоб специфікувати будь-який вибір дочірніх елементів, що допускається або <sequence> — щоб вказати, що дочірні елементи повинні з'являтися в тому ж порядку, як вони перераховані в схемі. Якщо елемент може з'являтися більше одного разу (як елемент <book>) потрібно буде помістити атрибут maxOccurs всередину батьківського елементу. Установка його в "unbounded" означає, що елемент може з'являтися необмежену кількість разів. І, нарешті, будь-які атрибути мають бути представлені елементами

<attribute>, включаючи ваш атрибут schemalocation, який повідомляє аналізатору, де шукати схему. Помістіть його після кінця списку дочірніх елементів.

Використання XML у додатку NET Framework включає множину просторів імен і класів, які полегшують завдання читання, маніпулювання і записи XML.

Об'єктна модель документа XML (XML Document Object Model – XML DOM) – це набір класів, що використовуються для маніпулювання XML інтуїтивно зрозумілим чином. Класи, складові DOM, знаходяться в просторі імен System.Xml. У цьому просторі міститься декілька класів і просторів імен, але в даному розділі зосередимо увагу тільки на деяких класах, які дозволяють легко маніпулювати XML. Ці класи описані в табл. 3.1.

Таблиця 3.1

Важливі класи з простору імен System. Xml

Клас	Опис
XmlNode	Представляє окремий вузол у документі. Служить базовим для багатьох класів, описаних в цьому розділі. Якщо вузол представляє корінь документа XML ви можете виконати з нього навігацію до будь-якої позиції документа
XmlDocument	Розширює клас XmlNode, але часто є першим об'єктом, який використовується при роботі зXML. Це пов'язано з тим, що цей клас застосовується для завантаження і збереження даних з диска або ще звідкись
XmlElement	Представляє окремий елемент у документі XML. XmlElement успадкований від XmlLinkedNode, який, у свою чергу успадкований від XmlNode
XmlAttribute	Представляє окремий атрибут. Подібно до класу XmlDocument, успадкований від класу XmlNode
XmlText	Представляє текст між відкриваючим і закриваючим дескрипторами
XmlComment	Представляє спеціальний тип вузлів, які не призначені служити частиною документа, а лише надають інформацію читачеві про частини документа
XmlNodeList	Представляє колекцію вузлів

Клас XmlDocument

Зазвичай перше, що повинне робити ваше застосування з XML — це прочитати його з файла. Як описано в табл. 3.1, за це відповідає клас XmlDocument. Ви можете сприймати XmlDocument як представлення дискового файла, розташоване в пам'яті.

Скориставшись класом XmlDocument для завантаження файла в пам'ять, ви можете отримати кореневий вузол документа з нього і почати читання і маніпуляції XML:

```
using System.Xml;
XmlDocument document = new XmlDocument();
document.Load(@"C:\ Chapter 25\books.xml");
```

Два рядки коду створюють новий екземпляр класу XmlDocument і завантажують в нього файл books.xml. Пам'ятаєте, що клас XmlDocument знаходиться в просторі імен System.Xml, і тому ви повинні вставити рядок using System.Xml; у початок коду.

На додаток до завантаження і збереження XML, клас XmlDocument також відповідає за підтримку самої структури XML. Тому ви знайдете в ньому численні методи, що відповідають за створення, зміну і видалення вузлів дерева документа.

Клас XmlElement

Після того, як документ завантажений в пам'ять, з ним потрібно щось робити. Властивість DocumentElement екземпляра XmlDocument, створеного в попередньому коді, повертає екземпляр XmlElement, що представляє кореневий елемент XmlDocument.

Цей елемент важливий, оскільки забезпечує доступ до кожного фрагмента інформації в документі:

```
XmlDocument document = new XmlDocument ();
document.Load(@"C:\Beginning Visual C#\Chapter 25\books.xml") ;
XmlElement element = document. DocumentElement ;
```

Після отримання кореневого елементу документа ви готові до використання цієї інформації. Клас XmlElement містить методи і властивості для маніпуляції вузлами і атрибутами дерева.

Модуль 2. Використання основних бібліотек .NET при розробці додатків на мові C#

Тема 4. Основні бібліотеки .Net

4.1. Принципи перевантаження операцій

У C#, як й у будь-якій іншій мові програмування, передбачений набір спеціальних символів для виконання найпоширеніших операцій із внутрішніми типами даних. Наприклад, багато програмістів знають, що символ "+" можна використати для одержання одного числа із двох інших:

```
// Оператор додавання (+) у дії
int a = 100: int b = 240:
int c = a + b; // c = = 340
```

Чому той самий оператор "+" можна застосовувати до будь-яких вбудованих типів C#, і не тільки числових?

```
// Складаємо два рядкових значення
string sl = "Hello":
string s2 = " world!":
string s3 = s1 + s2; // s3 = = Hello world!
```

Відповідь проста – оператор "+" перевантажений таким чином, щоб він міг нормально працювати із найрізноманітнішими типами даних. Якщо застосовуємо його до числових типів, виконується додавання. Якщо він застосовується до строкових типів, виконується злиття текстових рядків – конкатенація. Мова C# дозволяє створювати користувальницькі класи й структури, які будуть реагувати по-своєму (як визначать автори) на ті ж самі вбудовані оператори, наприклад на оператор додавання "+". Цей прийом називається перевантаженням операторів.

Ще одна обставина, яку потрібно підкреслити – це те, що, говорячи про операції, маємо на увазі не тільки арифметичні операції. Існують також операції порівняння: ==, <, >,! =, >= й <=. Візьмемо, наприклад, конструкцію if (a==b). Для класів ця конструкція за умовчанням зрівнює два посилання на предмет того, чи вказують вони на те саме місце в

пам'яті, замість того, щоб перевірити, чи містять екземпляри об'єктів однакові дані. Для класу string ця поведінка перевизначена таким чином, що порівняння рядків насправді порівнює їхній зміст. Ви можете зробити щось подібне зі своїми власними класами. За умовчанням для структур операція == взагалі нічого не робить. Спроба зрівняти дві структури, щоб дізнатися, чи містять вони однакові значення, приведе до помилки компіляції, якщо тільки ви не перевантажите операцію ==, щоб повідомити компіляторові, як він повинен виконувати таке порівняння.

Також варто відзначити, що перевантажувати можна не всі доступні операції. Перелік операцій, які перевантажувати можна, наведено у табл. 4.1.

Таблиця 4.1

Перелік операцій, які можна перевантажувати

Категорія	**Операції**	**Обмеження**
Арифметичні бінарні	+, *, /, -, %	Немає
Арифметичні унарні	+, -, ++, --	Немає
Бітові бінарні	&, , ^, <<, >>	Немає
Бітові унарні	!, ~ **true, false**	Операції **true** й **false** повинні перевантажуватися в парі
Порівняння	**>=, <= >, <**	Операції порівняння повинні перевантажуватися попарно
Присвоювання	+=, -=, *=, /=, >>=, <<=, %=, &=, =, ^=	Ці операції не можна явно перевантажити; їхнє перева-нтаження здійснюється неявно при перевизначенні індиві-дуальних операцій +, -, % і т. д.
Індексація	[]	Операцію індексу не можна перевантажити явно. Тип члена, що індексує, дозволяє підтримувати операцію інде-ксації в користувальницьких класах і структурах
Приведення	()	Операцію приведення не можна перевантажити явно. Користувальницькі риведення дозволяють реалізувати при-ведення, що набудовують

Існує безліч ситуацій, у яких можливість перевантаження операцій дозволяє генерувати кращий для читання й інтуїтивно зрозумілий код.

Наприклад, якщо ви пишете програму, що виконує деяке математичне або фізичне моделювання, то майже напевно використаєте класи, що представляють математичні об'єкти, такі, як координати, вектори, матриці, тензори, функції і т. д.

Графічні програми, які використовують математичні або пов'язані з координатами об'єкти, коли обчислюються позиції на екрані.

Клас, що представляє грошові суми (наприклад, у фінансовій програмі).

Текстовий процесор або програма аналізу тексту, що використовує класи, які представляють речення, висловлення тощо; вам може знадобитися використати операції для комбінування речень (більш складна версія конкатенації рядків).

Однак існує також багато типів, для яких перевантаження операцій небажане. Необачне застосування перевантаження операцій породжує код, що набагато складніше зрозуміти. Наприклад, множення двох об'єктів DateTime навіть концептуально не має змісту.

Як працюють операції

Щоб зрозуміти, як перевантажувати операції, дуже корисно замислитися про те, що відбувається, коли компілятор зустрічає операцію в коді. Використовуючи операцію додавання (+) як приклад, припустимо, що компілятор обробляє наступні рядки коду:

```
int mylnteger = 3;
uint myUnsignedlnt = 2;
double myDouble = 4.0;
long myLong = mylnteger + myUnsignedlnt;
double myOtherDouble = myDouble + mylnteger;
```

Що відбувається, коли компілятор зустрічає представлений нижче рядок?

```
long myLong = mylnteger + myUnsignedlnt;
```

Компілятор зрозуміє, що йому потрібно скласти два цілих числа й присвоїти результат змінній типу long. Однак вираз myInteger + myUnsignedInt – це насправді інтуїтивний і зручний синтаксис виклику методу, що складає разом два числа. Метод приймає два параметри – myInteger й myUnsignedInt – і повертає їхню суму. Таким чином, компілятор виконує ті ж дії, що й при будь-якому виклику методу – шукає найбільш підходяще перевантаження операції додавання на основі типів параметрів; у цьому випадку – приймаючи два цілих. Як і зі звичайними перевантаженими методами, бажаний тип повернення не впливає на вибір версії методу, що повинен викликати компілятор. У даному прикладі викликається перевантаження, що приймає два параметри типу int і повертає int; потім повернуте значення перетворюється в long.

Наступний рядок змушує компілятор використати інше перевантаження операції додавання:

```
double myOtherDouble = myDouble + myInteger;
```

У цьому екземплярі параметри мають типи double й int, але так трапилося, що не існує перевантаження операції додавання, що приймає таку комбінацію параметрів. Замість цього компілятор ідентифікує у ролі найбільш підходящу версію операції додавання із двома параметрами типу double і неявно конвертує один int в double. Додавання двох double вимагає обробки, відмінної від додавання двох int. Числа із плаваючою точкою зберігаються у вигляді мантиси й експоненти. Їхнє додавання включає бітовий зсув мантиси одного із двох параметрів double – таким чином, щоб дві експоненти мали те саме значення, додавання мантис, потім зсув мантиси результату й настроювання його експоненти для забезпечення максимально можливої точності.

Тепер розглянемо, що відбудеться, якщо компілятор зустріне щось таке:

```
Vector vect1, vect2, vect3;
// ініціалізувати vect1 й vect2
vect3 = vect1 + vect2;
vect1 = vect1*2;
```

Тут Vector – структура, що визначена в наступному розділі. Компілятор бачить, що йому потрібно скласти разом два екземпляри

Vector – vect1 й vect2. Він шукає перевантаження операції додавання, що приймає як параметри два екземпляри Vector.

Якщо компілятор знаходить підходяще перевантаження, він викликає цю реалізацію операції. Якщо ж ні – він шукає інше перевантаження операції +, що найбільш точно підходить – можливо, що приймає два параметри інших типів даних, які можуть бути неявно перетворені в екземпляри Vector. Якщо компілятор знову не знаходить підходящого перевантаження, він видає помилку компіляції, як він робить завжди, коли не може знайти відповідне перевантаження при будь-якому виклику методу.

Перевантаження операцій рівності

Як пам'ятаємо, у C# часто виникає необхідність заміщати метод System.Object.Equals(), щоб можна було виконувати порівняння структурних типів (у первинній реалізації цей метод підтримує тільки порівняння посилальних типів).

Крім заміщення методів Equals() і GetHashCode() (при заміщенні Equals() заміщення GetHashCode() обов'язково) вам, швидше за все, буде потрібно замістити так само й оператори рівності (= = й !=). Як це можна зробити, продемонструємо на прикладі із крапкою.

```
public class Point
{
    public int x. y: public Point (){}
    public Point(int xPos. int yPos){x = xPos: y = yPos:}
    public override bool Equals(object o)
    {
        If( ((Point)o).x == this.x && ((Point)o).y == this.y)
            return true;
        else
            return false;
    }

    public override int GetHashCode()
    { return this.ToString().GetHashCode(); }

    // А ось і саме перевантаження операторів рівності
```

```
        public static bool operator = =(Point p1, Point p2)
        {
                return p1.Equals(p2):
        }
        public static bool operator !=(Point p1, Point p2)
        {
                return !pl.Equals(p2):
        }
}
```

У переважній більшості випадків програмісти прагнуть використати зрозумілі й знайомі оператори = = й !=. Застосування їх інтуїтивно, чого не скажеш про метод Equals(). Тому рекомендується в тих ситуаціях, коли ви замістили метод Equals(), подбати ще й про перевантаження операторів = = й !=.

Ще один момент, який необхідно обов'язково відзначити, - C# не дозволить вам перевантажити оператор = = без перевантаження оператора != або навпаки (точно так само, як методи Equals() і GetHashCode() можна перевантажувати тільки в парі). Такий підхід гарантує, що всі оператори рівності будуть застосовуватися однаково коректно.

Оператори перетворення

Іноді об'єкт класу потрібно використати у виразі, що включає інші типи даних. Такі засоби може забезпечити перевантаження одного або декількох операторів. Але в деяких випадках бажаного результату можна досягти за рахунок перетворення типів (із класового в потрібний). З метою обробки подібних ситуацій C# дозволяє створювати спеціальний тип операторного методу operator, іменованого *оператором перетворення.* Такий оператор перетворить об'єкт деякого класу в значення іншого типу. По суті, оператор перетворення перевантажує оператор приведення типів. Оператори перетворення сприяють повній інтеграції класових типів у C#-середовище програмування, дозволяючи об'єктам класу вільно змішуватися з даними інших типів за умови визначення операторів перетворення в ці "інші типи".

Існують дві форми операторів перетворення: явна й неявна. У загальному виді вони записуються так:

```
public static explicit operator тип_результату (
ПЕРВИННИЙ__ТИП v) [return значення; ]
public static implicit operator тип_результату (
первинний_тип v) [return значення;]
```

Тут елемент *тип_результату* становить тип, у який ви хочете виконати перетворення; елемент *первинний_тип* означає тип об'єкта, що підлягає перетворенню; елемент *v* – значення класу після перетворення. Оператори перетворення повертають дані типу *тип_результату,* причому специфікатор типу тут вказувати не дозволяється.

Якщо оператор перетворення визначений із ключовим словом implicit, перетворення виконується автоматично, тобто при використанні об'єкта у виразі, що включає дані типу *тип_результату.* Якщо оператор перетворення визначений із ключовим словом explicit, перетворення виконується при використанні оператора приведення типів. Для однієї й тієї ж пари типів, що беруть участь у перетворенні, не можна визначити як explicit-, так і implicit-оператор перетворення.

Для ілюстрації визначення й використання оператора перетворення створимо його для класу Three. Припустимо, необхідно перетворити об'єкт типу Three у цілочислове значення, щоб його можна було використовувати у виразах типу int. Це перетворення буде полягати в обчисленні добутку значень всіх трьох координат об'єкта. Для реалізації такого перетворення використаємо implicit-форму оператора, що буде мати такий вид:

```
public static implicit operator int(Three opl)
return opl.x * opl.y * opl.z;
```

Нижче наведений текст програми, у якій ілюструється використання цього оператора перетворення.

```
// Приклад використання implicit-оператора перетворення.
using System;
// Клас тривимірних координат,
class ThreeD {
```

```
int x, y, z; // тривимірні координати.
public ThreeD() { x = y = z = 0; }
public ThreeD(int i, int j , int k) {
x = i; y = j ; z = k; }
// Перевантажуємо бінарний оператор "+".
public static ThreeD operator +(ThreeD op1, ThreeD op2)
ThreeD result = new ThreeD();
result.x = op1.x + op2.x;
result.y = op1.y + op2.y;
result.z = op1.z + op2.z;
return result;
// Неявне перетворення з типу ThreeD у тип int.
public static implicit operator int(ThreeD op1)
return op1.x * op1.y * op1.z;
// Відображаємо координати X, Y, Z.
public void show()
Console.WriteLine(x + ", " + y + ", " + z);
class ThreeDDemo {
public static void Main() {,
ThreeD a = new ThreeD(1, 2, 3);
ThreeD b = new ThreeD(10, 10, 10);
ThreeD c = new ThreeD();
i n t i;
Console.Write("Координати точки a: " ) ;
a.show();
Console.WriteLine();
Console.Write("Координати точки b: " ) ;
b.show();
Console.WriteLine() ;
c = a + b; // Підсумуємо координати точок a й b.
Console.Write("Результат додавання a + b: " ) ;
c.show();
Console.WriteLine() ;
i = a; // Перетворимо в значення типу int.
Console.WriteLine(
"Результат присвоювання i = a: " + i) ;
Console.WriteLine();
i = a * 2 - b ; // Перетворимо в значення типу int.
Console.WriteLine("Результат обчислення виразу a * 2 - b: " + i)
```

При виконанні ця програма генерує наступні результати:

Координати точки a: 1, 2, 3

Координати точки b: 10, 10, 10

Результат додавання a + b: 11, 12, 13
Результат присвоювання i = a: 6
Результат обчислення виразу a * 2 - b : -988

Як видно за результатами виконання цієї програми, якщо об'єкт класу ThreeD використовується у виразі цілочислового типу (наприклад i = a), до цього об'єкта застосовується оператор перетворення. У цьому випадку результатом цього перетворення буде число 6, отримане при множенні всіх координат, збережених в об'єкті a. Але якщо для виразу не потрібне перетворення в int-значення, оператор перетворення не викликається. Тому при обчисленні виразу c = a + b операторний метод operator int() не викликається.

Можна створювати різні методи перетворення, що відповідають вашим потребам. Наприклад, можна визначити операторний метод перетворення об'єкта якого-небудь класу в double- або long-значення. При цьому кожне перетворення виконується автоматично й незалежно від інших.

Оператор неявного перетворення застосовується автоматично в тому випадку, коли у виразі потрібне перетворення, коли методу передається об'єкт, коли виконується присвоювання, а також коли використовується явно задана операція приведення об'єкта до результуючого типу. Як альтернативний варіант можна створити оператор явного перетворення, що викликається тільки у випадку явного приведення типів. Оператор явного перетворення не викликається автоматично. Ось, наприклад, як виглядає попередня програма, перероблена для використання оператора явного перетворення об'єкта в значення типу int:

```
// Використання оператора явного перетворення.
using System;
// Клас тривимірних координат.
class ThreeD {
int x, y, z; // тривимірні координати.
public ThreeD() { x = y = z = 0; }
public ThreeD(int i, int j, int k) {
x = i; y = j; z = k; }
// Перевантажуємо бінарний оператор "+".
```

```
public static ThreeD operator +(ThreeD op1, ThreeD op2)
{
ThreeD result = new ThreeD();
result.x = op1.x + op2.x;
result.y = op1.y + op2.y;
result.z = op1.z + op2.z;
return result;
//Цього разу перевантажуємо explicit-оператор,
public static explicit operator int(ThreeD op1)
{
return op1.x * op1.y * op1.z;
// Відображаємо координати X, Y, Z.
public void show()
{
Console.WriteLine(x + ", " + y + ", " + z) ;
class ThreeDDemo {
public static void Main() {
ThreeD a = new ThreeD(1, 2, 3);
ThreeD b = new ThreeD(10, 10, 10);
ThreeD c = new ThreeD();
int i;
Console.Write("Координати точки a: " ) ;
a.show();
Console.WriteLine();
Console.Write("Координати  b: " ) ;
b.show();
Console.WriteLine() ;
c = a + b; // Підсумуємо координати об'єктів a й b.
Console.Write("Результат додавання a + b: " ) ;
c. show() ;
Console.WriteLine();
i = (int) a; // Перетворимо об'єкт у значення
// типу int, оскільки явно задана
// операція приведення типів.
Console.WriteLine("Результат присвоювання i = a: " + i);
Console.WriteLine();
/
```

```
i = (int)a * 2 - (int)b; // Потрібне приведення типів.
Console.WriteLine("Результат обчислення виразу a * 2 - b: " + i ) ;
```

Оскільки оператор перетворення тепер визначений з використанням ключового слова explicit, перетворення об'єкта в значення типу int повинне бути задане як оператор приведення типів. Наприклад, якщо в рядку коду

```
i = (int) a;
```

видалити оператор приведення типів, програма не скомпілюється.

Визначення й використання операторів перетворення пов'язане з рядом обмежень.

Первинний тип об'єкта або тип результату перетворення повинен збігатися зі створюваним класом. Не дозволяється перевизначати такі перетворення, як з типу double у тип int.

Не можна визначати перетворення в клас Object або з нього.

Не можна задавати як явне, так і неявне перетворення одночасно для однієї й тієї ж пари вихідного й результуючого типів.

Не можна задавати перетворення з базового класу в похідний.

Не можна задавати перетворення з одного інтерфейсу в інший.

Крім перерахованих правил існують також рекомендації, якими звичайно користуються при виборі між явним і неявним операторами перетворення. Незважаючи на певні зручності до неявно заданих перетворень прибігають тільки в ситуаціях, коли перетворення гарантовано позбавлене помилок. Інакше кажучи, неявні оператори перетворення варто створювати тільки при дотриманні наступних умов. По-перше, таке перетворення повинне гарантувати відсутність втрати даних, що має місце при усіканні, переповненні або втраті знака. По-друге, перетворення не повинне стати причиною виникнення виняткових ситуацій. Якщо передбачуване перетворення не відповідає цим вимогам, варто використовувати перетворення явного типу.

Приклад перевантаження операції: структура Vector

Розглянемо приклад реалізації перевантаження операцій, у якому спробуємо розкрити якнайбільше всіляких аспектів їхнього застосування.

Ви можете складати або перемножувати вектори з іншими векторами або числами. Додавання векторів означає індивідуальне

додавання кожного з їхніх компонентів. У даному контексті математики пишуть c = a+b, де a й b – вектори, які складаються, а c – результуючий вектор. Добре б мати можливість так само поводитися зі структурами Vector.

Нижче наведене визначення Vector, що містить поля-члени, конструктори й перевизначені ToString(), так що ви можете легко побачити вміст Vector, а наприкінці – перевантажені операції.

```
namespace Wrox.ProCSharp.OOCSharp
{
        struct Vector
        {
        public double x, y, z;
        public Vector(double x, double y, double z)
        {
                this-x = x; this.y = y; this.z = z;
        }
        public Vector(Vector rhs)
        {
                x = rhs.x; y = rhs.y; z = rhs.z;
        }
        public override string ToString() {
        return " ( " + x + " , " + y + " , " + z + " ) ";
}
```

У цьому прикладі представлено два конструктори, які вимагають вказати початкове значення вектора – або у вигляді значень всіх компонентів, або за значенням іншого Vector, значення компонентів якого можна скопіювати.

Конструктори на зразок другого, приймаючі єдиний аргумент типу Vector, часто називають конструкторами копіювання, оскільки вони дозволяють ініціалізувати екземпляр класу або структури простим копіюванням іншого екземпляра. Відзначимо, що для простоти залишаємо поля структури загальнодоступними (public). Можна було б зробити їх приватними (private) і написати відповідні властивості для доступу до них, але це ніяк не вплинуло б на даний приклад, за винятком того, що зробило б код більше.

Найцікавіша частина структури Vector – перевантаження операції додавання:

```
public static Vector operator + (Vector lhs, Vector rhs)
{
	Vector result = new Vector (lhs);
	result.x += rhs.x;
	result.y += rhs.y;
	result.z += rhs.z;
	return result;
}
```

Перевантаження операції оголошується майже так само, як метод, за винятком того, що ключове слово operator повідомляє компіляторові, що це насправді визначення перевантаження операції. За ключовим словом operator йде символ операції, у цьому випадку – знак додавання (+). Тип, що повертає – це тип, який ви одержите, застосувавши цю операцію. Додавання двох векторів дає в результаті вектор, тому, що повертає тут типом, є Vector. У даному конкретному випадку перевантаження операції додавання тип, що повертається, збігається з класом, що його включає, але це зовсім не обов'язково, як буде показано пізніше. Два параметри – це ті операнди, з якими операція буде працювати. Для бінарних операцій (які приймають два параметри) на зразок додавання й віднімання перший параметр – це значення, що записується ліворуч від знака операції, а другий – значення, що записується праворуч.

C# вимагає щоб перевантажені операції оголошувалися як public й static; це значить, що вони асоційовані із класом або структурою, а не з їхніми екземплярами. З цієї причини тіло перевантаженої операції не має доступу до нестатичних членів, а також до ідентифікатора this. І це нормально, тому що параметри представляють всі вхідні дані, необхідні операції для рішення свого завдання.

Тепер, якщо ви зрозуміли синтаксис оголошення операції додавання, то можете глянути на те, що відбувається усередині неї:

```
{
	Victor result = new Vector (lhs);
	result.x += rhs.x;
	result.y += rhs.y;
```

```
        result.z += rhs.z;
        return result;
}
```

Ця частина коду – точно така ж, яка могла б бути при визначенні методу, і легко здогадатися, що вона повертає вектор, що представляє суму lhs й rhs. Ви просто індивідуально складаєте члени x, y і z.

Тепер нам знадобиться написати деякий простий код, щоб протестувати структуру Vector:

```
static void Main()
{
        Vector vect1, vect2, vect3;
        Vect1 = new Vector (3.0, 3.0, 1.0);
        vect2 = new Vector (2.0, -4.0, -4.0);
        vect3 = vect1 + vect2;
        Console.WriteLine("vect1 = " + vect1.ToString());
        Console.WriteLine("vect2 = " + vect2.ToString()) ;
        Console.WriteLine("vect3 = " + vect3.ToString());
}
```

Ось що одержимо при виконанні цього коду:

```
Vectors
vect1 =(3,3,1)
vect2 =(2,-4,-4)
vect3 =(5,-1,-3)
```

Додавання додаткових перевантажень

На додаток до додавання векторів, можна було б множити й віднімати їх, а також порівнювати між собою. Потім розів'ємо приклад Vector далі, додавши ще кілька перевантажень операцій.

Зараз не станемо розробляти повний набір, що, імовірно, знадобився б для повнофункціонального типу Vector, а просто продемонструємо інші аспекти перевантаження операцій. Для початку перевантажимо операцію множення, щоб додати підтримку множення векторів на скаляри й векторів на інші вектори.

Множення вектора на скаляр просто означає індивідуальне множення кожного його компонента на скаляр; наприклад, 2 * (1.0, 2.5, 2.0) поверне (2.0, 5.0, 4.0). Відповідне перевантаження операції виглядає в такий спосіб:

```
public static Vector operator * (double lhs, Vector rhs)
{
    return new Vector (lhs * rhs.x, lhs * rhs.y, lhs * rhs.z);
}
```

Однак цього недостатньо. Якщо a й b оголошені як Vector, то це дозволить вам написати такий код:

```
b = 2 * a;
```

Компілятор просто неявно перетворить ціле число 2 в double, щоб вираз відповідав сигнатурі перевантаження. Однак наступний код не скомпілюється:

```
b = a * 2;
```

Справа в тому, що компілятор трактує перевантажені операції так само, як перевантаження методів. Він перевіряє всі доступні перевантаження даної операції, щоб відшукати найбільш підходящий. А останній оператор вимагає, щоб перший параметр мав тип Vector, а другий – ціле число або щось, неявно перетворене в ціле. Але не було надано такого перевантаження. Компілятор не може поміняти місцями параметри, хоча в цьому випадку їхній порядок не важливий. Доведеться явно визначити перевантаження, що приймає спочатку Vector, а потім – double. Існують два способи її реалізації. Перший спосіб припускає написання операції множення вектора на число точно так само, як це було зроблено в першому випадку:

```
public static Vector operator *  (Vector lhs, double rhs)
{
    return new Vector (rhs * lhs.x, rhs * lhs.y, rhs *lhs.z);
}
```

Але, маючи вже написаний код, що реалізує точно ту ж операцію, ви можете зволіти повторно використати його в такий спосіб:

```
public static Vector operator * (Vector lhs, double rhs)
{
    return rhs * lhs;
}
```

Цей код повідомляє компіляторові, що якщо він бачить множення – Vector на double, то може просто поміняти порядок операндів і викликати інше перевантаження.

Далі нам знадобиться перевантажити операцію множення для підтримки перемножування векторів. Математики пропонують безліч способів перемножування векторів, але нас цікавить один, відомий як скалярний добуток, що повертає в результаті скаляр. Це привід продемонструвати на даному прикладі, що арифметичні операції не зобов'язані повертати тип, що збігається з типом класу, у якому вони визначені.

У математичній термінології, якщо у вас є два вектори – (x, y, z) і (X, Y, Z), – то значення їхнього скалярного добутку визначається як значення виразу x*X + y*Y + z*Z. Що зараз нас цікавить – так це дати можливість людям, що використовують наш Vector, писати вираз на зразок double X = a * b для обчислення скалярного добутку двох об'єктів Vector (a й b). Відповідне перевантаження буде виглядати так:

```
public static double operator * (Vector lhs, Vector rhs)
{
    return lhs.x * rhs.x + lhs.y * rhs.y + lhs.z * rhs.z;
}
```

Тепер, коли розібралися з арифметичними операціями, можна перевірити їхню працездатність за допомогою простого тестового методу:

```
static void Main ()
{
    // демонстрація арифметичних операцій
    Vector vect1, vect2, vect3;
    vect1 = new Vector (1.0, 1.5, 2.0);
    vect2 = new Vector (0.0, 0.0, -10.0);
    vect3 = vect1 + vect2;
```

```
        Console.WriteLine("vect1 = " + vect1);
        Console.WriteLine("vect2 = " + vect2);
        Console.WriteLine("vect3 = vect1 + vect2 = " + vect3);
        Console.WriteLine("2*vect3 = " + 2*vect3);
        vect3 += vect2;
        Console.WriteLine("vect3+=vect2 дає " + vect3);
        vect3 = vectl*2;
        Console.WriteLine ("присвоювання vect3=vect1*2 дає " + vect3);
        double dot = vect1*vect3;
        Console.WriteLine("vect1*vect3 = " + dot);
}
```

Запуск цього коду дасть наступний результат:

```
Vectors2
vect1 = ( 1 , 1.5 , 2)
vect2 = ( 0 , 0 , -10 )
vect3 = vect1 + vect2 = ( 1 , 1.5 , -8 )
2*vect3 =(2,3, -16 )
vect3+=vect2 дає (1,1.5, -18 )
Присвоювання vect3=vect1*2 дає (2,3,4)
vect1*vect3 =14.5
```

Це показує, що перевантажені операції видають коректні результати, але якщо ви подивитеся на тестовий код уважно, то можете здивуватися, помітивши, що він використає операцію, що не була перевантажена – а саме операцію додавання із присвоюванням, +=.

```
vect3 += vect2;
Console.WriteLine("vect3 += vect2 дає " + vect3) ;
```

Хоча += звичайно вважається єдиною операцією, вона може бути розбита на два окремих кроки: додавання й присвоювання. На відміну від мови C++, C# у дійсності не дозволяє перевантажувати операцію =, але якщо ви перевантажите операцію +, то компілятор автоматично використовує це ваше перевантаження й в операції +=. Той же принцип застосовується до всіх операцій присвоювання, тобто -=, *=, /=, &= і т. д.

Останні зауваження про перевантаження

Перевантаження операторів може бути дуже зручним, коли хочемо змусити свої користувальницькі типи реагувати на ті ж оператори, що й вбудовані типи. Однак варто пам'ятати, що переважна більшість мов програмування (і в їхньому числі Java) перевантаження операторів не підтримують. Вимоги забезпечити можливість перевантаження операторів немає й у Common Language Specification. Тому якщо ви працюєте з великим проектом, різні модулі якого створені різними мовами, будьте готові зіштовхнутися з тим, що в деяких модулях використати перевантаження операторів буде неможливо.

Із цієї ситуації є очевидний вихід – зробити так, щоб кожному перевантаженому операторові відповідав звичайний метод з "нормальною" назвою. У модулях, написаних мовами, що підтримують перевантаження операторів, будуть використовуватися відповідні оператори, а в інших модулях – найбільш звичайні методи зі стандартними назвами. Для класу Point таке рішення може виглядати в такий спосіб:

```
// Крім перевантаженого оператора, у цьому визначенні класу передбачений
// звичайний метод з тими ж можливостями public class Point
// Метод AddPoints працює точно так, як перевантажений оператор //додавання
public static Point AddPoints (Point p1, Point p2)
return new Point(p1.x + p2.x, p1.y + p2.y);
}
// ...а метод SubtractPoints – як перевантажений оператор віднімання
public static Point SSubtractPoints (point p1, Point p2)
{
// Обчислюємо значення нової координати x
int new = p1.x - p2.x: if(new < 0)
throw new ArgumentOutOfRangeException;
// Обчислюємо значення нової координати y
int new = p1.y - p2.y; if(new < 0)
throw new ArgumentOutOfRangeException:
return new Point(new. new):
```

```
}}
```

Тепер точки можна складати ще й так:
Point finalPt = Point.AddPoints(ptOne, ptTwo);
Console. WriteLine("My final point: {0}", finalPt.ToString);

Таким чином, перевантаження необхідне, для того щоб спростити роботу над програмами, які використовують ті самі методи, але з різними вхідними даними. Тому, щоб не писати громіздкі функції або методи для певних операцій, аналоги яких є в будь-якій мові програмування, але які діють інакше, ніж необхідно в конкретній ситуації, і використовується перевантаження.

4.2. Індексатори та властивості

Властивості

Властивість (property) – це спеціальний тип членів класу, завдяки якому спрощується реалізація однієї із основних ідей інкапсуляції даних – дані класу мають бути захищені, а методи доступу до них – відкритими. Властивості забезпечують можливість контролю за введенням значень даних-членів класу. Пригадаємо ідею використання закритих членів класу. У наступному прикладі в класі **`MyClass`** маємо закрите поле **`field`** та пару відкритих методів, які забезпечують можливість зчитування цього поля та запису у нього певної величини з контролем її значення.

```
using System;
namespace Use_Methods
{
    class MyClass
    {
        private int field;       // закрите поле в класі
        public int get_field()    // відкритий метод для
зчитування поля field
        {
            return field;
        }
        public void set_field(int value) // відкритий метод
для запису поля field
        {
```

```
                if (value > 0) field = value;     // якщо
запропоноване для поля значення
                                                  // додатне, присвоюємо
                                              його значенню поля
                else field = 1;          // інакше встановлюємо
значення за замовчуванням
            }
        }
        class Program
        {
            static void Main()
            {
                MyClass m = new MyClass();
                Console.WriteLine("Введіть поле");
                int i = int.Parse(Console.ReadLine());
                m.set_field(i);//метод записує у поле field
значення i, якщо воно додатне
                Console.WriteLine("поле = {0}", m.get_field());//
метод друкує
              // як збільшити поле на одиницю?
                m.set_field(m.get_field() + 1);
                Console.WriteLine("тепер поле збільшене на 1 :
{0}", m.get_field());
            }
        }
    }
```

Практично реалізація цієї ідеї означає, що кожне закрите поле повинне бути забезпечене парою відкритих методами для роботи з ними. Але мова C# має більш зручний інструмент – це саме властивості, які були анонсовані вище. Властивість становить пару аксесорів (accessor) – **`get`** та **`set`**, які забезпечують доступ до певного поля класу. Властивість визначається за наступним синтаксисом:

```
    <модифікатор доступу властивості> <тип властивості>
<ідентифікатор властивості>
    {
          get             // acsessor get
          {
              // код acsessor'у get
          }
          set             // acsessor set
          {
              // код acsessor'у set
           }
    }
```

Тут у аксесорі **get** поміщають код, що повертає значення певного закритого поля класу, з яким пов'язана ця властивість. У аксесорі **set** визначаються дії, що забезпечують присвоєння тому самому полю певного значення. Причому саме тут можливо передбачити засоби контролю за значенням, яке записується у це поле. Зрозуміло, що **<тип властивості>** має бути ідентичним типу поля, з яким ця властивість пов'язана. Властивість не має параметрів, звертатись до неї можна просто за ідентифікатором. Важливо розуміти, що коли властивість використовується у лівій частині оператора присвоєння, то ***автоматично*** викликається аксесор **set**, який приймає змінну з наперед визначеним ідентифікатором **value**. В іншому випадку так само автоматично викликається аксесор **get**. З технічної точки зору при компіляції властивості генеруються два методи, подібні до тих, що було використано у попередньому прикладі. Слід розуміти, що властивість не має доступу до пам'яті, отже, її використання без деякого поля класу не має сенсу. Властивість лише керує доступом до цього поля. Прийнятий стиль оформлення властивості полягає у використанні для ідентифікатора властивості поля імені, яке збігається із назвою самого поля з точністю до першого символу – у поля це маленька літера, а у властивості цього поля – велика.

Виконаємо таке ж завдання, як і у попередньому прикладі, проте використаємо замість методів доступу до закритого поля **field** властивість. Тут клас **MyPropertyClass** має закрите поле **field** та властивість, що ним керує – **Field**. Аксесор цієї властивості **get** просто повертає значення поля **field**, а аксесор **set** перевіряє значення змінної **value**, яка автоматично потрапляє у нього, – якщо це значення додатне, то воно присвоюється полю **field**, в іншому разі у **field** встановлюється прийняте за умовчанням значення 1.

```
using System;
namespace Use_Properties
{
    class MyPropertyClass
    {
        private int field;  // закрите поле в класі - ним
керуватиме властивість Field
```

```
            public int Field    // властивість для поля field
            {
                get             // acsessor get
                {
                    return field;
                }
                set             // acsessor set
                {  // якщо запропоноване для поля значення
додатне, присвоюємо
                  // його значенню поля
                    if (value > 0) field = value;
                    else field = 1;
                }
            }
        }
        class Program
        {
            static void Main()
            {
                MyPropertyClass mp = new MyPropertyClass();
                Console.WriteLine("Введіть поле");
                mp.Field = int.Parse(Console.ReadLine());
                Console.WriteLine("поле = {0}", mp.Field);
                        // Як збільшити поле на одиницю? Тепер це
простіше!
                Console.WriteLine("тепер поле збільшене на 1 :
{0}", ++mp.Field);
            }
        }
    }
```

Зверніть увагу, коли ідентифікатор властивості фігурує у виразі

```
mp.Field = int.Parse(Console.ReadLine());
```

спрацьовує **set**, який отримує у змінній **value** (вона цілого типу, бо такого типу сама властивість) значення, введене з клавіатури методом **Console.ReadLine()**. В інструкції

```
Console.WriteLine("поле = {0}", mp.Field);
```

автоматично викликається **get**, що повертає значення **field**. Проте в обох випадках звертання до властивості **Field** виглядає однаково. В той же час, у попередньому прикладі звертались до двох різних методів – **set_field(i)** та **get_field()**. Крім того, в останньому прикладі для того, щоб збільшити (або зменшити) значення

поля **field** можна використовувати звичайні оператори інкременту (декременту): **++mp.Field чи mp.Field++**. Порівняйте, як це відбувалось у прикладі **Use_Methods**.

У нашому прикладі властивість **Field** призначена як для читання так і для запису поля **field**. Проте в деяких випадках властивість може бути використана ***лише для читання*** або ***лише для запису***. В такому разі відповідний аксесор просто відсутній. У наступному прикладі властивість **Name** може бути використана лише для читання значення поля **name**, адже аксесор **set** у ній відсутній. При спробі присвоєння цій властивості будь-якого значення виникне синтаксична помилка.

```
class MyPropertyClass
    {
        private string name = "MyName";
        public string Name // Ця властивість лише для читання
        {
            get { return name; }
        }
    }
class Program
    {
        static void Main()
        {
            MyPropertyClass mp = new MyPropertyClass();
            Console.WriteLine("name = " + mp.Name);
    //      mp.Name = "NewName";  // тут буде помилка -
властивість лише для читання !
        }
    }
```

Основна зручність використання властивостей полягає в тому, що користувач класу звертається до них просто як до звичайних полів класу. З'ясуємо схоже та відмінне між ***полями*** (даними-членами) та ***властивостями*** класу:

1. Використовуються поля та властивості з точки зору синтаксису однаково.
2. Поле розміщується у пам'яті, а властивість – ні.
3. Властивість є логічним полем, доступ до якого здійснюється через аксесори **get** та **set**.
4. Властивості, як і поля, мають модифікатор доступу.

5. Властивості, як і поля, можуть бути статичними.

Ознаки схожості ***властивості*** проявляють і з ***методами*** класу, тому порівняємо і їх.

1. Властивість, як і метод, містить код.
2. Властивість, як і метод, приховує деталі свого функціонування.
3. Властивість, як і метод, може бути статичною.
4. Властивість, на відміну від методу, не може мати тип **`void`**.
5. Властивість, на відміну від методу, не має дужок із списком параметрів.
6. Властивість, на відміну від методу, не може перевантажуватись.

Останнє, про що варто нагадати, – формальна змінна **`value`** є видимою та може бути використана лише в межах властивості, – звертання до змінної поза властивістю призведе до синтаксичної помилки.

Індексатори

Індексатор (indexer) – це ще один спеціальний тип членів класу, завдяки якому є можливість індексного доступу до екземплярів класу. Синтаксично це може виглядати як звертання до елементів масиву, які є екземплярами класу. Причому, на відміну від звичайних масивів, що походять від типу **`System.Array`**, в якості індексів тут може бути використаний довільний тип, а не лише цілий, починаючи з нуля. Визначення індексатора схоже на визначення властивості та має наступний синтаксис:

```
    <модифікатор доступу > <тип індексатору> this [<тип індексу>
< ідентифікатор індексу>]
    {
          get             // acsessor get
          {
              // код acsessor'y get
          }
          set             // acsessor set
          {
              // код acsessor'y set
           }
    }
```

Тут **<тип індексатору>** означає базовий тип об'єктів, що будуть індексуватись через індексатор (аналог базового типу елементів масиву). В аксесорі **set** знаходиться код, що автоматично активізується, коли індексатор знаходиться в лівій частині оператору присвоєння. В інших випадках автоматично викликається аксесор **get**. Як і у властивостей аксесор **set** індексатору приймає параметр **value**.

