Міністерство освіти і науки України
Львівський національний університет імені Івана Франка

Б. М. Голуб

C#.
Концепція та синтаксис

Навчальний посібник

Львів
Видавничий центр ЛНУ імені Івана Франка
2006

УДК 004.432 C# (075.8)
Г-62
ББК 3973.2-018.1я73-1 C#

Рецензенти:

Д-р. фіз.-мат. наук, професор Михайло Ярославович Бартіш
(Львівський національний університет імені Івана Франка)

Канд. фіз.-мат. наук Ярослав Володимирович Любінець

Остап Романович Радковський

Рекомендовано до друку вченою радою
факультету прикладної математики та інформатики.
Протокол № 2 від 21.06.2005р.

Голуб Б.М.
Г-62 C#. Концепція та синтаксис. Навч. посібник. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2006. – 136 с.

У посібнику подано опис основних конструкцій алгоритмічної мови C#: типи даних, змінні та константи, операції, оператори керування, класи, вказівники, делегати, події. Значну увагу приділено архітектурі .NET. Розглянуто основні класи базової бібліотеки .NET, складені модулі, метадані та механізми відображення, конфігураційні файли, взаємодію з COM. Описано принципи розробки проектів в середовищі програмування Visual Studio .NET.

Для студентів факультету прикладної математики та інформатики, а також усіх бажаючих навчитися програмувати алгоритмічною мовою C# у середовищі .NET.

ББК 3973.2-018.1я73-1 C#

Вступ

Технологія .NET – порівняно нова 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 і мобільних телефонів.

Microsoft – безумовний лідер у розвитку операційних систем і технологій програмування – відзначає .NET як головну платформу розробки програмного забезпечення на найближчі роки. Цей факт зумовлює для сучасного програміста необхідність володіти архітектурою та алгоритмічними мовами .NET.

Мова C# – нова алгоритмічна мова, розроблена для написання програм у середовищі .NET. Основою C# є мови Java та C++. За детальнішого ознайомлення з C# бачимо, що вона успішно акумулювала кращі особливості Java, C++ та інших сучасних мов. Водночас C# не є надлишковою мовою. Вона містить лише необхідні конструкції.

C# поза технологією .NET не існує. Мова ґрунтується на типах даних базової бібліотеки .NET. Для виконання C#-програм необхідне загальномовне середовище виконання (CLR).

Посібник містить, окрім опису конструкцій мови, вибіркову інформацію про архітектуру .NET та базову бібліотеку класів. Детальний опис цих елементів можна зробити лише в багатотомному виданні. Значний обсяг інформації та обмежений розмір посібника зумовили деяку конспективність викладок. Отож посібник не призначено для вивчення першої алгоритмічної мови.

Читачеві необхідно володіти навичками програмування на платформі операційної системи Windows, а також мати досвід використання однієї з сучасних мов програмування: C++, Java, Object Pascal, Visual Basic.

Посібник має вузьке спрямування: вступ до алгоритмічної мови C# та опис необхідних елементів .NET для написання простих Windows-проектів у середовищі Visual Studio .NET. Він не містить

таких складових .NET-технологій, як ADO.NET, ASP.NET, Web-сервіси, XML та інших.

Базова бібліотека .NET містить тисячі класів та десятки тисяч методів, більшість з яких мають декілька реалізацій. Цю інформацію неможливо запам'ятати. Отож при розробці проектів під .NET необхідний доступ до довідкової системи MSDN Library.

БАЗОВІ ПОНЯТТЯ ТЕХНОЛОГІЇ .NET

Ключові терміни

CLR (Common Language Runtime) – середовище виконання .NET. Можна розглядати як код, який завантажує програму, виконує та надає їй усі необхідні служби.

Керований код (Managed Code) – довільний код, розроблений для виконання в середовищі CLR. Код, який виконується безпосередньо під операційною системою і не потребує .NET платформи, називають некерованим.

IL (Intermediate Language, MSIL) – проміжна мова, на якій написано код, якщо він призначений для завантаження та виконання середовищем .NET. При обробці керованого коду компілятор генерує код на IL, а CLR виконує завершальну стадію компіляції в машинні коди безпосередньо перед виконанням.

CTS (Common Type System) – спільна система типів даних .NET. Розроблена для забезпечення сумісності адаптованих до .NET алгоритмічних мов. CTS надає також правила для означення нових типів даних.

CLS (Common Language Specification) – загальна специфікація алгоритмічних мов. CLS є підмножиною CTS і мінімальним набором стандартів, який повинні підтримувати усі компілятори для .NET.

Простір імен – логічна схема групування типів відповідної функціональності. Простори імен мають ієрархічну структуру.