Розглянемо простий приклад створення та використання індексатора. Клас **MyIndexClass** містить два закритих члени класу: **size**, який зберігає кількість елементів та ідентифікатор масиву **arr**. У конструкторі ініціалізується **size** та створюється масив відповідного розміру. Індикатор за значенням індексу **ind** (контролюється його значення, яке має бути між 0 та **size**) повертає через аксесор **get** або встановлює через аксесор **set** значення відповідного елементу масиву.

```
using System;
namespace Use_Indexer_0
{
class MyIndexClass
    {
        private int size;                // розмір масиву
        private int[] arr;               // декларація масиву
        public MyIndexClass(int size_)   // конструктор
        {
            size = size_;
            arr = new int[size_];        // тут масив
створюється
        }
        public int this[int ind]         // це і є індексатор
           {
             get    // повертаємо елемент масиву як результат
звертання до індексатору
            {
                if ((ind >= 0) && (ind < size)) { return
arr[ind]; }
                return 0;
            }
            set     // присвоюємо елемент масиву як результат
звертання до індексатору
            {
                if ((ind >= 0) && (ind < size)) { arr[ind] =
value; }
```

```
                }
            }
        }
        class Program
        {
            static void Main()
            {
                MyIndexClass mic = new MyIndexClass(5);
                for (int i = 0; i < 5; i++)
                {
                    mic[i] = i;                 // тут працює set
індексатору
                    Console.WriteLine("mic [{0}] = {1}", i,
mic[i]);  // тут працює get
                }
            }
        }
    }
```

У методі **Main()** створений екземпляр **mic** класу **MyIndexClass**. Цей об'єкт містить масив із 5 елементів. Вираз **mic[i]** є звертанням до індексатору. Таким чином, використовуємо об'єкт **mic** як індексований масив. Результат роботи цього прикладу наступний:

```
mic [0] = 0
mic   = 1
mic   = 2
mic   = 3
mic   = 4
```

Розглянемо ще один приклад. У ньому описаний клас **Roots**, призначений для визначення коренів квадратного рівняння. Саме рівняння задається своїми коефіцієнтами **a, b, c** – вони є закритими членами класу разом із масивом **r** коренів, який може містити 0, 1 або два корені. У випадку нескінченної кількості коренів (рівняння вироджується) цей масив також порожній. Крім того, закритими членами цього класу є **discr** – дискримінант квадратного рівняння та **num** – кількість коренів рівняння. Останнім полем керує властивість **Num** призначена лише для читання. Конструктор класу ініціалізує поля **a, b, c** , визначає **discr** та викликає закритий метод класу **Calc_Roots()** , в

якому зосереджена логіка розв'язування квадратного рівняння та створений (якщо рівняння має корені) масив коренів **r**. Індексатор, визначений у цьому класі, дозволяє після створення екземпляру **roots** класу **Roots** використовувати корені рівняння як елементи масиву **roots[i]**, де **i** пробігає індекси від 0 до **roots.Num** – кількість коренів рівняння. Нагадаємо, що у випадку нескінченної кількості коренів властивість **Num** містить максимальне ціле число, а масив коренів не визначений.

```
using System;
namespace Use_Indexer
{
    class Roots                         // Клас містить корені
квадратного рівняння
    {
        private double a, b, c;         // кофіцієнти рівняння
        private double[] r;             // деклація масиву
коренів (якщо вони будуть)
        private int num;                // кількість коренів
        private double discr;           // дискримінант рівняння
        public Roots(double a_, double b_, double c_)   //
конструктор
        {
            a = a_; b = b_; c = c_; discr = b * b - 4 * a *
c;
            Calc_Roots();
        }
        public int Num                  // властивість -
кількість коренів
        {
            get { return num; }         // лише для читання
        }
        private void Calc_Roots()   // тут визначаємо корені
        {
            if ((a == 0) && (b == 0) && (c == 0))
            {
                num = int.MaxValue;
                Console.WriteLine("Безліч коренів");
            }
            if ((a == 0) && (b == 0) && (c != 0))
            {
                num = 0;
```

```
                    Console.WriteLine("Немає коренів");
                }
                if ((a == 0) && (b != 0))
                {
                    num = 1;
                    r = new double ;
                    r[0] = -c / b;
                }
                if (a != 0)
                    if (discr == 0)
                    {
                        num = 1;
                        r = new double[num];
                        r[0] = -b / (2 * a);
                    }
                    else if (discr > 0)
                    {
                        num = 2;
                        r = new double[num];
                        r[0] = (-b + Math.Sqrt(discr)) / (2 * a);
                        r  = (-b - Math.Sqrt(discr)) / (2 * a);
}
                    else
                    {
                        num = 0;
                        Console.WriteLine("Немає коренів");
                    }
            }
            public double this[int index]  // індексатор
              {
                    get
                {
                    if ((index >= 0) && (index <= num))
                        return r[index];
                    else return float.NaN;
                }
              }
        }
        class Program
        {
            static void Main()
            {
                    Console.WriteLine("Введіть коефіцієнти
рівняння");
```

```
                    Console.Write("a = "); double a =
double.Parse(Console.ReadLine());
                    Console.Write("b = "); double b =
double.Parse(Console.ReadLine());
                    Console.Write("c = "); double c =
double.Parse(Console.ReadLine());
                    Roots roots = new Roots(a, b, c);
                    if ((roots.Num < int.MaxValue)&&(roots.Num >
0))
                    {
                        Console.WriteLine("Корені :");
                        for (int i = 0; i < roots.Num; i++)
                        {
                            Console.WriteLine(roots[i]); // Тут
працює індексатор
                        }
                    }
            }
        }
    }
```

Оскільки, як вже було сказано, індексатори проявляють певну схожість із властивостями (головна спільна риса – всі правила для аксесорів властивостей справедливі і для індексаторів), перелічимо спільне та різне між ними:

1. Доступ до властивості здійснюється за її ідентифікатором, індексатор не має власного ідентифікатору, доступ здійснюється за індексом елементу.

2. Аксесор **`get`** властивості не має параметрів, в той час як **`get`** індексатора має один (або більше) параметрів – індекс.

3. Аксесор **`set`** властивості має неявний параметр **`value`**, а **`set`** індексатора крім **`value`** має ті самі індекси, що і його **`get`**.

4. Властивість може бути статичним членом класу, індексатор – ніколи, оскільки він визначається через посилання **`this.`**

5. Властивості не перевантажуються в той час, як індексатор може бути перевантажений за рахунок використання індексу іншого типу, або іншої кількості індексів.

У наступному прикладі визначений клас з індексатором з двома індексами, причому не цілого типу.

```
    using System;
    namespace Use_Indexer_2
    {
        using System;
        class Grid
        {
            const int NumRows = 26;         // кількість символів
у латинському алфавіті
            const int NumCols = 10;         // кількість
десятклвих цифр
            int[,] cells = new int[NumRows, NumCols];
            public int this[char symb, char dig] // двовимірний
індексатор
            {
                get
                {
                 symb = Char.ToUpper(symb);    // символ - завжди
велика літера
                 if (((symb >= 'A') && (symb <= 'Z')) && ((dig >=
'0') && (dig <= '9')))
                     return cells[symb - 'A', dig - '0'];
                 else return 0;
                }
                set
                {
                 symb = Char.ToUpper(symb);    // символ - завжди
велика літера
                 if (((symb >= 'A') && (symb <= 'Z')) && ((dig >=
'0') && (dig <= '9')))
                 cells[symb - 'A', dig - '0'] = value;
                }
            }
        }
        class Program
        {
            static void Main()
            {
                Grid gr = new Grid();
                for (char c = 'A'; c <= 'Z'; c++)
                {
                   for (char d = '0'; d <= '9'; d++)
                   {
                   gr[c, d] = (int)(c - 'A') * (int)(d - '0'); //
Тут працює set indexer'a
```

```
                    Console.Write(gr[c, d] + " \t");            //
Тут працює get indexer'a
                }
            }
        }
    }
}
```

Тут у класі **Grid** визначений двовимірний індексатор з індексами символьного типу. Таким чином екземпляр цього класу можна сприймати як двовимірну матрицю, у якої рядки індексуються символами літер, а стовпчики – символами цифр. Сама таблиця заповнена цілими числами, рівними добуткам номерів рядків та стовпчиків.

Зауваження. Індексатор насправді не вимагає існування базового масиву в класі, головне, щоб у його аксесорах була прописана логіка функціонування, яка для користувача класу виглядає як звертання до елементів масиву. Зокрема у класі **Roots** масив **r** може бути взагалі не визначений при деяких наборах коефіцієнтів **a, b, c** .

4.3. Обробка виключень

Помилки за своєю сутністю далеко не завжди трапляються з вини того, хто кодує додаток. Іноді програма генерує помилку з вини дій кінцевого користувача. У будь-якому випадку користувач завжди повинен очікувати на виникнення помилок у своїх програмах, а програмісти спираючись на це, кодувати у відповідності з цими очікуваннями.

У платформі .NET Framework передбачена розвинена система обробки помилок. Механізм, який влаштовано в C# для обробки помилок, дозволяє закодувати спеціальну обробку для кожного типу помилкових умов, а також відокремити код, який потенційно може народити помилки, від коду, який займається обробкою останніх.

Незалежно від того, наскільки добре написано програмний код, програми, які програмуються, повинні вміти долати помилки будь-якого роду, які можуть виникнути при їх виконанні. Наприклад, посеред деякого процесу складної обробки, програмний код може виявити, що не має прав доступу для читання файла, на який останній посилається, або ж під час передачі даних по мережі станеться обрив зв'язку. У таких виняткових ситуаціях недостатньо, щоб метод повернув деякий код помилки, бо для звичайного користувача це може стати критичним

випадком, причини якого він не зможе зрозуміти, а продукт у його уявленні почне набувати недоброї слави.

Може так статися, що до моменту виникнення виключної ситуації в програмі виконано 15-20 вкладених викликів методів, тому в дійсності потрібно "перестрибнути" назад через усі ці 15 або 20 викликів для того, щоб коректно вийти з завдання та прийняти відповідні заходи по обробці ситуації. Мова C # має дуже гарний засіб для того, щоб справлятися з подібними ситуаціями, а саме – механізм, відомий як обробка виключень.

Порівняно з іншими мовами програмування, виключення в C # відкривають для користувача зовсім новий "світ" можливостей для обробки помилок в програмах. Великий плюс, який отримала мова C# порівняно з C++, це те що в мові C++ обробка виключень може значно впливати на продуктивність процесору. В C# обробка виключних ситуацій побудована таким чином, що на продуктивність процесору зовсім не впливає.

Класи виключень

У мові програмування C # виняток – це об'єкт, створений при настанні певної виняткової помилкової ситуації. Цей об'єкт містить інформацію, яка може допомогти відстежити причину виникнення проблеми. Хоч в мові можливо створити свої власні класи винятків, .NET представляє безліч визначених класів винятків.

Усі класи є частиною простору імен System, за винятком IOException і класів, успадкованих від IOException, які становлять частину простору імен System.IO. Простір імен System.IO має справу з читанням і записом даних у файли. Взагалі не існує безумовного простору імен для винятків; класи виключень повинні бути поміщені в ті простори, де знаходяться класи, які можуть їх генерувати, тому виключення, пов'язані з введенням-виведенням (IO) знаходяться в System.IO, а інші класи виключень можна знайти в інших просторах імен базових класів.

Найбільш загальний клас винятків, System.Exception, успадкований від System.Object, як і треба було чекати від класу .NET. Взагалі програмісту не потрібно створювати об’єкти винятків System.Exception у своєму коді, тому що вони не несуть в собі ніякої специфіки конкретної помилкової ситуації.

Від System.Exception в ієрархії успадковується два важливих класи:

1. System. SystemException – цей клас передбачений для винятків, які зазвичай генерує виконуюча система .NET, або ж які мають деяку загальну природу і можуть бути згенеровані майже будь-яким додатком. Наприклад, StackOverflowException породжує виконуюча система .NET, коли виявляє переповнення стеку. З іншого боку, у своєму коді програміст можете порушити ArgumentException, або якийсь з його підкласів, якщо виявиться, що метод був викликаний з неправильними аргументами. До підкласу System.SystemException відносяться виключення, що представляють як фатальні, так і не фатальні помилки.

2. System.ApplicationException – цей клас важливий, тому що його призначення – служити базовим для будь-якого класу винятків, обумовлених незалежними розробниками. Тому якщо програміст визначає будь-яке виключення, що обслуговує унікальні помилкові ситуації у своїх програмах, то він повинен наслідувати їх – напряму або ж через посередників – від System.ApplicationException.

Серед інших корисних класів винятків варто сказати про описані нижче:

1. StackOverflowException – цей виняток генерується, якщо область пам'яті, відведена для стека, переповнена. Переповнення стека може статися, якщо метод нескінченно рекурсивно викликає самого себе. Зазвичай це вважається фатальною помилкою, оскільки не дає вашій програмі робити нічого, окрім негайного завершення (в цьому випадку малоймовірно навіть, що виконається блок finally). Спроби обробити помилки, подібні до цієї, як правило, безпорадні, і замість цього вам краще забезпечити коректний вихід із програми.

2. EndOfStreamException – зазвичай причиною цього винятку є спроба читання за кінцем файла. Потік (stream) представляє потік даних між їх джерелами.

3. OverflowException – це те, що трапляється при спробі приведення змінної int, що має значення, скажімо *40*, до типу uint: в checked-контексті.

Ієрархія класів винятків дещо незвична, тому що більшість цих класів не додають ніякої функціональності до своїх базових класів. Однак у випадку обробки виключень основною причиною долучення класів спадкоємців є вказівка більш специфічних помилкових умов, що часто не вимагає перевизначення старих методів або додавання нових

(хоча немає нічого незвичайного в додаванні додаткових властивостей для обслуговування додаткової інформації про умови помилки). Наприклад, програміст може мати базовий клас ArgumentException, призначений для виклику методів, коли їм передаються неправильні аргументи, і успадкований від нього клас ArgumentNullException, який обслуговує частковий випадок передачі в параметрі значення null.

Перехоплення виключень

Беручи до уваги, що .NET Framework включає велику кількість вже існуючих класів винятків, розглянемо питання як їх використовувати у своєму коді для перехоплення помилкових умов? Для того, щоб долати можливі помилкові ситуації в коді C #, програма зазвичай ділиться на блоки трьох різних типів:

1. try блок інкапсулює код, що формує частину нормальних дій програми, і який, однак, потенційно може зіткнутися з серйозними помилковими ситуаціями.

2. catch – блок інкапсулює код, який обробляє помилкові ситуації, що відбуваються в коді блоку try. Це також зручне місце для протоколювання помилок.

3. finally – блок інкапсулює код, очищуючий будь-які ресурси або виконуючий інші дії, що зазвичай потрібно виконати в кінці блоків try або catch. Важливо розуміти, що цей блок виконується незалежно від того, було порушено виключення чи ні. Оскільки метою блоку finally є виконання коду очищення, який повинен бути виконаний у будь-якому випадку, компілятор видасть помилку, якщо ви помістите оператор return всередину блоку finally. Наприклад, у блоці finally ви можете закрити будь-яке з'єднання, яке було відкрито в блоці try. Важливо також зрозуміти, що блок finally необов'язковий. Якщо у вас немає вимог по очищенню коду (таких як закриття будь-яких відкритих об'єктів), то в цьому блоці немає необхідності.

Розглянемо рекомендації до того як зібрати всі ці три блоки для перехоплення помилкових умов:

1. Потік управління спочатку входить у блок try.

2. Якщо у блоці try не виникає ніяких помилок, виконання триває нормально до кінця блоку, і по досягненні його кінця передається в блок

finally, якщо такий присутній. Однак якщо в блоці try виникає помилка, потік управління переходить на блок catch.

3. Помилкова умова обробляється в блоці catch.

4. Наприкінці блоку catch управління автоматично передається в блок finally, якщо такий присутній.

5. Виконується блок finally (якщо такий є).

У загальному вигляді синтаксис мовою C# буде мати вигляд як представлено нижче:

```
try { // код при нормальному виконанні }
catch {  // обробка помилок }
finally { // очистка }
```

Насправді існує декілька варіантів цієї конструкції:

1. Програміст може пропустити блок finally, оскільки він не обов’язковий.

2. Програміст може застосувати стільки блоків catch, скільки бажає, для обробки специфічних типів помилок. Однак ідея полягає в тому, що вам не потрібно особливо захоплюватися цим і будувати величезну кількість блоків catch, оскільки від цього може постраждати продуктивність створюваного додатка.

3. Ви можете пропустити всі блоки catch. При цьому синтаксис служить не для ідентифікації винятків, а як спосіб гарантії того, що код блоку finally буде виконаний, коли потік управління покине блок try. Це зручно, якщо в блоці try присутні кілька точок виходу.

Поки все добре, але є питання, на яке необхідно відповісти: коли виконання увійшло в блок try, як воно дізнається, коли варто перемкнутися на блок catch, якщо виникне помилка? Якщо виявлена помилка, код робить те, що називається збудження виключення. Іншими словами, він створює екземпляр об'єкта виключення і активізує його:

```
throw new OverflowException();
```

Тут створюється екземпляр об'єкта класу винятку OverflowException. Як тільки комп’ютер досягає оператору throw всередині блоку try, він негайно шукає блок catch, асоційований з цим блоком try. Якщо є більш одного блоку catch асоційованого з цим try, він ідентифікує коректний catch-блок, перевіряючи, який клас винятків перехоплює кожен з них.

Наприклад, коли порушується об'єкт OverflowException, потік виконання переходить на наступний блок catch:

```
        catch (OverflowException ex)
 {/ / Тут йде обробка виключення }
```

Іншими словами, комп'ютер шукає такий блок catch, який приймає об'єкт-виключення відповідного класу (або його базового класу).

З урахуванням сказаного можемо розширити блок try, продемонстрований вище. Припустимо, що в блоці try можуть виникнути дві серйозні помилки: переповнення і вихід за кордон масиву. Припустимо, що наш код містить дві bool змінних – Overflow та OutOfBounds, – які вказують на наявність цих умов. Як було видно раніше для індикації переповнювання існує визначений клас винятків (OverflowException); аналогічно, на випадок виходу за кордон масиву передбачений клас виключення IndexOutOfRangeException.

Тепер блок try буде виглядати так:

```
try
{// код для нормального виконання
        if  (Overflow==true)
        {
          throw new OverflowException();
        }
        //подальша обробка
        if  (OutOfBounds==true)
        {throw new IndexOutOfRangeException();}
}
        //в протилежному випадку продовжується нормальна обробка
        catch (OverflowException ex)
        {
        //обробка помилок переповнення
        }
        catch (IndexOutOfRangeException ex)
        {
        // обробка помилок виходу за межу масиву
        }
```

```
finally
{
// очистка
}
```

C # представляє потужний і гнучкий механізм обробки помилок. Справа в тому, що оператор throw може бути вкладений послідовно у кілька викликів методів всередині блоку try, але цей же блок try залишається в силі, навіть коли потік управління входить до вкладених методів. Якщо комп'ютер зустрічає оператор throw, він негайно передає управління назад – через всі вкладені виклики методів стеку, – намагаючись знайти кінець блоку try і запустити відповідний блок catch. Протягом цього процесу всі локальні змінні проміжних викликів методів коректно залишають контекст. Це робить архітектуру try…catch придатною до ситуацій, на зразок тієї, що описана на початку, коли помилка виникає всередині виклику методу, який має 15-й або 20-й рівень вкладеності і обробка повинна припинитися негайно.

З цього пояснення стає зрозуміло що, блоки try можуть відігравати дуже істотну роль у контролі потоку управління коду. Однак важливо розуміти, що виключення призначені для виняткових умов, про що і говорить їх назва. Не варто користуватися ними як способом управління виходом із циклів.

System.Exception

Дуже часто бібліотечний код збуджує виключення, якщо стикається з якимись проблемами, або коли метод викликається неправильно, або ж коли передаються неправильні параметри. Однак бібліотечний код рідко намагається перехопити виключення. Це розглядається як область відповідальності клієнтського коду.

Під час налагодження часто трапляється так, що виняток збуджується в бібліотеці базових класів. Процес налагодження в деяких випадках передбачає визначення причин виникнення виключень з тим, щоб позбутися від них. Тоді, метою програміста домогтися щоб до моменту поставки коду споживачеві було гарантовано, що виключення виникають лише в дуже рідкісних випадках і, наскільки можливо, обробляються у вашому коді якимось розумним чином.

У System.Exception є багато властивостей, які перераховано в табл. 4.2.

Таблиця 4.2

Властивості System.Exception

Властивості	Опис
Data	Надає можливість додавати конструкції "ключ-значення" до виключень, які можуть бути використані для постачання винятків деякою додатковою інформацією
HelpLink	Зв'язок з довідковим файлом, у якому представлена більш детальна інформація про виключення
InnerException	Якщо виняток було порушено всередині блоку catch, то InnerException містить об'єкт винятку, який був посланий в цей catch-блок
Message	Текст, що описує умова помилки
Source	Ім'я програми або об'єкта, який викликав виняток
StackTrace	Інформація про виклики методів у стек (для того, щоб допомогти в пошуку методу, збуджуючий виняток)
TargetSite	Об'єкт рефлексії .NET, що описує метод, який збудив виняток

З усіх цих властивостей тільки StackTrace і TargetSite застосовуються автоматично виконуючим середовищем .NET, якщо доступне трасування стеку. Source завжди заповнюється виконуючою системою .NET ім’ям збірки, в якій виникло виключення (хоча програміст може модифікувати цю властивість у своєму коді, щоб надати більш специфічну інформацію), у той час як Data, Message, HelpLink і InnerException повинні бути заповнені кодом, який був порушений винятком, шляхом встановлення потрібних значень безпосередньо перед збудженням виключення.

Наприклад, код, який збуджує виняток, може виглядати приблизно так:

```
if (ErrorCondition == true)
{
Exception myException = new ClassmyException (``Help!!!``);
myException.Source = (``ім’я мого додатку``);
myException.HelpLink = ``MyHelpFile.txt``;
myException.Data[``ErrorDate``] = DateTime.Now;
```

```
myException.Data.Add(``Додаткова інформація``);
throw myException;
}
```

Тут ClassMyException – ім'я класу, який збуджується виключенням. Можна відзначити, що класи виключень прийнято іменувати із закінченням Exception. Також зверніть увагу, що значення властивості Data може присвоюватися двома можливими способами.

Що відбувається з необробленими виключеннями?

Іноді виключення може бути порушено, але у вашому коді немає відповідного блоку catch, який би обробляв виключення подібного роду. Приклад SimpleExceptions може служити ілюстрацією сказаного. Припустимо, наприклад, що опустимо catch-блок для FormatException і catch-блок без параметрів, а залишили тільки блок, який бере IndexOutOfRangeException. Що відбудеться у такому випадку, у випадку виникнення виключення FormatException? Відповідь: його перехопила виконуюча система .NET. Приблизно це відбиває те, як можна вкладати блоки try один в одного. Виконуюча система .NET поміщає всю обробляєму програму всередину величезного блоку try і робить це з кожною програмою .NET. Цей блок try має оброблювач catch, який може перехопити виключення будь-якого типу. Якщо трапляється виключення, яке не обробляє ваша програма, то потік винятку просто передається з програми і перехоплюється catch-блоком виконуючої системи .NET. Однак те, що вийде в результаті, навряд чи нас влаштує. Виконання такого коду буде перервано і користувач побачить діалогове вікно зі скаргою на те, що ваш код не обробив виняток, разом з інформацією про виняток, яку виконуюча система .NET зможе витягти. Але, важливо те, що виняток буде перехоплено.

Взагалі кажучи, якщо програміст пише програму, то повинен намагатися перехопити стільки винятків, скільки зможете, і обробити їх осмисленим чином. Якщо ж програміст пише бібліотеку, то йому звичайно краще не перехоплювати виключення (якщо тільки певний виняток не уявляє щось неправильне у його коді, що код може обробити), а припускати, що їх перехопить і обробить код викликаємий програмою. Однак, тим не менше, краще перехоплювати виключення, визначені Microsoft, і порушувати власні винятки, що надають більш специфічну інформацію клієнтського коду.

4.4. Введення-виведення даних

C#-програми виконують операції введення-виведення за допомогою потоків, які побудовані на ієрархії класів. *Потік* (stream) – це абстракція, яка генерує і приймає дані. За допомогою потоку можна читати дані з різних джерел (клавіатура, файл) і записувати в різні джерела (принтер, екран, файл). Не дивлячись на те, що потоки зв'язуються з різними фізичними пристроями, характер поведінки всіх потоків однаковий. Тому класи і методи введення-виведення можна застосувати до багатьох типів пристроїв.

На найнижчому рівні ієрархії потоків введення-виведення знаходяться потоки, що оперують байтами. Це пояснюється тим, що багато пристроїв при виконанні операцій введення-виведення орієнтовано на байти. Проте для людини більш звично оперувати символами, тому розроблені символьні потоки, які фактично є оболонками, що виконують перетворення байтових потоків у символьні навпаки. Окрім цього, реалізовані потоки для роботи з int-, double-, short-значеннями, які також представляють оболонку для байтових потоків, та працюють не з самими значеннями, а з їх внутрішнім представленням у вигляді двійкових кодів.

Центральну частину потокової C#-системи займає клас Stream простору імен System.IO. Клас Stream представляє байтовий потік і є базовим для решти всіх потокових класів. З класу Stream виведені такі байтові класи потоків, як:

1) FileStream – байтовий потік, розроблений для файлового введення-виведення;

2) BufferedStream – укладає в оболонку байтовий потік і додає буферизацію, яка у багатьох випадках збільшує продуктивність програми;

3) MemoryStream – байтовий потік, який використовує пам'ять для зберігання даних.

Програміст може вивести власні потокові класи. Проте для переважної більшості додатків досить вбудованих потоків.

Детально розглянемо клас FileStream, класи StreamWriter і StreamReader, що є оболонками для класу FileStream і дозволяють перетворювати байтові потоки в символьні, а також класи BinaryWriter і BinaryReader, що є оболонками для класу FileStream і дозволяють

перетворювати байтові потоки в двійкові для роботи з int-, double-, short- і так далі значеннями.

Байтовий потік

Щоб створити байтовий потік, пов'язаний з файлом, створюється об'єкт класу FileStream. При цьому в класі визначено декілька конструкторів. Найчастіше використовується конструктор, який відкриває потік для читання і/або запису:

FileStream(string filename, FileMode mode)

де:

1) параметр filename визначає ім'я файла, з яким буде пов'язаний потік введення-виведення даних; при цьому filename визначає або повний шлях до файла, або ім'я файла, який знаходиться в папці bin/debug вашого проекту;

2) параметр mode визначає режим відкриття файла, який може приймати одне з можливих значень, визначених переліком FileMode:

a) FileMode.Append – призначений для додавання даних в кінець файла;

b) FileMode.Create – призначений для створення нового файла, при цьому якщо існує файл з таким же ім'ям, то він буде заздалегідь видалений;

c) FileMode.CreateNew – призначений для створення нового файла, при цьому файл з таким же ім'ям не повинен існувати;

d) FileMode.Open – призначений для відкриття існуючого файла;

e) FileMode.OpenOrCreate – якщо файл існує, то відкриває його, інакше створює новий;

f) FileMode.Truncate – відкриває існуючий файл, але урізає його довжину до нуля

Якщо спроба відкрити файл виявилася не успішною, то генерується одне з виключень: FileNotFoundException – файл неможливо відкрити внаслідок його відсутності, IOException – файл неможливо відкрити через помилку введення-виведення, ArgumentNullException – ім'ям файла є null-значення, ArgumentException – некоректний параметр mode, SecurityException – користувач не володіє правами доступу, DirectoryNotFoundException – некоректно заданий каталог.

Інша версія конструктора дозволяє обмежити доступ тільки читанням або тільки записом:

FileStream (string filename, FileMode mode, FileAccess how)

де:

1) параметри filename і mode мають те ж призначення, що і в попередній версії конструктора;

2) параметр how визначає спосіб доступу до файла і може приймати одне із значень, визначених переліком FileAccess:

a) FileAccess.Read – тільки читання;
b) FileAccess.Write – тільки запис;
c) FileAccess.ReadWrite – і читання, і запис.

Після встановлення зв'язку байтового потоку з фізичним файлом внутрішній покажчик потоку встановлюється на початковий байт файла.

Для читання чергового байта з потоку, пов'язаного з фізичним файлом, використовується метод ReadByte (). Після прочитання чергового байта внутрішній покажчик переміщається на наступний байт файла. Якщо досягнуто кінця файла, то метод ReadByte() повертає значення – 1.

Для побайтового запису даних у потік використовується метод WriteByte ().

Після закінчення роботи з файлом його необхідно закрити. Для цього достатньо викликати метод Close (). При закритті файла звільняються системні ресурси, раніше виділені для цього файла, що дає можливість використовувати їх для роботи з іншими файлами.

Розглянемо приклад використання класу FileStream, для копіювання одного файла в інший. Але спочатку створимо текстовий файл text.txt у папці bin/debug поточного проекту. І внесемо до нього довільну інформацію, наприклад:

```
12   456
Hello!
23,67 4: Message
```

```
using System;
using System.Text;
using System.IO;        //для роботи з потоками
```

```
namespace MyProgram
{
  class Program
  {
    static void Main()
    {
      try
      {
      FileStream fileIn = new FileStream("text.txt", FileMode.Open,
FileAccess.Read);
      FileStream fileOut = new FileStream("newText.txt", FileMode.Create,
FileAccess.Write);
      int i;
      while ((i = fileIn.ReadByte())!=-1)
      {
         //запис чергового файла в потік, пов'язаний з файлом flleOut
         fileOut.WriteByte((byte)i);
      }
      fileIn.Close();
      fileOut.Close();
      }
      catch (Exception EX)
      {
         Console.WriteLine(EX.Message);
      }
    }
  }
}
```

Символьний потік

Щоб створити символьний потік потрібно помістити об'єкт класу Stream (наприклад, FileStream) "всередину" об'єкта класу StreamWriter або об'єкта класу StreamReader. В цьому випадку байтовий потік автоматично перетворюватиметься в символьний.

Клас StreamWriter призначений для організації вихідного символьного потоку. У ньому визначено декілька конструкторів. Один з них записується таким чином:

StreamWriter(Stream stream);

де параметр stream визначає ім'я вже відкритого байтового потоку.

Наприклад, створити екземпляр класу StreamWriter можна таким чином:

StreamWriter fileOut=new StreamWriter (new FileStream ("text.txt", FileMode. Create, FileAccess.Write));

Цей конструктор генерує виключення типу ArgumentException, якщо потік stream не відкритий для виводу, і виключення типу Argument NullException, якщо він (потік) має null-значення.

Інший вид конструктора дозволяє відкрити потік відразу через звернення до файла:

StreamWriter(string name);

де параметр name визначає ім'я файла, що відкривається.

Наприклад, звернутися до даного конструктора можна таким чином:

StreamWriter fileOut=new StreamWriter("c:\temp\t.txt");

І ще один варіант конструктора StreamWriter:

StreamWriter(string name, bool appendFlag);

де параметр name визначає ім'я файла, що відкривається;

параметр appendFlag може набувати значення true – якщо потрібно додавати дані в кінець файла, або false – якщо файл необхідно перезаписати.

Наприклад:

StreamWriter fileOut=new StreamWriter ("t.txt", true);

Тепер для запису даних у потік fileOut можна зернутися до методу WriteLine. Це можна зробити таким чином:

fileOut.WriteLine("test");

У даному випадку в кінець файла t.txt буде дописано слово test.

Клас StreamReader призначений для організації вхідного символьного потоку. Один з його конструкторів виглядає таким чином:

StreamReader (Stream stream);

де параметр stream визначає ім'я вже відкритого байтового потоку.

Цей конструктор генерує виключення типу ArgumentException, якщо потік stream не відкритий для введення.

Наприклад, створити екземпляр класу StreamWriter можна таким чином:

StreamReader fileIn = new StreamReader (new FileStream ("text.txt", FileMode.Open, FileAccess.Read));

Як і у випадку з класом StreamWriter біля класу StreamReader є і інший вид конструктора, який дозволяє відкрити файл безпосередньо:

StreamReader (string name);

де параметр name визначає ім'я файла, що відкривається.

Звернутися до даного конструктора можна таким чином:

StreamReader fileIn=new StreamReader ("c:\temp\t.txt");

У C# символи реалізуються кодуванням Unicode. Для того, щоб можна було обробляти текстові файли, що містять символи кирилиці, створені, наприклад, в Блокноті, рекомендується викликати наступний вид конструктора StreamReader:

StreamReader fileIn=new StreamReader ("c:\temp\t.txt", Encoding.GetEncoding(1251));

Параметр Encoding.GetEncoding(1251) говорить про те, що виконуватиметься перетворення з коду Windows-1251 (одна з модифікацій коду ASCII, що містить символи кирилиці) в Unicode. Encoding.GetEncoding(1251) реалізований в просторі імен System.Text.

Тепер для читання даних з потоку fileIn можна скористатися методом ReadLine. При цьому якщо буде досягнутий кінець файла, то метод ReadLine поверне значення null.

Розглянемо приклад, в якому дані з одного файла копіюються в інший, але вже з використанням класів StreamWriter і StreamReader.

```
static void Main ()
{
        StreamReader fileIn = new StreamReader ("text.txt",
Encoding.GetEncoding(1251));
        StreamWriter fileOut=new StreamWriter ("newText.txt", false);
        string line;
        while ((line=fileIn.ReadLine())!=null) //поки потік не порожній
        {
                fileOut.WriteLine(line);
```

```
        }
        fileIn.Close();
        fileOut.Close();
    }
```

Таким чином, даний спосіб копіювання одного файла в інший дасть нам той же результат, що і при використанні байтових потоків. Проте, його робота буде менш ефективною, оскільки витрачатиметься додатковий час на перетворення байтів у символи. Але у символьних потоків є свої переваги. Наприклад, можна використовувати регулярні вирази для пошуку заданих фрагментів тексту у файлі.

```
static void Main()
{
    StreamReader fileIn = new StreamReader ("text.txt");
    StreamWriter fileOut=new StreamWriter ("newText.txt", false);
    string text=fileIn.ReadToEnd();
    Regex r= new Regex (@"[-+]?\d+");
    Match integer = r.Match(text);
    while (integer.Success)
    {
        fileOut.WriteLine(integer);
        integer = integer.NextMatch();
    }
    fileIn.Close();
    fileOut.Close();
}
```

Двійкові потоки

Двійкові файли зберігають дані в тому ж вигляді, в якому вони представлені в оперативній пам'яті, тобто у внутрішньому представленні. Двійкові файли не застосовуються для перегляду людиною, вони використовуються тільки для програмної обробки.

Вихідний потік BinaryWriter підтримує довільний доступ, тобто є можливість виконувати запис у довільну позицію двійкового файла. Найбільш важливі методи потоку BinaryWriter наведені у табл. 4.3.

Таблиця 4.3

Основні методи потоку BinaryWriter

Член класу	Опис
BaseStream	Визначає базовий потік, з яким працює об'єкт BinaryWriter
Close	Закриває потік
Flush	Очищає буфер
Seek	Встановлює позицію в поточному потоці
Write	Записує значення в поточний потік

Найбільш важливі методи вихідного потоку BinaryReader наведені у табл. 4.4.

Таблиця 4.4

Основні методи потоку BinaryReader

Член класу	Опис
BaseStream	Визначає базовий потік, з яким працює об'єкт BinaryReader
Close	Закриває потік
PeekChar	Повертає наступний символ потоку без переміщення внутрішнього покажчика в потоці
Read	Прочитує черговий потік байтів або символів і зберігає в масиві, передаваному у вхідному параметрі
ReadBoolean, ReadByte, ReadInt32 і т.д	Прочитує з потоку дані певного типу

Двійковий потік відкривається на основі базової протоки (наприклад, FileStream), при цьому двійковий потік перетворюватиме байтовий потік у значення int-, double-, short- і т. д.

Розглянемо приклад формування двійкового файла:

```
static void Main ()
{
    //відкриваємо двійковий потік
    BinaryWriter fOut=new BinaryWriter (new FileStream
("t.dat",FileMode.Create));
    //записуємо дані в двійковий потік
    for (int i=0; i<=100; i+=2)
```

```
        {
            fOut.Write(i);
        }
        fOut.Close(); //закриваємо двійковий потік
    }
```

Спроба проглянути двійковий файл за допомогою текстового редактора є неінформативною. Двійковий файл можна переглянути програмним шляхом наприклад таким чином:

```
    static void Main ()
    {
        FileStream f=new FileStream ("t.dat",FileMode.Open);
        BinaryReader fIn=new BinaryReader (f);
        long n=f.Length/4; //визначаємо кількість чисел у двійковому
потоці
        int a;
        for (int i=0; i<n; i++)
        {
            a=fIn.ReadInt32();
            Console.Write(a+" ");
        }
        fIn.Close();
        f.Close();
    }
```

Двійкові файли є файлами з довільним доступом, при цьому нумерація елементів у двійковому файлі ведеться з нуля. Довільний доступ забезпечує метод Seek. Розглянемо його синтаксис:

```
Seek (long newPos, SeekOrigin pos)
```

де параметр newPos визначає нову позицію внутрішнього покажчика файла в байтах щодо вихідної позиції покажчика, яка визначається параметром pos. У свою чергу параметр pos має бути заданий одним із значень перерахування SeekOrigin, які наведені в табл. 4.5.

Таблиця 4.5

Значення перерахування SeekOrigin

Значення	Опис
SeekOrigin.Begin	Пошук від початку файла
SeekOrigin.Current	Пошук від поточної позиції покажчика
SeekOrigin.End	Пошук від кінця файла

Після виклику методу Seek наступні операції читання або запису виконуватимуться з нової позиції внутрішнього покажчика файла.

Розглянемо приклад організації довільного доступу до двійкового файла (на прикладі файла t.dat):

```
static void Main()
{
    //зміна даних у двійковому потоці
    FileStream f=new FileStream ("t.dat",FileMode.Open);
    BinaryWriter fOut=new BinaryWriter (f);
    long n=f.Length; //визначаємо кількість байт у байтовому потоці
    int a;
    //зсув на дві позиції, оскільки тип int займає 4 байти
    for (int i=0; i<n; i+=8)
    {
        fOut.Seek(i,SeekOrigin.Begin);
        fOut.Write(0);
    }
    fOut.Close();
    //читання даних з двійкового потоку
    f=new FileStream ("t.dat",FileMode.Open);
    BinaryReader fIn=new BinaryReader (f);
    n=f.Length/4; // визначаємо кількість чисел у двійковому потоці
    for (int i=0; i<n; i++)
    {
        a=fIn.ReadInt32();
        Console.Write(a+" ");
    }
    fIn.Close();
    f.Close();
}
```

Потік BinaryReader не має методу Seek, проте використовуючи можливості потоку FileStream можна організувати довільний доступ при читанні двійкових файлів. Розглянемо наступний приклад:

```
static void Main ()
```

```
{
    //Записуємо у файл t.dat цілі числа від 0 до 100
    FileStream f=new FileStream ("t.dat",FileMode.Open);
    BinaryWriter fOut=new BinaryWriter (f);
    for (int i=0; i<100; ++i)
    {
       fOut.Write(i);;
    }
    fOut.Close();
    //Об'єкти f і fIn пов'язані з одним і тим же файлом
    f=new FileStream ("t.dat",FileMode.Open);
    BinaryReader fIn=new BinaryReader (f);
    long n=f.Length; // визначаємо кількість байт у потоці
    //Читаємо дані з файла t.dat, переміщаючи внутрішній покажчик на 8
байт, тобто на два цілі числа
    for (int i=0; i<n; i+=8)
    {
       f.Seek(i,SeekOrigin.Begin);
       int a=fIn.ReadInt32();
       Console.Write(a+" ");
    }
    fIn.Close();
    f.Close();
}
```

Перенаправлення стандартних потоків

Трьома стандартними потоками, доступ до яких здійснюється через властивості Console.Out, Console.In і Console.Error, можуть користуватися всі програми, що працюють в просторі імен System. Властивість Console.Out відноситься до стандартного вихідного потоку. За умовчанням це консоль. Наприклад, при виклику методу Console.WriteLine() інформація автоматично передається в потік Console.Out. Властивість Console.In відноситься до стандартного вхідного потоку, джерелом якого за умовчанням є клавіатура. Наприклад, при введенні даних з клавіатури інформація автоматично передається потоку Console.In, до якого можна звернутися за допомогою методу Console.ReadLine(). Властивість Console.Error відноситься до помилок у стандартному потоці, джерелом

якого також за умовчанням є консоль. Проте ці потоки можуть бути перенаправлені на будь-який сумісний пристрій введення-виведення, наприклад, на роботу з фізичними файлами.

Перенаправити стандартний потік можна за допомогою методів SetIn (), SetOut () і SetError (), які є членами класу Console:

```
static void SetIn (TextReader input)
static void SetOut (TextWriter output)
static void SetError (TextWriter output)
```

Приклад перенаправлення потоків проілюстрований наступною програмою, в якій двовимірний масив вводиться з файла input.txt, а виводиться у файл output.txt

```
static void Main ()
{
   try
   {
      int [,] MyArray;
      StreamReader file=new StreamReader ("input.txt");
      Console.SetIn(file);              // перенаправляємо стандартний вхідний потік на file
      string line=Console.ReadLine();
      string []mas=line.Split(' ');
      int n=int.Parse(mas[0]);
            int m=int.Parse(mas );
      MyArray = new int [n,m];
      for (int i = 0; i < n; i++)
      {
       line = Console.ReadLine();
       mas = line.Split(' ');
       for (int j = 0; j < m; j++)
       {
            MyArray [i,j]= int.Parse(mas [j]);
       }
      }
      PrintArray ("початковий масив:", MyArray, n, m);
      file.Close();
```

```
}

static void PrintArray (string a, int [,] mas, int n, int m)
{
   StreamWriterfile=new StreamWriter ("output.txt"); // перенаправляємо
стандартний вхідний потік на file
   Console.SetOut(file);
   Console.WriteLine(a);
   for (int i = 0; i < n; i++)
   {
      for (int j=0; j<m; j++) Console.Write("{0} ", mas [i,j]);
      Console.WriteLine();
   }
   file.Close();
}
```

___input.txt_____

```
3 4
1 4 2 8
4 9 0 1
5 7 4 2
```

За необхідності відновити початковий стан потоку Console.In можна таким чином:

```
TextWriter str = Console.In; // спочатку зберігаємо початковий стан
вхідного потоку
Console.SetIn(str);    // за необхідності відновлюваний початковий стан
вхідного потоку
```

Аналогічним чином можна відновити початковий стан потоку Console.Out:

```
TextWriter str = Console.Out;  // спочатку зберігаємо початковий стан
вихідного потоку
// за необхідності відновлюємо початковий стан вихідного потоку
Console.SetOut(str);
```

4.5. Колекції

Під *колекцією розуміють* групу об’єктів. Простір імен System.Collections містить безліч інтерфейсів і класів, які визначають і реалізують колекції різних типів. Колекції спрощують програмування, пропонуючи вже готові рішення для побудови структур даних, розробка яких "з нуля" відрізняється великою трудомісткістю. Мова йде про убудовані колекції, які підтримують, наприклад, функціонування стеків, черг і хеш-таблиць.

Основна перевага колекцій полягає в тому, що вони стандартизують спосіб обробки груп об'єктів у прикладних програмах. Усі колекції розроблені на основі набору чітко певних інтерфейсів. Ряд вбудованих реалізацій таких інтерфейсів, як ArrayList, Hashtable, Stack і Queue, можна використовувати "як є". У кожного програміста також є можливість реалізувати власну колекцію, але в більшості випадків досить убудованих.

Платформа Microsoft .NET Framework підтримує три основних типи колекцій: загального призначення, спеціалізовані й орієнтовані на побітову організацію даних. Колекції загального призначення реалізують ряд основних структур даних, включаючи динамічний масив, стек і чергу. Сюди також відносяться словники, призначені для зберігання пар ключ/значення. Колекції загального призначення працюють із даними типу object, тому їх можна використовувати для зберігання даних будь-якого типу.

Колекції спеціального призначення орієнтовані на обробку даних конкретного типу або на обробку унікальним способом. Наприклад, існують спеціалізовані колекції, призначені тільки для обробки рядків або односпрямованого списку.

Класи колекцій, орієнтованих на побітову організацію даних, служать для зберігання груп бітів. Колекції цієї категорії підтримують такий набір операцій, що не характерний для колекцій інших типів. Наприклад, у біт-орієнтованої колекції Bit Array визначені такі побітові операції, як "і" та "або".

Основним для всіх колекцій є реалізація *перечисельника (нумератора),* що підтримується інтерфейсами IEnumerator і IEnumerable.

Перечисельник забезпечує стандартизований спосіб поелементного доступу до вмісту колекції. Оскільки кожна колекція повинна реалізувати інтерфейс IEnumerable, до елементів будь-якого класу колекції можна

одержати доступ за допомогою методів, визнаних в інтерфейсі IEnumerator. Отже, після внесення невеликих змін код, що дозволяє циклічно опитувати колекцію одного типу, можна успішно використовувати для циклічного опитування колекції іншого типу.

Простір імен System.Collections містить класи й інтерфейси, які визначають різні колекції об'єктів. Основні класи та інтерфейси колекцій наведені у табл. 4.6.

Таблиця 4.6

Основні класи та інтерфейси колекцій

Клас	Опис
1	2
ArrayList	Служить для реалізації інтерфейсу IList за допомогою масиву з динамічною зміною розміру на вимогу
BitArray	Управляє компактним масивом двійкових значень, представлених логічними величинами, де значення true відповідає 1, а значення false відповідає 0
CaseInsensitiveComparerpa	Перевіряє рівність двох об'єктів без урахування регістру рядків
CaseInsensitiveHashCodeProvider	Надає хеш-код об'єкта, використовуючи алгоритм хешування, при якому не враховується регістр рядків
CollectionBase	Надає абстрактний базовий клас для колекції зі строгим типом
Comparer	Перевіряє рівність двох об'єктів з урахуванням регістра рядків
DictionaryBase	Надає абстрактний базовий клас для колекції пар "ключ-значення" зі строгим типом
Hashtable	Надає колекцію пар "ключ-значення", які впорядковані за хеш-кодом ключа
Queue	Надає колекцію об'єктів, що обслуговується за принципом "першим прийшов — першим вийшов" (FIFO)
ReadOnlyCollectionBase	Надає абстрактний базовий клас для колекції зі строгим типом, що доступна тільки для читання

Закінчення табл. 4.6

1	2
SortedList	Надає колекцію пар "ключ-значення", які впорядковані по ключах. Доступ до пар можна одержати за ключем та за індексом
Stack	Представляє колекцію об'єктів, що обслуговується за принципом "останнім прийшов — першим вийшов" (LIFO)
ICollection	Визначає розмір, перечисельники й методи синхронізації для всіх колекцій
IComparer	Надає іншим додаткам метод для порівняння двох об'єктів
IDictionary	Надає колекцію пар "ключ-значення"
IDictionaryEnumerator	Здійснює нумерацію елементів словника
IEnumerable	Надає перечисельник, що підтримує просте переміщення по колекції
IEnumerator	Підтримує просте переміщення по колекції
IHashCodeProvider	Надає хеш-код об'єкта, використовуючи користувальницьку хеш-функцію
IList	Надає колекцію об'єктів, до яких можна одержати доступ окремо, за індексом

Інтерфейси колекцій

У просторі імен System. Collections визначена безліч інтерфейсів. Розглянемо інтерфейси колекцій, оскільки вони визначають функції, загальні для всіх класів колекцій.

Інтерфейс ICollection

Інтерфейс ICollection можна назвати фундаментом, на якому побудовані всі колекції. У ньому оголошені основні методи й властивості, без яких не може обійтися жодна колекція. Він успадковує інтерфейс IEnumerable. Не знаючи суті інтерфейсу ICollection, неможливо зрозуміти механізм дії колекції.

В інтерфейсі ICollection визначені наступні властивості:

int Count { get; } – кількість елементів колекції в цей момент;

bool isSynchronized { get; } – приймає значення true, якщо колекція синхронізована, і значення false у противному випадку.

За умовчанням колекції не синхронізовані. Але для більшості колекцій можна одержати синхронізовану версію:

object syncRoot { get; } – об'єкт, для якого колекція може бути синхронізована.

Властивість Count – найбільш затребувана, оскільки містить кількість елементів, збережених у колекції в цей момент. Якщо властивість Count дорівнює нулю, виходить, колекція порожня. В інтерфейсі ICollection визначений наступний метод:

void CopyTo (Array *target,* mt *startldx).*

Метод CopyTo () копіює вміст колекції в масив, заданий параметром *target,* починаючи з індексу, заданого параметром startldx. Можна сказати, що метод CopyTo () забезпечував перехід від колекції до стандартного масиву.

Оскільки інтерфейс ICollection успадковує інтерфейс IEnumerable, він також включає його єдиний метод GetEnumerator ( ) :

IEnumerator GetEnumerator()

Цей метод повертає нумератор колекції.

Інтерфейс IList

Інтерфейс IList успадковує інтерфейс ICollection і визначає поводження колекції, доступ до елементів якої дозволений за допомогою індексу з відліком від нуля. Крім методів, визначених в інтерфейсі ICollection, інтерфейс Ilist визначає й власні методи. Деякі із цих методів служать для модифікації колекції. Якщо ж колекція призначена тільки для читання або має фіксований розмір, виклик цих методів приведе до генерування виключення типу NotSupportedException.

Інтерфейс IDictionary

Інтерфейс IDictionary визначає поводження колекції, що встановлює відповідність між унікальними ключами й значеннями. Ключ – це об'єкт, що використовується для одержання відповідного йому значення. Отже, колекція, що реалізує інтерфейс IDictionary, служить для зберігання пар ключ/значення. Збережену один раз пару можна потім витягти за заданим ключем. Інтерфейс IDictionary успадковує інтерфейс ICollection.

Класи колекцій загального призначення

Класи колекцій діляться на три основних категорії: загального призначення, спеціалізовані й орієнтовані на побітову організацію даних. Класи загального призначення можна використовувати для зберігання об'єктів будь-якого типу. Бітові призначені для зберігання бітової інформації. Колекції спеціального призначення розробляються для обробки даних конкретного типу і наведені у табл. 4.7.

Таблиця 4.7

Класи колекцій загального призначення

Клас	***Опис***
Stack	Стек – окремий випадок однонаправленого списку, що діє за принципом: останнім прийшов – першим вийшов
Queue	Черга – окремий випадок однонаправленого списку, що діє за принципом: першим прийшов – першим вийшов
ArrayList	Динамічний масив, тобто масив, який за необхідності може збільшувати свій розмір
Hash table	Хеш-кодування-таблиця для пар ключ/значення
SortedList	Відсортований список пар ключ/значення

Розглянемо дані колекції детальніше.

Зауваження. Абстрактний тип даних (АТД) *список* – це послідовність елементів a_1, a_2.,a_n ($n \geq 0$) одного типу. Кількість елементів *n* називається *довжиною списку*. Якщо $n > 0$, то a_1 називається *першим елементом списку*, а a_n – *останнім елементом списку*. У разі $n = 0$ маємо *порожній список*, який не містить елементів. Важлива властивість списку полягає в тому, що його елементи лінійно впорядковані відповідно до їх позиції в списку. Так елемент a_i *передує* a_{i+1} для i=1, 2 .n-1 і a_i *слідує* за a_{i-1} для i=2 .n. Список називається *однонаправленим,* якщо кожен елемент списку містить посилання на наступний елемент. Якщо кожен елемент списку містить два посилання (одну на наступний елемент в списку, другу – на попередній елемент), то такий список називається *двонаправленим* (двозв'язковим). А якщо останній елемент зв'язати покажчиком з першим, то вийде кільцевий список.

Клас Stack

АТД стек – це окремий випадок однонаправленого списку, додавання елементів у який і вибірка елементів з якого виконуються з одного кінця, так званого вершиною стека (головою – head). При вибірці елемент виключається із стека. Інші операції із стеком не визначені. Говорять, що стек реалізує принцип обслуговування LIFO (last in – fist out, останнім прийшов, – першим вийшов). Стек найпростіше уявити собі у вигляді піраміди, на яку надягають кільця (рис. 4.1).

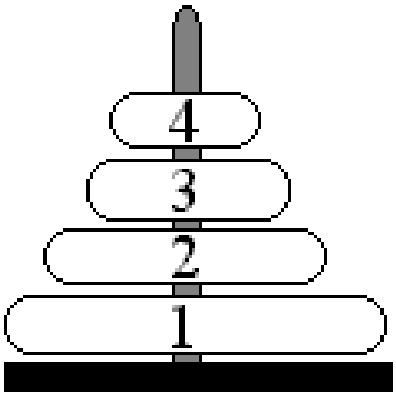

Рис. 4.1. **Представлення стеку у вигляді піраміди**

Дістати перше кільце можна тільки після того, як будуть зняті всі верхні кільця.

У C# реалізацію АТД стек представляє клас Stack, який реалізує інтерфейси ICollection, IEnumerable і ICloneable. Stack – це динамічна колекція, розмір якої змінюється.

У класі Stack визначені наступні конструктори:

public Stack(); // створює порожній стек, початкова місткість якого дорівнює 10

public Stack(int capacity); // створює порожній стек, початкова місткість якого рівна capacity

public Stack(ICollection c); // створює стек, який містить елементи колекції, заданої параметром c

Окрім методів, визначених в інтерфейсах, що реалізовуються класом Stack, в цьому класі визначені власні методи, які наведені у табл. 4.8.

Таблиця 4.8

Методи класу Stack

Метод	Опис
public virtual bool Contains(object v)	Повертає значення true, якщо об'єкт v міститься в стеку, що визивається, інакше повертає значення false
public virtual void Clear()	Встановлює властивість Count рівним нулю, тим самим очищаючи стек
public virtual object Peek()	Повертає елемент, розташований у вершині стека, але не витягуючи його із стека
public virtual object Pop()	Повертає елемент, розташований у вершині стека, і витягує його із стека
public virtual void Push(object v)	Поміщає об'єкт v у стек
public virtual object[] ToArray()	Повертає масив, який містить копії елементів стека, що визивається

Розглянемо декілька прикладів використання стека.

Приклад 1. Для заданого значення n запишемо в стек всі числа від 1 до n, а потім витягуватимемо із стека:

```
using System;
using System.Collections;

namespace ConsoleApplication
{
  class Program
  {
    public static void Main ()
    {
      Console.Write("n=");
      int n=int.Parse(Console.ReadLine());
      Stack intStack = new Stack();
      for (int i = 1; i <= n; i++)
        intStack.Push(i);
      Console.WriteLine("Розмірність стека " + intStack.Count);

      Console.WriteLine("Верхній елемент стека = " + intStack.Peek());
      Console.WriteLine("Вміст стека " + intStack.Count);
```

```
      Console.Write("Вміст стека = ");
      while (intStack.Count != 0)
      Console.Write("{0} ", intStack.Pop());
      Console.WriteLine("\nНова розмірність стека " + intStack.Count);
    }
  }
}
```

Приклад 2. У текстовому файлі міститься математичний вираз. Перевірити баланс круглих дужок у даному виразі.

```
using System;
using System.Collections;
using System.IO;

namespace MyProgram
{
  class Program
  {
    public static void Main()
    {
      StreamReader fileIn=new StreamReader("t.txt",);
      string line=fileIn.ReadToEnd();
      fileIn.Close();
      Stack skobki=new Stack();
      bool flag=true;
        //перевіряємо баланс дужок
      for ( int i=0; i<line.Length;i++)
      {
        //якщо поточний символ дужка, що відкривається, то поміщаємо її в
стек
        if (line[i]== '(') skobki.Push(i);
          else if (line[i]== ')') // якщо поточний символ дужка, що
закривається, то
        {
        //якщо стек порожній, то для дужки, що закривається, не вистачає
парної, що відкривається
```

```
        if (skobki.Count == 0)
          { flag = false; Console.WriteLine("Можливо в позиції " + i + "зайва ) дужка"); }
            else skobki.Pop(); //інакше витягуємо парну дужку
        }
      }
      //якщо після проглядання рядка стек виявився порожнім, то дужки збалансовані
      if (skobki.Count == 0) { if (flag) Console.WriteLine("дужки збалансовані"); }
      else // інакше баланс дужок порушений
      {
        Console.Write("Можливо зайва ( дужка в позиції:");
        while (skobki.Count != 0)
        {
              Console.Write("{0} ", (int)skobki.Pop());
        }
        Console.WriteLine();
      }
    }
  }
}
```

_____t.txt_____

```
(1+2)-4*(a-3)/(2-7+6)
```

Клас Queue

АТД черга – це окремий випадок однонаправленого списку, додавання елементів у який виконується в один кінець (хвіст), а вибірка проводиться з іншого кінця (голови). Інші операції з чергою не визначені. При вибірці елемент виключається з черги. Говорять, що чергу реалізує принцип обслуговування FIFO (fist in – fist out, першим прийшов, – першим вийшов). Чергу найпростіше представити у вигляді вузької труби, в один кінець якою кидають м'ячі, а з іншого кінця якої вони вилітають (рис. 4.2). Зрозуміло, що м'яч, який був кинутий у трубу першим, першим і вилетить з іншого кінця.

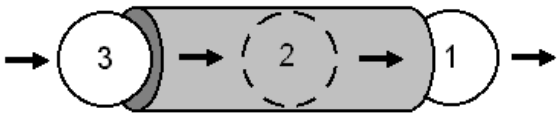

Рис. 4.2. **Представлення черги**

У C# реалізацію АТД чергу представляє клас Queue, який також як і стек реалізує інтерфейси ICollection, IEnumerable і ICloneable. Queue – це динамічна колекція, розмір якої змінюється. За необхідності збільшення місткості черги відбувається з коефіцієнтом зростання за умовчанням рівним 2.0.

У класі Queue визначені наступні конструктори:

public Queue(); //створює порожню чергу, початкова місткість якої дорівнює 32
public Queue (int capacity); // створює порожню чергу, початкова місткість якої рівна capacity
public Queue (int capacity, float n);//створює порожню чергу, початкова місткість якої рівна capacity, і коефіцієнт зростання
//встановлюється параметром n
public Queue (ICollection c);//створює чергу, яка містить елементи колекції, заданої параметром c, і аналогічною
//місткістю

Окрім методів, визначених в інтерфейсах, що реалізовуються класом Queue, в цьому класі визначені власні методи, які наведені у табл. 4.9.

Таблиця 4.9

Методи класу Queue

Метод	Опис
1	2
public virtual bool Contains (object v)	Повертає значення true, якщо об'єкт v міститься в черзі, що визивається, інакше повертає значення false
public virtual void clear ()	Встановлює властивість Count рівним нулю, тим самим очищаючи чергу

Закінчення табл. 4.9

1	2
public virtual object Dequeue ()	Повертає об'єкт з початку черги, що визивається, видаляючи його з черги
public virtual object Peek ()	Повертає об'єкт з початку черги, що визивається, не видаляючи його з черги
public virtual void Enqueue(object v	Додає об'єкт v в кінець черги
public virtual object [ ] ToArray ()	Повертає масив, який містить копії елементів із черги
public virtual void TrimToSize()	Встановлює властивість Capacity рівним значенню властивості Count

Розглянемо декілька прикладів використання стека.

Приклад 1. Для заданого значення n запишемо в чергу всі числа від 1 до n, а потім витягуватимемо їх з черги:

```
using System;
using System.Collections;
namespace MyProgram
{
  class Program
  {
    public static void Main()
    {
      Console.Write("n=");
      int n=int.Parse(Console.ReadLine());
      Queue intQ = new Queue();
      for (int i = 1; i <= n; i++)
        intQ.Enqueue(i);
      Console.WriteLine("Розмірність черги " + intQ.Count);

      Console.WriteLine("Верхній елемент черги = " + intQ.Peek());
      Console.WriteLine("Розмірність черги " + intQ.Count);

      Console.Write("Вміст черги = " );
      while (intQ.Count!=0)
        Console.Write("{0}  ", intQ.Dequeue());
```

```
      Console.WriteLine("\nНова розмірність черги " + intQ.Count);
    }
  }
}
```

Приклад 2. У текстовому файлі записана інформація про людей (прізвище, ім'я, по батькові, вік, вага через пробіл). Вивести на екран спочатку інформацію про людей молодше 40 років, а потім інформацію про всіх останніх.

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace MyProgram
{
  class Program
  {
    public struct one //структура для зберігання даних про одну людину
    {
      public string f;
      public string i;
      public string про;
      public int age;
      public float massa;
    }

    public static void Main()
    {
        StreamReader fileIn = new
StreamReader("t.txt",Encoding.GetEncoding(1251));
        string line;
        Queue people = new Queue();
        one a;
        Console.WriteLine("ВІК МЕНШЕ 40 РОКІВ");
        while ((line = fileIn.ReadLine()) != null) //читаємо до кінця файла
        {
              string [] temp = line.Split(' '); //розбиваємо рядок на складові
елементи
```

```
            //заповнюємо структуру
            a.f = temp[0];
            a.i = temp ;
            a.o = temp ;
            a.age = int.Parse(temp );
            a.massa = float.Parse(temp );
            // якщо вік менше 40 років, то виводимо дані на екран, інакше
поміщаємо їх у
            //чергу для тимчасового зберігання
            if (a.age<40)
                  Console.WriteLine(a.f + "\t"+ a.i + "\t"+ a.o + "\t"+a.age + "\t"
+ a.massa);
            else people.Enqueue(a);
      }
      fileIn.Close();

      Console.WriteLine("ВІК 40 РОКІВ І СТАРШЕ");
      while (people.Count != 0) //витягуємо з черги дані
      {
            a = (one)people.Dequeue();
            Console.WriteLine(a.f + "\t"+ a.i + "\t"+ a.o + "\t"+a.age + "\t" +
a.massa);
      }

   }
  }
}
```

_____t.txt_____

```
Іванов Сергій Миколайович 21 64
Петров Ігор Юрійович 45 88
Семенов Михайло Олексійович 20 70
Піманов Олександр Дмитрович 53 101
```

Клас ArrayList

У C# стандартні масиви мають фіксовану довжину, яка не може змінитися під час виконання програми. Клас ArrayList призначений для

підтримки динамічних масивів, які за необхідності можуть збільшуватися або скорочуватися.

Об'єктом класу ArrayList є масив змінної довжини, елементами якого є об'єктні посилання. Будь-який об'єкт класу ArrayList створюється з деяким початковим розміром. При перевищенні цього розміру колекція автоматично подвоюється. У разі видалення об'єктів масив можна скоротити.

Клас ArrayList реалізує інтерфейси ICollection, IList, IEnumerable і ICloneable. У класі ArrayList визначені наступні конструктори:

```
//створює порожній масив з максимальною ємкістю рівної 16 елементам, при поточній розмірності 0
public ArrayList()
public ArrayList(int capacity) // створює масив із заданою ємкістю capacity, при поточній розмірності 0
public ArrayList(ICollection c)        // створює масив, який ініціалізувався елементами колекції c
```

Окрім методів, визначених у інтерфейсах, які реалізує клас ArrayList, в ньому визначені і власні методи, наведені у табл. 4.10.

Таблиця 4.10

Методи класу ArrayList

Метод	Опис
1	2
public virtual void AddRange (ICollection c)	Додає елементи з колекції c в кінець колекції, що визивається
public virtual int BinarySearch (object v)	У відсортованій колекції, що визивається, виконує пошук значення, заданого параметром v. Повертає індекс знайденого елементу. Якщо шукане значення не виявлене, повертає від'ємне значення
public virtual int BinarySearch (object v, IComparer comp)	У відсортованій колекції, що визивається, виконує пошук значення, заданого параметром v, на основі методу порівняння об'єктів, заданого параметром comp. Повертає індекс знайденого елементу. Якщо шукане значення не виявлене, повертає від'ємне значення

Продовження табл. 4.10

1	2
public virtual int BinarySearch (int startIdx nt count, object v, IComparer comp)	У відсортованій колекції, що визивається, виконує пошук значення, заданого параметром v, на основі методу порівняння об'єктів, заданого параметром comp. Пошук починається з елементу, індекс якого дорівнює значенню startIdx, і включає count елементів. Метод повертає індекс знайденого елементу. Якщо шукане значення не виявлене, повертає від'ємне значення
public virtual void CopyTo(Array ar, int startIdx)	Копіює вміст колекції, що визивається, починаючи з елементу, індекс якого дорівнює значенню startIdx, в масив, заданий параметром ar. Приймальний масив має бути одновимірним і сумісним за типом з елементами колекції
public virtual void CopyTo(int srcIdx, Array ar int destIdx, int count)	Копіює count елементів колекції, що визивається, починаючи з елементу, індекс якого дорівнює значенню srcIdx, в масив, заданий параметром ar, починаючи з елементу, індекс якого дорівнює значенню destIdx. Приймальний масив має бути одновимірним і сумісним за типом з елементами колекції
public virtual ArrayList GetRange(int idx int count)	Повертає частину колекції, що визивається, типу ArrayList. Діапазон повертаної колекції починається з індексу idx і включає count елементів. Повертаний об'єкт посилається на ті ж елементи, що і об'єкт, що визивається
public static ArrayList FixedSize(ArrayList ar)	Перетворює колекцію ar на ArrayList-масив з фіксованим розміром і повертає результат
public virtual void InsertRange(int startIdx ICollection c)	Вставляє елементи колекції, заданої параметром c, в колекцію, що визивається, починаючи с індексу, заданого параметром startIdx
public virtual int LastIndexOf(object v)	Повертає індекс останнього входження об'єкту v в колекції, що визивається. Якщо шуканий об'єкт не виявлений, повертає негативне значення
public static ArrayList Readonly(ArrayList ar)	Перетворює колекцію ar на ArrayList-массив, призначений тільки для читання
public virtual void ReraoveRange(int idx int count)	Видаляє count елементів із колекції, що визивається, починаючи з елементу, індекс якого дорівнює значенню idx
public virtual void Reverse()	Розташовує елементи колекції, що визивається, в зворотному порядку

Закінчення табл. 4.10

1	2
public virtual void Reverse(int startIdx, int count)	Розташовує в зворотному порядку count елементів колекції, що визивається, починаючи з індексу startIdx
public virtual void SetRange(int startIdx ICollection c	Замінює елементи колекції, що визивається, починаючи з індексу startIdx, елементами колекції, заданої параметром c
public virtual void Sort()	Сортує колекцію за збільшенням
public virtual void Sort(IComparer comp)	Сортує колекцію, що визивається, на основі методу порівняння об'єктів, заданого параметром comp. Якщо параметр comp має нульове значення, для кожного об'єкта використовується стандартний метод порівняння
Public virtual void Sort (int startidx, int endidx icomparer comp)	Сортує частину колекції, що визивається, на основі методу порівняння об'єктів, заданого параметром comp. Сортування починається з індексу startidx і закінчується індексом endidx. Якщо параметр comp має нульове значення, для кожного об'єкта використовується стандартний метод порівняння
public virtual object [ ] ToArray ()	Повертає масив, який містить копії елементів об'єкта, що визивається
public virtual Array ToArray (Type type)	Повертає масив, який містить копії елементів об'єкта, що визивається. Тип елементів у цьому масиві задається параметром type
public virtual void TrimToSize()	Встановлює властивість Capacity рівним значенню властивості Count

Властивість Capacity дозволяє дізнатися або встановити ємкість динамічного масиву, що визивається, типу ArrayList. Ємкістю є кількість елементів, які можна зберегти в ArrayList-масиві без його збільшення. Якщо вам заздалегідь відомо, скільки елементів повинно міститися в ArrayList-масиві, то розмірність масиву можна встановити використовуючи властивість Capacity, зекономивши тим самим системні ресурси. Якщо потрібно зменшити розмір ArrayList-масиву, то шляхом установки властивості Capacity можна зробити його меншим. Але встановлюване значення не має бути менше значення властивості Count, інакше згенерує виключення ArgumentOutOfRangeException. Щоб зробити ємкість ArrayList-масиву рівною дійсній кількості елементів, що зберігаються в ньому в даний момент, треба встановити значення властивості Capacity

рівним властивості Count. Того ж ефекту можна добитися, викликавши метод TrimToSize ().

Розглянемо декілька прикладів використання динамічного масиву.

```
using System;
using System.Collections;

namespace MyProgram
{
  class Program
  {
    static void ArrayPrint(string s, ArrayList a)
    {
      Console.WriteLine(s);
      foreach (int i in a)
        Console.Write(i + " ");
      Console.WriteLine();
    }

    static void Main(string[] args)
    {
      ArrayList myArray = new ArrayList();
      Console.WriteLine("Початкова ємкість масиву: " + myArray.Capacity);
      Console.WriteLine("Початкова кількість елементів: " + myArray.Count);

      Console.WriteLine("\nДобавили 5 цифр");
      for (int i = 0; i < 5; i++) myArray.Add(i);
      Console.WriteLine("Поточна ємкість масиву: " + myArra.Capacity);
      Console.WriteLine("Поточна кількість елементів: " + myArray.Count);
      ArrayPrint("Вміст масиву", myArray);

      Console.WriteLine("\nОптимізуємо ємкість масиву");
      myArray.Capacity=myArray.Count;
      Console.WriteLine("Поточна ємкість масиву: " + myArray.Capacity);
      Console.WriteLine("Поточна кількість елементів: " + myArray.Count);
      ArrayPrint("Вміст масиву", myArray);

      Console.WriteLine("\nДобавляємо елементи в масив");
```

```
      myArray.Add(10);
      myArray.Insert(1, 0);
      myArray.AddRange(myArray);
      Console.WriteLine("Поточна ємкість масиву: " + myArray.Capacity);
      Console.WriteLine("Поточна кількість елементів: " + myArray.Count);
      ArrayPrint("Вміст масиву", myArray);

      Console.WriteLine("\nВилучаємо елементи з масиву");
      myArray.Remove(0);
      myArray.RemoveAt(10);
      Console.WriteLine("Поточна ємкість масиву: " + myArray.Capacity);
      Console.WriteLine("Поточна кількість елементів: " + myArray.Count);
      ArrayPrint ("Вміст масиву", myArray);

      Console.WriteLine("\nВидаляємо весь масив");
      myArray.Clear();
      Console.WriteLine("Поточна ємкість масиву: " + myArray.Capacity);
      Console.WriteLine("Поточна кількість елементів: " + myArray.Count);
      ArrayPrint("Вміст масиву", myArray);
    }
  }
}
```

Приклад 2. У текстовому файлі записана інформація про людей (прізвище, ім'я, по батькові, вік, вага через пробіл). Вивести на екран інформацію про людей, відсортовану за віком.

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace MyProgram
{
  class Program
  {
    public struct one //структура для зберігання даних про одну людину
    {
```

```
    public string f;
    public string i;
    public string про;
    public int age;
    public float massa;
  }

  public class SortByAge : IComparer //реалізація стандартного
інтерфейсу
  {
    int IComparer.Compare(object x, object y) //перевизначення методу
Compare
    {
      one t1 = (one)x;
      one t2 = (one)y;
      if (t1.age > t2.age) return 1;
      if (t1.age < t2.age) return -1;
      return 0;
    }
  }

  static void ArrayPrint(string s, ArrayList a)
  {
    Console.WriteLine(s);
    foreach (one x in a)
      Console.WriteLine(x.f + "\t"+ x.i + "\t"+ x.o + "\t"+x.age + "\t" + x.massa);
  }

  static void Main(string[] args)
  {
      StreamReader fileIn = new
StreamReader("t.txt",Encoding.GetEncoding(1251));
      string line;
      one a;
      ArrayList people = new ArrayList();
      string[] temp = new string;
```

```
        while ((line=fileIn.ReadLine())!=null) //цикл для організації обробки
файла
        {
          temp = line.Split(' ');
          a.f = temp[0];
          a.i = temp;
          a.o = temp;
          a.age = int.Parse(temp );
          a.massa = float.Parse(temp );
          people.Add(a);
        }
        fileIn.Close();

        ArrayPrint("Початкові дані: ", people);
        people.Sort(new Program.SortByAge());    //виклик сортування
        ArrayPrint("Відсортовані дані: ", people);
    }
  }
}
```

_____t.txt_____

```
Іванов Сергій Миколайович 21 64
Петров Ігор Юрійович 45 88
Семенов Михайло Олексійович 20 70
Піманов Олександр Дмитрович 53 101
```

Зауваження. Зверніть увагу на те, що в даному прикладі був розроблений вкладений клас SortByAge, що реалізовує стандартний інтерфейс IComparer. У цьому класі був переобтяжений метод Compare, що дозволяє порівнювати між собою два об'єкти типу one. Створений клас використовувався для сортування колекції по заданому критерію (за віком).

Клас Hashtable

Клас Hashtable призначений для створення колекції, в якій для зберігання об'єктів використовується хеш-кодування-таблиця. У хеш-таблиці для зберігання інформації використовується механізм, що іменується хешуванням (hashing). Суть хешування полягає в тому, що для визначення унікального значення, яке називається хеш-кодом,

використовується інформаційний вміст відповідного йому ключа. Хеш-кодування-код потім використовується як індекс, по якому в таблиці відшукуються дані, відповідні цьому ключу. Перетворення ключа в хеш-кодування-код виконується автоматично, тобто саме хеш-кодування-код ви навіть не побачите. Але перевага хешування в тому, що воно дозволяє скорочувати час виконання таких операцій, як пошук, прочитування і запис даних, навіть для великих об'ємів інформації.

У класі Hashtable, окрім властивостей, визначених у реалізованих ним інтерфейсах, визначено дві власні public-властивості:

```
public virtual ICollection Keys { get; } //дозволяє отримати колекцію ключів
public virtual ICollection Values { get; } //дозволяє отримати колекцію значень
```

Для додавання елементу в хеш-таблицю необхідно викликати метод Add(), який приймає два окремі аргументи: ключ і значення. Важливо відзначити, що хеш-таблиця не гарантує збереження порядку елементів, оскільки хешування зазвичай не застосовується до відсортованих таблиць.

Розглянемо приклад, який демонструє використання Hashtable колекції:

Приклад 1: розглянемо прості операції з хеш-таблицею

```
using System;
using System.Collections;

namespace MyProgram
{
  class Program
  {

    static void printTab(string s, Hashtable a)
    {
      Console.WriteLine(s);
      ICollection key = a.Keys; //Прочитали всі ключі
      foreach (string i in key) //використання ключа для набуття значення
      {
        Console.WriteLine(i+"\t"+a[i]);
```

```
    }
    Console.WriteLine();
  }
static void Main(string[] args)
  {
```

Клас Hashtable реалізує стандартні інтерфейси IDictionary, ICollection, IEnumerable, ISerializable, IDeserializationCallback і ICloneable. Розмір хеш-таблиці може динамічно змінюватися. Розмір таблиці збільшується тоді, коли кількість елементів перевищує значення, рівне добутку місткості таблиці і її коефіцієнта заповнення, який може набувати значення на інтервалі від 0,1 до 1,0. За умовчанням встановлений коефіцієнт рівний 1,0.

У класі Hashtable визначено декілька конструкторів:

public Hashtable() // створює порожню хеш-кодування-таблицю

public Hashtable(IDictionary c)// будує хеш-таблицю, яка ініціалізувалася елементами колекції c

public Hashtable(int capacity) // створює хеш-таблицю з місткістю *capacity*

public Hashtable(int capacity, float n) //створює хеш-таблицю місткістю capacity і коефіцієнтом заповнення n

Окрім методів, визначених в інтерфейсах, які реалізує клас Hashtable, в ньому визначені і власні методи, які наведені у табл. 4.11.

Таблиця 4.11

Методи класу Hashtable

Метод	Опис
public virtual bool ContainsKey (object k)	Повертає значення true, якщо в хеш-таблиці, що визивається, міститься ключ, заданий параметром k. Інакше повертає значення false
public virtual bool ContainsValue (object v)	Повертає значення true, якщо в хеш-таблиці, що визивається, міститься значення, задане параметром v. Інакше повертає значення false
public virtual IDictionaryEnumerator GetEnumerator()	Повертає для хеш-таблиці, що визивається, нумератор типу IDictionaryEnumerator

```
      Hashtable tab = new Hashtable();
      Console.WriteLine("Початкова кількість елементів: " + tab.Count);
      printTab("Вміст таблиці: ", tab);

      Console.WriteLine("Додали в таблицю записи");
      tab.Add("001","ПЕРШИЙ");
      tab.Add("002","ДРУГИЙ");
      tab.Add("003","ТРЕТІЙ");
      tab.Add("004", "ЧЕТВЕРТИЙ");
      tab.Add("005", "П'ЯТИЙ");
      Console.WriteLine("Поточна кількість елементів: " + tab.Count);
      printTab("Вміст заповненої таблиці", tab);
      tab["005"]= "НОВИЙ П'ЯТИЙ";
      tab["001"]= "НОВИЙ ПЕРШИЙ";
      printTab("Вміст зміненої таблиці", tab);
    }
  }
}
```

Приклад 2. Розробимо простий записник, в який можна додавати і видаляти телефони, а також здійснювати пошук номера телефону за прізвищем і прізвищем по номеру телефону.

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace MyProgram
{
  class Program
  {
        static void printTab(string s, Hashtable a)
    {
      Console.WriteLine(s);
      ICollection key = a.Keys; //Прочитали всі ключі
      foreach (string i in key) //використання ключа для набуття значення
      {
        Console.WriteLine(i + "\t" + a[i]);
```

```
    }
  }

  static void Main(string[] args)
  {
    StreamReader fileIn = new
StreamReader("t.txt",Encoding.GetEncoding(1251));
    string line;
    Hashtable people = new Hashtable();
    while ((line = fileIn.ReadLine()) != null) //цикл для організації обробки
файла
    {
      string [] temp = line.Split(' ');
      people.Add(temp[0],temp );
    }
    fileIn.Close();
    printTab("Початкові дані: ", people);

    Console.WriteLine("Введіть номер телефону");
    line = Console.ReadLine();
    if (people.ContainsKey(line)) Console.WriteLine(line + "\t" +
people[line]);
    else
    {
      Console.WriteLine("Такого номера немає в записнику.\nВведіть
прізвище: ");
      string line2=Console.ReadLine();
      people.Add(line,line2);
    }
    printTab("Початкові дані: ", people);
    Console.WriteLine("Введіть прізвище для видалення");
    line = Console.ReadLine();
    if (people.ContainsValue(line))
    {
      ICollection key =people.Keys; //Прочитали всі ключі
      Console.WriteLine(line);
      string del="";
```

```
        foreach (string i in key) //використання ключа для набуття значення
            if (string.Compare((string)people[i], line) == 0)
            {
                del = i;
                break;
            }

        Console.WriteLine(del + "\t" + people[del]+ "- дані видалені!!!");
        people.Remove(del);
        printTab("Змінені дані: ", people);
      }
      else Console.WriteLine("Такого абонента в записнику немає ");
    }
  }
}
```

_____t.txt_____

```
12-34-56 Іванов
78-90-12 Петров
34-56-78 Семенов
90-11-12 Піманов
```

4.6. Рядки та регулярні вирази

Обробка текстової інформації є одним з найпоширеніших завдань сучасного програмування. C# надає для її вирішення широкий набір засобів: символи char, незмінні рядки string, змінні рядки StringBuider і регулярні вирази Regex.

Символьний тип char призначений для зберігання символу в кодуванні Unicode. Символьний тип відноситься до вбудованих типів даних C# і відповідає стандартному класу Char бібліотеки .Net з простору імен System. У цьому класі визначені статичні методи, що дозволяють задавати вигляд і категорію символу, а також перетворювати символ у верхній або нижній регістр, в число. Розглянемо основні методи, які наведені у табл. 4.12.

Таблиця 4.12

Основні методи класу Char

Метод	Опис
GetNumericValue	Повертає числове значення символу, якщо він є цифрою, і -1 інакше
GetUnicodeCategory	Повертає категорію Unicode-символа. У Unicode символи розділені на категорії, наприклад цифри (DecimalDigitNumber), римські цифри (LetterNumber), роздільники рядків (LineSeparator), букви в нижньому регістрі (LowercaseLetter) і т. д.
IsControl	Повертає true, якщо символ є таким, що управляє
IsDigit	Повертає true, якщо символ є десятковою цифрою
IsLetter	Повертає true, якщо символ є буквою
IsLetterOrDigit	Повертає true, якщо символ є буквою або десятковою цифрою
IsLower	Повертає true, якщо символ заданий у нижньому регістрі
IsNumber	Повертає true, якщо символ є числом (десятковим або шістнадцятирічним)
IsPunctuation	Повертає true, якщо символ є розділовим знаком
IsSeparator	Повертає true, якщо символ є роздільником
IsUpper	Повертає true, якщо символ заданий у нижньому регістрі
IsWhiteSpace	Повертає true, якщо символ є пробільним (пропуск, переклад рядка, повернення каретки)
Parse	Перетворить рядок у символ (рядок повинен складатися з одного символу)
ToLower	Перетворить символ у нижній регістр
ToUpper	Перетворить символ у верхній регістр

У наступному прикладі розглянемо застосування даних методів:

```
static void Main()
    {
      try
       {
        char b = 'B', з = '\x64', d = '\uffff';
        Console.WriteLine("{0}, {1}, {2}", b, з, d);
        Console.WriteLine("{0}, {1}, {2}", char.ToLower(b), char.ToUpper(c),
char.GetNumericValue(d));
        char a;
        do     //цикл виконується до тих пір, поки не ввели символ е
         {
```

```
        Console.WriteLine("Введіть символ: ");
        a = char.Parse(Console.ReadLine());
        Console.WriteLine("Введений символ {0}, його код  {1}, його
категорія {2}", a, (int) a, char.GetUnicodeCategory(a));
        if (char.IsLetter(a)) Console.WriteLine("Буква");
        if (char.IsUpper(a)) Console.WriteLine("Верхній регістр");
        if (char.IsLower(a)) Console.WriteLine("Нижній регістр");
        if (char.IsControl(a)) Console.WriteLine("Символ, що управляє");
        if (char.IsNumber(a)) Console.WriteLine("Число");
        if (char.IsPunctuation(a)) Console.WriteLine("Роздільник");
       } while (a != 'e');
     }
    catch
     {
         Console.WriteLine("Виникло виключення");
      }
  }
```

Використовуючи символьний тип можна організувати масив символів і працювати з ним на основі базового класу Array:

```
static void Main()
      {
         char[] a ={ 'm', 'a', 'X',  'i', 'M', 'u', 'S' '!', '!', '!' };
         char [] b="кол біля дзвону".ToCharArray(); //перетворення рядка в
масив символів
         PrintArray("Початковий масив a:", a);
         for (int x=0;x<a.Length; x++)
            if (char.IsLower(a[x])) a[x]=char.ToUpper(a[x]);
         PrintArray("Змінений масив a:", a);
         PrintArray("Початковий масив b:", b);
         Array.Reverse(b);
         PrintArray("Змінений масив b:", b);
     }

static void PrintArray(string line, Array a)
     {
```

```
        Console.WriteLine(line);
        foreach( object x in a) Console.Write(x);
        Console.WriteLine('\n');
    }
```

Тип string, призначений для роботи із стоками символів в кодуванні Unicode, є вбудованим типом C#. Йому відповідає базовий тип класу System.String бібліотеки .Net. Кожен об'єкт string – це незмінна послідовність символів Unicode, тобто методи, призначені для зміни рядків, повертають змінені копії, початкові ж рядки залишаються незмінними.

Створити рядок можна декількома способами:

```
1) string s;                                  // ініціалізація відкладена
2) string s="програмування";                  //ініціалізація строковим літералом
3) string s=@"Привіт!                         //символ @ повідомляє конструктор string, що рядок
   Сьогодні гарна погода!!! "                 // потрібно сприймати буквально, навіть якщо він
                                              // займає декілька рядків
4) string s=new string (' ', 20);             //конструктор створює рядок з 20
   пропусків
5) int x = 12344556;                          //ініціалізували цілочисельну змінну
   string s = x.ToString();                   //перетворили її до типу string
6) char [] a={'a', 'b', 'c', 'd', 'e'};       //створили масив символів
   string v=new string (a);                   // створення рядка з масиву символів
7) char [] a={'a', 'b', 'c', 'd', 'e'};       // створення рядка з частини масиву
   символів, при цьому: 0
   string v=new string (a, 0, 2)              // показує з якого символу, 2 – скільки
   символів
                                              //використовувати для ініціалізації
```

Клас string володіє багатим набором методів для порівняння рядків, пошуку в рядку і інших дій з рядками. Основні методи наведені у табл. 4.13.

Нагадуємо, що виклик статичних методів відбувається через звернення до імені класу, наприклад, String.Concat(str1, str2), у решті випадків через звернення до екземплярів класу, наприклад, str.ToLower().

Таблиця 4.13

Основні методи класу String

Назва	Вигляд	Опис
1	2	3
Compare	Статичний метод	Порівняння двох рядків в лексикографічному (алфавітному) порядку. Різні реалізації методу дозволяють порівнювати рядки з обліком або без урахування регістра
CompareTo	Метод	Порівняння поточного екземпляра рядка з іншим рядком
Concat	Статичний метод	Злиття довільного числа рядків
Copy	Статичний метод	Створення копії рядка
Empty	Статичне поле	Відкрите статичне поле, що представляє порожній рядок
Format	Статичний метод	Форматування рядка відповідно до заданого формату
IndexOf IndexOfAny, LastIndexOf LastIndexOfAny	Екземпляри методи	Визначення індексів першого і останнього входження заданого підрядка або будь-якого символу із заданого набору в даний рядок
Insert	Екземпляр метод	Вставка підрядка в задану позицію
Join	Статичний метод	Злиття масиву рядків в єдиний рядок. Між елементами масиву вставляються роздільники
Length	Властивість	Повертає довжину рядка
PadLeft, PadRigth	Екземпляри методи	Вирівнюють рядки по лівому або правому краю шляхом вставки потрібного числа пропусків на початку або в кінці рядка
Remove	Екземпляр метод	Видалення підрядка із заданої позиції
Replace	Екземпляр метод	Заміна всіх входжень заданого підрядка або символу новим підрядком або символом
Split	Екземпляр метод	Розділяє рядок на елементи, використовуючи різні роздільники. Результати поміщаються в масив рядків
StartWith, EndWith	Екземпляри методи	Повертають true або false залежно від того, починається або закінчується рядок заданим підрядком

Закінчення табл. 4.13

1	2	3
Substring	Екземпляр метод	Виділення підрядка, починаючи із заданої позиції
ToCharArray	Екземпляр метод	Перетворить рядок в масив символів
ToLower ToUpper	Екземпляри методи	Перетворення рядка до нижнього або верхнього регістра
Trim, TrimStart, TrimEnd	Екземпляри методи	Видалення пропусків на початку і кінці рядка або тільки з однієї неї кінця

На прикладі розглянемо використання даних властивостей і методів.

```
static void Main()
  {
    string str1 ="Перший рядок";
    string str2 = string.Copy(str1);
    string str3 = "Другий рядок";
    string str4 = "ДРУГИЙ рядок";
    string strUp, strLow;
    int result, idx;
    Console.WriteLine("str1: " + str1);
    Console.WriteLine("Довжина рядка str1: " +str1.Length);

    // Створюємо версії рядка str1 великими і малими літерами.
    strLow = str1.ToLower();
    strUp = str1.ToUpper();
    Console.WriteLine("Версія рядка str1 великими літерами: " +strLow);
    Console.WriteLine("Версія рядка str1 малими літерами: " +strUp);
    Console.WriteLine();

    // Порівнюємо рядки
    result = str1.CompareTo(str3);
    if (result == 0) Console.WriteLine("str1 і str3 рівні.");
      else if (result < 0) Console.WriteLine("str1 менше, ніж str3");
            else Console.WriteLine("str1 більше, ніж str3");
    Console.WriteLine();
    //порівнюємо рядки без урахування регістра
```

```
    result = String.Compare(str3,str4,true);
    if (result == 0) Console.WriteLine("str3 і str4 рівні без урахування
    регістра.");
      else Console.WriteLine("str3 і str4 не рівні без урахування регістра.");
    Console.WriteLine();

    //порівнюємо частини рядків
    result = String.Compare(str1, 4, str2, 4, 2);
    if (result == 0) Console.WriteLine("частина str1 і str2 рівні");
     else Console.WriteLine("частина str1 і str2 не рівні");
    Console.WriteLine();

    // Пошук рядків.
    idx = str2.IndexOf("рядок");
    Console.WriteLine("Індекс першого входження підрядка рядка: " + idx);
    idx = str2.LastIndexOf("о");
    Console.WriteLine("Індекс останнього входження символу про: " + idx);

    //конкатенація
    string str=String.Concat(str1, str2, str3, str4);
    Console.WriteLine(str);

    //видалення підрядка
    str=str.Remove(0,str1.Length);
    Console.WriteLine(str);

    //заміна підрядка "рядок" на порожній підрядок
    str=str.Replace("рядок"," ");
    Console.WriteLine(str);
    }
```

Дуже важливими методами обробки рядків є методи розділення рядка на елементи Split і злиття масиву рядків в єдиний рядок Join.

```
static void Main()
  {
    string poems = "хмаринки небесні вічні мандрівники";
```

```
char[] div = { ' '}; // створення масиву роздільників
    // Розбиваємо рядок на частини
    string[] parts = poems.Split(div);
    Console.WriteLine("Результат розбиття рядка на частини: ");
    for (int i = 0; i < parts.Length; i++)
     Console.WriteLine(parts[i]);
    // Тепер збираємо ці частини в один рядок, як роздільник
    використовуємо символ
    string whole = String.Join("   ", parts);
    Console.WriteLine("Результат збірки: ");
    Console.WriteLine(whole);
  }
```

У загальному випадку рядок може містити й інші роздільники:

```
static void Main()
  {
    string poems = "Хмаринки небесні, вічні мандрівники...";
    char[] div = { ' ', ',, '.'}; // створення масиву роздільників
    // Розбиваємо рядок на частини
    string[] parts = poems.Split(div);
    Console.WriteLine("Результат розбиття рядка на частини: ");
    for (int i = 0; i < parts.Length; i++)
    Console.WriteLine(parts[i]);
    // Тепер збираємо ці частини в один рядок
    string whole = String.Join("   ", parts);
    Console.WriteLine("Результат збірки: ");
    Console.WriteLine(whole);
  }
```

Розглянемо інший приклад – використовуючи метод Split вводити двовимірний масив можна не поелементно, а порядково:

```
 static void Main()
       {
          try
          {
             int[][] MyArray;
```

```
            Console.Write("введіть кількість рядків: ");
            int n = int.Parse(Console.ReadLine());
            MyArray = new int[n][];
            for (int i = 0; i < MyArray.Length; i++)
            {
                string line = Console.ReadLine();
                string[] mas = line.Split(' ');
                MyArray[i]= new int[mas.Length];
                for (int j = 0; j < MyArray[i].Length; j++)
                {
                    MyArray[i][j]= int.Parse(mas[j]);
                }
            }
            PrintArray("початковий масив:", MyArray);
            for (int i = 0; i < MyArray.Length; i++) Array.Sort(MyArray[i]);
            PrintArray("підсумковий масив", MyArray);
        }
        catch
        {
            Console.WriteLine("виникло виключення");
        }
    }

    static void PrintArray(string a, int[][] mas)
    {
        Console.WriteLine(a);
        for (int i = 0; i < mas.Length; i++)
        {
            foreach (int x in mas[i]) Console.Write("{0} ", x);
            Console.WriteLine();
        }
    }
```

У даному прикладі можуть виникнути виняткові ситуації, якщо введений рядок елементів масиву міститиме зайві пропуски. Отже, від цих пропусків потрібно позбавитися:

```
 static void Main()
{
    try
    {
        int[][] MyArray;
        Console.Write("введіть кількість рядків: ");
        string line= Console.ReadLine()
        int n = int.Parse(line.Trim());
        MyArray = new int[n][];
        for (int i = 0; i < MyArray.Length; i++)
        {
            line = Console.ReadLine();
            line=line.Trim(); //вилучаємо пропуски на початку і кінці рядка
            //видаляємо зайві пропуски усередині рядка
       n = line.IndexOf("  ");
            while (n > 0)
            {
                line = line.Remove(n, 1);
                n = line.IndexOf("  ");
            }
            string[] mas = line.Split(' ');
            MyArray[i]= new int[mas.Length];
            for (int j = 0; j < MyArray[i].Length; j++)
            {
                MyArray[i][j]= int.Parse(mas[j]);
            }
        }
        PrintArray("початковий масив:", MyArray);
        for (int i = 0; i < MyArray.Length; i++) Array.Sort(MyArray[i]);
        PrintArray("підсумковий масив", MyArray);
    }
    catch
    {
        Console.WriteLine("виникло виключення");
    }
}
```

```
static void PrintArray(string a, int[][] mas)
{
   Console.WriteLine(a);
   for (int i = 0; i < mas.Length; i++)
   {
      foreach (int x in mas [i]) Console.Write("{0} ", x);
      Console.WriteLine();
   }
}
```

При роботі з об'єктами класу string потрібно враховувати їх властивість незмінності, тобто той факт, що методи змінюють не самі рядки, а їх копії. Розглянемо фрагмент програми:

```
string a="";
for (int i = 1; i <= 100; i++) a +="!";
Console.WriteLine(a);
```

У цьому випадку в пам'яті комп'ютера буде сформовано 100 різних рядків вигляду:

```
!
!!
!!!
.
!!!...!!
```

І лише останній рядок зберігатиметься в змінній a. Посилання на решту всіх рядків будуть втрачені, але ці рядки зберігатимуться в пам'яті комп'ютера і засмічуватимуть пам'ять. Боротися з таким засміченням доведеться складальникові сміття, що позначатиметься на продуктивності програми. Тому якщо потрібно змінювати рядок, то краще користуватися класом StringBuilder.

Щоб створити рядок, який можна змінювати, в C# передбачений клас StringBuilder, визначений в просторі імен System.Text. Об'єкти цього класу завжди оголошуються з явним викликом конструктора класу (через операцію new). Приклади створення змінних рядків:

StringBuilder a =new StringBuilder(); // створення порожнього рядка, розмір за умовчанням 16 символів
StringBuilder b = new StringBuilder("abcd");
//ініціалізація рядка і виділення необхідної пам'яті
StringBuilder c = new StringBuilder(100); //створення порожнього рядка і виділення пам'яті під 100 символів
StringBuilder d = new StringBuilder("abcd", 100); //ініціалізація рядка і виділення пам'яті під 100 символів
StringBuilder d = new StringBuilder("abcd", 1, 3,100); //ініціалізація підрядком "bcd", і виділення пам'яті під 100 символів

Основні елементи класу наведені в табл. 4.14.

Таблиця 4.14

Основні елементи класу StringBuilder

Назва	Вигляд	Опис
1	2	3
Append	Метод екземпляра	Додавання даних у кінець рядка. Різні варіанти методу дозволяють додавати в рядок величини будь-яких вбудованих типів, масиви символів, рядка і підрядка string
AppendFormat	Метод екземпляра	Додавання форматованого рядка в кінець рядка
Capacity	Властивість	Отримання і установка ємкості буфера. Якщо встановлюване значення менше поточної довжини рядка або більше максимального, то генерується виключення ArgumentOutOfRangeException
Insert	Метод екземпляра	Вставка підрядка в задану позицію
Length	Змінна властивість	Повертає довжину рядка. Присвоювання йому значення 0 скидає вміст і очищає рядок
MaxCapacity	Незмінна властивість	Повертає найбільшу кількість символів, яка може бути розміщена в рядку
Remove	Метод екземпляра	Видалення підрядка із заданої позиції
Replace	Метод екземпляра	Заміна всіх входжень заданого підрядка або символу новим підрядком або символом
ToString	Метод екземпляра	Перетворення в рядок типу string

Закінчення табл. 4.14

1	2	3
Chars	Змінна властивість	Повертає з масиву або встановлює в масиві символ із заданим індексом. Замість нього можна користуватися квадратними дужками []
Equals	Метод екземпляра	Повертає true, тільки якщо об'єкти мають одну і ту ж довжину і складаються з одних і тих же символів
CopyTo	Метод екземпляра	Копіює підмножину символів рядка в масив char

Як бачимо, методи класу StringBuilder менш розвинені, ніж методи класу String, але вони дозволяють ефективніше використовувати пам'ять за рахунок роботи із змінними рядками. Розглянемо приклади використання даних методів.

```
static void Main()
    {
        try
        {
        StringBuilder str=new StringBuilder("Площа");
        PrintString(str);
        str.Append(" трикутника рівна");
        PrintString (str);
        str.AppendFormat(" {0:f2} см ", 123.456);
        PrintString(str);
        str.Insert(8, "даного ");
        PrintString(str);
        str.Remove(7, 21);
        PrintString(str);
        str.Replace("a", "o");
        PrintString(str);
        StringBuilder str1=new StringBuilder(Console.ReadLine());
        StringBuilder str2=new StringBuilder(Console.ReadLine());
        Console.WriteLine(str1.Equals(str2));
        }
        catch
        {
            Console.WriteLine("Виникло виключення");
        }
    }
```

```
static void PrintString(StringBuilder a)
{
  Console.WriteLine("Рядок: "+a);
  Console.WriteLine("Поточна довжина рядка " +a.Length);
  Console.WriteLine("Об'єм буфера "+a.Capacity);
  Console.WriteLine("Максимальний об'єм буфера "+a.MaxCapacity);
  Console.WriteLine();
}
```

Із змінним рядком можна працювати не тільки як з об'єктом, але і як з масивом символів:

```
static void Main()
        {
        StringBuilder a = new StringBuilder("2*3=3*2");
        Console.WriteLine(a);
        int k=0;
        for (int i = 0; i < a.Length; ++i )
              if (char.IsDigit(a[i])) k+=int.Parse(a[i].ToString());
        Console.WriteLine(k);
        }
```

На практиці часто комбінують роботу із змінними і незмінними рядками. Проте якщо необхідно змінювати рядок, то в цьому випадку використовують StringBuilder.

Приклад. Даний рядок, у якому міститься осмислене текстове повідомлення. Слова повідомлення розділяються пропусками і розділовими знаками. Вивести всі слова повідомлення, які починаються і закінчуються на одну і ту ж букву.

```
static void Main()
    {
    Console.WriteLine("Введіть рядок: ");
    StringBuilder a = new StringBuilder(Console.ReadLine());
    Console.WriteLine("Початковий рядок: "+a);
    for (int i=0; i<a.Length;)
          if (char.IsPunctuation(a[i])) a.Remove(i,1);
```

```
        else ++i;
    string str=a.ToString();
    string []s=str.Split(' ');
  Console.WriteLine("Шукані слова: ");
    for (int i=0; i<s.Length; ++i)
    if (s[i][0]==s[i][s.Length-1]) Console.WriteLine(s[i]);  }
```

Стандартний клас string дозволяє виконувати над рядками різні операції, зокрема пошук, заміну, вставку і видалення підрядків. Проте, є класи завдань з обробки символьної інформації, де стандартних можливостей явно не вистачає. Щоб полегшити вирішення подібних завдань, в .Net Framework вбудований могутній апарат роботи з рядками, заснований на регулярних виразах.

Регулярні вирази призначені для обробки текстової інформації і забезпечують:

1. Ефективний пошук у тексті за заданим шаблоном.
2. Редагування тексту.
3. Формування підсумкових звітів за наслідками роботи з текстом.

Детально розглянемо перші два аспекти застосування регулярних виразів.

Метасимволи в регулярних виразах

Регулярний вираз – це шаблон, по якому виконується пошук відповідного фрагмента тексту. Мова опису регулярних виразів складається з символів двох видів: звичайних символів і метасимволів. Звичайний символ представляє у виразі сам себе, а метасимвол – деякий клас символів. Найбільш споживані метасимволи наведені у табл. 4.15.

Таблиця 4.15

Метасимволи в регулярних виразах

Клас символів	**Опис**	**Приклад**
1	2	3
.	Будь-який символ, окрім \n	Вираз c.t відповідає фрагментам: cat, cut, c#t, c{t і т. д.

Закінчення табл. 4.15

1	2	3
[]	Будь-який одиночний символ з послідовності, записаної усередині дужок. Допускається використання діапазонів символів	Вираз c [aui]t відповідає фрагментам: cat, cut, cit. Вираз c[a-c]t відповідає фрагментам: cat, cbt, cct
[^]	Будь-який одиночний символ, що не входить в послідовність, записану усередині дужок. Допускається використання діапазонів символів	Вираз c [^aui]t відповідає фрагментам: cbt, cct, c2t і т. д. Вираз c[^a-c]t відповідає фрагментам: cdt, cet, c%t і т. д.
\w	Будь-який алфавітно-цифровий символ	Вираз c\wt відповідає фрагментам: cbt, cct, c2t і т. д., але не відповідає фрагментам c%t, c{t і т. д.
\W	Будь-який не алфавітно-цифровий символ	Вираз c\Wt відповідає фрагментам: c%t, c{t, c.t і т. д., але не відповідає фрагментам cbt, cct, c2t і т. д.
\s	Будь-який пробільний символ	Вираз \s\w\w\w\s відповідає будь-якому слову з трьох букв, оточеному пробільними символами
\S	Будь-який не пробільний символ	Вираз \s\S\S\S\s відповідає будь-яким трьом непробільним символам, оточеним пробільними
\d	Будь-яка десяткова цифра	Вираз c\dt відповідає фрагментам: c1t, c2t, c3t і т. д.
\D	Будь-який символ, що не є десятковою цифрою	Вираз c\Dt не відповідає фрагментам: c1t, c2t, c3t і т. д.

Окрім метасимволів, що позначають класи символів, можуть застосовуватися уточнюючі метасимволи, які наведені у табл. 4.16.

Таблиця 4.16

Уточнюючі метасимволи в регулярних виразах

Уточнюючі символи	**Опис**
1	2
^	Фрагмент, співпадаючий з регулярними виразами, слід шукати тільки на початку рядка
$	Фрагмент, співпадаючий з регулярними виразами, слід шукати тільки в кінці рядка

Закінчення табл. 4.16

1	2
\A	Фрагмент, співпадаючий з регулярними виразами, слід шукати тільки на початку багаторядкового рядка
\Z	Фрагмент, співпадаючий з регулярними виразами, слід шукати тільки в кінці багаторядкового рядка
\b	Фрагмент, співпадаючий з регулярними виразами, починається або закінчується на границі слова, тобто між символами, відповідними метасимволам \w і \W
\B	Фрагмент, співпадаючий з регулярними виразами, не повинен зустрічатися на межі слів

У регулярних виразах часто використовуються повторювачі (табл. 4.17) – метасимволи, які розташовуються безпосередньо після звичайного символу або групи символів і задають кількість його повторень у виразі.

Таблиця 4.17

Повторювачі

Повторю-вачі	**Опис**	**Приклад**
*	Нуль або більш за повторення попереднього елементу	Вираз ca*t відповідає фрагментам: ct, cat, caat, caaat і т. д.
+	Одне або більш за повторення попереднього елементу	Вираз ca+t відповідає фрагментам: cat, caat, caaat і т. д.
?	Не більш за одне повторення попереднього елементу	Вираз ca?t відповідає фрагментам: ct, cat
{n}	Рівно n повторень попереднього елементу	Вираз ca{3}t відповідає фрагменту: caaat. Вираз (cat){2} відповідає фрагменту: catcat
{n,}	Принаймні n повторень по-переднього елементу	Вираз ca{3,}t відповідає фрагментам: caaat, caaaat, caaaaaaat і т. д. Вираз (cat){2,} відповідає фрагментам: catcat, catcatcat і т. д.
{n, m}	От n до m повторень попереднього елементу	Вираз ca{2, 4}t відповідає фрагментам: caat, caaat, caaaat

Регулярний вираз записується у вигляді рядкового літерала, причому перед рядком необхідно ставити символ @, який говорить про те, що рядок потрібно буде розглядати і в тому випадку, якщо він займатиме декілька рядків на екрані. Проте символ @ можна не ставити, якщо як шаблон використовується шаблон без метасимволів.

Зауваження. Якщо потрібно знайти якийсь символ, який є метасимволом, наприклад, крапку, можна це зробити захистивши її зворотним слешем. Тобто просто крапка означає будь-який одиночний символ, а \. означає просто крапку.

Приклади регулярних виразів:

g) слово ukr – @"ukr" або "ukr"

h) номер телефону у форматі xxx-xx-xx – @"\d\d\d-\d\d-\d\d" або @"\d{3}(-\d\d){2}"

i) номер автомобіля - @"[A-Z]\d{3}[A-Z]{2}\d{2,3}UKR"

Пошук в тексті за шаблоном

Простір імен бібліотеки базових класів System.Text.RegularExpressions містить всі об'єкти платформи .NET Framework, що мають відношення до регулярних виразів. Найважливішим класом, що підтримує регулярні вирази, є клас Regex, який представляє незмінні регулярні вирази, що відкомпілювалися. Для опису регулярного виразу в класі визначено декілька переобтяжених конструкторів:

1. Regex() – створює порожній вираз
2. Regex(String) – створює заданий вираз.
3. Regex(String, RegexOptions) – створює заданий вираз і задає параметри для його обробки за допомогою елементів перерахування RegexOptions (наприклад, розрізняти чи не розрізняти великі і малі букви).

Пошук фрагментів рядка, відповідних заданому виразу, виконується за допомогою методів IsMach, Mach, Matches класу Regex.

Метод ***IsMach*** повертає true, якщо фрагмент, відповідний виразу, в заданому рядку знайдений, і false інакше. Наприклад, спробуємо визначити, чи зустрічається в заданому тексті слово *собака*:

```
static void Main()
    {
       Regex r = new Regex("собака",RegexOptions.IgnoreCase);
       string text1 = "Кіт в будинку, собака в будці.";
       string text2 = "Котик в будинку, собачка в будці.";
```

```
        Console.WriteLine(r.IsMatch(text1));
        Console.WriteLine(r.IsMatch(text2));
    }
```

Зауваження. RegexOptions.IgnoreCase – означає, що регулярний вираз застосовується без урахування регістра символів.

Можна використовувати конструкцію вибору з декількох елементів. Варіанти вибору перераховуються за допомогою символу "|". Наприклад, спробуємо визначити, чи зустрічається в заданому тексті слів *собака* або *кіт*:

```
static void Main(string[] args)
    {
        Regex r = new Regex("собака кіт",RegexOptions.IgnoreCase);
        string text1 = "Кіт в будинку, собака в будці.";
        string text2 = "Котик в будинку, собачка в будці.";
        Console.WriteLine(r.IsMatch(text1));
        Console.WriteLine(r.IsMatch(text2));
    }
```

Спробуємо визначити, чи є в заданих рядках номера телефону у форматі xx-xx-xx або xxx-xx-xx:

```
static void Main()
    {
        Regex r = new Regex(@"\d{2,3}(-\d\d){2}");
        string text1 = "tel:123-45-67";
        string text2 = "tel:no";
        string text3 = "tel:12-34-56";
        Console.WriteLine(r.IsMatch(text1));
        Console.WriteLine(r.IsMatch(text2));
        Console.WriteLine(r.IsMatch(text3));
    }
```

Метод ***Match*** класу Regex не просто визначає, чи міститься текст, відповідний шаблону, а повертає об'єкт класу Match – послідовність фрагментів тексту, що збіглися з шаблоном. Наступний приклад дозволяє знайти всі номери телефонів у вказаному фрагменті тексту:

```
 static void Main()
{
   Regex r = new Regex(@"\d{2,3}(-\d\d){2}");
   string text = @"Контакти у Києві tel:123-45-67, 123-34-56; fax:123-
56-45
                        Контакти у Харкові tel:12-34-56; fax:12-56-45";
   Match tel = r.Match(text);
   while (tel.Success)
   {
      Console.WriteLine(tel);
      tel = tel.NextMatch();
   }
}
```

Наступний приклад дозволяє підрахувати суму цілих чисел, що зустрічаються в тексті:

```
static void Main()
{
   Regex r = new Regex(@"[-+]?\d+");
   string text = @"5*10=50 -80/40=-2";
   Match teg = r.Match(text);
   int sum = 0;
   while (teg.Success)
   {
      Console.WriteLine(teg);
      sum += int.Parse(teg.ToString());
      teg = teg.NextMatch();
   }
   Console.WriteLine("sum=" + sum);
}
```

Метод ***Matches*** класу Regex повертає об'єкт класу MatchCollection – колекцію всіх фрагментів заданого рядка, що збіглися з шаблоном. При цьому метод Matches багато разів запускає метод Match, кожного разу починаючи пошук з того місця, на якому закінчився попередній пошук.

```
static void Main(string[] args)
```

```
{
        string text = @"5*10=50 -80/40=-2";
        Regex theReg = new Regex(@"[-+]?\d+");
        MatchCollection theMatches = theReg.Matches(text);
        foreach (Match theMatch in theMatches)
        {
                Console.Write("{0} ", theMatch.ToString());

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

Редагування тексту

Регулярні вирази можуть ефективно використовуватися для редагування тексту. Наприклад, метод Replace класу Regex дозволяє виконувати заміну одного фрагмента тексту іншим або видалення фрагментів тексту:

***Приклад** 1*. Зміна номерів телефонів:

```
static Main(string[] args)
        {
        string text = @"Контакти у Києві tel:123-45-67, 123-34-56; fax:123-56-
45.
                Контакти у Харкові tel:12-34-56; fax:11-56-45";
        Console.WriteLine("Старі дані\n"+text);
        string newText=Regex.Replace(text, "123-", "890-");
        Console.WriteLine("Нові дані\n" + newText);
        }
```

Приклад 2. Видалення всіх номерів телефонів з тексту:

```
static void Main(string[] args)
        {
        string text = @"Контакти у Києві tel:123-45-67, 123-34-56; fax:123-56-
45.
```

```
            Контакти у Харкові tel:12-34-56; fax:11-56-45";
        Console.WriteLine("Старі дані\n"+text);
        string newText=Regex.Replace(text @"\d{2,3}(-\d\d){2}", "");
        Console.WriteLine("Нові дані\n" + newText);
        }
```

Приклад 3. Розбиття початкового тексту на фрагменти:

```
static void Main ()
{
        string text = @"Контакти у Києві tel:123-45-67, 123-34-56; fax:123-56-
45.
            Контакти у Харкові tel:12-34-56; fax:11-56-45";
        string []newText=Regex.Split(text"[,.:;]+");
        foreach( string a in newText)
            Console.WriteLine(a);
}
```

Тема 5. Особливості застосування платформи .NET при розробці програмного забезпечення

5.1. Управління пам’яттю та вказівники

Змінна, яка представляє клас або масив, містить адресу пам'яті, у якій зберігається об'єкт (дані екземпляра). Це посилання синтаксично трактується так, немов змінна сама безпосередньо зберігає дані об'єкта. І лише через посилання можна отримати ці дані. Посилання C# розроблені так, щоб спростити код і мінімізувати можливість внесення помилок несвідомого псування даних у пам'яті.

В окремих випадках виникає потреба безпосередньої роботи з пам'яттю з допомогою покажчиків, добре відомих у C++ та інших алгоритмічних мовах. Цю функціональність можна використовувати для забезпечення високої продуктивності окремих фрагментів коду або для звертання до функцій у зовнішній (не .NET) DLL, які вимагають передавання вказівника як параметра (наприклад, функції Windows API).

C# дає змогу використовувати вказівники лише у спеціальних блоках, які помічаються як незахищені (небезпечні) за допомогою ключового слова unsafe:

```
unsafe class C {
 / /довільний метод класу може використовувати вказівник }
 unsafe void M() {
  //метод може використовувати вказівники }
  class A {
unsafe int *p //оголошення поля-вказівника у класі }
unsafe {
  //незахищений код
}
```

Не можна оголосити локальну змінну як unsafe. Якщо така потреба виникає, то цю змінну потрібно розмістити всередині незахищеного блоку.

Компілятор C# не буде компілювати код, який містить вказівники за межами блоків unsafe. Для використання режиму unsafe проект повинен містити увімкнену опцію Project I Properties I Build I Allow Unsafe Code.

Синтаксис вказівників

Для оголошення вказівників використовують символ *:

```
int *pX, pY;
double *pResult;
void *pV;
```

На відміну від C++ символ * діє на всі оголошувані у стрічці змінні. Тобто pY також буде вказівником.

Для роботи з вказівниками використовують дві унарні операції:

адресна операція & перетворює тип даних за значенням у вказівник (наприклад, int у *int);

операція розіменування * перетворює вказівник у тип даних за значенням (наприклад, *int у int).

Розглянемо код:

```
int X = 0; int *pX; pX  = &X; *pX
= 10;
```

Оскільки pX містить адресу змінної X (після виконання оператора pX = &X), то код *pX = 10 запише значення 10 на місце X. Тобто в результаті змінна X набуде значення 10.

Вказівник можна привести до цілочисельного типу

```
uint ui = (uint)pX;
```

Вказівники гарантовано можна привести лише до типів uint, long або ulong, а для 64-розрядних процесорів – лише до типу ulong.

Вказівники на структуру

Вказівник можна утворити лише на типи за значенням. Причому для структур існує обмеження: структура не повинна містити типів за посиланням.

Означимо наступну структуру:

```
struct Complex {
   public double Re;
   public double Im; }
```

Ініціалізуємо вказівник на цю структуру:

```
Complex *pComplex;
Complex complex = new Complex();
*pComplex = Scomplex;
```

Доступ до членів структури можна здійснити за допомогою вказівника:

```
(*pComplex).Re = 1;
```

Однак такий синтаксис дещо ускладнений. Отож C# передбачає іншу операцію доступу до членів структури через вказівник:

```
pComplex->Re = 1;
```

Вказівники на члени класу

У C# неможливо утворити вказівник на клас, однак можна утворити вказівники на члени класу, які мають тип за значенням. Це вимагає використання спеціального синтаксису з огляду на особливості механізму прибирання "сміття". У довільний момент часу може бути прийняте рішення про переміщення об'єктів класу на нове місце з метою упорядкування динамічної пам'яті. Оскільки члени класу розташовані в динамічній пам'яті, вони також будуть переміщені. А якщо на них були утворені вказівники, то з цього моменту їх значения стануть некоректними.

Щоб уникнути цієї проблеми, використовують ключове слово fixed, яке повідомляє прибиральника "сміття" про можливе існування вказівників на деякі члени окремих екземплярів класу. У цьому випадку такі об'єкти переміщатися в пам'яті не будуть.

Перепишемо структуру Complex як клас:

```
public class Complex {
   public double Re;
   public double Im; }
```

Синтаксис використання fixed у випадку одного вказівника такий:

```
Complex complex = new Complex();
fixed (double *pRe = &(complex.Re)) { … }
```

Область видимості вказівника pRe розповсюджується лише на блок у фігурних дужках. Доки виконується код усередині блоку fixed, прибиральник "сміття" не чіпатиме об'єкт complex.

Якщо потрібно оголосити декілька таких вказівників, то всі вони описуються як fixed до блоку використання:

```
fixed (double *pRe = &(complex.Re))
fixed (double *pIm = &(complex.Im)) { … }
```

Якщо змінні однотипні, їх можна ініціалізувати всередині одного fixed:

```
fixed (double *pRe = &(complex.Re),
double *pIm = &(complex.Im)) { … }
```

Блоки fixed можуть бути вкладені один в інший.

Вказівники можуть показувати на поля в одному і тому ж екземплярі класу, у різних екземплярах або на статичні поля, які існують незалежно від екземплярів класу.

Арифметичні операції над вказівниками

До вказівників можна додавати та віднімати цілочисельні значення. У цьому випадку вказівник змінює своє значення на відповідне ціле число, помножене на довжину типу в байтах. Якщо додається число X до вказівника на тип T зі значенням P, то в результаті вказівник міститиме адресу P + X*(sizeof(T)).

З вказівниками можна використовувати операції +, -, +=, -=, ++ та –, де змінна з правого боку цих операторів буде long або ulong.

Можна віднімати вказівники на один і той же тип даних. Результатом такої операції буде різниця значень вказівників, поділена на довжину типу.

Для демонстрації арифметичних операцій над вказівниками утворимо високопродуктивний одномірний масив.

Усі масиви C# є об'єктами за посиланням і розміщуються у динамічній пам'яті. Процес вибірки з цієї пам'яті, запису в неї та її

обслуговування є доволі об'ємним. Якщо є потреба утворити масив на короткий проміжок часу без втрат продуктивності через розташування в динамічній пам'яті, доцільно виконати це у стеку.

Для виділення деякої кількості пам'яті у стеку можна використати ключове слово stackalloc. Ця команда використовує два параметри: тип змінної, яку потрібно зберігати, і кількість змінних. Утворимо з її допомогою масив з n елементів типу double:

```
int n = 20;
double *pDoubles = stackalloc double[n];
```

У результаті виконання цього коду середовище виконання .NET виділить 160 байт (20*sizeof (double)) і запише у pDoubles адресу першого з них. Наступний код демонструє механізм доступу до елементів масиву:

```
*pDoubles = 0; //0-ий елемент
int k = 10;
*(pDoubles+k) = 1; //k-ий елемент
```

C# дає також альтернативний синтаксис доступу до елементів масиву. Якщо деяка змінна p має тип вказівника, а к є довільним числовим типом, то вираз p [к] завжди інтерпретується як *(p+к). Наприклад, останню стрічку коду можна записати так:

```
pDoubles[k] = 5;
```

Зазначимо, що, на відміну від звичайних масивів, ця стрічка не ініціює виняток, якщо к буде більшим за 19, тобто відбудеться вихід за межі масиву. Інформація у відповідних байтах буде затерта новим значенням. І найкращий випадок у цій ситуації – виникнення винятку в тій частині коду, де цю інформацію використовують. У найгіршому випадку отримаємо правдоподібні, проте неправильні результати. Недарма такий код необхідно свідомо оголосити небезпечним.

5.2. Атрибути

Ключові слова public і private задають поведінку членів класу, описуючи їх доступність ззовні. Оскільки компілятори розпізнають лише зумовлені ключові слова, ви не маєте можливості створювати свої власні. Проте CLR дозволяє додавати оголошення, які називають

атрибутами, для коментування таких елементів коду як типи, поля, методи і властивості.

Коли ви компілюєте свій код, він перетвориться в Microsoft Intermediate Language (MSIL) і поміщається у файл формату Portable Executable (PE) разом з метаданими, що згенерували компілятором. Атрибути дозволяють додати до метаданих додаткову інформацію, яка потім може витягуватися за допомогою механізму рефлексії. Компілятор створює атрибути, коли ви оголошуєте екземпляри спеціальних класів, що успадковуються від System.Attribute.

.NET Framework широко використовує атрибути. Атрибути описують правила серіалізації даних, управляють безпекою і обмежують оптимізацію JIT-компіляторів для полегшення відладки коду. Атрибути також можуть містити ім'я файла або автора, або управляти видимістю елементів управління і класів при розробці форм призначеного для користувача інтерфейсу.

Ви можете використовувати атрибути для довільного коментування коду і управління поведінкою компонентів. Атрибути дозволяють додавати описові елементи C#, керовані розширення для C++, Microsoft Visual Basic.NET або в будь-яку іншу мову, що підтримує CLR, без необхідності переписувати компілятор.

Крім того, атрибути можна використовувати в ATL проектах.

Використання атрибутів

Більшість атрибутів застосовуються до таких елементів мови, як класи, методи, поля і властивості. Але деякі атрибути є глобальними – вони впливають на всю збірку або модуль. Глобальні атрибути в текстах програм оголошуються після using директив верхнього рівня перед визначеннями типів і просторів імен. Вони можуть використовуватися в різних вихідних файлах однієї програми.

Вживання атрибутів на рівні класів і методів

Атрибути в програмному коді використовуються таким чином:

1. Визначається новий або береться той, що існує в .NET Framework атрибут.

2. Ініціалізувався конкретний екземпляр атрибуту за допомогою виклику конструктора атрибуту.

Атрибут поміщається в метадані при компіляції коду і стає доступний CLR і будь-яким інструментальним засобам і застосуванням через механізми рефлексії.

За угодою, імена всіх атрибутів закінчуються словом Attribute. Проте, мови з VisualStudio.NET не вимагають завдання повного імені атрибуту. Наприклад, якщо потрібно ініціалізувати атрибут System.-ObsoleteAttribute, досить написати Obsolete.

Наступний приклад показує, як використовувати атрибут System.ObsoleteAttribute, позначаючий код як застарілий. Атрибуту передається рядок "Буде видалено в наступній версії". Цей атрибут заставляє компілятор видати переданий рядок як попередження при компіляції поміченого кода.

```
using System;
public class MainApp
{
  public static void Main()
  {
     int MyInt = Add(2,2);
  }
  [Obsolete("Буде видалено в наступній версії ")]
  public static int Add(int a, int b)
  {
    return (a + b);
  }
}
```

Вживання атрибутів на рівні складань

Для вживання атрибутів на рівні складань використовується ключове слово Assembly. Наступний приклад показує, як використовується атрибут AssemblyNameAttribute:

```
using System.Reflection;
[assembly:AssemblyName("Моє складання")]
```

При компіляції коду рядок "Моє складання" поміщається в маніфест збірки в секції метаданих. Цей атрибут можна побачити за допомогою дизасемблера MSIL (Ildasm.exe) або за допомогою призначених для користувача засобів.

Вживання атрибутів на рівні модулів

Для вживання атрибутів на рівні модулів використовується ключове слово Module, у останньому все як на рівні збірок.

Призначені для користувача атрибути

Аби розробляти власні атрибути, не потрібно вивчати щось принципово нове. Якщо ви знайомі з об'єктно-орієнтованим програмуванням і знаєте, як розробляти класи, ви знаєте вже практично все. Призначені для користувача атрибути – це класи, які тим або іншим чином успадковують від System.Attribute. Також як і всі інші класи, призначені для користувача атрибути містять методи для запису і читання даних. Розглянемо процес створення призначеного для користувача атрибуту по кроках.

Використання атрибуту AttributeUsageAttribute

Оголошення призначеного для користувача атрибуту починається з AttributeUsageAttribute, який задає ключові характеристики нового класу. Наприклад, він визначає, чи може атрибут успадковуватися іншими класами, або до яких елементів він може застосовуватися. Наступний фрагмент коду ілюструє вживання атрибуту AttributeUsageAttribute

```
[AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple = true)]
[AttributeUsage(AttributeTargets::All, Inherited = false, AllowMultiple = true)]
```

Клас System.AttributeUsageAttribute містить три члени, які важливі для створення користувальницьких атрибутів: AttributeTargets, Inherited і AllowMultiple.

Поле AttributeTargets

У попередньому прикладі використовується прапор Attribute Targets.All. Цей прапор означає, що цей атрибут може застосовуватися до будь-яких елементів програми. З іншого боку, можна задати прапор AttributeTargets.Class, що означає, що атрибут застосовується тільки до класів, або AttributeTargets.Method – для методів класів та інтерфейсів. Подібним чином можна застосовувати свої атрибути. Також можна використовувати декілька примірників атрибуту AttributeTargets. У наступному прикладі показано, як користувальницький атрибут може застосовуватися до будь-якого класу або методу:

```
[AttributeUsage(AttributeTargets.Class  AttributeTargets.Method)]
[AttributeUsage(AttributeTargets::Class  AttributeTargets::Method)]
```

Властивість Inherited

Ця властивість визначає, чи буде атрибут вспадкувати класами, спадкоємцями того, до якого цей атрибут застосований. Ця властивість може приймати два значення: true або false.

```
public class MyAttribute : Attribute
{
}
[AttributeUsage(Inherited = false)]
public class YourAttribute : Attribute
{
}
```

Вищеописані атрибути потім застосовуються до методу класу MyClass:

```
public class MyClass
{
  [MyAttribute][YourAttribute]
  public void MyMethod() {
    //...
  }
}
```

І, нарешті, розглянемо клас YourClass – спадкоємець MyClass. З методом MyMethod цього класу буде пов'язаний тільки атрибут MyAttribute.

```
public class YourClass : MyClass
{
  public void MyMethod()
  {
    //...
  }
}
```

Властивість AllowMultiple

Це властивість показує, чи може атрибут застосовуватися багаторазово до одного елементу. За замовчуванням воно дорівнює false, що значить – атрибут може використовуватися тільки один раз. Розглянемо наступний приклад:

```
public class MyAttribute : Attribute
{
}
```

```
[AttributeUsage(AllowMultiple = true)]
public class YourAttribute : Attribute
{
}
```

Якщо використовується декілька екземплярів атрибутів MyAttribute заставляє компілятор видати повідомлення про помилку. Наступний фрагмент коду ілюструє правильне використання атрибуту YourAttribute і неправильне – MyAttribute:

```
public class MyClass
{
  [MyAttribute, MyAttribute]
  public void MyMethod()
  {
    //...
  }
  [YourAttribute, YourAttribute]
  public void YourMethod()
  {
    //...
  }
}
```

Якщо властивості AllowMultiple і Inherited встановлені в true, клас може успадковувати атрибут і мати ще екземпляри, застосовані безпосередньо до нього. Якщо ж властивість AllowMultiple рівно false, значення атрибутів батьківського класу будуть переписані значеннями цього ж атрибуту класа-спадкоємця.

Типи даних, допустимі в атрибутах

Атрибут може містити поля наступних типів:

- Bool
- Byte
- Char
- Double
- Float
- Int
- Long
- Short
- String

- Object
- System.Type
- Відкриті перераховуючі типи.

Спроба використовувати в класі, що реалізовує атрибут інших типів, наводить до помилок компіляції.

Визначення атрибутивного класу

Тепер можна приступити до визначення самого класу. Це визначення виглядає подібно до визначення звичайного класу, що демонструє наступний приклад:

```
public class MyAttribute : System.Attribute
{
  // ...
}
```

Цей приклад показує наступні положення:

Атрибутивні класи повинні оголошуватися як відкриті.

За угодою, імена класів повинні закінчуватися словом Attribute. Хоча це і необов’язково, рекомендується робити так для поліпшення читаності тексту.

Всі атрибутивні класи повинні, так або інакше, успадковувати від System.Attribute.

Визначення конструкторів

Атрибути ініціалізувалися конструкторами, так само як звичайні класи. Наступний фрагмент коду ілюструє типовий конструктор атрибуту. Цей відкритий конструктор приймає один параметр і ініціалізував змінну класу.

```
public MyAttribute(bool myvalue)
{
   this.myvalue = myvalue;
}
```

Конструктори можна перевантажувати, щоб приймати різні комбінації параметрів. Якщо для атрибутивного класу визначені властивості, для ініціалізації можна використовувати комбінацію позиційних та іменованих параметрів. Зазвичай всі обов'язкові параметри оголошуються як позиційні, а необов'язкові як іменовані. Наступний приклад показує приклади використання параметризованого конструктора для ініціалізації атрибуту. Тут передбачається, що атрибут має обов'язковий параметр типу Boolean і необов'язковий типу String.

```
[MyAttribute(false, OptionalParameter = "додаткові дані")]
[MyAttribute(false)]
```

Параметри, визначені як властивості, можуть передаватися в довільному порядку. Але обов'язкові параметри повинні передаватися в тому порядку, в якому вони описані в конструкторі. Наступний фрагмент коду показує, як необов'язковий параметр може передаватися перед обов'язковим.

```
// Іменований параметр поміщається перед позиційним.
[MyAttribute(OptionalParameter = "додаткові дані", false)]
```

Визначення властивостей

Властивості визначаються, якщо потрібно передавати іменовані параметри в конструктори або легко та зручно отримувати значення полів атрибуту. Наступний приклад показує, як реалізувати просту властивість для настроюваного атрибута:

```
public bool MyProperty
{
  get { return this.myvalue; }
  set { this.myvalue = value; }
}
```

Приклад настроюваного атрибута

Атрибут з цього прикладу містить інформацію про ім'я і рівні програміста, а також про час останнього перегляду коду. Він містить три закритих змінних, в яких зберігаються дані. Кожна мінлива пов'язана з відкритою властивістю для читання і запису значень. Також є конструктор з двома обов'язковими параметрами.

```
[AttributeUsage(AttributeTargets.All)]
public class DeveloperAttribute : System.Attribute
{
  private string name;
  private string level;
  private bool reviewed;
  public DeveloperAttribute(string name, string level)
  {
    this.name = name;
    this.level = level;
    this.reviewed = false;
  }
```

```
  public virtual string Name
  {
    get { return name; }
  }
  public virtual string Level
  {
    get { return level; }
  }
  public virtual bool Reviewed
  {
    get { return reviewed; }
    set { reviewed = value; }
  }
}
```

Застосовувати цей атрибут можна, використовуючи як повне ім'я DeveloperAttribute, так і скорочене – Developer:

```
[Developer("Іван Семенов", "1")]
[Developer("Іван Семенов", "1", Reviewed = true)]
```

У першому прикладі показано вживання атрибуту з одним обов'язковим параметром, а в другому – з обома типами параметрів.

Отримання одиночного атрибуту

У наступному прикладі атрибут DeveloperAttribute (розглянутий вище) застосовується до класу MainApp в цілому. Метод GetAttribute використовує Attribyte.GetCustomAttribute для отримання стану атрибуту DeveloperAttribute перед тим, як вивести інформацію на консоль.

```
using System;
[Developer("Іван Семенов", "42", Reviewed = true)]
class MainApp
{
  public static void Main()
  {
    GetAttribute(typeof(MainApp));
  }
  public static void GetAttribute(Type t)
  {
    DeveloperAttribute MyAttribute =
```

```
        (DeveloperAttribute) Attribute.GetCustomAttribute(t,
typeof(DeveloperAttribute));
      if (MyAttribute == null)
      {
        Console.WriteLine("Атрибут не знайдений");
      }
      else
      {
        Console.WriteLine("Ім’я: {0}." , MyAttribute.Name);
        Console.WriteLine("Рівень: {0}." , MyAttribute.Level);
        Console.WriteLine("Перевірено: {0}." , MyAttribute.Reviewed);
      }
    }
  }
```

Якщо атрибут не знайдено, метод GetCustomAttribute повертає нульове значення. У цьому прикладі припускається, що атрибут визначений у поточному просторі імен, якщо це не так, не забудьте імпортувати відповідний простір імен.

Отримання списку однотипних атрибутів

У попередньому прикладі посилання на клас і атрибут передавалися у метод GetCustomAttribute. Цей код прекрасно працює, якщо на рівні класу визначений лише один атрибут. Але якщо на тому ж рівні визначено декілька однотипних атрибутів, цей метод поверне не всю інформацію. У таких випадках потрібно використовувати метод Attribute.GetCustomAttributes, який повертає масив атрибутів. Наприклад, якщо на рівні класу визначені два примірники атрибуту DeveloperAttribute, можна модифікувати метод GetAttribute, щоб отримати обидва. Як це зробити, показано в наступному прикладі:

```
  public static void GetAttribute(Type t)
  {
    DeveloperAttribute[] MyAttribute =
      (DeveloperAttribute[]) Attribute.GetCustomAttributes(t,
typeof(DeveloperAttribute));
    if (MyAttribute == null)
      Console.WriteLine("Атрибут не знайдений");
```

```
    else
      for (int i = 0 ; i < MyAttribute.Length ; i++) {
        Console.WriteLine("Ім’я: {0}." , MyAttribute[i].Name);
        Console.WriteLine("Рівень: {0}." , MyAttribute[i].Level);
        Console.WriteLine("Перевірено: {0}.", MyAttribute[i].Reviewed);
      }
  }
```

Отримання списку різнотипних атрибутів

Методи GetCustomAttribute та GetCustomAttributes не можуть шукати атрибут у всьому класі і повертати всі його екземпляри. Вони дивляться тільки один метод або поле за раз. Тому, якщо є клас з одним атрибутом для всіх методів і потрібно одержати всі примірники цього атрибуту, не залишається нічого робити, як передавати ці методи один за іншим як параметри GetCustomAttribute та GetCustomAttributes. У наступному фрагменті коду показано, як отримати всі примірники атрибуту DeveloperAttribute, визначеного як на рівні класу, так і на рівні методів.

```
using System;
using System.Reflection;
public static void GetAttribute(Type t)
{
  DeveloperAttribute att =
   (DeveloperAttribute) Attribute.GetCustomAttribute (t,
typeof(DeveloperAttribute));
  if (att == null)
    Console.WriteLine("Клас {0} не має атрибута Developer.\n",
t.ToString());
  else
  {
    Console.WriteLine("Атрибут Ім’я на рівні класу: {0}.", att.Name);
    Console.WriteLine("Атрибут Рівень на рівні класу: {0}.", att.Level);
    Console.WriteLine("Атрибут Перевірено на рівні класу: {0}.\n",
att.Reviewed);
  }
  MemberInfo[] MyMemberInfo = t.GetMethods();
  for (int i = 0; i < MyMemberInfo.Length; i++) {
```

```
        att = (DeveloperAttribute)
Attribute.GetCustomAttribute(MyMemberInfo[i], typeof (DeveloperAttribute));
        if (att == null)
          Console.WriteLine("Метод {0} не має атрибута Developer.\n" ,
MyMemberInfo[i].ToString());
        else {
          Console.WriteLine("Атрибут Ім’я на рівні методу {0}: {1}.",
MyMemberInfo[i].ToString(), att.Name);
          Console.WriteLine("Атрибут Рівень на рівні методу {0}: {1}.",
MyMemberInfo[i].ToString(), att.Level);
          Console.WriteLine("Атрибут Перевірено на рівні методу {0}:
{1}.\n", MyMemberInfo[i].ToString(), att.Reviewed);
         }
       }
     }
```

Для доступу до методів і полів класу, що перевіряється використовуються методи класу System.Type. У цьому прикладі спочатку через Type запитується інформація про атрибути, визначених на рівні класу, а потім, через метод Type.GetMethods виходить інформація про всі атрибути, визначених на рівні методів. Ця інформація міститься в масив об'єктів типу System.Reflection.MemberInfo. Якщо потрібні атрибути властивостей, використовується метод Type.GetProperties, а для конструкторів – Type.GetConstructors. Клас Type має безліч методів для доступу до елементів типу, тут описана лише дуже невелика частина.

5.3. Збереження та відновлення стану об’єктів у .NET

Серіалізація – запис структури, класу або даних у потік. Це може бути передача структури по протоколу TCP / IP для якої-небудь гри. Або може бути простий запис структури / дамп у файл, у бінарному / двійковому форматі. Це може бути звичайний файл налаштувань для програми, файл збереження гри (save file) або власний варіант бази даних.

На відміну від програм на некерованому коді, додатки .NET Framework не обов'язково виконуються у вигляді окремих процесів, а можуть існувати в межах одного процесу операційної системи у своїх

власних областях, які називаються доменами програми. Такі області можна розглядати як деякі логічні процеси віртуальної машини CLR. Використання керованого коду дозволяє при цьому гарантувати ізоляцію програм у межах своїх областей. При передачі між доменами додатків деякого об'єкта для його класу повинна бути визначена процедура серіалізації, яка дозволяє зберегти стан об'єкта в деякому зовнішньому сховищі (наприклад, у файлі, або в повідомленні транспортного протоколу) за допомогою потоків вводу-виводу, і процедура десеріалізації, що створює копію об'єкта за збереженого станом. Слід зазначити, що в загальному випадку це можуть бути об'єкти різних класів, і навіть створені в різних системах розробки додатків.

Задача серіалізації об'єкта, що включає тільки поля з елементарних типів значень (спадкоємців класу System.ValueType) і строк, не становить принципових труднощів. Для такого об'єкта в ході серіалізації в потік записуються самі значення всіх полів об'єкта. Однак у загальному випадку об'єкт містить посилання на інші об'єкти, які, в свою чергу, можуть посилатися один на одного, утворюючи так званий граф об'єктів *(object graph).* Самі посилання не можуть бути збережені в потоці вводу-виводу, тому основне питання серіалізації – це спосіб заміни посилань.

Класи, що виробляють серіалізацію і десеріалізацію в .NET Framework, називаються класами форматування (formatters). В .NET Framework виділяється три різних незалежних класи форматування: XmlSerializer, SoapFormatter, BinaryFormatter. Вони використовуються як для запису та читання об'єктів з потоків вводу-виводу, так і для побудови розподілених систем:

технологія веб-служб ASP.NET використовує XmlSerializer;

технологія Remoting використовує SoapFormatter, BinaryFormatter або створений користувачем клас;

при роботі з повідомленнями MSMQ використовується XmlSerializer (через XMLMessageFormatter), або BinaryFormatter (через BinaryMessage-Formatter), або створений користувачем клас;

технологія Enterprise Services заснована на Remoting і використовує BinaryFormatter.

Можна провести класифікацію можливих методів серіалізації за наступними основними ознаками.

1. Класифікація за видом оброблюваного графа:

універсальні методи: граф довільного виду з циклами;

довільний ациклічний граф;

дерево.

2. Класифікація за форматом зберігання інформації в сховищі:

бінарні методи, які використовують двійковий формат зберігання даних, який не придатний для читання людиною без використання спеціальних засобів;

текстові методи, використовують XML або інші текстові формати, придатні для читання або редагування людиною при використанні текстового редактора.

3. Класифікація за специфікацією формату даних, отриманого в результаті серіалізації:

закриті методи: специфікація задається тільки за допомогою інтерфейсів всіх класів, об'єкти яких утворюють граф; в цьому випадку обидві вилучені компоненти повинні бути створені на одній мовній платформі, причому обидві сторони повинні мати як мінімум опис інтерфейсу серіалізуємих класів;

відкриті методи: специфікація може бути задана у вигляді загальноприйнятого формату, наприклад схеми XSD.

За даною класифікацією XmlSerializer реалізує відкритий текстовий неуніверсальний метод, BinaryFormatter – закритий двійковий універсальний метод, SoapFormatter – текстовий відкритий метод. Додатковою особливістю кожного з класів, за винятком BinaryFormatter, є обмеження на класи, які підлягають серіалізації.

Клас серіалізації Soap.SoapFormatter використовується виключно в середовищі. NET Remoting, а клас System.Runtime.Serialization.Formatters. Binary.BinaryFormatter може також використовуватися в середовищі MSMQ замість XMLSerializer. Обидва класи форматування за наведеною класифікацією є універсальними. Клас форматування BinaryFormatter реалізує двійковий закритий метод серіалізації, клас SoapFormatter – текстовий і відкритий, заснований на специфікації кодування SOAP-RPC.

Клас SoapFormatter не підтримує одне з важливих нововведень - параметризованих типів даних *(generic types).*

Обидва зазначених класи в найпростішому випадку при серіалізації зберігають усі поля класу (але не його властивості), незалежно від їх

видимості Поля, що мають атрибут System.NonSerializeAttribute, ігноруються. Клас повинен мати атрибут System.SerializableAttribute. У ході серіалізації класу форматування використовують методи класу System.Runtime.Serialization.FormatterServices. Серіалізуємий клас повинен містити конструктор без параметрів, який викликається при створенні нового об'єкта в ході десеріалізації.

Якщо ж обробляється клас, що реалізує інтерфейс ISerializable, то він серіалізується викликом методу GetObjectData (SerializationInfo info, StreamingContext context) цього інтерфейсу, всередині якого зазвичай так же викликаються методи FormatterServices. Десеріалізація таких класів здійснюється викликом конструктора ISerializable (SerializationInfo info, StreamingContext context), заповнює поля об'єкта значеннями з info.

Про завершення своєї десеріалізації об'єкт може отримати повідомлення, реалізувавши інтерфейс System.Runtime.Serialization. IDeserializationCallback з єдиним методом OnDeserialization.

Отриманий таким чином на першому кроці серіалізації об'єкт класу SerializationInfo містить імена та значення серіалізуємих полів. Розглянуті класи форматування, що реалізують інтерфейс IFormatter, перетворюють ці імена в деякий вид, який передається між доменами програми через потоки вводу-виводу.

Розглянемо приклад створення класу з інтерфейсом ISerializable і власним механізмом серіалізації.

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Serialization.Formatters.Binary;
using System.Reflection;

[Serializable]
public class Person: ISerializable

     public String name;

public Person ()
    (
```

Метод GetObjectData використовується на першому кроці серіалізації класу. У ході його роботи в об'єкт класу SerializationInfo

додається інформація про поля класу, що підлягають серіалізації. Для отримання метаданих про поля класу використовується статичний метод GetSerializableMembers класу FormatterServices.

```
public     void     GetObjectData     (SerializationInfo     info,
StreamingContext context)
 (
Type thisType = this.GetType ();
       MemberInfo [] serializableMembers =
FormatterServices.GetSerializableMembers (thisType, context);
foreach (MemberInfo serializableMember in serializableMembers)
        (
/ / Не обробляти поля з атрибут NonSerializedAttribute
             if (! (Attribute.IsDefined (serializableMember,
typeof (NonSerializedAttribute))))
            (
info.AddValue (serializableMember.Name,
                         ((FieldInfo)          serializableMember).
GetValue (this));
)
        )
)
```

Для проведення десеріалізації клас містить конструктор спеціального виду, заповнюються поля класу значеннями з об'єкта класу SerializationInfo.

```
protected Person (SerializationInfo info,
                         StreamingContext context)
    (
Type thisType = this.GetType ();
MemberInfo [] serializableMembers =
                FormatterServices.GetSerializableMembers
(thisType, context);
foreach (MemberInfo serializableMember in serializableMembers)
 (
FieldInfo fieldInformation = (FieldInfo) serializableMember;
             if (! (Attribute.IsDefined (serializableMember,
                                         typeof
(NonSerializedAttribute))))
            (
              fieldInformation.SetValue (this,
                              info.GetValue (fieldInformation.Name,
fieldInformation.FieldType));
```

```
            )
        )
    )
/ / Person
```

Нижченаведений приклад використання створеного класу Person.

```
public class SampleApp
  (
     public static void Main ()
 (
using (Stream stream = new MemoryStream ())
 (
             IFormatter formatter = new BinaryFormatter ();

Person person = new Person ();
Console.WriteLine ( "Збережено: (0)", person.name);
formatter.Serialize (stream, person);

stream.Position = 0;
Person personRestored = (Person) formatter.Deserialize (stream);
Console.WriteLine ( "Відновлено: (0)", personRestored.name);
)
)
)
```

Класи форматування мають механізм, що дозволяє змінити процедури серіалізації і десеріалізації для об'єктів певного класу і його нащадків. Це необхідно, зокрема, при використанні віддалених об'єктів, які маршалізуються по посиланню або не перетинають межі домену програми. Такі об'єкти знаходяться на сервері, а на стороні клієнта для їх використання повинен бути створений певний посередник, що реалізує весь інтерфейс віддаленого об'єкта, включаючи доступ до його полів та властивостей. Для реалізації маршалізації за посиланням до об'єкта форматування через поле SurrogateSelector можна приєднати клас, який реалізує інтерфейс System.Runtime.Serialization.ISurrogateSelector. Він повинен пов'язувати тип віддаленого об'єкта зі спеціальною процедурою його серіалізації і десеріалізації. Використання цього механізму в. NET Remoting призводить до того, що спадкоємці класу MarshalByRefObject не покидають свого домену програми. При використанні ж Binary-Formatter у середовищі MSMQ спадкоємці MarshalByRefObject серіалізується звичайним чином, оскільки використовує об'єкти форматування

клас BinarryMessageFormatter не створює пов'язаний з типом MarshalByRefObject об'єкт класу SurrogateSelector.

Використання класу BinaryFormatter є найбільш ефективним і універсальним, але й найбільш закритим способом серіалізації. Цей клас дозволяє передавати між доменами програми довільний граф об'єктів, але при його використанні розподілена система втрачає властивість відкритості. У разі застосування цього класу взаємодіючі компоненти можуть бути створені тільки на платформі CLI, причому обом сторонам необхідно мати збірку з серіалізуємим типом. При використанні в якості параметрів типів з стандартної бібліотеки чи класів, що їх використовують, бажано щоб обидві сторони були реалізовані на одній версії CLI. Тому для передачі складних типів найкраще використовувати XML. Однак, стандартний клас System.Xml.XmlDocument не може бути серіалізований класами BinaryFormatter і SoapFormatter, оскільки даний клас не має атрибута Serializable. Для серіалізації об'єктів класу XmlDocument найпростіше перетворити його в рядок, а потім серіалізувати його. Можна так само створити спадкоємця XmlDocument, який буде реалізовувати інтерфейс ISerializable.

Нижче наводиться приклад допоміжного класу з двома статичними методами, що перетворить об'єкт класу XmlDocument в рядок і навпаки. Оскільки метод XmlDocument.ToString () проти очікувань не повертає текст XML-документа і у нього немає методу, зворотного LoadXml, то слід використовувати клас StringWriter.

Приклад 1. Двійкова серіалізація

SevaXmlUtils.cs

```
using System;
using System.IO;
using System.Xml;

namespace Seva.Xml
 (
public static class XmlUtils
 (
        public static String XmlToString (XmlDocument xml)
 (
StringWriter xmlLine = new StringWriter ();
            return xmlLine.ToString ();
)
```

```
public static XmlDocument XmlFromString (String xmlLine)
        (
XmlDocument xml = new XmlDocument ();
xml.LoadXml (xmlLine);
return xml;
)
)
)
```

Приклад 2. Двійкова серіалізація

```
using System;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
using System.Runtime.Serialization;

[Serializable]
class MouseEvent : Event
{
    //...
}

 [Serializable]
abstract class Event
{
    public void Save()
    {
        BinaryFormatter ser = new BinaryFormatter();
        Stream      f      =      new      StreamWriter("Test.gaga",
false).BaseStream;

        ser.Serialize(f, this);
        f.Close();
    }
}
BinaryFormatter ser = new BinaryFormatter();
Stream        f        =        new        StreamReader("Test.gaga",
System.Text.Encoding.Default).BaseStream;
Event ev = (Event)ser.Deserialize(f);
 [Serializable]
class MouseEvent : Event, Iserializable
```

```
{
    int XCoordinate;
    public void GetObjectData(SerializationInfo info,
StreamingContext context)
    {
        info.SetType(typeof(MouseEvent));
        info.AddValue("XCoordinate", XCoordinate,
XCoordinate.GetType());
    }
    public  MouseEvent(SerializationInfo  info,  StreamingContext
context)
    {
        XCoordinate = (int)info.GetValue("XCoordinate",
typeof(int));
    }
    //…
}
 [field: NonSerializedAttribute()]
int YCoordinate;
[OnDeserializedAttribute()]
private void RunThisMethod(StreamingContext context)
{
    YCoordinate = 777;
}
```

Усі класи серіалізації бібліотеки. NET Framework мають свої особливості і обмеження, що може викликати значні зміни в програмному коді при переході з одного проміжного середовища на інше. Один із способів боротьби з цією проблемою полягає у відмові від серіалізації нетривіальних класів (містять що-небудь, крім примітивних типів-значень і рядків), і особливо складних облікових структур. Замість них, ймовірно, варто використовувати набори даних (клас System.Data.Dataset) або документи XML (клас System.Xml.XMLDocument). Хоча такий спосіб може бути не зовсім зручним для розробників, він гарантує створення незалежного від класу форматування програмного коду.

Модуль 3. Використання концепцій ООП щодо розробки додатків з графічним інтерфейсом користувача на мові C#

Тема 6. Основи використання технології Windows Forms

6.1. Делегати та події

Делегати мають дуже велику сферу використання, в тому числі і для реалізації моделі подій. Делегати – об'єктно-орієнтована, безпечна концепція, яка не порушує типову захищеність.

Делегат – це об'єкт, який може посилатись на метод. Таким чином, при створенні делегату, по суті створюється об'єкт, котрий може містити посилання на метод. Цей метод можна викликати з допомогою відповідного посилання. Таким чином, делегат може викликати метод, на який він посилається.

Делегати використовуються через дві основні причини.

По-перше, делегати забезпечують підтримку функціонування подій.

По-друге, делегати дозволяють під час виконання програми виконувати метод, котрий точно не відомий в період компіляції. Ця можливість особливо корисна коли потрібно створити оболонку, до якої могли б підключитись програмні компоненти.

Делегати – це безумовно, дуже корисна синтаксична конструкція C#, яка дозволяє визначати ім'я функції, що викликається під час виконання, а не при компіляції. Але ще більш важлива роль делегатів заключається в тому, що на них основана модель подій C #.

Подія – це автоматичне повідомлення про виконання деякої дії.

Модель подій стає невід'ємною частиною сучасного програмування. Вона частіше всього використовується при створенні інтерфейсу користувача, коли кожна дія користувача представляється в виді події і передається конкретному об'єкту.

Делегати є посиланнями на методи, які виконують інкапсуляцію справжніх покажчиків і пропонують зручний сервіс для роботи з ними.

Існує два типи делегатів – одиночні (Singlecast) і комбіновані (Multicast). В синтаксисі вони відрізняються лише типом значення, що повертається. При опису одиночних делегатів створюється клас, який

унаслідується від System.Delegate, а при описі комбінованих делегатів – унаслідується від System.MulticastDelegate.

Об’явлення делегату створює клас – а не просто вказівник на метод, саме тому не порушується об’єктно-орієнтована концепція.

Усі делегати є об’єктами типу System.Delegate або System.MulticastDelegate, який є похідним від першого.

Різниця між цими класами заключається в тому, що екземпляри першого (делегати) можуть зберігати лише одне посилання на методи, екземпляри другого можуть містити одразу декілька посилань на методи. Завдяки цьому можна під єднати до одного делегату декілька методів, кожен з котрих при єдиному зверненні до делегату буде викликатися по черзі. Таким чином, з програми буде видно лише один делегат, який викликає декілька методів.

Ця можливість зручна для підтримки подій, так як дозволяє без використання додаткових механізмів приєднати до події декілька функцій обробників. Фактично делегат становить об’єкт – чорний ящик, який скриває в собі вказівники на функції.

Об’явлення делегатів має наступний синтаксис:

[атрибути] [модифікатори] delegate повертаємий_тип ідентифікатор ([параметри]);

де модифікатори – це модифікатори доступу

Усі методи в середовищі .NET можна розділити на 2 групи: статичні (static) і екземплярні (instance).

Якщо делегат посилається на статичний метод, то в цьому випадку відома вся необхідна інформація для виклику методу: адреса методу, параметри. Якщо делегат посилається на екземплярний метод, то для того щоб викликати екземплярний метод, делегату необхідно знати посилання на об’єкт, до якого прив’язаний даний конкретний метод. Це посилання зберігається в самому об’єкті делегату і вказується при його створенні. Протягом всього життя об’єкта делегату дане посилання не змінює свого значення, воно завжди постійне і може бути задано тільки при його створенні. Не залежно від того, посилається делегат на статичну функцію або на екземплярний метод, звернення до нього ззовні нічим відрізнятись не буде. Всю необхідну функціональність забезпечує сам делегат, разом з середовищем виконання. Це зручно, так як багато делегатів можна прив’язати до однієї події.

При опису делегата потрібно вказати прототип методу, на котрий будуть посилатись екземпляри даного делегату. В загальному випадку опис делегату буде виглядати так:

delegate void MyDelegate(string s);

Для того щоб використати делегат, необхідно створити екземпляр даного класу, який би вказував на потрібний нам метод. Сам по собі делегат є типом і ніяких посилань на метод містити не може, а відповідно не може і використовуватись для звернення до них. Для цього необхідно використати екземпляр делегату.

Створення екземпляру делегату:

myDelegate del = new MyDelegate(MyHandler);

де MyDelegate – тип делегату, del – екземпляр делегату, який буде створений в результаті виконання конструкції. MyHandler – метод, на який буде посилатись цей делегат. Після створення екземпляру делегату можна звертатись до методів, на які він посилається. В мовах високого рівня існує можливість звертатись до екземпляру делегата, як до самого методу. Наприклад:

del("Hello, world!");

Цей рядок викликає метод MyHandler, на який посилається наш делегат.

Приклад роботи з делегатами

```
using System;
// Головний клас додатку.
class App
{
   // Описуємо власний делегат.
   delegate void MyDelegate(string s);

   static void MyHandler(string s)
   {
      // Виводимо на консоль аргумент переданий функції
      Console.WriteLine(s);
   }
   static void Main()
   {
      // Створюємо екземпляр нашого делегата,
      MyDelegate del = new MyDelegate(MyHandler);
      // Викликаємо функцію через делегат.
```

```
        del("Hello World");
    }
};
```

У результаті роботи додатку на екран буде виведено наступний рядок:
Hello, World!

Виклик екземплярних методів відрізняється від виклику статичних тим, що при створенні екземпляру делегату необхідно вказати посилання на об’єкт, який буде використовуватись при виклику методу через даний делегат.

myDelegate del = new MyDelegate(sc.MyHandler);

де sc – посилання на екземпляр об’єкта, котрий повинен використовуватись при зверненні до методу MyHandler.

Приклад використання делегату для звернення до екземплярного методу.

```
using System;
class SomeClass
{
    public string SomeField;
    public void MyHandler(string s)
    {
        Console.WriteLine(SomeField + s);
    }
};
// Головний клас додатка
class App
{
    delegate void MyDelegate(string s);
    static void Main()
    {
        // Створюємо екземпляр класу SomeClass
        SomeClass sc = new SomeClass();
        //Створюємо екземпляр делегату, який містить
        //посилання на сам метод та на об’єкт,
        //для якого буде викликано метод
        MyDelegate del = new MyDelegate(sc.MyHandler);
        sc.SomeField = "Hello, World!";
```

```
        // Викликаємо метод через делегат.
        del(" - from Instance Delegate");
        // Еквівалентно наступному виклику
        // sc.MyHandler(" - from Instance Delegate");
        sc.SomeField = "Good bye, World!";
        // Викликаємо метод
        del(" - from Instance Delegate");
    }
};
```

У результаті роботи додатка на консоль буде виведено наступне:

```
Hello, World! - From Instance Delegate
Good bye, World! - From Instance Delegate
```

Клас MulticastDelegate

Цей клас містить основні сервіси по управлінню делегатами.

Властивості

Method	Повертає метод, на який посилається делегат
Target	Повертає об'єкт, до якого прив'язано метод, на який посилається делегат

Методи

DynamicInvoke	Дозволяє динамічно звертатись до функцій, які зв'язані з делегатом
GetInvocationList	повертає масив делегатів, які прив'язані до делегату, в порядку, у якому вони викликаються
Equality Operator	Оператор (==), дозволяє визначити рівність делегатів
Inequality Operator	Оператор (!=), дозволяє визначити чи різні делегати
Combine	Конкатинує два (або більше) делегати, створюючи новий делегат, список викликань якого включає списки делегатів, які об'єднуються. Початкові делегати не модифікуються
Remove	Видаляє список викликань одного делегата з списку викликань іншого. При цьому створюється новий делегат, список викликань котрого становить результат видалення. Початкові делегати не модифікуються
CreateDelegate	Дозволяє динамічно створювати делегат

MulticastDelegate.Method

Повертає опис методу, на який посилається делегат.

```
public MethodInfo Method { get; }
```

Особливість властивості в тому, що якщо делегат містить не одне посилання на функцію, а декілька, властивість завжди буде повертати останнє додане посилання.

MulticastDelegate.Target

Повертає об’єкт, з яким зв’язаний метод, на який посилається делегат.

public object Target { get; }

Якщо делегат посилається на статичний метод, властивість поверне порожнє посилання (null). Особливість властивості в тому, що якщо делегат містить не одне посилання, а декілька посилань на методи, то властивість поверне об’єкт, зв’язаний з останнім з доданих в чергу методів.

Приклад використання властивостей Method і Target

```
using System;
class SomeClass
{
    // Об’являємо екземплярний метод
    public void InstanceMethod()
    {
       Console.WriteLine("InstanceMethod was called – Hello, World!");
   }
};
// Головний клас додатка
class App
{
   // Опишемо делегат
   delegate void MyDelegate();
   public static void Main()
   {
      // Створюємо екземпляр класу SomeClass
      SomeClass sc = new SomeClass();
      // Створюємо екземпляр делегату
      MyDelegate del = new MyDelegate(sc.InstanceMethod);
      // Виводимо на консоль тип об’єкта, до якого прив’язаний метод,
      // на який посилається делегат.
      Console.WriteLine("Target type = {0}",
```

```
        del.Target.GetType().ToString());
        //Виводимо на консоль інформацію  по методу, на який посилається
        //делегат
        Console.WriteLine("Method = {0}",del.Method.ToString());
        Console.WriteLine();
        //Викликаємо сам  делегат.
        del();
    }
}
```

У результаті роботи додатка на консоль будуть виведені наступні рядки.

```
Target type = SomeClass
Method = Void InstanceMethod()
InstanceMethod was called – Hello, World!
```

MulticastDelegate.DynamicInvoke

Метод дозволяє динамічно звертатись до делегату.

```
public object Delegate.DynamicInvoke(
// Аргументи, які треба передати
// при виклику  функцій, зв’язаних з делегатом.
object[] args
 );
```

Приклад динамічного звернення до делегату

```
using System;

class App
{
    delegate void MyDelegate();
    static void Handler()
   {
       Console.WriteLine("Handler method was called – Hello, World!");
   }
    public static void Main()
   {
       // Створюємо екземпляр делегату.
       SomeDelegate sd = new SomeDelegate(Handler);
      // Звертаємось до делегату.
      sd.DynamicInvoke(new object[0]);
```

```
        // Викликаємо делегат, з неправильною кількістю параметрів
        sd.DynamicInvoke(new object );
    }
};
```

У результаті роботи додатка спочатку на консоль буде виведений рядок Handler method was called – Hello, World!
Потім буде виведено повідомлення про те, що при виклику делегату було передано неправильну кількість аргументів.

Оператори порівняння
Equality та Inequality

Оператори дозволяють вияснити, чи посилаються делегати на одну функцію.

```
public static bool operator == (Delegate d1, Delegate d2);
public static bool operator != (Delegate d1, Delegate d2);
```

При порівнянні делегатів, враховуються не тільки метод, на який посилається делегат, але і посилання на об’єкт, з котрим зв’язаний метод.

MulticastDelegate.Combine та MulticastDelegate.Remove

Методи застосовуються для підтримки делегатів, які посилаються одночасно на декілька функцій.

```
// Об’єднує два делегати в один
public static Delegate Delegate.Combine(
  // Перший делегат.
  Delegate,
  // Другий делегат.
  Delegate
);
// Об’єднує будь-яку кількість делегатів  в один.
public static Delegate Delegate.Combine(
  // Масив делегатів, які будуть об’єднані в один.
  Delegate[]
);
```

У результаті роботи методу буде повернено новий делегат, який в своєму списку викликань буде містити функції всіх вказаних делегатів.

При цьому в старих делегатів при виконанні даної операції списки викликань залишаться такими як є.

Існує функція видалення вказаних методів зі списку викликань деякого делегату.

```
public static Delegate Delegate.Remove(
  // Делегат-джерело.
  Delegate source,
  // Функції, зв’язані з цим делегатом,
  // будуть видалені з делегата-джерела
  Delegate value
);
```

Можна оперувати делегатами за допомогою звичайних операцій додавання та віднімання (+,-,+=,-=). При цьому компілятор автоматично буде генерувати код, який звертається до методів Combine та Remove.

Приклад роботи делегатів, які посилаються на декілька методів

```
using System;
class App
{
   delegate void MyDelegate(string s);
   static void MyHandler(string s)
   {
      Console.WriteLine(s);
   }
    static void Main()
   {
      // Створюємо екземпляр делегату
      // У ролі параметру конструктора
      // передамо посилання на метод MyHandler.
      MyDelegate del = new MyDelegate(MyHandler);
      // Створюємо новий делегат і комбінуємо його з
      // раніше створеним делегатом
      del += new MyDelegate(MyHandler);
      del = del + new MyDelegate(MyHandler);
      // Видаляємо зі списку викликань делегату
      // одне з посилань на метод MyHanlder.
     del -= new MyDelegate(MyHanlder);
     // Звертаємось до кінцевого делегату, при цьому по порядку будуть
```

```
        // викликані всі методи, зв’язані з ним.
        del("Hello, world!");
    }
};
```

У результаті роботи додатка на консоль будуть виведені наступні рядки

```
Hello, World!
Hello, World!
```

MulticastDelegate.CreateDelegate

Метод дозволяє динамічно створювати делегат заданого типу.

```
public static Delegate Delegate.CreateDelegate(
  Type type,
  MethodInfo method);
public static Delegate Delegate.CreateDelegate(
  Type type,
  Object target,
  string method);
public static Delegate Delegate.CreateDelegate(
  Type type,
  Type target,
  String method);
public static Delegate Delegate.CreateDelegate(
  Type type,
  Object target,
  String method,
  bool ignoreCase);
```

Цей метод відомий у рамках підтримки технології відображення, він дозволяє динамічно створювати делегати будь-якого типу. Він використовується якщо на момент компіляції не відомо, скільки аргументів приймає метод зворотного виклику і якого він типу.

Події в C#

Частіше за все події (events) використовуються в додатках під Windows з графічним інтерфейсом користувача, в яких такі елементи управління, як Button (кнопка) або Calendar (календар), реагуючи на подію видають інформацію на тій же панелі, де вони розташовані. Як приклад такої події можна привести натискання мишкою на кнопку.

Однак використання подій зовсім не обмежено програмами з графічним інтерфейсом – вони можуть бути виключно корисними в звичайних консольних програмах.

Події працюють наступним чином. Об’єкт, якому необхідна інформація про деяку подію, реєструє обробник для цієї події. Коли очікувана подія відбувається, викликаються всі зареєстровані обробники.

Події – це члени класу, які об’являються з використанням ключового слова event.

Найбільш розповсюджена форма об’явлення події має вид:

event делегат_події об’єкт;

Елемент делегат_події означає ім’я делегату, який використовується для підтримки події, яка об’являється, а елемент об’єкт – ім’я об’єкта події, що створюється.

Методи екземпляру класу і статичні методи можуть служити обробниками подій, але в їх використанні є різниці. Якщо як обробник використовується статичний метод, повідомлення про подію застосовується до класу (і не явно до всіх об’єктів цього класу). Якщо як обробник подій використовується метод екземпляру класу, події посилаються до конкретних екземплярів цього класу. Кожний об’єкт класу, котрий повинен отримати повідомлення про подію, необхідно реєструвати окремо.

Приклад використання полів-подій

```
using System;
// Вводимо власний делегат, який не приймає ніяких значень
delegate void MyDelegate();
class Button
{
   // Вводимо загальнодоступну подію, до якої можуть підключатись всі
   // бажаючі
   public event MyDelegate Click;
   // Функція, яка симулює подію натиснення на кнопку
   public void SimulateClick()
  {
      // Викликаємо функції, які зв’язані з подією Click,
      if (Click != null)
          Click();
  }
```

```
};
class App
{
   static void Main()
  {
      // Створюємо екземпляр класу/компонента
      Button sc = new Button();
      // Додаємо обробник до його події
      sc.Click += new MyDelegate(Handler);
      // Викликаємо функцію, яка ініціює виникнення події натиснення на кнопку
      sc.SimulateClick();
  }
    //А це функція – обробник події натиснення на кнопку
  static void Handler()
  {
    Console.WriteLine("Hello, World!");
  }
};
```

У результаті виконання роботи додатку, на консоль буде виведений рядок:

Hello, World!

Так як делегати події можуть призначатись для багатоадресної передачі. В цьому випадку на одне повідомлення про подію може відповідати декілька об’єктів.

Приклад використання події призначеної для багатоадресної передачі

```
using System;
// Об’являємо  делегат для події
delegate void MyEventHandler();
// Об’являємо клас події
class MyEvent
{
    public event MyEventHandler SomeEvent;
    //Цей метод викликається для генерації події
    public void OnSomeEvent()
    {
        if(SomeEvent != null)
```

```
            SomeEvent();
    }
}
class A
{
     public void AhandlerO
    {
        Console.WriteLine("Подія, отримана об’єктом A.") ;
   }
}
class B
{
     public void BhandlerO
    {
        Console.WriteLine("Подія, отримана об’єктом B.);
    }
}
class EventDemo
{
     static void handler()
    {
        Console.WriteLine("Подія, отримана класом EventDemo.");
    }
    public static void Main()
     {
        MyEvent evt = new MyEvent();
        A aOb = new A();
        B вOb = new B () ;
         //Додаємо обробник у список подій
        evt.SomeEvent += new MyEventHandler(handler);
        evt.SomeEvent += new MyEventHandler(aOb.Ahandler);
        evt.SomeEvent += new MyEventHandler(вOb.Bhandler);
        //Генеруємо подію
        evt.OnSomeEvent();
        Console.WriteLine();
       //Видаляємо один обробник
       evt.SomeEvent -= new MyEventHandler(aOb.Ahandler);
```

```
        evt.OnSomeEvent();
    }
}
```

Результати виконання програми:

Подія, отримана класом EventDemo.
Подія, отримана об’єктом A.
Подія, отримана об’єктом B.
Подія, отримана класом EventDemo.
Подія, отримана об’єктом B.

При цьому створюються два додаткових класи A, B, у яких також визначаються обробники подій, які сумісні з сигнатурою делегату MyEventHandler.

Контроль над подіями

Події .NET повністю закриті для програміста. Вони приховують всю брудну роботу і контролювати звернення до подій при стандартному способі використання подій неможливо.

Існує додатковий розширений режим їх використання, де програміст може самостійно об’являти функції, які управляють підключенням і відключенням делегатів – обробників. Для цього в мові C# передбачена спеціальна конструкція, з використанням двох додаткових ключових слів add і remove.

```
event DelegateName SomeEvent
{
  add
  {
    // Код, який реалізує додавання делегата до списку виклику події
  }
  remove
  {
    // Код, який реалізує вилучення делегату зі списку викликів події
  }
}
```

При використанні розширеного режиму компілятор не буде автоматично описувати поле-делегат, і його доведеться вводити самостійно.

Приклад підтримки подій у ручному режимі.

```
using System;
```

```
// Описуємо делегат
delegate void MyDelegate();
class Button
{
  // Закрите поле – посилання на екземпляр делегату, котрий буде
  //обслуговувати подію
  private MyDelegate _click;
  // Подія користувача, з можливістю контролю доступу до нього
  public event MyDelegate Click
  {
      // Ця функція буде викликана при спробі додавання делегату в
список
       //виклику події
      add
     {
        // Повідомлення користувачеві про те, що відбулась спроба додати
делегат
         Console.WriteLine("add handler was invoked");
         // Додаємо делегат у список викликів
         _click += value;
     }
     // Ця функція буде викликана при спробі виключення
     // делегата зі списку викликів події
     remove
    {
       // Повідомлення користувачеві про те, що відбулась спроба
виключити
        // делегат зі списку викликів.
        Console.WriteLine("remove handler was invoked");
       // Видаляємо цю функцію зі списку обробки
       _click -= value;
    }
 }
 //Функція симулювання події натиснення на кнопку
 public void SimulateClick()
 {
   // Викликаємо функції, пов’язані з подією Click
```

```
    if (Click != null)
      Click();
  }
};
class App
{
  public static void Main()
  {
    // Створюємо екземпляр компонента/ класу.
    Button sc = new Button();
    // Додаємо обробник події.
    sc.Click += new MyDelegate(Handler);
    // Посередньо викликаємо функцію нашого обробника
    sc.SimulateClick();
    // Удаляємо функцію-обробник з списку викликань
    sc.Click -= new MyDelegate(Handler);
    // Посередньо викликаємо функцію нашого обробника
    sc.SimulateClick();
  }
  // Функція-обробник для компонента / класу
  public static void Handler()
  {
    Console.WriteLine("Hello World - Handler was invoked");
  }
};
```

У результаті роботи прикладу отримаємо на консолі наступні рядки

```
add handler was invoked
Hello World - Handler was invoked
remove handler was invoked
```

Як видно, програма діє по заданій схемі, контролюючи при цьому додавання та виключення делегатів.

Така можливість може бути корисною для аналізу і контролю за функціями, які прикріплені до події.

Події можна визначати в інтерфейсах. "Доставкою" подій повинні займатись відповідні класи. Події можна визначати, як абстрактні.

Забезпечити реалізацію такої події повинен похідний клас. Але події реалізовані з допомогою засобів доступу add та remove,

абстрактними бути не можуть. Будь-яку подію можна визначити з допомогою ключового слова sealed. Подія може бути віртуальною, тобто її можна перевизначити в похідному класі.

Стандартний делегат загальної бібліотеки

C# дозволяє програмісту створювати події будь-якого типу. Але в цілях компонентної сумісності з середовищем .NET Framework необхідно слідувати рекомендаціям, підготовленим Microsoft спеціально для цих цілей. Центральне місце в цих рекомендаціях займає вимога того, щоб обробники подій мали два параметри. Таким чином, .NET – сумісні обробники подій повинні мати наступну загальну форму запису:

```
public delegate void EventHandler(
  // Посилання на об’єкт, який викликав подію
  object sender,
  // Параметри, які описують подію
  EventArgs e
);
```

При зверненні до події, яка має тип даного делегату, як параметр потрібно передати посилання на поточний об’єкт (this), а в другому параметрі екземпляр класу EventArgs або похідний від нього.

```
someEvent(this, EventArgs.Empty);
```

де властивість Empty повертає порожній екземпляр типу EventArgs. Його можна створити і самому, використавши оператор new.

```
SomeEvent(this, new EventArgs());
```

3.3 Використання вбудованого делегату EventHandler

Для багатьох подій параметр типу EventArgs не використовується. Для спрощення процесу створення коду в таких випадках середовище .NET Framework включає вбудований тип делегату – EventHandler. Його можна використовувати для об’явлення обробників подій, яким не потрібна додаткова інформація.

Приклад використання вбудованого типу делегату EventHandler

```
using System;
// Об’являємо клас події
class MyEvent
{
    public event EventHandler SomeEvent; // Об’явлення використовує
делегат
                                         // EventHandler.
```

```
    // Метод викликається для генерації SomeEvent-події
    public void OnSomeEvent()
   {
       if(SomeEvent != null)
           SomeEvent(this, EventArgs.Empty);
    }
}
class EventDemo
{
    static void handler(object source, EventArgs arg) {
    Console.WriteLine("Подія відбулась");
    Console.WriteLine("Джерелом є клас " +source + ".") ;
    public static void Main()
    {
       MyEvent evt = new MyEvent();
       //Додаємо обробник handler() в список подій
       evt.SomeEvent +- new EventHandler(handler);
       //Генеруємо подію
       evt.OnSomeEvent();
    }
}
```

У даному випадку параметр типу EventArgs не використовується і замість нього передається об’єкт – заповнювач EventArgs.Empty. Результати виконання цієї програми:

Подія відбулась
Джерелом є клас MyEvent

Використання делегатів та подій у звичайних додатках неважко. Вище були переглянуті всі найбільш важливі аспекти проблеми. Але при проектуванні стійких до збоїв додатків, можна задуматись про створення власного механізму звернення до делегатів, з використанням багато-поточної моделі і більш захищених алгоритмів. Делегати широко застосовуються для асинхронної обробки і додавання нестандартного коду до коду класів. Делегати можуть використовуватись для багатьох цілей, включаючи методи зворотного виклику, визначення статичних методів і обробку подій.

6.2. Введення до Windows Forms

Мова C# і .NET Framework-бібліотека Forms пропонують повністю об'єктно-орієнтований підхід до Windows-програмування. Бібліотека Forms визначає простий, інтегрований і логічний спосіб управління розробкою Windows-додатків. Цей рівень інтеграції став можливим лише завдяки таким унікальним засобам мови C#, як делегати і події. Більш того, завдяки специфічному в C# використанню системи збору сміття, практично усунена проблема "витоки пам'яті".

У .NET передбачено два головні простори імен, що забезпечують інструментами для створення додатків з графічним інтерфейсом. Перший простір імен – System.Windows.Forms – призначено для створення звичайних застосувань .NET з графічним інтерфейсом. Це можуть бути як окремі настільні застосування, що працюють абсолютно незалежно, так і клієнтські частини з великими можливостями (так звані "товсті клієнти", fat clients) у розподілених клієнт-серверних застосуваннях. Типи простору імен Windows.Forms ховають від нас виклики Win32 API, дозволяючи зосередитися не на технічних складнощах, а на функціональних можливостях нашого застосування. Другий простір імен, який також може використовуватися для створення додатків з графічним інтерфейсом, – це System.Web.UI (і вкладене в нього System. Web.UI.WebControls). Ці простори імен використовуються для розробки додатків ASP.NET і дозволяють створювати клієнтські частини додатки, що працюють в будь-якому браузері. При цьому використовуються стандартні протоколи HTML, HTTP та ін.

Як і всі інші простори імен, System.Windows.Forms містить величезну кількість типів: класів, структур, делегатів, інтерфейсів і перерахувань. Усі типи простору імен System.Windows.Forms можно об’єднати за наступними категоріями:

Ключова інфраструктура. Це типи, які містять ключові операції .NET-програм Windows Forms (Form, Application та ін.), а також різні типи для взаємодії з унаслідованими елементами управління ActiveX.

Елементи управління, типи, необхідні для реалізації інтерфейса користувача з широкими можливостями (Button, MenuStrip, ProgressBar, DataGridView та ін.). Усі ці типи наслідуються від базового класу Control. Елементи управління налаштовуються на етапі розробки і будуть видимі за умовчанням під час виконання.

Компоненти. Типи, які не наслідуються від базового класу Control, але незважаючи на це надають візуальні можливості .NET-програм Windows Forms (ToolTip, ErrorProvider та ін.). Багато компонентів (такі, як Timer) невидимі під час виконання, але можуть налаштовуватися візуально на етапі розробки.

Стандартні діалогові вікна. Windows Forms має декілька готових діалогових вікна для стандартних операцій (OpenFileDialog, PrintDialog, та ін.). Звичайно, якщо стандартні діалогові вікна вам не підійдуть, ви маєте можливість створити свої діалогові вікна.

Ядром Windows-програм, написаних на C#, є форма. Форма інкапсулює основні функції, необхідні для створення вікна, його відображення на екрані і здобуття повідомлень. Форма створюється за допомогою реалізації об'єкта класу Form або класу, похідного від Form. Клас Form окрім поведінки, обумовленого власними членами, демонструє поведінку, успадковану від предків. Серед його базових класів виділяються своєю значущістю класи System.Componentmodel. Component і System.Windows.Forms.Control. Клас Control визначає межі, властиві всім windows-елементам управління. Той факт, що клас Form виведений із класу Control, дозволяє використовувати форми для створення елементів керування. Коли вікну посилається повідомлення, воно перетвориться в подію. Отже, щоб обробити Windows-повідомлення, досить для нього зареєструвати обробник подій. При здобутті цього повідомлення обробник подій викликатиметься автоматично.

Для створення в проекті нової форми необхідно відкрити вікно Solution Explorer (за умовчанням воно видно), в якому відображується структура всього проекту, і в ньому в контекстному меню вибрати пункти Add->Windows Form. Після цього треба вибрати потрібний тип форми, ввести її ім'я і натиснути OK. Якщо вікно Solution Explorer закрито, його можна викликати за допомогою пунктів головного меню View->Solution Explorer.

Під час життєвого циклу форми з нею відбуваються наступні події:

- Load;
- GotFocus;
- Activated;
- Closing;
- Closed;

- Deactivate;
- LostFocus;
- Dispose.

Відображеня головної форми відбувається під час запуску програми. Решту форм можна викликати за допомогою двох методів класу Form: Show та ShowModal. Метод Show відображає звичайну форму, ShowModal відображає модальну форму.

Для виведення яких-небудь повідомлень можна використовувати метод Show классу Messagebox з простору імен System.Windows.Forms. Використовуючи цей клас, можна організувати просту інтерактивність з користувачем.

Приклад. Виведення діалогового вікна з повідомленням за допомогою методу MessageBox.Show().

```
MessageBox.Show("MessageBox - класс");
```

Вікно додатка і діалогове вікно повідомлення мають вигляд, наведений на рис. 6.1.

Рис. 6.1. **Вигляд вікна додатка і діалогового вікна повідомлення**

За допомогою методу Show (є перевантаженим, 26 перевантажень) класу Messagebox показується діалогове вікно з повідомленням, яке містить текст, кнопки і значки, які інформують і інструктують користувача.

Метод Show відображує модальне діалогове вікно повідомлень в центрі екрану. Після натиснення будь-якої з кнопок діалогове вікно закривається і функція Show дозволяє отримати відповідь користувача, повертаючи одну з констант типу Dialogresult. Проаналізувавши цю повернену константу можна управляти виконанням додатка.

Клас перерахування Dialogresult має наступні поля:

Yes – значення, повернене діалогом	*None* – нічого не повертає діалог, діалог продовжується	
No	*Retry* – повернути	*Ignore* – ігнорувати
OK	*Cancel* – відмінити	*Abort* – перервати

Приклад. Закриття форми з підтвердженням. Після клацання по кнопці "Close" на формі з'являється діалогове вікно з кнопками Yesno і піктограмою "Question". Після клацання по кнопці Yes батьківська форма закривається.

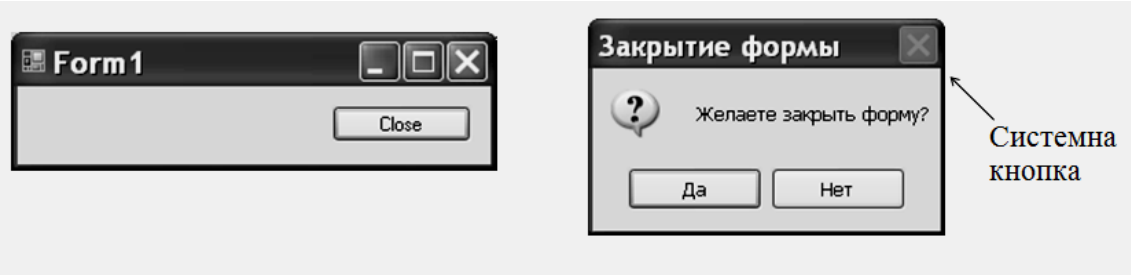

Рис. 6.2. **Вигляд основної форми та діалогового вікна**

Після клацання по кнопці No батьківська форма не закривається і продовжує функціонувати. Приклад демонструє передачу інформації від батьківської форми до діалогу і від діалогу батьківській формі.

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
      {
 if (MessageBox.Show("Бажаєте закрити форму?", "Закриття форми",
    MessageBoxButtons.YesNo,
    MessageBoxIcon.Question,
        MessageBoxDefaultButton.Button1) != DialogResult.Yes)
         {
            e.Cancel = true;
         }
               }
```

Компоненти панелі управління для windows-додатків представлені на рис. 6.3.

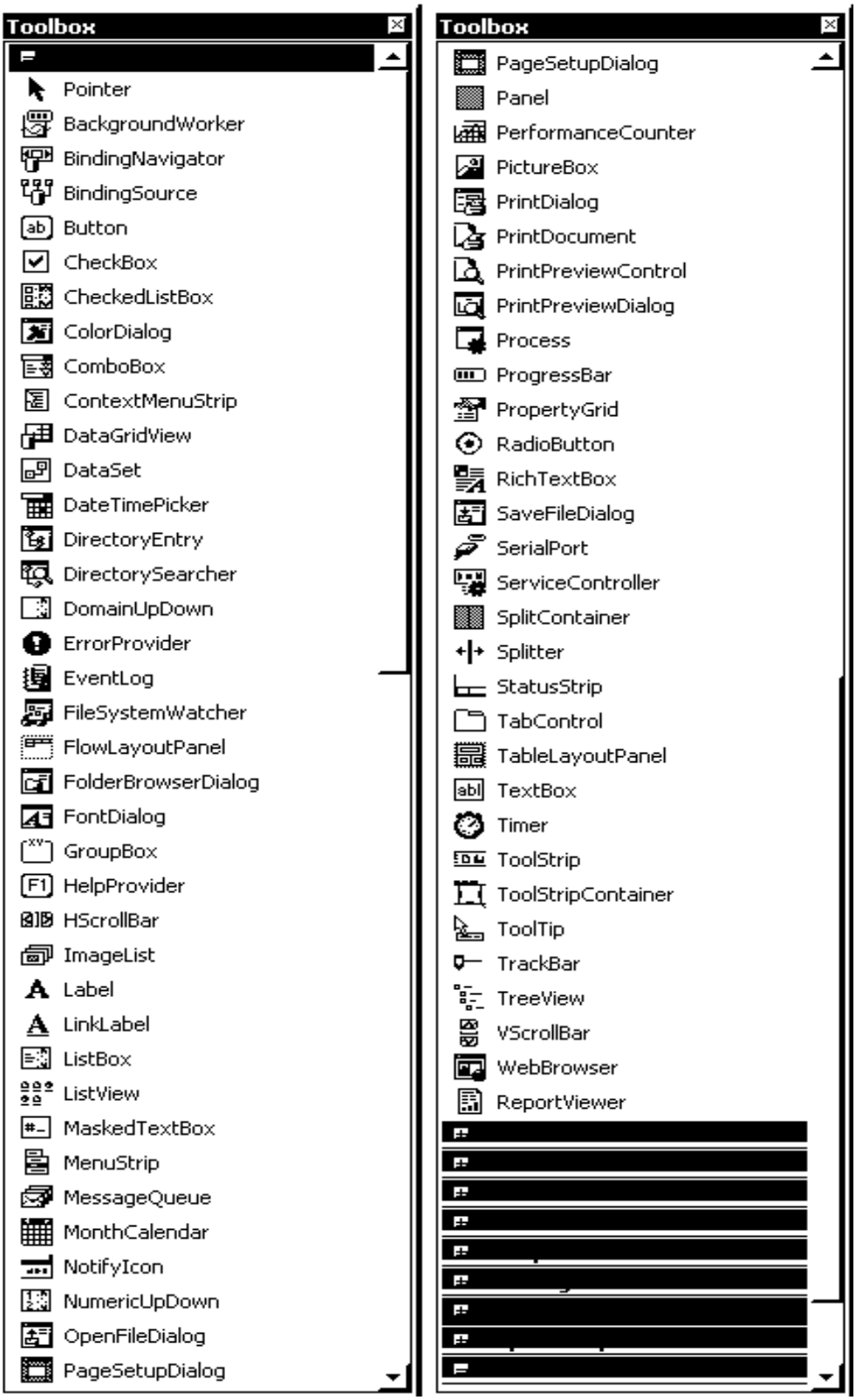

Рис. 6.3. **Панель Toolbox**

Клас Application можна розглядати як "клас нижчого рівня", що дозволяє нам управляти поведінкою додатка Windows Forms. Крім того, цей клас визначає набір подій рівня всього застосування, наприклад закриття додатка або простій центрального процесора. В більшості

випадків нам не доведеться безпосередньо взаємодіяти з цим типом, проте інколи його члени можуть виявитися виключно корисними.

Клас Application також визначає безліч статичних властивостей, більшість з яких доступні лише для читання.

6.3. Основи використання елементів управління

Форма – це графічний елемент, що з'являється на екрані. Форма служить контейнером для решти елементів управління.

Кожний додаток може мати одну або декілька форм, одна з яких є головною і відображається першою при запуску програми. При закритті головного вікна (форми) додатка припиняється робота всього додатка, при цьому також закриваються всі інші вікна додатка.

У програмі форма представляється об'єктом класу Form з простору імен System.Windows.Forms. Основні властивості, методи і події класу Form наведені в табл. 6.1 – 6.3 відповідно.

Таблиця 6.1

Основні властивості класу Form

Властивість	**Призначення**
1	2
BackColor	Колір фону
BackgroundImage	Фоновий малюнок форми
ControlBox	Встановлює значення, що є індикатором наявності у форми екранних кнопок управління станом вікна (має значення True або False)
Cursor	Тип курсора миші
FormBorderStyle	Зовнішній вигляд форми. Встановлює значення, яке задає стиль меж форми: Значення властивості — Вид межі вікна Fixed3D — Розмір вікна міняти не дозволяється. Вид меж – як у стандартних діалогових вікон Windows. FixedSingle — Розмір вікна міняти не дозволяється. Вид меж – тонка смуга. None — Розмір вікна міняти не дозволяється. Видима межа відсутня. Sizeable — Стандартна межа, що допускає зміну розмірів вікна (за умовчанням). FixedToolWindow — Аналогічно FixedSingle, але висота заголовка вікна зменшена. SizeToolWindow — Аналогічно Sizeable, але висота заголовка вікна зменшена

Закінчення табл. 6.1

1	2
Font	Шрифт написів на формі
Height	Висота форми
Icon	Піктограма розміру 32 x 32
Location	Координати (X, Y) лівого верхнього кута форми щодо її контейнера. Властивість впливає на місцеположення форми, якщо властивість StartPosition=Manual
MaximizeBox	Використовується для вказівки про наявність або відсутність у форми кнопки максимізації
MinimizeBox	Використовується для вказівки про наявність або відсутність у форми кнопки мінімізації
Opacity	Ступінь прозорості форми
Text	Заголовок форми
StartPosition	Встановлює значення, яке задає початкову позицію вікна форми на екрані
Width	Ширина форми
WindowState	Стан форми (згорнута, розгорнена, нормальний розмір)

Таблиця 6.2

Основні методи класу *Form*

Метод	Опис
Close()	Закриває форму
CenterToScreen()	Розміщує форму в центрі екрану
Hide()	Приховує форму
ShowDialog()	Відображає форму у вигляді модального діалогового вікна
Show()	Відображає форму у вигляді немодального вікна

Таблиця 6.3

Основні події класу Form

Властивість	Призначення
1	2
Activated	Форма стала активною, виникає у момент активізації вікна (при отриманні формою фокусу введення)
FormClosing	Виникає перед закриттям форми
FormClosed	Виникає після закриття форми
Load	Відбувається після того, як форма розміщується в пам'яті, але поки залишається невидимою на екрані. Подія виникає один раз. У обробник події зазвичай включається код, що виконує початкову ініціалізацію форми і компонентів форми на додаток до параметрів, встановлених на етапі розробки програми. Крім того, в обробник включаються додаткові операції, які повинні відбуватися одноразово при створенні форми.
MouseClick	Клацання кнопкою миші на формі

Закінчення табл. 6.3

1	2
KeyDown	Виникає при натисненні клавіші на клавіатурі
KeyPress	Виникає, якщо форма має фокус введення і користувач натискає, а потім відпускає клавішу на клавіатурі
KeyUp	Виникає при відпуску клавіші на клавіатурі
MouseClick	Клацання кнопкою миші на формі
Resize	Виникає при зміні розмірів вікна

Стандартні діалогові вікна повідомлень

Приклад. Виведення діалогового вікна з повідомленням за допомогою методу MessageBox.Show().

MessageBox.Show("MessageBox - клас");

За допомогою методу Show (є перевантаженим, 26 перевантажень) класу MessageBox показується діалогове вікно з повідомленням, яке може містити текст, кнопки і значки, які інформують і інструктують користувача.

Метод Show відображає модальне діалогове вікно повідомлень в центрі екрану. Після натиснення будь-якої з кнопок діалогове вікно закривається, а функція Show дозволяє отримати відповідь користувача, повертаючи одну з констант типу DialogResult. Проаналізувавши значення цієї константи можна управляти виконанням додатка.

Приклад. Закриття форми з підтвердженням.

Після клацання по кнопці "Close" на формі з'являється діалогове вікно з кнопками "YesNo" і піктограмою "Question". Після клацання по кнопці "Yes" батьківська форма закривається. Після клацання по кнопці "No" батьківська форма не закривається і продовжує функціонувати. Приклад демонструє передачу інформації від батьківської форми до діалогу і від діалогу батьківській формі.

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
if (MessageBox.Show("Бажаєте закрити форму?", "Закриття форми"
    MessageBoxButtons.YesNo,
    MessageBoxIcon.Question,
```

```
        MessageBoxDefaultButton.Button1) != DialogResult.Yes)
          {
             e.Cancel = true;
 }
```

Створення простого Windows-додатка за допомогою MS Visual Studio
Перший спосіб

1. Створити проект типу Console Application.
2. У властивостях проекту (пункт меню Project->Properties) вказати тип додатка: Windows Application.
3. Додати посилання на необхідні простори імен (для цього у вікні Solution Explorer виділити проект і в його контекстному меню вибрати Add References. У вікні, що з'явилося, на вкладці .NET вибрати System.Windows.Forms і System.Drawing).
3. Набрати в редакторові наступний код:

```
using System;
using System.Windows.Forms;
using System.Drawing;

namespace ConsoleWin3
{
   class Program
   {

      static void f_Click(object sender, EventArgs e)
      {
         System.Windows.Forms.MessageBox.Show ("Виконано клацання формою");
      }

      static void Main(string[ ] args)
      {
         Form f = new Form();
         f.Text = "Моя перша форма";
         f.Width = 300;
         f.Height = 300;
         f.Cursor = Cursors.Hand;
         f.Top = 200;
```

```
            f.Left = 200;
            f.BackColor = Color.Cyan;
            f.  MouseClick += new MouseEventHandler(f_Click);
            Application.Run(f);
        }

    }
}
```

4. Запустити програму на виконання і перевірити її працездатність (реакцію на клацання миші на формі).

Другий спосіб

1. Створити проект типу Windows Application.
2. Виділити форму.
3. Відкрити вікно Properties і встановити значення відповідних властивостей. Наприклад, змінити значення властивості Text з "Form1" на "Моя перша програма".
4. У вікні Properties перемкнутися на вкладку Events і вибрати подію MouseClick.
5. Двічі клацнути в полі напроти події MouseClick. Середовище програмування автоматично відкриє файл Form1.cs і автоматично створить заготівку методу – обробника події MouseClick:

```
private void Form1_MouseClick(object sender, EventArgs e) { }
```

6. Додати в метод Form1_MouseClick наступний код:

```
MessageBox.Show("Виконано клацання формою");
```

Тема 7. Розробка та використання елементів управління

7.1. Використання основних елементів управління

Середовище MS Visual Studio надає велику кількість компонентів, серед яких можна виділити елементи управління і невізуальні компоненти. Елементи управління – це візуальні компоненти, що забезпечують взаємодію між користувачем і програмою. Невізуальні компоненти не мають графічного інтерфейсу і є класами, що забезпечують додаткові функціональні можливості для додатка.

У вікні Toolbox всі компоненти розбиті на декілька груп:

елементи управління загального призначення (Common Controls) – елементи управління, які часто використовуються;

контейнери (Containers) – елементи управління, які можуть містити інші елементи управління;

меню і панелі інструментів (Menus & Toolbars);

дані (Data) – елементи управління для роботи з даними;

компоненти (Components) – компоненти, що не мають графічного інтерфейсу;

друк (Printing) – стандартні діалогові вікна для управління процесом друку документів;

діалоги (Dialogs) – стандартні діалогові вікна загального призначення.

Для додавання компонентів на форму можна використовувати спосіб перетягання мишею.

Елемент управління "Кнопка" (група Common Controls)

Кнопка (Button) – це найпростіший зі всіх елементів управління і при цьому найчастіше використовуваний. Кнопка є елементом, що управляє, і використовується для видачі команд на виконання певних функціональних дій, тому часто її називають командною кнопкою. На поверхні кнопки може міститися текст і/або графічний малюнок.

Безпосередній предок класу System.Windows.Form.Button у ієрархії класів .NET – це клас ButtonBase, що забезпечує загальні можливості для цілої групи похідних від нього елементів управління.

Сам клас Button не додає яких-небудь додаткових можливостей крім успадкованих від ButtonBase, окрім важливої властивості DialogResult. Ця властивість дозволяє повертати значення при закритті діалогового вікна, наприклад, при натисненні кнопок ОК або Cancel (Відмінити).

Вирівнювання тексту і зображень щодо країв кнопки

У переважній більшості випадків вирівнювання тексту, розміщеного на кнопці, проводиться по центру, так що текст буде розміщений строго посередині кнопки. Проте якщо нам необхідно використовувати інший стиль вирівнювання, в нашому розпорядженні – властивість *TextAlign*, визначена в класі *ButtonBase*.

Усі властивості будь-якого компоненту перераховані у вікні *Properties* (зробити компонент активним, клацнути по кнопці *Properties*). Внизу вікна *Properties* дана коротка характеристика активної властивості.

Події кнопки

Основною для кнопки є подія Click, що виникає при її натисненні. При цьому кнопка приймає відповідний вигляд, підтверджуючи дію, що відбувається, візуально. Дії, що виконуються в обробнику події Click, відбуваються відразу після відпуску кнопки.

Діалогові вікна для роботи з файлами (група Dialogs)

Компоненти OpenFileDialog і SaveFileDialog є невізуальними і дозволяють використовувати в програмі стандартні діалогові вікна Windows. Ці компоненти застосовуються для отримання від користувача бажаних значень настройок, наприклад, для введення повного імені файла разом зі шляхом пошуку.

Компонент "Відкрити файл" (OpenFileDialog) призначений для вибору файла з метою подальшого відкриття. Компонент SaveFileDialog виконує діалог "Зберегти файл як**".** Всі властивості цих компонентів однакові, тільки їх сенс декілька різний для відкриття і закриття файлів. Основна властивість, в якій повертається у вигляді рядка вибраний користувачем файл, – FileName.

Приклад: обробник пункту меню "Зберегти".

```
private void saveMenuItem_Click(object sender, EventArgs e)
{
   string FName;
   saveFileDialog1.Filter = "Текстові файли  *.txt Всі файли *.*";
   if ( saveFileDialog1.ShowDialog() == DialogResult.OK)
   {
      FName = saveFileDialog1.FileName;
      StreamWriter sw = new StreamWriter(FName);
      sw.Write(textBox1.Text);
      sw.Close();
    }
}
```

Приклад: обробник пункту меню "Відкрити".

```
private void toolStripMenuItem4_Click(object sender, EventArgs e)
{
   string FName="";
   openFileDialog1.Filter = "Текстові файли  *.txt Всі файли *.*";
```

```
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
      FName = openFileDialog1.FileName;
      StreamReader sr = new StreamReader(FName);
      textBox1.Text = sr.ReadToEnd();
      sr.Close();
    }
}
```

7.2. Використання таблиць у графічному інтерфейсі користувача

Елемент управління "Таблиця" (клас DataGridView) призначений для виведення на форму даних у табличному вигляді. Він володіє величезною функціональністю – реалізує 118 власних подій на додаток до 69 подій, успадкованих від класу Control.

Додавання колонок – крок, абсолютно необхідний перед тим, як таблиця буде пред'явлена кінцевому користувачеві. Якщо не було додано жодної колонки, DataGridView буде тьмяно-сірим прямокутником на формі, абсолютно позбавлений якої-небудь функціональності. Існує ряд способів додавання колонок. У найзагальнішому випадку стикаємося з чотирма можливими сценаріями:

1. Є джерело даних, воно доступне під час розробки, і можна додавати колонки в цей час.
2. Немає джерела даних, але вже під час розробки знаємо склад і тип колонок, і готові додавати їх.
3. Є джерело даних, але воно доступне тільки під час виконання, а під час розробки нічого невідомо ні про нього, ні про склад колонок.
4. Немає джерела даних, а склад/тип колонок з'ясовується динамічно, під час виконання, а під час розробки невідомий тип і, можливо, навіть кількість колонок.

Усі інші варіації зводяться до комбінацій цих чотирьох базисних сценаріїв.

1. Джерело даних доступне під час розробки

Після завдання значень властивостей *DataSource* і *DataMember* таблиця автоматично вивчає схему джерела і генерує по колонці для кожної колонки таблиці або властивості об'єкта, колекція яких використовується як джерело даних. Причому робить це "розумно",

підбираючи не тільки відповідний заголовок колонки, але і тип колонки. Тобто якщо тип колонки буде чимось ніби int/decimal/string, то додасться колонка типу DataGridViewTextBoxColumn. А якщо така колонка матиме тип boolean, то додасться вже DataGridViewCheckBoxColumn. Зрозуміло, маємо право видалити "зайві" на погляд автора колонки, поправити текст заголовка, а також тип колонки. От як це робиться. Після завдання значень властивостей *DataSource* і, за необхідності, *DataMember,* вже маємо колонки, що згенерували по описаному вище алгоритму. Виділивши таблицю в дизайнерові, натиснемо її "розумний ярлик" (smart tag). "Розумний ярлик" знаходиться у верхньому правому кутку елементу управління і надає доступ до меню, склад елементів якого можна охарактеризувати як "найбільш часто використовувані настройки".

Меню дозволяє робити з таблицею багато цікавих речей, але в даному сценарії найцікавіший пункт Edit Columns. При виборі цього пункту відкривається діалог редагування колонок.

У ньому можна видалити зайві колонки (кнопка Remove), змінити заголовок колонки (властивість HeaderText), тип колонки (властивість ColumnType) і ряд інших властивостей кожної колонки. У списку Selected Columns зліва показуються всі колонки, причому їх порядок "зверху-вниз" відповідає порядку "зліва-направо" реальної таблиці. Парою кнопок із стрілками можна міняти їх порядок у цьому списку.

У даному сценарії основну роботу виконує візуальний дизайнер MS Visual Studio.

2. Відсутність джерела даних під час розробки

У цьому сценарії не встановлюємо значення властивостей *DataSource* і *DataMember,* а готуємося працювати у вільному, неприв'язаному режимі. Відкриваємо все ті ж smart tag-меню, але вибираємо пункт Add Columns.

Оскільки джерела даних немає (властивості *DataSource* і *DataMember* виставлені в null), перемикач Databound column не працює. Отже, нам доступний тільки перемикач Unbound column і підлеглі йому поля. Задаючи значення в цьому вікні, фактично, задаємо наступні властивості нової колонки:

Ім'я колонки (не плутайте із заголовком) – це ім'я створюваної у формі змінної (тип якої DataGridViewColumn або його спадкоємець), в яку поміщається посилання на колонку, що додається.

Тип колонки.

Текст, який буде показаний у заголовку даної колонки.

Три перемикачі, що залишилися, допомагають "на льоту" задати відповідні властивості колонки.

Після натиснення на кнопку Add даний діалог не закривається, а пропонує нові значення за умовчанням – Column2, Column3 і т. д.

У даному сценарії програміст указує, які колонки і в якому вигляді він хоче бачити.

3. Готове джерело даних, що підключається під час виконання

Сценарій, що змагається по простоті зі сценарієм номер 1. Якщо властивість DataGridView.A*utoGenerateColumns* виставлено в true (а за умовчанням так і є), то під час виконання будь-яка зміна властивостей DataSource/DataMember ***викликає*** генерацію колонок по алгоритму сценарію 1. Можна також запустити (перезапустити) цей процес генерації і додавання, встановивши згадану властивість в false, а потім повернувши його в true.

У даному сценарії програміст користується колекцією колонок, що заздалегідь згенерували, вносячи мінімальні зміни.

4. Відсутність джерела даних під час виконання

У даному випадку немає іншого виходу, окрім як скласти власну колекцію колонок. Перший крок – визначити тип колонок, які хотілося б бачити таблиці. Оскільки колонки всіх типів (і вбудовані, і призначені для користувача) додаються в таблицю однаково, в даному розділі буде розглянута робота з найпоширенішим типом колонки – DataGridViewTextBoxColumn. Робота зі всіма іншими типами колонок абсолютно аналогічна.

Є наступні шляхи програмного додавання колонок:

Метод *Add()* колекції колонок. Цей метод перевантажений і дозволяє додавати як готову колонку (екземпляр класу DataGridViewColumn або його спадкоємця), так і пару "ім'я – заголовок":

```
_grid.Columns.Add("MyColumnName", "MyColumnHeaderText");
_grid.Columns.Add(new DataGridViewColumn(...));
```

Просто встановити властивість ColumnCount таблиці в яке-небудь значення більше нуля:

```
_grid.DataSource = null; //якщо до цього була прив'язка до джерела
_grid.ColumnCount = 5;
```

При цьому будуть створені колонки, що ініціалізували значеннями за умовчанням.

У даному сценарії програміст повністю відповідає за створення нових колонок і за їх внесення до колекції. Настройка нових колонок також повністю лежить на ньому.

Додавання рядків

Властивість DataGridView.*Rows* забезпечує доступ до колекції рядків. Користуючись нею, можна додати рядки в таблицю. Але на відміну від цілих чотирьох можливих сценаріїв додавання колонок, у разі додавання рядків сценарій всього один. Додати рядки в DataGridView можна або програмно, скориставшись методом Add колекції рядків, або підключивши до нього деяке джерело даних.

Метод Add() має чотири варіанти:

```
// додає один рядок, заповнюючи його значеннями за умовчанням
int Add();

// додає один рядок, заповнюючи його значеннями з масиву values
int Add(params object[] values);

// додає декілька рядків, заповнюючи їх значеннями за умовчанням
int Add(int count);

// додає заздалегідь створений рядок
int Add(DataGridViewRow dataGridViewRow);
```

DataGridView допускає наявність в одній колонці чарунок різних типів. Для цього спочатку об'єкт типу DataGridViewRow повинен бути створений і заповнений окремо. І тільки тоді доданий у таблицю. При цьому кількість колонок у рядку повинна відповідати числу колонок таблиці.

Нижче наведений приклад додавання перемикача в першу колонку третього рядка :

```
_grid.DataSource = null;
// створимо 3 колонки типу DataGridViewTextBoxColumn
_grid.ColumnCount = 3;
_grid.Rows.Add();
_grid.Rows.Add();
```

```
DataGridViewRow newRow = new DataGridViewRow();
// Створюємо чарунку типу CheckBox
DataGridViewCheckBoxCell checkCell = new DataGridViewCheck
BoxCell();
checkCell.Value = true;
// Додаємо як першу чарунку нового рядка чарунку типу
CheckBox
newRow.Cells.Add(checkCell);
// Решту чарунок заповнюємо чарунками типу TextBox
newRow.Cells.Add(new DataGridViewTextBoxCell());
newRow.Cells.Add(new DataGridViewTextBoxCell());
// цей рядок буде з перемикачем у першій колонці
_grid.Rows.Add(newRow);
```

7.3. Розробка елементів управління

Елемент управління "Меню" (клас MenuStrip)

Створення меню за допомогою вбудованого редактора:

1. Помістити на форму проекту елемент управління MenuStrip (група Menus & Toolbars). За умовчанням меню займе місце вверху клієнтської частини форми.

2. Виклик контекстно-залежного редактора меню – клацнути по піктограмі з маленькою стрілкою (розташована справа вгорі елементу меню). За допомогою редактора меню можна:

вставити стандартне меню (вкладка Insert Standard Items);

розташувати меню в потрібному місці на формі (властивість Dock);

викликати редактор меню (вкладка Edit Items*).*

Елемент управління "Рядок стану" (клас StatusStrip)

Рядок стану зазвичай розміщується в нижній частині форми, може ділитися на будь-яке число "панелей" з текстовою або графічною інформацією, що містить пояснення для пунктів меню, поточний час або спеціальні дані застосування.

Елемент управління "Панель інструментів" (клас ToolStrip)

Панелі інструментів забезпечують альтернативний спосіб активізації відповідних пунктів меню. Наприклад, при клацанні на кнопці "Зберегти", результат буде тим же, що і при виборі "Файл> Зберегти" з меню.

Елемент управління ToolStrip надає більшість гнучких сучасних функцій панелей інструментів, присутній у складних додатках, таких, як Microsoft Office. ToolStrip підтримує не тільки кнопки із зображеннями, але і поля редагування тексту, поля із списками і випадаючі меню. Можна розмістити у вікні декілька таких елементів управління і дозволити користувачам переміщати їх і прикріплювати до будь-якого краю вікна, або переміщати з однієї панелі ToolStrip *на* іншу.

Подібно до інших елементів управління Windows Forms, ToolStrip підтримує вбудований редактор.

За умовчанням елемент ToolStrip розташовується у верхній частині клієнтської області форми.

Стандартна панель інструментів містить наступні кнопки – New, Open, Save, Print, Cut, Copy, Paste, Help.

Редактор колекції елементів елементу ToolStrip для стандартного набору кнопок панелі інструментів (викликається клацанням по кнопці Edit Items. вбудованого редактора елементу ToolStrip) дозволяє редагувати властивості елементів панелі інструментів.

Тема 8. Використання графічних можливостей технології Windows Forms

8.1. Використання графічних можливостей Windows Forms

Графічні інструменти Windows об'єднані під однією назвою – GDI (Graphic Device Interface – інтерфейс графічних пристроїв). GDI – це підсистема Windows, призначена для виведення графічних зображень на екран і на принтер.

GDI+ – це новий набір програмних інтерфейсів, що використовуэться в .NET. *GDI+* є керованою альтернативою *Win32 GDI*.

У .NET передбачена безліч просторів імен, призначених для висновку двовимірних графічних зображень. Крім очікуваних стандартних типів (наприклад, для роботи з кольором, з шрифтами, з олівцем і щіткою (пензлем), із зображеннями) в цих просторах передбачені типи для виконання досить витончених операцій, таких, як геометричні перетворення, згладжування нерівностей, підготовка палітри, підтримка виведення на принтер і багато інших.

Простори імен Windows Forms для роботи з графікою наведені в табл. 8.1. – 8.2.

Таблиця 8.1

Простори імен *GDI+*

Простір імен	Опис
System.Drawing	Основний простір імен GDI+, що визначає безліч типів для основних операцій візуалізації (шрифти, пера, пензлі і т. д.), а також клас Graphics для підтримки Windows-графіки
System.Drawing.Drawing2D	Пропонує типи для складнішої двовимірної і векторної графіки – градієнтні пензлі, стилі кінців ліній для пер, геометричні перетворення і т. д.
System.Drawing.Imaging	Пропонує типи, що забезпечують обробку графічних зображень, – зміна палітри, витягання метаданих зображення, робота з метафайлами і т. д.
System.Drawing.Printing	Пропонує типи, що забезпечують відображення графіки на друкарській сторінці, безпосередню взаємодію з принтером і визначення повного формату завдання друку
System.Drawing.Text	Дозволяє працювати з колекціями системних шрифтів

Таблиця 8.2

Основні типи простору імен System.Drawing

Тип	Опис
1	2
Bitmap	Інкапсулює дані зображення і визначає набір методів для виконання різних операцій з цим зображенням
Brush, Brushes, SystemBrushes, HatchBrush, LinearGradientBrush, SolidBrush, TextureBrush	Об'єкти Brush (пензлі) використовуються для заповнення внутрішнього простору графічних форм (прямокутників, еліпсів або багатокутників). Тип Brush – це абстрактний базовий клас, решта типів є похідними від Brush і визначає різні набори можливостей. Додаткові типи Brush визначені в просторі імен System. Drawing. Drawing2D
Color	Структура Color визначає набір статичних полів, які можуть бути використані для настройки кольору
Font, FontFamily	Тип Font інкапсулює характеристики шрифту (ім'я, розмір, зображення і т.п.). FontFamily представляє набір шрифтів, які відносяться до одного сімейства, але мають деякі невеликі відмінності
Graphics	Цей найважливіший клас визначає набір методів для виведення тексту, зображень і геометричних фігур. Методи малювання класу вимагають об'єкт Pen (перо зображає контур фігури) або Brush (пензель створює залиті фігури) для візуалізації заданої фігури. Можна вважати цей тип еквівалентом типу Hdc в Win32

Закінчення табл. 8.2

1	2
Icon, Systemlcons	Ці класи призначені для роботи з призначеними для користувача і системними піктограмами
Image, ImageAnimator	Image – це абстрактний базовий клас, який забезпечує можливості типів Bitmap, Icon і Cursor. ImageAnimator дозволяє проводити показ зображень (типів, похідних від Image) через вказані вами інтервали часу
Pen, Pens,SystemPens	Pen (перо) – це клас, за допомогою якого можна малювати прямі і криві лінії. У класі Pen визначений набір статичних властивостей, за допомогою яких можна отримати об'єкт Pen із заданими властивостями (наприклад, зі встановленим кольором)
Point, POINTF	Ці структури забезпечують роботу з координатами точки. Point працює із значеннями типу int, а POINTF — із значеннями типу float
Rectangle, RECTANGLEF	Ці структури призначені для роботи з прямокутними областями (int/float)
Size, SIZEF	Ці структури забезпечують роботу з розмірами (висотою і шириною). Size використовує значення типу int, а SIZEF – типу float

За умовчанням верхній лівий кут компоненту *GDI+ (*наприклад, *Form* або *Panel*) має координати (0, 0). Координата *x* – це відстань по горизонталі (управо) від верхнього лівого кута. Координата *y* – це відстань по вертикалі (вниз) від верхнього лівого кута. Координати вимірюються в пікселях, що є найменшими одиницями дозволу монітора комп'ютера (рис. 8.1).

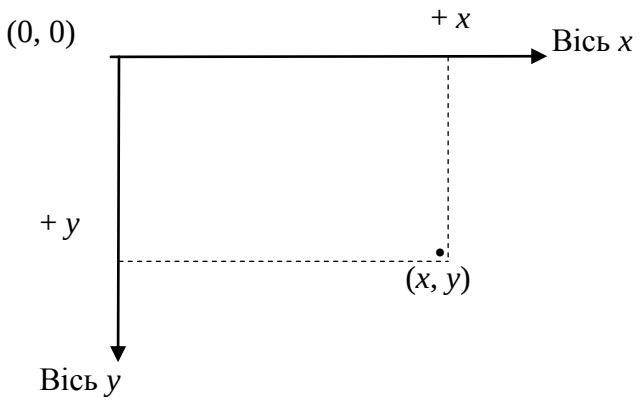

Рис. 8.1. **Система координат *GDI+***

Можливості класу Graphics

У класі Graphics інкапсульовані поверхні малювання GDI+. Є три основних типи поверхонь малювання:

Вікна і управляючі елементи на екрані.

Сторінки, що посилаються на принтер.

Растрові зображення в пам'яті.

У класі Graphics передбачені функції, які дозволяють малювати на будь-якій з цих поверхонь. Цей клас дозволяє також малювати дуги, криві, криві Безьє (Bezier), еліпси, малюнки, прямі, прямокутники і текст.

Клас System.Drawing.Graphics – це "вхід" у функціональні можливості GDI+. Всі можливості виведення зображень в GDI+ зосереджені саме в цьому класі. Можна вважати цей клас якимсь віртуальним пристроєм, на який проводиться виведення графіки.

За допомогою властивостей і методів класу Graphics можна малювати на поверхні видимих об'єктів, які включають цей клас і, відповідно, мають властивість Graphics. Наприклад, властивість Graphics мають такі об'єкти, як форма (Form), напис (Label), кнопка (Button).

Клас Graphics має велике число властивостей і методів, які дозволяють переміщатися по елементу управління, малювати графічні примітиви, копіювати зображення і їх окремі області, а також виводити текстову й іншу графічну інформацію.

Клас Graphics забезпечує:

завантаження і зберігання графічних зображень;

створення нових і зміна зображень, що зберігаються, за допомогою пера, пензля і шрифту;

малювання і зафарбовування різних фігур, ліній і текстів;

комбінування різних зображень.

Найбільш часто використовувані методи цього класу представлені в табл. 8.3.

Таблиця 8.3

Деякі методи класу Graphics

Метод	Призначення
FromHdc(), FromHwnd(), Fromlmage()	Статичні методи, що забезпечують можливість отримання об'єкта Graphics з елементу управління або зображення
Clear()	Заповнює об'єкт Graphics вибраним користувачем кольором, видаляючи його попередній вміст
DrawArc(), DrawBezier(), DrawBeziers(), DrawCurve(), DrawEllipse(), DrawIcon(), DrawLines(), DrawLine(), DrawPie(), DrawPath(), DrawRectange(), DrawRectangles(), DrawString()	Ці методи (як і багато інших) призначені для виведення зображень і геометричних фігур
FillElllpse(), FillPath(), FillPie, FillPolygon(), FillRectangle()	Ці методи призначені для заповнення внутрішніх областей графічних об'єктів

Як тільки потрібне оновлення вікна (унаслідок його перекриття або часткового псування), для нього генерується подія Paint (подія Paint успадкована від класу Control). Подія Paint наступає, коли приходить повідомлення Windows про необхідність перемальовувати зіпсоване зображення. У обробнику цієї події і потрібно перемальовувати зображення. Проте, по обробникові події форми Paint перемальовувалося зображення всієї форми, а це може бути трудомістка операція.

Способи перемальовування:

1. Для малювання написати процедуру (метод форми), яку викликати в обробнику події Paint форми.

2. Перемальовування істотно прискориться, якщо перемальовувати тільки зіпсовану частину елементу управління.

До аргументів події Paint відноситься об'єкт PaintEventArgs, з якого можна отримати об'єкт Graphics для управління. Об'єкт Graphics необхідно отримувати для кожного звернення до методу Paint, тому що властивості графічного контексту, що представляються графічним об'єктом, можуть мінятися.

Об'єкт Graphics для деякого вікна можна одержати двома шляхами. Перший полягає в перевизначенні події OnPaint() – віртуального методу, який клас Forms успадковує від класу control. В цьому випадку об'єкт Graphics одержуємо з PaintEventArgs, який передається разом з подією:

```
protected override void OnPaint(PaintEventArgs e)
{
 Graphics g = e.Graphics;
 // Малюємо тут
}
```

У деяких ситуаціях потрібно виконувати малювання у вікні безпосередньо, не чекаючи настання події OnPaint(). Це може виявитися актуальним у тому випадку, якщо створюємо код, призначений для вибору у вікні яких-небудь графічних об'єктів (аналогічний вибору ікон в Windows Explorer), або переносимо якийсь об'єкт за допомогою миші. В цьому випадку доступ до об'єкта Graphics можна дістати, звертаючись до методу CreateGraphics() даної форми, який є ще одним методом, успадкованим класом Forms від класу Control:

```
protected void Forml_Click(object sender, System.EventArgs e)
{
Graphics g = this.CreateGraphics();
// Малюємо тут
g.Dispose:();  // Це важливий момент
}
```

Тепер для зображення фігур і рядків на формі доступна змінна g.

При розкритті, закритті вікна або зміні його розмірів автоматично генерується подія Paint для форми. Так само при відображенні будь-якого елементу управління (наприклад, TextBox або Button) програми генерується подія Paint для цього елементу управління.

Подію Paint можна згенерувати примусово за допомогою виклику методу Invalidate, також успадкованого від класу Control. Даний метод оновлює клієнтську область елементу управління і неявно перемальовував всі графічні компоненти. У *.NET* містяться декілька перевантажених методів Invalidate, що дають можливість оновлення частин клієнтської області. Виклик методу Invalidate з параметром Rectangle оновлює тільки область, позначену прямокутником, що підвищує продуктивність програми.

Управління кольором – структура Color

Структура Color має поля і визначає методи і константи для маніпулювання з кольором.

Для завдання кольору використовується *ARGB-модель (A* – альфа-компонент прозорості, *R* – червоний компонент (кількість червоного кольору в підсумковому кольорі), *G* – зелений компонент, *B* – синій компонент). Всі чотири компоненти *ARGB-моделі* є байтами, що представляють цілі числа від 0 до 255.

Альфа-значення визначає непрозорість кольору. Якщо *A=0*, то колір прозорий, якщо *A=255* – колір насичений. Значення *A* між 0 і 255 дають зважений ефект поєднання *RGB-значення* кольору із значенням будь-якого фонового кольору, роблячи колір напівпрозорим.

Існує можливість вибору з порядка 17 млн кольорів. Якщо монітор не здатний відобразити всі ці кольори, то буде вибраний колір, найближчий до заданого.

Для створення кольору, заснованого на червоному, зеленому, синьому і альфа-каналі, можна використовувати статичний метод *Color.FromArgb*

Елемент управління "діалог вибору кольору" (ColorDialog)

Елемент управління ColorDialog використовується для:

вибору кольору з доступної палітри в 48 кольорів;

створення спеціальних кольорів і використання їх.

Метод ShowDialog() викликає модальне діалогове вікно ColorDialog, після вибір кольору і натиснення кнопки ОК додаток приймає вибір кольору через властивість Color елементу ColorDialog. Метод ShowDialog() повертає результат типу DialogResult, який може бути оброблений.

При натисненні кнопки "OK" метод ShowDialog() повертає значення DialogResult=OK, привласнює властивості Color значення вибраного кольору і закриває модальне вікно.

При натисненні кнопки "Відміна" метод ShowDialog() повертає значення DialogResult=Cancel, привласнює властивості Color значення кольору за умовчанням (колір за умовчанням встановлюється у вікні Properties і спочатку рівний Black) і закриває модальне вікно.

Для створення спеціального кольору необхідно клацнути по кнопці "Визначити колір". При цьому вікно збільшиться і відобразяться різні

відтінки кольорів. Можна вибрати потрібний відтінок і його насиченість. Після закінчення настройок необхідно натиснути на кнопку "Add to Custom Colors", після чого спеціальний колір додається в розділ спеціальних (призначених для користувача) кольорів діалогового вікна. При натисненні кнопки "OK" встановлюється властивість Color діалогового вікна.

Робота з класом Pen – робота з пером

Звичайне застосування об'єктів Pen (перо) полягає в малюванні ліній. Як правило, об'єкт Pen використовується не сам по собі – він передається як параметр численним методам виводу, визначеним у класі Graphics. Як правило, назви всіх цих методів, використовуючих *Pen*, починаються з Draw (схожі за функціональністю методи, що приймають об'єкт Brush (пензель), починаються на Fill).

У класі Pen передбачено декілька переобтяжених конструкторів, за допомогою яких можна задати початковий колір і товщину пера (об'єкт Pen можна також створити на основі існуючого об'єкта Brush). Велика частина можливостей *Pen* визначається властивостями цього класу.

Окрім класу Pen в GDI+ також можна використовувати колекцію певних пер (колекція Pens). За допомогою статичних властивостей колекції Pens можемо миттєво отримати вже готове перо, без необхідності створювати його вручну. Проте всі типи Pen, які створюються за допомогою колекції Pens, мають одну і ту ж однакову ширину, рівну 1.

Стиль об'єкта Pen визначається значенням з перерахування DashStyle (визначеного в System.Drawlng.Drawing2D). Значення з перерахування DashStyle представлені в табл. 8.4.

Таблиця 8.4

Значення перерахування DashStyle

Значення	Перерахування
Dash	Штрихова лінія
DashDot	Штрихпунктирна лінія: штрих – крапка – штрих
DashDotDot	Штрихпунктирна лінія: штрих – крапка – крапка – штрих
Dot	Пунктир з одних крапок
Solid	Суцільна лінія
Custom	Призначений для користувача стиль

Крім використання готових стилів пунктирних ліній з перерахування DashStyle, можемо також визначити свій власний стиль. Для цього використовується властивість DashPattern класу Pen.

Робота з класом Brush – робота з пензлем

Усі класи, похідні від абстрактного класу Brush, визначають об'єкти, що задають колір внутрішньої частини графічних фігур (наприклад, конструктор об'єкта Color – колір, яким буде виконано зображення). У більшості методів Fill пензлі (Brush) заповнюють простір кольором, узором (текстурою) або графічним зображенням. Сам клас Brush є абстрактним, і створювати об'єкти цього класу не можемо. У табл. 8.5 представлені похідні від класу Brush пензлі і їх функції.

Таблиця 8.5

Класи, похідні від класу Brush
(визначені в просторі імен System.Drawing.Drawing2D)

Пензель	Функція
SolidBrush	Заповнення області одним кольором. Визначається об'єктом Color
HatchBrush	Використання прямокутного пензлю для заповнення області штрихуванням. Штрихування визначається членом перерахування HatchStyle, кольором переднього плану (яким зображений узор) і фоновим кольором
LinearGradientBrush	Заповнення області поступовим змішуванням одного кольору з іншим. Лінійні градієнти визначаються уздовж лінії. Їх можна задати двома кольорами, кутом градієнта або шириною прямокутника або двома крапками
TextureBrush	Заповнення області повторенням заданого зображення (Image) уздовж поверхні

Крім того, створювати об'єкти пензлів (вибравши із заздалегідь готового набору) можна за допомогою типів-колекцій Brushes і System.Brushes, також визначених у просторі імен System.Drawing. Створення об'єктів з цих типів-колекцій проводиться за допомогою їх статичних властивостей. Далі можемо передати створений об'єкт пензля як параметр відповідному методу об'єкта Graphics.

Штрихові пензлі

Складніше "закрашення" можна провести за допомогою похідного від *Brush* класу HatchBrush, визначеного в просторі імен System.Drawing.-Drawing2D. Цей тип дозволяє закрасити внутрішню область об'єкта за допомогою великої кількості шаблонів штрихування (55 шаблонів штрихування), визначених у перерахуванні HatchStyle. У табл. 8.6 наведені деякі з цих шаблонів.

Таблиця 8.6

Значення перерахування HatchStyle (стилі штрихування)

Значення	Опис
BackwardDiagonal	Діагональне штрихування з нахилом управо
Cross	"Хрестоподібне" штрихування, що складається з пересічних вертикальних і горизонтальних ліній
DiagonalCross	Ще один різновид "хрестоподібного" штрихування, що складається з пересічних діагональних ліній
ForwardDiagonal	Діагональне штрихування з нахилом вліво
Hollow	"Порожній" пензель, який нічого не малює
Horizontal	Горизонтальне штрихування
Pattern	Штрихування, яке створюється на основі вказаного користувачем растрового зображення
Solid	Звичайний "щільний" пензель без всякого штрихування (аналогічно звичайному типу SolidBrush)
Vertical	Вертикальне штрихування

При створенні об'єкта *HatchBrush* обов'язково потрібно буде вказати два кольори: колір "переднього плану" (колір штрихування) і колір фону (що штрихуємо).

Градієнтні пензлі

Градієнтний пензель представлений типом LinearGradientBrush. Основне призначення цього типу – забезпечити плавне змішення двох кольорів для отримання градієнтного переходу. Єдина особливість використання цього типу полягає в тому, що потрібно вказати напрям колірного переходу за допомогою значень з перерахування LinearGradientMode (табл. 8.7).

Таблиця 8.7

Значення перерахування LinearGradientMode

Значення	Опис
BackwardDiagonal	З верхнього правого кута в нижній лівий
ForwardDiagonal	З верхнього лівого кута в нижній правий
Horizontal	Зліва направо
Vertical	Зверху вниз

Пензлі текстур

Пензлі текстур представлені типом TextureBrush. Ці пензлі "зафарбовують" відведену для цього область текстурою – тобто вказаним програмістом растровим зображенням. Для TextureBrush використовується посилання на об'єкт Image, що представляє зображення, – конструктор TextureBrush вимагає посилання на об'єкт Image. Зображення може бути зовнішнім файлом (у форматі bmp, gif або jpg) або збіркою *.NET*.

Робота зі шрифтами

Основний клас, який використовується для роботи зі шрифтами в GDI+, – це клас System.Drawing.Font. Об'єкти цього класу представляють конкретні шрифти, встановлені на комп'ютері. У цьому класі передбачена безліч перевантажених конструкторів.

Зазвичай конструктор шрифту вимагає:

назву гарнітури шрифту (font name);

розмір гарнітури шрифту (font size);

стиль гарнітури шрифту (font style з перерахування FontStyle)

Стилі гарнітури можна комбінувати за допомогою операції . Змінити властивості об'єкта Font не можна, для використання іншої гарнітури необхідно створити новий об'єкт Font.

Приклад. Два найчастіше використовуваних конструктора мають вигляд:

```
// Створюємо об'єкт Font, указуючи ім'я шрифту і його розмір
Font  f1 = new Font("Times New Roman", 12);

// Створюємо об'єкт Font, указуючи ім'я, розмір і стиль
```

Font f2 = new Font("Courier New", 16, FontStyle.Bold FontStyle.-Underline);

При створенні f2 використовуються стилі з перерахування FontStyle. Можна задавати відразу декілька стилів одночасно. Значення з перерахування FontStyle представлені в табл. 8.8.

Таблиця 8.8

Доступні стилі з перерахування FontStyle

Елемент перерахування FontStyle	**Стиль**
Bold	Напівжирний
Italic	Курсив
Regular	Звичайний текст
Strikeout	Закреслений
Underline	Підкреслений

Після того, як настроєні необхідні параметри об'єкта Font, наступне завдання – передати їх методу Graphics.DrawString(). Не дивлячись на те, що цей метод багато разів перевантажений, як правило, доводиться указувати стандартний набір інформації:

текстовий рядок, який виводитиметься;

шрифт;

пензель (товщина лінії);

область виводу.

Клас FontDialog

Клас FontDialog призначений для виведення діалогового вікна, в якому користувач зможе вибрати потрібний йому шрифт і встановити необхідні параметри шрифту.

Щоб викликати на екран вікно FontDialog, необхідно так само скористатися методом ShowDialog(). Потім для отримання інформації про те, що користувач вибрав, слід скористатися властивістю Font діалогу.

Виведення зображень (клас Image*)*

Для виведення зображень використовується клас Image простору імен System.Drawing. Тип Image визначає безліч властивостей і методів,

які можна використовувати для настройки параметрів зображення, що виводиться.

Наприклад, за допомогою властивостей Width, Height і Size можна отримати або встановити розміри зображення. Крім того, в просторі імен System.Drawing.Imaging визначена безліч типів для проведення складних перетворень зображень.

Клас Image є абстрактним, і створювати об'єкти цього класу не можна. Зазвичай оголошені змінні Image привласнюються об'єктам класу Bitmap. Крім того, можна створювати об'єкти класу Bitmap безпосередньо і використовувати їх замість об'єктів класу Image.

Виведення отриманих зображень проводиться за допомогою спеціального методу класу Graphics, який називається, – DrawImage(). Цей метод багато разів переобтяжений, тому в нашому розпорядженні безліч варіантів того, як помістити зображення в потрібне нам місце на формі.

Клас Bitmap дозволяє виводити зображення, які зберігаються у файлах найрізноманітнійшого формату, – *bmp*, *jpg*, *gif*, *ico*.

Приклад. Розробити Windows-додаток, який будує графік функції y=sin(x).

```
private void Form1_Paint(object sender, PaintEventArgs e)
      {
         Graphics g = this.CreateGraphics();
        Pen p = new Pen(Color.Green);
        for (int i = 0; i < this.ClientSize.Width; i+=30)
        g.DrawLine(p,i,0,i,this.ClientSize.Height);
       for (int i = 0; i < this.ClientSize.Height; i+=30)
        g.DrawLine(p,0,i,this.ClientSize.Width,i);
     Pen p1 = new Pen(Color.Red,3);
     double angle1 = 0,angle2;
     double y1, y2;
     for (int i = 1; i < this.Width; i++)
     {
      angle2 = i * 1.0 / 180 * Math.PI;
      y1 = this.ClientRectangle.Height / 2 * (1 - Math.Sin(angle1));
      y2 = this.ClientRectangle.Height / 2 * (1 - Math.Sin(angle2));
      g.DrawLine(p1, i - 1, (int) y1, i, (int) y2);
      angle1 = angle2;
```

```
    }
}

private void Form1_Resize(object sender, EventArgs e)
{
  this.Invalidate();
}
```

Тема 9. Реалізація багатопотоковості у .NET

9.1. Реалізація багатопотоковості у .NET

C # підтримує паралельне виконання коду через багатопотоковість. Потік – це незалежний шлях виконання, здатний виконуватися одночасно з іншими потоками.

Програма на C# запускається як єдиний потік, автоматино створюваний CLR та операційною системою ("головний" потік), і стає багатопотоковою за допомогою створення додаткових потоків.

З програми можна зробити мережний виклик, який часто займає деякий час. Блокувати користувальницький інтерфейс і просто дозволити користувачеві очікувати повернення відповіді з сервера – не дуже гарний варіант. Користувач міг би виконати в цей час якісь інші дії або навіть відмінити відправлений на сервер запит. Допомогти тут може використання багатопотоковості.

Для застосування багатопотоковості існує кілька причин. Одна з них – не примушувати користувача чекати. Для всіх видів активності, що вимагають очікування, наприклад, через необхідність отримання доступу до файла, бази даних або мережі, може запускатися новий потік, який дозволяє виконувати в цей час інші завдання. Навіть якщо мова йде про завдання, насичених у плані обробки, то і тут багатопотоковість може допомогти. Численні потоки одного процесу запросто можуть одночасно виконуватися на різних ЦП або, що стало можливим в наші дні, навіть у різних ядрах одного багатоядерного ЦП.

Зрозуміло, необхідно знати і про деякі наслідки виконання численних потоків. Оскільки вони виконуються в один і той же час, у разі отримання ними доступу до одних і тих же даних можуть виникати проблеми. Щоб цього не відбувалося, потрібно обов'язково реалізовувати механізми синхронізації.

У даній роботі наводиться інформація, якою необхідно володіти при програмуванні додатків з безліччю потоків. Зокрема, тут розглядаються наступні питання:

- огляд багатопотоковості;
- полегшений варіант багатопотоковості з використанням делегатів;
- клас Thread;
- пули потоків;
- наслідки багатопотоковості.

Не слід плутати багатопотоковість та багатозадачність або навіть багатопроцесорність. Багатопотоковість – це не магічний засіб, що дозволяє збільшити швидкість вашої роботи, це скоріше засіб, який збільшує зручність і ефективність роботи з вашим додатком. Ось простий приклад:

```
using System;
using System.Threading;

class ThreadTest
{
  static void Main()
  {
    Thread t = new Thread(WriteY);
    t.Start();          // Виконати WriteY у новому потоці
    while (true)
      Console.Write("x"); // Весь час друкувати 'x'
  }

  static void WriteY()
  {
    while (true)
      Console.Write("y"); // Весь час друкувати 'y'
  }
}
```

У головному потоці створюється новий потік t, виконуючий метод, який безперервно друкує символ 'y'. Одночасно головний потік безперервно друкує символ 'x'.

CLR призначає кожному потоку свій стек, так що локальні змінні зберігаються окремо. У наступному прикладі визначаємо метод з локальної змінної, а потім виконуємо його одночасно в головному та у знову створеному потоках:

```
static void Main()
{
  new Thread(Go).Start();      // Виконати Go() у новому потоці
  Go();                        // Виконати Go() у головному потоці
}

static void Go()
{
  // Визначаємо та використовуємо локальну змінну 'cycles'
  for (int cycles = 0; cycles < 5; cycles++)
    Console.Write('?');
}
```

Результат:
??????????

Окремий примірник змінної cycles створюється в стек кожного потоку, так що виводиться, як і очікувалося, десять знаків '?'.

Потік (thread) – це незалежна послідовність інструкцій у програмі. До цього моменту всі програми C #, які були розглянуті, мали тільки одну точку входу – метод Main (). Виконання починалося з першого оператора методу Main() і тривало до виходу з нього.

Така структура дуже добре підходить для програм, які виконують лише одну ідентифіковану послідовність завдань, але найчастіше програмами необхідно виконувати більше однієї задачі одночасно. Потоки важливі як для клієнтських, так і для серверних додатків. При введенні в редакторі Visual Studio коду на C # у вікні Dynamic Help (Динамічна довідка) негайно відображається перелік підходящих для вводимого коду тем. Фоновий потік виконує пошук по довідковій системі. Те ж саме робить і засіб перевірки орфографії в Microsoft Word. Один потік очікує вводу від користувача, в той час як інший у фоновому режимі виконує якийсь аналіз. Третій потік може зберігати записані дані в який-

небудь тимчасовий файл, а четвертий – завантажувати якісь додаткові дані з Internet.

У програмі, яка виконується на сервері, один потік завжди чекає запиту від клієнта і називається потоком-слухачем (listener thread). Як тільки такий запит надходить, він відразу переадресовується окремому робочому потоку (worker thread), який продовжує взаємодію з клієнтом. Потік-слухач при цьому негайно повертається на місце для отримання запиту від наступного клієнта.

Операційна система планує потоки. У потоку є пріоритет, програмний лічильник для розміщення програми, де він насправді обробляється, і стек для зберігання його локальних змінних. У кожного потоку є свій власний стек, однак пам'ять для програмного коду та купа розподіляються між усіма потоками одного процесу. Це робить взаємодію між потоками одного процесу швидким, тому що всі потоки процесу звертаються до однієї і тієї ж віртуальної пам'яті. Однак це також і ускладнює справу, оскільки численні потоки можуть змінювати одну і ту ж комірку пам'яті.

Процес керує ресурсами, до числа яких належать віртуальна пам'ять і дескриптори Windows, і містить як мінімум один потік. Для виконання програми наявність потоку є обов'язковим.

У .NET керований потік визначається класом Thread. Керований потік зовсім необов'язково відображається на один потік операційної системи. Таке можливо, але відображення керованих потоків на фізичні потоки операційної системи є обов'язком хоста виконуючою середовища .NET. У цьому поведінка заснованого на SQL Server 2005 хоста виконуючого середовища дуже відрізняється від поведінки хоста виконуючого середовища, який використовується Windows-додатками. Інформацію про власні потоки можна отримувати за допомогою класу ProcessThread, але у випадку керованих додатків зазвичай для цього краще застосовувати керовані потоки.

Асинхронні делегати

Найбільш простим способом створення потоку є визначення делегата і його виклик асинхронним чином. Клас Delegate підтримує асинхронний виклик методів. Він просто створює в тлі потік, який і виконує зазначену задачу:

Щоб продемонструвати асинхронні можливості делегатів, візьмемо метод, виконання якого займає певний час. Наприклад, через метод Thread. Sleep () показаному нижче методу TakesAWhile для завершення необхідно, щоб із другим аргументом було передано хоча б декілька мілісекунд.

```
static int TakesAWhile (int data, int ms)
{
        Console.WriteLine("TakesAWhile запущений");
        Thread.Sleep(ms);
        Console.WriteLine("TakesAWhile завершено");
        return ++data;
}
```

Щоб викликати цей метод з делегату, потрібно визначити делегат з тим же параметром і повертаними типами, як показано нижче на прикладі делегата TakesAWhileDelegate:

```
public delegate int TakesAWhileDelegate (int data, int ms);
```

Далі вже можна використовувати інші прийоми, викликаючи делегат асинхронним чином і повертаючи результати.

Опитування

Одним із прийомів є опитування та перевірка, чи завершив делегат свою роботу. Створений авторами вище клас delegate надає метод BeginInvoke (), в якому можна передати вхідні параметри з типом делегати. Метод BeginInvoke () завжди отримує два додаткових параметри типу AsyncCallback і object. Зараз важливо інше – повертається тип BeginInvoke (), яким є IAsyncResult. За допомогою IAsyncResult можна отримати інформацію про делегатів і перевірити, завершив він уже свою роботу, для чого необхідно ще додатково використовувати і властивість IsCompleted. Основний потік програми буде продовжувати виконувати цикл while до тих пір, поки делегат не завершить свою роботу.

```
static void Main ()
{
        // синхронний
        // TakesAWhile (1, 3000);
```

```
// асинхронний
TakesAWhileDelegate dl = TakesAWhile;
IAsyncResult ar = dl.BeginInvoke (1, 3000, null, null);
while (!ar.IsCompleted)
{
  // виконання в основному потоці ще яких-небудь операцій
  Console.Write(".");
  Thread.Sleep(50);
}
int result = dl.EndInvoke(ar);
Console.WriteLine("result:   {0}",   result); }
```

Запустивши цей додаток, можна побачити, що основний потік і потік делегат виконуються паралельно, і що після завершення роботи потоком делегата основний потік припиняє прохід по циклу:

```
TakesAWhile запущений
TakesAWhile завершено
result:  2
```

Замість того щоб перевіряти, чи завершив делегат свою роботу, можна також просто викликати метод EndInvoke () типу делегати після завершення роботи, яка може бути виконана основним потоком. Метод EndInvoke () сам по собі очікує, коли делегат завершить свою роботу. Щоб завершити виконання основного потоку, не чекаючи того, коли свою роботу завершить делегат, потік делегата буде зупинений.

Дескриптор очікування

Іншим способом очікування результату від асинхронного делегата є використання дескриптора очікування (wait handle), який асоціюється з IAsyncResult. Отримати доступ до цього дескриптора очікування можна за допомогою властивості AsyncWaitHandle. Ця властивість повертає об'єкт типу WaitHandle, в якому як раз і можна чекати, поки потік делегата завершить свою роботу. Метод WaitOne () перший необов'язковий параметр приймає значення тайм-ауту, в якому можна вказати максимальний період очікування: тут ставимо 50 мілісекунд. У разі виникнення тайм-ауту метод WaitOne () повертається зі значенням false, і прохід по циклу while триває. Якщо під час очікування до тайм-

ауту справа не доходить, цикл while завершується з допомогою break, і результат виходить за допомогою методу EndInvoke ().

```
static void Main()
{
  TakesAWhileDelegate dl = TakesAWhile;
  lAsyncResult ar = dl.BeginInvoke(1, 3000, null, null);
  while (true)
  {
      Console.Write (".");
      if (ar.AsyncWaitHandle.WaitOne(50, false))
      {
        Console.WriteLine("Зараз можна повернути result") ;
        break;
      }
  }
  int result = dl.EndInvoke(ar);
  Console.WriteLine("result: {0}",   result);
}
```

Клас Thread

За допомогою класу Thread можна створювати й управляти потоками. Наведений тут код є дуже простим прикладом створення та запуску нового потоку. Конструктор класу Thread приймає параметр делегата типу ThreadStart і Paгагае terizedThreadStart. Делегат ThreadStart визначає метод з типом void і без будь-яких аргументів. Після створення об'єкта Thread потік можна запустити з допомогою методу Start ().

```
using System;
using System.Threading;
namespace Wrox.ProCSharp.Threading
{
  class Program
  {
    static void Main()
    {
      Thread tl = new Thread(ThreadMain);
      tl. Start ();
```

```
        Console.WriteLine("Головний потік.");
    }
    static void ThreadMain()
    {
      Console.WriteLine("Виконання в потоці.");
  }
}
```

Запустивши цей додаток, отримуємо результат двох потоків:

Головний потік.

Виконання у потоці.

Сказати точно, висновок якого потоку буде перший, не можна. Виконання потоків планується операційною системою, а це означає, що кожен раз черговість потоків може бути різною.

Як анонімний метод можна використовувати з асинхронним делегатом, вже показувалося. Проте його також можна використовувати і з класом Thread, передаючи реалізацію методу потоку аргументу конструктора Thread:

```
using System;
using System.Threading;

namespace Wrox.ProCSharp.Threading
{
  class Program
  {
    static void Main()
    {
        Thread tl = new Thread (
        delegate()
        { Console.WriteLine("Виконання в потоці.");
        });
    tl.Start ();
    Console.WriteLine("Головний потік.");
    }
  }
}
```

Якщо змінна, що посилається на потік і дозволяє керувати ним після його створення, не потрібна, те ж саме можна написати і коротше, а саме: створити новий об'єкт Thread за допомогою конструктора, передати конструктору анонімний метод і за допомогою повернення об'єкта Thread викликати метод Start ( ) прямо:

```
using System.Threading;
namespace Wrox.ProCSharp.Threading
{
  class Program
  {
    static void Main ()
    {
        new Thread (
          delegate()
          {
            Console.WriteLine("Виконання в потоці.");
          }) .Start() ;
        Console.WriteLine("Головний потік.");
    }
  }
}
```

Однак є кілька вагомих причин залишити змінну, що посилаються на об'єкт Thread. Одна з них пов'язана з більш зручним керуванням потоками: можна призначити потоку ім'я, встановивши перед його запуском властивість Name. Щоб отримати ім'я поточного потоку, можна використовувати статичний властивість Thread.CurrentThread для того, щоб дістатися до примірника Thread поточного потоку, і звернутися до властивості Name для доступу з читання. У потоку також є керований ідентифікатор потоку, який можна вважати за допомогою властивості ManagedThreadId.

```
static void Main ()
{
  Thread tl = new Thread(ThreadMain);
  tl. Name = "MyNewThreadl";
  tl.Start ();
```

```
  Console.WriteLine("Главный поток.");
}
static void ThreadMain()
{
  Console.WriteLine("Виконання в потоці {0}, id: {1}.",
  Thread.CurrentThread.Name,Thread.CurrentThread.ManagedThreadId);
```

У такому разі у висновку програми також буде відображатися ім'я та ідентифікатор потоку:
Головний потік.
Виконання у потоці MyNewThreadI, id: 3.

Призначення потоку імені дуже допомагає при налагодженні потоків, дозволяючи включати під час сеансу налагодження в Visual Studio панель інструментів Debug Location (Розміщення налагодження), яка показує ім'я потоку.

Передача даних потокам

За необхідності передати потоку якісь дані, це можна зробити двома способами: або скористатися конструктором Thread з делегатом Parameterized ThreadStart, або створити спеціальний клас і визначити метод потоку як метод екземпляру для того, щоб мати можливість ініціалізувати дані примірника перед запуском потоку.

Для передачі даних потоку необхідний будь-який клас чи структура, що містить дані. Тут використовується структура Data, що містить рядок, але ви можете передавати будь-який об'єкт, який захочете.

```
public struct Data
{
  public string Message;
}
```

Якщо використовується делегат ParameterizedThreadStart, точка входу потоку повинна мати параметр об'єкта типу і повертається тип void. Об'єкт може бути приведений до потрібного типу, а на консоль може бути виведено відповідне повідомлення:

```
static void ThreadMainWithParameters(object o)
{
```

```
  Data d = (Data)o;
  Console.WriteLine("Виконання в потоці, отримано {0}",d.Message);
}
```

За допомогою конструктора класу Thread можна призначити нову точку входу ThreadMainWithParameters і викликати метод Start (),що передає змінну d:

```
static void Main()
{
  Data d = new Data();
  d.Message = "Info";
  Thread t2 = new Thread(ThreadMainWithParameters);
  t2.Start(d);
}
```

Інший спосіб передачі даних нового потоку передбачає визначення класу (тут це клас MyThread) з необхідними полями, а також визначення головного методу потоку як метод примірника цього класу:

```
public class MyThread
{
  private string data;
  public MyThread(string data)
  {
    this.data = data;
  }
  public void ThreadMain ()
  {
    Console.WriteLine("Виконання в потоці, data: {0}", data);
  }
}
```

Таким чином, можна створити об'єкт класу My Thread та передати його і метод ThreadMain () конструктору класу Thread. Після цього потік зможе отримувати доступ до даних:

```
MyThread obj = new MyThread ("info");
Thread t3 = new Thread(obj.ThreadMain);
t3.Start ();
```

Фонові потоки

Процес програми продовжує виконуватися до тих пір, поки виконується хоча б один пріоритетний потік. Якщо виконується більше одного пріоритетного потоку, і метод Main завершується, процес додатки продовжує залишатися активним до тих пір, поки свою роботу не завершать всі пріоритетні потоки.

Потік, що створюється за допомогою класу Thread, за умовчанням є пріоритетним. Потоки, які створюються за допомогою класу ThreadPool, завжди є фоновими.

При створенні потоку з допомогою класу Thread можна вказувати шляхом встановлення відповідного значення для властивості IsBackground, яким повинен бути потік – пріоритетним або фоновим. Показаний нижче метод Main () встановлює для властивості IsBackground потоку tl значення false (яке є значенням за умовчанням). Після запуску нового потоку основний потік просто виводить на консоль повідомлення про завершення. Новий потік виводить на консоль повідомлення про запуск і повідомлення про завершення, засинаючи в проміжку між ними на 3 секунди. Ці три секунди надають для основного потоку прекрасну можливість завершити свою роботу до того, як це зробить новий потік.

```
class Program
{
  static void Main()
  {
    Thread tl = new Thread(ThreadMain);
    tl.Name = "MyNewThreadl";
    tl.IsBackground = false;
    tl.Start ();
    Console.WriteLine("Головний потік зараз завершується...");
  }
  static void ThreadMain()
  {
    Console.WriteLine("Потік {0} запущений", Thread.CurrentThread.Name);
    Thread.Sleep(3000);
```

```
    Console.WriteLine("Потік {0} завершений", Thread.CurrentThread.Name);
  }
}
```

Запустивши цей додаток, ви все одно побачите виведення на консоль повідомлення про завершення, незважаючи на те, що основний потік завершив свою роботу раніше. А все тому, що новий потік теж є пріоритетним.

Головний потік зараз завершується...
Потік MyNewThreadl запущений
Потік MyNewThreadl завершений

Якщо ви змініте значення властивості IsBackground, що використовується для запуску нового потоку, на true, відображений на консолі результат буде виглядати по-іншому. На системі автора після зміни на консолі завжди відображалося повідомлення про запуск нового потоку і ніколи – про його завершення. Може статися і так, що повідомлення про запуск теж не буде відображатися, наприклад, якщо робота нового потоку буде з будь-якої причини завершена передчасно, ще до того, як він навіть встигне запуститися.

Головний потік зараз завершується...
Потік MyNewThreadl запущений

Фонові потоки дуже зручні для виконання фонових завдань. Наприклад, після закриття програми Microsoft Word в продовженні роботи засобу перевірки орфографії немає ніякого сенсу. Тому при закритті програми потік засоби перевірки орфографії може запросто знищуватися. Однак потік, що відповідає за організацію сховища повідомлень Outlook, повинен залишатися активним до самого кінця, тобто до тих пір, поки він повністю не завершить свою роботу, навіть в тому випадку, якщо сама програма Outlook закривається.

Керування потоками

Потік створюється шляхом виклику методу Start () об'єкта Thread. Однак після виклику методу Start () новий потік все ще знаходиться не в змозі Running, а в змозі Unstarted. У стан Running потік переходить відразу ж, як тільки планувальник потоків операційної системи вибере

його для виконання. Прочитати інформацію про поточний стан потоку можна через властивість Thread.ThreadState.

За допомогою методу Thread.Sleep () потік переходить в стан WaitSleepJoin і очікує поновлення протягом проміжку часу, зазначеного у методі Sleep ().

Зупинити потік можна, викликавши метод Thread.Abort (). Коли викликається цей метод, у вказаному потоці генерується виключення. При наявності обробника, що перехоплює цей виняток, потік може виконати перед завершенням які-небудь операції з очищення. У потоку також є шанс продовжити виконання після отримання виключення ThreadAbortException у результаті виклику методу Thread.ResetAbort (). Стан потоку, що отримує запит на переривання, змінюється з AbortRequested на Aborted, якщо потік не скидає переривання.

Якщо необхідно дочекатися завершення роботи потоку, можна викликати метод Thread.Join (). Метод Thread.Join () блокує поточний потік і переводить його в стан WaitSleepJoin до тих пір, поки завершиться приєднаний потік.

В .NET 1.0 також підтримувалися методи Thread.Suspend () і Thread.Resume (), які дозволяли, відповідно, призупиняти та відновлювати роботу потоку. Однак дізнатися, що робить потік, який отримав запит Suspend, було неможливо, а він міг знаходитися в синхронізованому розділі з блокування, що легко могло призвести і до взаємоблокування. Саме тому зараз ці методи вважаються застарілими. Замість них тепер можна відправляти потоки за допомогою об'єктів синхронізації відповідний сигнал, так щоб той сам міг припиняти свою роботу. У такому випадку потік краще знає, коли йому слід переходити в стан очікування.

9.2. Синхронізація потоків

Програмування з застосуванням багатопоточності є далеко не просте завдання. При запуску безлічі потоків, які отримують доступ до одних і тих же даних, відразу ж можуть виникнути проблеми, відшукати які буде дуже важко. Щоб уникнути такої ситуації, слід уважно вивчити питання синхронізації і проблеми, які можуть виникати при використанні безлічі потоків. У наступних розділах описуються такі можливі наслідки застосування багатопотоковості, як змагання за ресурси і взаємоблокування.

Змагання за ресурси може виникнути у випадку, якщо два або більше потоків отримують доступ до одних і тих самих об'єктів, і доступ до спільно використовуваного стану не синхронізується.

Щоб продемонструвати змагання за ресурси, нижче наводиться приклад, у якому визначається клас StateObject з полем int і метод ChangeState. В реалізації Changestate мінлива стану перевіряється на предмет того, чи не містить вона значення 5. Якщо вона містить значення 5, це значення збільшується на один (інкрементується). Далі відразу ж слідує оператор Trace.Assert, який тут же засвідчується в тому, що в змінної стану тепер міститься значення 6. Здається очевидним, що після збільшення змінної, що містить значення 5, на 1, в ній повинно знаходитися значення 6. Однак це зовсім не обов'язково може бути саме так. Наприклад, якщо один потік тільки що виконав оператор if (state = = 5), планувальник може витіснити його і запустити ще один потік. Тепер цей другий потік потрапить у тіло if, і оскільки в змінній стану, як і раніше, міститься значення 5, воно буде збільшено на 1 до значення 6. Після цього знову настане черга виконання першого потоку, в результаті чого в наступному операторі значення змінної стану буде збільшено вже до 7. Ось тут якраз і виникне змагання за ресурси і з'явиться відповідне повідомлення.

```
public class StateObject
{
  private int state = 5;
  public void ChangeState(int loop)
  { if   (state == 5)
    { state++;
        Trace.Assert(state == 6,  "Race condition occurred after " +
loop + " loops") ;
// Trace.Assert(state == 6,   "Змагання за ресурси виникло після " + // loop
+ " циклів");
    }
    state = 5;
  }
}
```

Давайте упевнемося в цьому, визначивши метод потоку. Нижче показано метод RaceCondition () класу SampleThread, який як параметр

отримує StateObject. Усередині нескінченного циклу while викликається метод ChangeState (). Змінна і використовується тільки для відображення в повідомленні номеру циклу.

```
public class SampleThread
{
  public void RaceCondition(object o)
  {
    Trace.Assert(o is StateObject, "o повинна мати тип StateObject");
    StateObject state = o as StateObject;
    int i = 0;
    while (true)
    {
      state.ChangeState(i++) ;
    }
  }
}
```

У методі Main цієї програми створюється новий об'єкт StateObject, який поділяється між всіма потоками. Об'єкти Thread створюються шляхом передачі адреси RaceCondition з об'єктом типу SampleThread в конструкторі класу Thread. Далі запускається новий потік за допомогою методу Start (), що передає об'єкт state.

```
static void Main()
{
  StateObject state = new StateObject ();
  for (int i = 0; i < 20; i++)
  {
    new Thread(new SampleThread().RaceCondition).Start(state);
  }
}
```

Запустивши цю програму, ви побачите змагання за ресурси. Те, скільки часу буде проходити між ними, залежить від системи і того, чи компонувати програму як остаточну або відлагоджувальну. У випадку, якщо створювалася остаточна версія, проблема буде повторюватися частіше, тому що код був оптимізований. Якщо в системі встановлено

кілька ЦП або двоядерне ЦП, що дозволяють безлічі потоків виконуватись паралельно, проблема також буде виникати частіше, ніж у системі з одноядерним ЦП. У системі з одноядерним ЦП ця проблема виникати теж буде, оскільки виконання потоків планується з переривання на підставі пріоритетів, але звичайно ж не так часто.

Результуюче повідомлення програми інформує про те, що змагання за ресурси виникло після 3816 циклів. Якщо запустити цей додаток кілька разів, то результати будуть завжди різними.

Уникнути цієї проблеми можна, заблокувавши спільно використовуваний об'єкт. Зробити це можна всередині потоку, заблокувавши за допомогою оператора lock змінну стану, яка спільно використовується потоками, як показано нижче. Всередині блоку коду, відповідального за блокування об'єкта стану, може знаходитися тільки один потік. Оскільки цей об'єкт використовується спільно всіма потоками, потік повинен переходити в режим очікування при блокуванні, якщо стан уже заблоковано в іншому потоці. При блокуванні потік починає володіти цим блокуванням і знімає його тільки по закінченні відповідаючого за блокування блоку коду. Коли кожен потік, змінюючий об'єкт, на який посилається змінна стану, використовує блокування, проблема, яка пов'язана із змаганням за ресурси, більше не виникає.

```
public class SampleThread
{
  public void RaceCondition(object o)
  { Trace.Assert(o is StateObject,"o повинна мати тип StateObject");
    StateObject state = o as StateObject;
    int i = 0;
    while (true)
    {
      lock (state) // при такому блокуванні змагання за ресурси немає
      {
           state.Changestate(i++);
      }
    }
  }
}
```

Замість того, щоб застосовувати блокування при використанні розділяючого об'єкта можна також просто зробити цей розділяючий об'єкт безпечним щодо потоків. Тут метод ChangeState () містить оператор lock. Оскільки блокувати саму змінну state не можна (тільки посилочні типи можуть використовуватися для блокування), було визначено змінну sync типу object та вказано її в операторі lock. Якщо при кожній зміні значення state буде застосовуватися блокування з використанням того самого об'єкта синхронізації, проблема змагання ресурсів більше ніколи не виникне.

```
public class StateObject
{
  private int state = 5;
  private object sync = new object ();
  public void ChangeState(int loop)
  {
    lock (sync)
    {
      if (state == 5)
      {
        state++;
        Trace.Assert(state == 6,"Змагання за ресурси виникло після " +
            loop + " циклів");
      }
      state = 5;
    }
  }
}
```

Взаємоблокування

Занадто велика кількість блокувань теж може викликати наслідки. Взаємоблокування (deadlock) – це коли як мінімум два потоки зупиняються і чекають один від одного зняття блокування. Оскільки обидва потоки чекають виконання відповідної дії один від одного, виходить, що вони блокують один одного, в результаті чого їх очікування може тривати нескінченно.

Для демонстрації взаємоблокування нижче наводиться приклад, в якому створюються і за допомогою конструктора класу SampleThread

передаються два об'єкти типу StateObject. Також у ньому створюються два потоки: один виконує метод Deadlock! (), А інший — метод Deadlocks ().

```
StateObject statel = new StateObject ();
StateObject state2 = new StateObject ();
new Thread(new SampleThread(statel,   state2) .Deadlockl) .Start ();
new Thread(new SampleThread(statel,   state2) .Deadlock2). Start ();
```

Далі методи Deadlockl () і Deadlock2 ()змінюють стан двох об'єктів – s1 і s2. Тому і виконуються два блокування. Метод Deadlockl () спочатку блокує об'єкт s1, а потім — об'єкт s2. Метод Deadlock2 (), навпаки, спочатку блокує об'єкт s2, а потім — об'єкт s1. Через це періодично може відбуватися наступне: наприклад, спочатку дозволяється блокування для s1 в Deadlockl (), далі відбувається перемикання потоків, в результаті чого починає виконуватися метод Deadlock2 (), і блокується об'єкт s2. Після цього другий потік, відповідно, очікує блокування об'єкта s1. Оскільки йому потрібно очікувати, планувальник потоків знову відновлює виконання першого потоку, який тепер чекає блокування об'єкта s2. У результаті виходить, що обидва потоки знаходяться в стані очікування і не можуть зняти блокування, оскільки блок коду, який відповідає за блокування, ще не пройдено. Це і є типовий приклад взаємоблокування.

Тема 10. Використання додаткових можливостей платформи .NET

10.1. Модулі компіляції

У середовищі .NET код групується у *складені модулі* – елементарні одиниці в середовищі аплікації. Складений модуль є набором ресурсів і типів, інтегрованих в єдиний логічний об'єкт, який володіє визначеними функціональними можливостями. Складений модуль містить маніфест, метадані типів, код MSIL та ресурси (не всі ці складові є необхідними).

Модуль може налічувати один або кілька файлів. Один з цих файлів містить *маніфест* модуля – частину метаданих (даних, які описують дані), в яких перелічено вміст складеного модуля:

• Версія складеного модуля.

• Культура. Це поле присутнє у маніфесті, якщо модуль призначено для підтримки глобалізації та містить реалізацію функцій для деякої конкретної культури.

• Інформація про стійке ім'я. Якщо модулю надано сильне ім'я, це поле містить відкритий ключ провайдера модуля.

• Список файлів. Кожен файл модуля ідентифікується за кеш-кодом, що робить модулі значно захищенішими від несанкціонованої модифікації чи заміни файлів.

• Інформація про типи. Якщо зі складеного модуля експортують типи, у маніфесті перелічуються файли, які містять оголошення та реалізацію відповідних типів.

• Інформація про зовнішні складені модулі. Для кожного модуля, який використовує цей складений модуль, зазначають ім'я, версію, культуру та відкритий ключ.

• Набір вимог на права. Якщо модуль використовує сторонній код або ресурси, то маніфест містить вимоги на право такого використання.

Типи даних є унікальними всередині складеного модуля. За його межами імена типів уточнюють назвою складеного модуля.

Права доступу надають (або не надають) для модуля загалом.

Перевірка версій здійснюється на рівні модуля. У ньому можуть бути вказані версії інших модулів, необхідні зазначеному модулю для роботи. Файли, які формують модуль, версій не мають.

Складений модуль може мати лише одну точку входу: Main, WinMain або DllMain. Для перегляду вмісту модуля можна використати утиліту ILDasm.exe.

Приватні та розподілені складені модулі

За умовчанням під час компіляції програми утворюється *приватний* складений модуль, доступ до якого має лише одна аплікація. Такий модуль розташовується в каталозі аплікації або його підкаталозі. Щоб до приватного модуля мала доступ інша програма, необхідно утворити копію модуля в каталозі програми.

На відміну від приватного, *розподілений* складений модуль може використовуватися декількома аплікаціями, розташованими в одній файловій системі. Розподілений модуль необхідно забезпечити унікальним ім'ям і розташувати в *глобальному кеші модулів* (область файлової системи в каталозі WinNT\Assembly).

Розгорнути модуль в глобальному кеші можна з допомогою інструмента .NET Framework Configuration, утиліт Al. exe та gacutil. exe або інсталятора, який працює з глобальним кешем модуля. У будь-якому випадку необхідно виконати такі дії:

• Створення криптографічної пари. З цією метою використовують утиліту розподілених імен sn. exe. Наприклад, команда sn -k newkey.snk утворить файл newkey.snk, який міститиме персональний і відкритий ключі.

• Завдання стійкого імені. У файлі Assemblylnfo.cs проекту потрібно надати атрибуту assembly: AssemblyKeyFile значення назви файла, який містить криптографічну пару, і перекомпілювати проект.

• Розташування складеного модуля в кеші. Доцільно використати утиліту gacutil.exe. Наприклад: gacutil /i:MyAssembly.dll.

Для надання значення атрибуту складеного модуля у C# використовують такий синтаксис:

[assembly:AttributeName("Value")]

Наприклад:

using System.Reflection; [assembly : AssemblyVersionAttribute ("1.0.0.1")]

Утворення складених модулів

Усі типи проектів в MS Visual Studio утворюють складені модулі у вигляді виконуваного файла (EXE) або бібліотеки (DLL). Складені модулі можна також утворювати безпосередньо компілятором командної стрічки scs. Додатково компілятор дає змогу утворювати прості модулі – DLL без атрибутів складеного модуля. Простой модуль також має маніфест, однак всередині маніфеста відсутня позиція .assembly.

Компілятор csc має значну кількість опцій. Їхній перелік і зміст можна отримати з допомогою команди csc.exe /help (або csc . exe /?).

Для утворення складеного модуля з декількох файлів можна використати утиліту компонування ai. Наприклад, команда

al.exe Ml.netmodule M2.netmodule/embed:My.bmp /main:M1.Main /out:MyApp.exe/t:exe

утворює файл складеного модуля MyApp.exe. Модуль складається з двох простих модулів і графічного ресурсу My.bmp, який додано з допомогою ключа /embed. Ключ /main зазначає повну назву точки входу (клас і метод).

10.2. Розгортання додатків

Розгортання – це процес розповсюдження готового додатка або компоненту для установки на інші комп'ютери. У Visual Studio 2008 можна розгортати додатки або компоненти за допомогою технології розгортання ClickOnce або технології розгортання установника Windows.

Visual Studio надає дві різні стратегії по розгортанню додатків Windows: публікація додатків за допомогою технології ClickOnce або розгортання з традиційною установкою за допомогою установника Windows. При використанні розгортання ClickOnce здійснюється публікація додатка в деяке централізоване розташування, і користувач встановлює або запускає додаток з цього розташування. При використанні розгортання за допомогою установника Windows додаток упаковується у файл Setup.exe, який розповсюджується серед користувачів і за допомогою якого вони можуть запустити установку.

Існують наступні чинники, які необхідно враховувати при виборі стратегії розгортання: тип додатка, тип і розташування користувачів, частота оновлення додатка і вимоги до установки.

У більшості випадків розгортання ClickOnce зручніше для кінцевого користувача і вимагає менше зусиль з боку розробника. Проте в деяких випадках розгортання за допомогою установника Windows необхідне.

Інструменти розгортання Visual Studio призначені для обробки типових корпоративних потреб в розгортанні; вони можуть не підійти для яких-небудь специфічних сценаріїв розгортання. Для додаткових сценаріїв розгортання можливо потрібний засіб розгортання від незалежного виробника або засобу для розповсюдження програмного забезпечення, такі, як Systems Management Server (SMS).

Функціональні можливості розгортання ClickOnce

У загальному випадку розгортання ClickOnce значно спрощує процес установки і оновлення додатків, але в цьому випадку всі можливості розгортання за допомогою установника Windows будуть недоступні.

Додатки, розгорнені за допомогою ClickOnce, оновлюються автоматично, що дуже зручно для застосувань, що часто змінюються. Хоча додатки ClickOnce можуть бути спочатку встановлені за допомогою компакт-дисків, користувачі повинні мати підключення до мережі, щоб скористатися можливостям автоматичного оновлення.

Нарешті, можуть існувати аспекти безпеки, які можуть вплинути на вибір стратегії. Розгортання ClickOnce в деяких випадках може зажадати

від користувача ухвалення рішення з питання, пов'язаного з безпекою, який може виявитися складним для непідготовленого користувача.

Функціональні можливості розгортання за допомогою установника Windows

При використанні установника Windows до рішення додається проект установки для створення файла установки, поширюваного між користувачами; користувач запускає файл установки і виконує кроки майстра для установки додатка. При використанні ClickOnce додаток упаковується за допомогою Майстра публікацій і публікується на веб-вузлі або в загальній мережній теці; користувач в один етап встановлює і запускає додаток безпосередньо з цього розташування.

При використанні установника Windows створюється пакет установника, поширюваний між користувачами; користувач запускає файл установки і виконує кроки майстра для установки додатка. При розгортанні ClickOnce створюється додаток і маніфести розгортання, які потім публікуються разом з файлами додатка на веб-вузлі або загальному мережному ресурсі; користувач встановлює і запускає додаток безпосередньо з цього розташування.

Технологія розгортання ClickOnce дозволяє створювати додатки Windows, що самообновляються, які здатні встановлюватися і працювати при мінімальній участі користувача. Технологія розгортання ClickOnce служить для подолання трьох основних проблем розгортання:

Труднощі оновлення додатків. У разі розгортання за допомогою установника Microsoft кожного разу коли додаток оновлюється, користувач повинен переустановити все застосування. Технологія розгортання ClickOnce дозволяє надавати оновлення автоматично. Завантажуються тільки ті частини додатка, які змінилися, а потім повне, оновлене застосування повторно встановлюється з нової розташованої поряд теки.

Вплив на комп'ютер користувача. У разі розгортання за допомогою установника Windows застосування часто засновані на загальних компонентах, з можливістю конфліктів контролю версій. У технології розгортання ClickOnce кожне застосування самодостатньо і не робить впливу на інші застосування.

Дозволи безпеці. Розгортання за допомогою установника Window повинне виконуватися з правами адміністратора і допускає тількиобмежену призначену для користувача установку. Розгортання за технологією ClickOnce надає користувачам, що не мають прав

адміністратора, можливість установки, і дозволяє наділяти тільки тими правами системи безпеки доступу до коду, які необхідні для цього.

У минулому ці проблеми іноді змушували розробників ухвалювати рішення про створення веб-додатків замість додатків Windows, приносячи в жертву простоті установки призначений для користувача інтерфейс з широкими можливостями і оперативність відгуку Windows Forms. Використовуючи додатки, що розгортаються за технологією ClickOnce, можна використовувати переваги обох методів.

Що таке додаток ClickOnce?

Простіше кажучи, додаток ClickOnce – це будь-який додаток Windows Presentation Foundation, Windows Forms або консолі, опубліковане за допомогою технології ClickOnce. Додаток ClickOnce можна опублікувати трьома різними способами: з веб-сторінки, із загального мережного файлового ресурсу або з носія, такого, як компакт-диск. Додаток ClickOnce може бути встановлений на комп'ютер кінцевого користувача і може виконуватися локально, навіть коли комп'ютер знаходиться в автономному режимі, або додаток може виконуватися тільки в інтерактивному режимі без постійної установки яких-небудь компонентів на комп'ютер кінцевого користувача.

Додатки ClickOnce можуть бути такими, що самообновляються; вони можуть перевіряти наявність нових версій у міру їх доступності і автоматично замінювати будь-які оновлені файли. Розробник може задати поведінку оновлення. Мережний адміністратор також може управляти стратегіями оновлення, наприклад позначаючи оновлення як обов'язкове. Кінцевий користувач і адміністратор можуть також відкатувати оновлення до ранішої версії.

Оскільки додатки ClickOnce ізольовані, установка і виконання додатка ClickOnce не можуть порушити роботу існуючих застосувань. Додатки ClickOnce самодостатні; кожен додаток ClickOnce встановлюється і виконується з безпечного кеша, виділеного для кожного користувача, для кожного застосування. За умовчанням додатки ClickOnce виконуються в зонах безпеки Інтернету або інтрамережі. За необхідності додаток може запитати підвищені рівні дозволів.

Як працює розгортання ClickOnce

Основна архітектура розгортання ClickOnce заснована на двох XML-файлах маніфестів: маніфести додатка і маніфести розгортання.

Маніфест додатка описує саме застосування. Опис містить складки, залежності і файли, які складають додаток, необхідні дозволи і

місцеположення, де будуть доступні оновлення. Розробник додатка створює маніфест додатка за допомогою майстра публікацій в Visual Studio 2008 або засобах створення маніфесту (Mage.exe) в SDK (пакет засобів розробки програмного забезпечення) для Windows. Маніфест розгортання описує порядок розгортання додатка. Опис містить місцеположення маніфесту додатку і версію додатка, яка повинна запускатися клієнтами. Адміністратор розробляє маніфест розгортання, використовуючи засіб (Mage.exe) створення маніфесту в SDK для Windows.

Після того, як маніфест розгортання створений, він копіюється в місцеположення розгортання. Це може бути веб-сервер, загальний мережний файловий ресурс або носій, такий, як компакт-диск. Маніфест додатка і всі файли додатка копіюються також в місцеположення розгортання, яке задається в маніфесті розгортання. Це місцеположення може співпадати з місцеположенням розгортання або воно може знаходитися у іншому місці. При використанні **Майстра публікацій** в Microsoft Visual Studio 2005 операції копіювання виконуються автоматично.

Після його розгортання в місцеположенні розгортання кінцеві користувачі можуть завантажити і встановити додаток, клацнувши значок, що представляє файл маніфесту розгортання на веб-сторінці або в теці в більшості випадків для кінцевого користувача виводиться просте діалогове вікно із запитом на підтвердження установки, після якого виконується установка, і додаток запускається без додаткових дій користувача. У випадках, коли додатку необхідні дозволи більш високого рівня, в діалоговому вікні виводиться також запит користувачеві на надання дозволу до того, як продовжиться установка.

Додаток додається в призначене для користувача меню **Пуск** і в групу **Установка і видалення програм** в **Панелі управління**. На відміну від інших технологій розгортання нічого не додається в теку **Файли програм**, в реєстр і на робочий стіл. До того ж, для установки не потрібні права адміністратора.

Коли розробник додатка створює оновлену версію додатка, він (вона) генерує також новий маніфест додатка і копіює файли в місцеположення розгортання – зазвичай теку, споріднену для теки розгортання початкового застосування. Адміністратор оновлює маніфест розгортання, щоб націлити його на місцеположення нової версії додатка.

Крім місцеположення розгортання маніфест розгортання містить також місцеположення оновлення (веб-сторінку або загальний мережний

файл). Властивості пункту **Публікація** додатка ClickOnce використовуються, щоб вказати час і частоту перевірки додатком наявності оновлень. Поведінка оновлення може бути задана в маніфесті розгортання, або воно може бути представлене у вигляді варіантів вибору в призначеному для користувача інтерфейсі додатку за допомогою функцій API ClickOnce. Крім того, властивості пункту **Публікація** можуть використовуватися, щоб зробити оновлення обов'язковими або щоб повернути їх назад до ранішої версії.

10.3. Локалізація додатків

Різні культури мають різні календарі і використовують різні формати чисел і дат. До того ж сортування рядків може давати різні результати, тому що порядок A-Z може варіюватися залежно від культури. Щоб додатки відповідали вимогам глобального ринку, їх потрібно глобалізувати і локалізувати.

Глобалізація стосується інтернаціоналізації додатків, тобто підготовки їх для міжнародних ринків. Застосування, що глобалізують, підтримують безліч форматів чисел і дат, залежно від культури, різні календарі і т. п. *Локалізація* – це адаптація додатка до вимог конкретних культур. Для перекладу рядків можна використовувати ресурси.

.NET підтримує глобалізацію і локалізацію як Windows, так і Web-додатків. Щоб глобалізувати додаток, можна використовувати класи з простору імен System.Globalization; щоб локалізувати додаток, можна використовувати ресурси, які підтримує простір імен System.Resources.

Простір імен System.Globalization включає всі класи культур і регіонів, що підтримують різні формати дат, чисел, і навіть різні календарі, які представлені такими класами, як GregorianCalendar, HebrewCalendar, JapaneseCalendar і т. д. Використовуючи ці класи, можна отримати різні відображення інформації залежно від локальних настройок користувача.

Символи Unicode займають 16 бітів, тому в цьому кодуванні є місце для 65 536 символів. Чи достатньо цього для всіх мов, використовуваних в інформаційних технологіях? У разі китайської мови, наприклад, необхідно більше 80 000 символів. Проте Unicode спроектована так, що справляється з цим завданням. Unicode розрізняє базові символи і складені символи. Можна додати безліч комбінованих символів до одного базового, щоб побудувати окремий символ, що відображається або текстовий елемент.

Комбіновані символи визначені в діапазоні від 0x0300 до 0x0345. Для ринків Америки і Європи існують зумовлені символи, що полегшують роботу із спеціальними символами. Для азіатського ринку, де тільки для однієї китайської мови необхідно більше 80 000 символів, такі зумовлені символи не існують. У разі азіатських мов завжди використовуються комбіновані символи. Проблема, витікаюча з цього, полягає в тому, що важко отримати правильне число символів, що відображаються, або текстових елементів, а також в отриманні базових символів замість комбінованих. Простір імен System.Globalization представляє клас StringInfo, який можна використовувати, щоб справитися з цим.

У табл. 10.1 представлений список статичних методів класу StringInfo, які допомагають працювати з комбінованими символами.

Таблиця 10.1

Статичні методи класу StringInfo

Метод	Опис
GetNextTextElement	Повертає перший текстовий елемент (базовий символ і всі комбіновані) вказаного рядка
GetTextElementEnumerator	Повертає об'єкт TextElementEnumerator, що дозволяє виконати ітерацію по всіх текстових елементах рядка
ParseCombiningCharacters	Повертає масив цілих чисел, що посилаються на всі базові символи рядка

Світ розділений на безліч культур і регіонів, і додатки повинні враховувати їх відмінності. Культурою називаємо набір особливостей, заснований на мові і культурних традиціях. Документ RFC 1766 визначає найменування культур, які використовуються, залежно від мови і країни або регіону. Деякі приклади: en-AU, en-CA, en-GB і en-US для англійської мови в Австралії, Канаді, Великобританії і Сполучених Штатах відповідно.

Можливо, найважливішим класом у просторі імен System.Globalization є CultureInfo. Клас CultureInfo представляє культуру і визначає календарі, форматування дат і чисел, а також порядок сортування рядків, використовувані в культурі.

Клас RegionInfo представляє регіональні настройки (такі, як валюта), а також указує, чи використовується в регіоні метрична система. Деякі регіони використовують декілька мов. Одним з прикладів може бути Іспанія, в якій є баскська (eu-ES), каталонська (ca-ES), іспанська (es-ES) і галісійська (gl-ES) культури. Подібно до того, як один

регіон може мати безліч мов, також і однією мовою можуть говорити в багатьох регіонах, – наприклад, іспанською говорять в Мексиці, Іспанії, Гватемалі, Аргентині, Перу, і не тільки.

Працюючи з .NET Framework, потрібно розрізняти три типи культур: *специфічні*, *нейтральні* й *інваріантні*.

Специфічна культура асоційована з реально існуючою культурою, визначеною в RFC 1766. Специфічна культура може бути відображена на нейтральну культуру. Наприклад, de – це нейтральна культура для специфічних культур de-AT, de-DE, de-CH та ін. Тут ‘de’ – скорочення, символізує німецьку мову, а AT, DE і CH – скорочення для Австрії, Німеччини і Швейцарії.

При перекладі додатків, як правило, немає необхідності виконувати переклад для кожного регіону; немає великої різниці між варіантами німецької мови, прийнятими в Германії і Австрії. Замість використання специфічних культур для локалізації додатків досить використовувати нейтральні культури.

Інваріантна культура незалежна від реальної культури. Збереження форматованих чисел і дат у файлах, або відправка їх по мережі на сервер, використовуючи культуру, незалежну від будь-яких призначених для користувача настройок, – якнайкращий вибір.

Коли ви встановлюєте культуру, необхідно розрізняти культуру для призначеного для користувача інтерфейсу і культуру для формату чисел і дат. Культури асоційовані з потоками, і з цими двома типами культур до потоку можна застосовувати по дві настройки культур. Клас Thread має властивість CurrentCulture і CurrentUICulture. Властивість CurrentCulture призначена для установки поточної культури, використовуваної для форматування і сортування, в той час, як CurrentUICulture застосовується для установки мови призначеного для користувача інтерфейсу.

Користувачі можуть змінювати установки за умовчанням Current-Culture, використовуючи регіональні і мовні настройки в панелі управління Windows. За допомогою цієї конфігурації також можна змінювати значення форматів за умовчанням, вживаних для виводу чисел, часу і дат у даній культурі.

CurrentUICulture не залежить від цих настройок. Установки CurrentUI-Culture залежать від мови операційної системи. Всі ці настройки визначають дуже хороші умовчання, і в більшості випадків немає необхідності змінювати їх поведінку. Якщо культура повинна бути змінена, це легко зробити, змінивши обидві властивості потоку, пов'язані

з культурою, скажімо, на іспанську культуру, як показано в наступному фрагменті коду:

```
System.Globalization.CultureInfo ci = new
System.Globalization.CultureInfo("es-ES");
System.Threading.Thread.CurrentThread.CurrentCulture = ci;
System.Threading.Thread.CurrentThread.CurrentUICulture = ci;
```

Форматування чисел

Числові структури Int16, Int32, Int64 і т. п. в просторі імен System включають перевантажений метод ToString(). Цей метод може бути використаний для створення різних представлень чисел. Для структури Int32 ToString() перевантажений наступними чотирма версіями:

```
public string ToString();
public string ToString(IFormatProvider);
public string ToString(string);
public string ToString(string, IFormatProvider);
```

ToString() без аргументів повертає рядок без опцій форматування. Можна також передати рядок і клас, що реалізовує інтерфейс IFormatProvider.

Рядок специфікує формат представлення. Формат може бути заданий стандартним рядком форматування чисел, або ж шаблонним рядком форматування чисел. Для стандартного рядка ‘C’ указує нотацію валюти, ‘D’ – число в десятковому форматі, ‘E’ – наукову нотацію, ‘F’ – формат з фіксованою точкою, ‘G’ – загальний вигляд, ‘N’ – виведення числа. Шаблонне форматування виведення чисел дозволяє специфікувати кількість розрядів, роздільники груп і дробової частини, процентну нотацію тощо. Рядок шаблонного формату "###.###" означає два 3-значних десяткових блоки, розділених груповим роздільником.

Інтерфейс IFormatProvider реалізований класами NumberFormatInfo, DateTimeFormatInfo і CultureInfo. Цей інтерфейс визначає єдиний метод – GetFormat(), повертаючий об'єкт формату.

NumberFormatInfo може бути використаний для визначення спеціалізованих замовлених форматів чисел. Конструктор за умовчанням NumberFormatInfo створює незалежний або інваріантний об'єкт. Використовуючи NumberFormatInfo, можна змінити всі опції форматування, – такі, як знак додатніх чисел, символ відсотка, роздільник груп числа, символ валюти і багато ін. Доступний тільки на читання, незалежний об'єкт NumberFormatInfo повертається статичною властивістю InvariantInfo. Об'єкт NumberFormatInfo, в якому значення форматування

засновані на властивості CultureInfo поточного потоку, повертається статичною властивістю CurrentInfo.

Така ж підтримка, як для чисел, передбачена і для дат. Структу-ра DateTime включає деякі методи для перетворення дат в рядки. Відкриті нестатичні методи ToLongDateString(), ToLongTimeString(), ToShortDateString() і ToShortTimeString() створюють рядкові представ-лення, використо-вуючи поточну культуру. Можна використовувати метод ToString() для призначення іншої культури:

```
public string ToString();
public string ToString(IFormatProvider);
public string ToString(string);
public string ToString(string, IFormatProvider);
```

У рядковому аргументі методу ToString() можна специфікувати зумовлений символ формату або замовлений рядок формату для конвертації дати в рядок. Клас DateTimeFormatInfo специфікує можливі значення. Аргументом IFormatProvider можна вказати культуру.

Застосування перевантажень цього методу без аргументу IFormatProvider припускає використання культури поточного потоку:

```
DateTime d = new DateTime( 2005, 08, 09 );
// поточна культура
Console.WriteLine( d.ToLongDateString() );
// використовувати IFormatProvider
Console.WriteLine( d.ToString( "D", new CultureInfo( "fr-FR" )) );
// використовувати культуру поточного потоку
CultureInfo ci = Thread.CurrentThread.CurrentCulture;
Console.WriteLine( ci.ToString() + ": " + d.ToString( "D" ) );
ci = new CultureInfo( "es-ES" );
Thread.CurrentThread.CurrentCulture = ci;
Console.WriteLine( ci.ToString() + ": " + d.ToString( "D" ) );
```

Сортування

Порядок сортування рядків залежить від культури. Деякі культури мають порядки сортування, що відрізняються. Одним прикладом може служити фінська мова, в якій V і W трактуються, як одне і те ж. Алгоритми, що порівнюють рядки для сортування, за умовчанням використовують культуро-залежне сортування, коли сортування залежить від культури.

Щоб продемонструвати поведінку сортування для фінської мови, наступний код представляє маленький приклад консольного застосування,

в якому деякі найменування штатів США поміщаються в масив у довільному порядку. Автори використовуватимуть класи з просторів імен System.Collections, System.Threading і System.Globalization, тому вони повинні бути доступні. Приведений нижче метод DisplayNames() застосовується для відображення елементів масиву або колекції на екрані:

```
static void DisplayNames( IEnumerable e )
{
  foreach ( string s in e )
    Console.Write( s + " - " );
  Console.WriteLine();
}
```

У методі Main() після створення масиву з найменуваннями деяких штатів США властивості потоку CurrentCulture привласнюється культура Finnish, так що подальший виклик Array.Sort() використовує фінський порядок сортування рядків. Виклик методу DisplayNames() відобразить всі найменування штатів на консолі:

```
static void Main( string[] args )
{
  string[] names = {"Alabama", "Texas", "Washington"
                    "Virginia", "Wisconsin", "Wyoming"
                    "Kentucky", "Missouri", "Utah", "Hawaii"
                    "Kansas", "Lousiana", "Alaska", "Arizona"};
  Thread.CurrentThread.CurrentCulture =
    new CultureInfo( "fi-FI" );
  Array.Sort( names );
  Console.WriteLine( "\n відсортовано..." );
  DisplayNames( names );
```

Після першого відображення найменувань штатів у порядку фінського сортування, масив сортується знову. Якщо потрібно отримати сортування, незалежне від культури користувача, що може бути зручне, коли сортований масив відправляється на сервер, або десь зберігається, то в цьому випадку можна використовувати інваріантну культуру.

Це можна зробити, передавши другий аргумент Array.Sort(). Метод Sort() приймає в другому аргументі об'єкт, що реалізовує інтерфейс IComparer. Клас Comparer з простору імен System.Collections реалізує IComparer. Comparer.DefaultInvariant повертає об'єкт Comparer, що використовує інваріантну культуру для порівняння значень елементів масиву для незалежного від культури сортування:

```
// сортування з використанням інваріантної культури
Array.Sort(names, Comparer.DefaultInvariant);
Console.WriteLine("\відсортовано для інваріантної культури...");
DisplayNames(names);
}
```

На додаток до форматування і системи одиниць вимірювання, прийнятих у тій або іншій місцевості, також залежно від культури можуть розрізнятися тексти і картинки, використовувані програмою. І тут в гру вступають ресурси.

Ресурси

Ресурси – такі, як картинки і таблиці рядків, – можуть бути поміщені в ресурсні файли або підлеглі складки. Ці ресурси можуть бути дуже корисні при локалізації додатків, і .NET має вбудовану підтримку для пошуку локалізованих ресурсів.

Перш, ніж побачимо, як використовувати ресурси для локалізації додатків, у наступному розділі поговоримо про те, як створюються і читаються ресурси, не зважаючи на аспекти мови.

Створення ресурсних файлів

Ресурсні файли можуть містити такі речі, як картинки і таблиці рядків. Ресурсний файл створюється або як звичайний тестовий файл, або як файл з розширенням .resX, використовуючий XML. Почнемо цей розділ з прикладу текстового файла.

Ресурс, що зберігає таблицю рядків, може бути створений як звичайний текстовий файл. У ньому рядкам просто призначаються ключі. Ключ – це ім'я, яке може бути використане програмою для набуття значення. Як у ключах, так і в значеннях допускаються пропуски.

Наступний приклад показує просту таблицю рядків у файлі strings.txt:

```
Title = Professional C#
Chapter = Localization
Author = Christian Nagel
Publisher = Wrox Press
```

Генератор ресурсних файлів

Генератор ресурсних файлів (утиліта Resgen.exe) може бути використаний для створення ресурсного файла з

strings.txt. Наступна команда

```
resgen strings.txt
```

створює strings.resources. Результуючий ресурсний файл може бути доданий до збірки, – або як зовнішній файл, або, як вбудований в DLL або EXE. Утиліта Resgen також підтримує створення ресурсних файлів .resX , що базуються на XML. Простий спосіб створення XML-файла:

resgen strings.txt strings.resX

Ця команда створює ресурсний файл XML strings.resX. У .NET 2.0 Resgen підтримує ресурси, що строго типізуються. Ресурс, що строго типізується, представляється класом, який звертається до ресурсів. Цей клас може бути створений автоматично, якщо застосувати опцію /str до утиліти Resgen:

resgen /str:C#,DemoNamespace,DemoResource,DemoResource.cs strings.resX

Разом з опцією /str указуються мова, простір імен, ім'я класу і ім'я файла початкової коди.

Утиліта Resgen не підтримує додавання картинок. Серед прикладів SDK .NET Framework є приклад ResGen, що включає керівництво. За допомогою ResGen можна включити посилання на зображення у файл .resX. Додавання картинок також може бути виконане програмно, якщо використовувати класи ResourceWriter або ResXResourceWriter, як побачимо це пізніше.

Простір імен System.Resources містить класи, які мають справу з ресурсами:

Клас ResourceManager може бути використаний для отримання ресурсів для поточної культури з складок або ресурсних файлів. Застосовуючи ResourceManager, можна отримати ResourceSet для певної конкретної культури.

Клас ResourceSet представляє набір ресурсів для певної культури. Коли створюється екземпляр ResourceSet, він проходить по класу, що реалізовує інтерфейс IResourceReader, і зберігає всі ресурси в HashTable.

Інтерфейс IResourceReader використовується з ResourceSet для перерахування ресурсів. Клас ResourceReader реалізує цей інтерфейс.

Клас ResourceWriter використовується для створення ресурсного файла. ResourceWriter реалізує інтерфейс IresourceWriter.

ResXResourceSet, ResXResourceReader і ResXResourceWriter подібні ResourceSet, ResourceReader, і ResourceWriter; проте, вони використо-вуються для створення ресурсного файла, заснованого на XML, – .resX, замість двійкового.

Запитання для самоперевірки

1. Що таке Common Language Runtime?
2. Що таке Framework Class Library?
3. Що таке Common Language Specification?
4. Що таке Intermediate Language?
5. Для чого використовують компілятор часу виконання?
6. Що таке керований код?
7. Що таке загальна система типів?
8. Що таке система метаданих?
9. Назвіть головні розбіжності між типами-значеннями і типами-посиланнями.
10. Для чого використовують вікно властивостей?
11. Що відображає вікно провідника рішення?
12. Опишіть методи введення/виведення даних у режимі консолі.
13. Що таке ідентифікатор?
14. Що розуміють під типом даних?
15. Що повідомляється компіляторові при визначенні змінних-/констант?
16. Що таке область визначення змінних?
17. Що таке час існування змінних?
18. Вкажіть і охарактеризуйте цілі типи даних.
19. Вкажіть і охарактеризуйте раціональні типи даних.
20. Що таке логічний тип?
21. Як представляються рядки символів?
22. Що таке константи?
23. Що таке вираз? Вкажіть правила обчислення виразу.
24. Наведіть приклади операцій з однаковим пріоритетом.
25. Наведіть приклади операцій порівняння.
26. Що таке логічне І/АБО?
27. Наведіть приклади функцій перетворення типів даних.
28. Наведіть приклади функцій перевірки типів.
29. Наведіть приклади математичних функцій.
30. Перерахуйте оператори керування.
31. Які форми має умовний оператор?
32. Запишіть і охарактеризуйте оператор циклу for.
33. Запишіть і охарактеризуйте оператор циклу do – while.
34. Запишіть і охарактеризуйте оператор циклу while.

35. У яких випадках застосовують оператор break?
36. Що таке масив?
37. Яка різниця між статичним і динамічним масивом?
38. Як описують одновимірний масив?
39. Як описують багатовимірний масив?
40. Опишіть основні методи класу Array.
41. Що таке структура? Як описують структуру?
42. Що таке перерахування?
43. Як порівнюються рядки символів?
44. Назвіть і охарактеризуйте методи класифікації символів класу Char.
45. З якою метою найкраще використовувати клас String?
46. З якою метою найкраще використовувати клас StringBuilder?
47. Дайте означення класу.
48. Що таке інкапсуляція?
49. З якою метою використовують поля класу?
50. Дайте означення властивості.
51. З якою метою використовують ключове слово private в оголошенні класу?
52. З якою метою використовують ключове слово public в оголошенні класу?
53. З якою метою використовують ключове слово protected в оголошенні класу?
54. Що таке методи класу?
55. З якою метою використовують ключове слово private в оголошенні властивості/методу?
56. З якою метою використовують ключове слово public в оголошенні властивості/методу?
57. Що таке перевантаження методу?
58. Дайте означення об’єкта.
59. Що таке посилання на об’єкт?
60. Дайте означення конструктора.
61. Що таке перевантаження конструктора?
62. Дайте означення деструктора.
63. З якою метою використовують статичні члени класу?
64. Що таке успадкуваня?
65. Що таке поліморфізм?
66. Охарактеризуйте головні методи класу Object.
67. Що таке абстрактний клас?

68. З якою метою використовують інтерфейс?
69. З якою метою використовують делегати?
70. Що таке подія? Як генерують подію?
71. Як зв’язати подію об’єкта з процедурою опрацювання події?
72. Що таке файл послідовного доступу?
73. Що таке двійковий файл?
74. Що таке файл довільного доступу?
75. Опишіть функції введення/виведення даних для файлів.
76. З якою метою використовують метод FileOpen?
77. З якою метою використовують метод Close?
78. З якою метою використовують клас FileSystemInfo?
79. З якою метою використовують клас Directory?
80. З якою метою використовують клас DirectoryInfo?
81. З якою метою використовують клас File?
82. З якою метою використовують клас FileInfo?
83. З якою метою використовують клас FileStream?
84. З якою метою використовують клас StreamReader?
85. З якою метою використовують клас StreamWriter?
86. З якою метою використовують клас BinaryReader?
87. З якою метою використовують клас BinaryWriter?
88. Опишіть головні події роботи з мишею.
89. Опишіть головні події роботи з клавіатурою.
90. Опишіть базові компоненти введення і відображення даних.
91. Як реалізовується мультивибір у ListBox?
92. Що таке кнопка? З якою метою її використовують?
93. Що таке залежний перемикач? З якою метою його використовують?
94. Що таке незалежний перемикач? З якою метою його використовують?
95. Що таке панель? З якою метою її використовують?
96. Що таке таймер? З якою метою його використовують?
97. Що таке елемент керування ComboBox? З якою метою його використовують?
98. Що таке елемент керування CheckListBox? З якою метою його використовують?
99. Що таке елемент керування TrackBar? З якою метою його використовують?

Література

Основна

1. Голуб Б. М. C#. Концепція та синтаксис : навч. посібник / Б. М. Голуб. – Львів : Видавничий центр ЛНУ імені Івана Франка, 2006. – 136 с.

2. Троелсен Э. C# и платформа.Net / Э. Троелсен ; пер. с англ. – СПб. : Питер, 2007. – 796 с.

3. Троелсен Э. Язык программирования C# 2005 и платформа .Net 2.0 / Э. Троелсен ; пер. с англ. – М. : Издательский дом "Вильямс", 2007. – 1168 с.

4. Шилдт Г. C#: учебный курс / Г. Шилдт. – СПб. : Питер, 2003. – 512 с.

Додаткова

5. Гради Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ / Буч Гради ; пер. с англ. – М. : Издательский дом "Вильямс", 2005. – 776 с.

6. Лабор В. В. Си Шарп: Создание приложений для Windows / В. В. Лабор. – Мн. : Харвест, 2003. — 384 с.

7. Либерти Джесс. Программирование на C# / Джесс Либерти ; пер. с англ. – СПб. : Символ-плюс, 2005. – 684 с.

8. Нейгел К. C# 2005 для профессионалов / К. Нейгел, Б. Ивьен, Дж. Глинн и др. / пер. с англ. – М. : Издательский дом "Вильямс", 2006. – 1376 с.

9. Петцольд Чарльз. Программирование для Microsoft Windows на C#. В 2-х т. Т. 1 / Чарльз Петцольд ; пер. с англ. – М. : Издательско-торговый дом Русская Редакция, 2002. – 576 с.

10. Петцольд Чарльз. Программирование для Microsoft Windows на C#. В 2-х т. Т. 2 / Чарльз Петцольд ; пер. с англ. – М. : Издательско-торговый дом Русская Редакция, 2002. – 624 с.

11. Рихтер Джеффри. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#. Мастер-класс / Джеффри Рихтер ; пер. с англ. – СПб. : Питер, 2007. – 656 с.

ЗМІСТ

НАВЧАЛЬНЕ ВИДАННЯ

Парфьонов Юрій Едуардович
Федорченко Володимир Миколайович
Лосєв Михайло Юрійович
Щербаков Олександр Всеволодович

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Конспект лекцій
для студентів напряму підготовки
"Комп'ютерні науки"
всіх форм навчання

Відповідальний за випуск **Пономаренко В. С.**
Відповідальний редактор **Сєдова Л. М.**

Редактор **Голінська О. Г.**
Коректор **Голінська О. Г.**

План 2010 р. Поз. № 107-К.
Підп. до друку Формат 60 х 90 1/16. Папір MultiCopy. Друк Riso.
Ум.-друк. арк. 19,5. Обл.-вид. арк. 24,37. Тираж прим. Зам. №_____
Видавець і виготівник — видавництво ХНЕУ, 61001, м. Харків, пр. Леніна, 9а
Свідоцтво про внесення до Державного реєстру суб'єктів видавничої справи ***Дк № 481 від 13.06.2001 р.***

Парфьонов Ю. Е.
Федорченко В. М.
Лосєв М. Ю.
Щербаков О. В.

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Конспект лекцій