Складений модуль (Assembly) – модуль, який містить компільований керований код. На відміну від виконуваних EXE чи DLL файлів, складені модулі містять також *метадані*: інформацію про модуль та всі визначені в ньому типи, методи та інше. Складений модуль може бути приватним (доступним для

використання однією аплікацією) або розподіленим (доступним для використання багатьма аплікаціями).

Глобальний кеш складених модулів (Global Assembly Cashe, GAC) – місце на диску, де зберігаються розподілені складені модулі.

Відображення (Reflection) – технологія, яка передбачає програмний доступ до метаданих складеного модуля.

Компіляція Just-In-Time (JIT) – процес виконання завершальної стадії компіляції з IL у машинний код. Передбачає компіляцію не коду загалом, а лише його частин за необхідністю.

Маніфест – область складеного модуля, що містить метадані.

Область застосування (AppDomain) – спосіб, за якого CLR дає змогу різним програмам виконуватися в одному і тому ж просторі процесів.

Прибирання сміття (Garbage collection) – механізм чищення пам'яті, реалізований у .NET.

Процес компіляції та запуску програми

Скомпільований код програми не містить інструкцій асемблера, а лише інструкції на IL. Ці інструкції розташовані у складеному модулі. Сюди ж долучають метадані:

- опис типів даних;
- методи всередині складеного модуля;
- простий кеш (будується на основі вмісту складеного модуля та може бути використаний для перевірки його цілісності);
- інформація про версії складеного модуля;
- інформація про необхідні зовнішні складені модулі;
- інформація про привілеї, необхідні для виконання коду складеного модуля.

Пакет програм збирається зі складених модулів. Один з цих модулів є виконуваним і містить точку входу основної програми, а інші є бібліотеками. .NET завантажує виконуваний модуль, перевіряє його цілісність та метадані, а також рівень привілеїв користувача на відповідність потребам складеного модуля.

На цьому ж етапі CLR також робить перевірку *безпеки коду за типом пам'яті* (memory type safety). Код вважають безпечним за типом пам'яті лише тоді, коли він звертається до пам'яті

способами, які може контролювати середовище CLR. Якщо CLR не впевнене у безпеці коду за типом пам'яті, то (залежно від локальної політики безпеки) може відмовити у виконанні коду.

Для виконання коду CLR утворює *процес* операційної системи Windows і зазначає область застосування, в якій розташовано головний *потік* програми. CLR вибирає першу частину коду, який необхідно виконати, компілює її з мови IL на мову асемблера та виконує з відповідного потоку програми. Коли під час виконання трапляється новий метод, він компілюється у виконуваний код. Процес компіляції цього методу відбувається лише один раз. У процесі виконання коду CLR відстежує стан пам'яті та періодично запускає процедуру прибирання „сміття", тобто звільнення пам'яті від об'єктів, на які відсутні вказівники у коді.

ПРОГРАМУВАННЯ В СЕРЕДОВИЩІ VS.NET

У цьому розділі коротко розглянемо елементи середовища програмування Microsoft Visual Studio .NET 2005 (далі VS).

Типи проектів

Новий проект можна створити з допомогою меню **File | New | Project** (або натисненням кнопки **New Project**). У діалоговому вікні потрібно вибрати тип проекту та мову програмування. Для C# означені такі типи проектів (конкретний набір проектів залежить від вибору, здійсненого в процесі налаштування середовища):

Тип проекту	**Стартовий код**
Windows	
Windows Application	Порожня форма.
Class Library	Бібліотека класів, яку можна використовувати в довільному коді, призначеному для платформи .NET.
Windows Control Library	Клас .NET, який можна активізувати іншим кодом .NET. Має інтерфейс користувача (подібно компонентам ActiveX).
Web Control Library	Елемент керування. Активізується сторінками ASP.NET для генерування HTML-коду представлення елемента керування при перегляді у вікні браузера.
Console Application	Аплікація, яка виконується з командної стрічки або у вікні консолі.
Windows Service	Служба, яка працює у фоновому режимі Windows.

Empty Project	Проект Windows Application без стартового коду.
Crystal Reports Windows Application	Порожня форма проекту для генератора звітів Crystal Reports
Office	
Проекти для Microsoft Excel та Word	
Smart Device	
Проекти для Pocket PC, Smartphone та Windows CE	
Database	
Проект для SQL Server	
Web Site	
ASP.NET Web Site	Web-сайт на основі ASP.NET
ASP.NET Web Service	Клас Web-служби
Empty Web Project	Проект ASP.NET Web Site без стартового коду.
Project From Existing Code	
Project From Existing Code	Нові файли для порожнього проекту. Використовують для конвертування існуючого коду C# у проект VS.NET.

Файли проекту

При створенні нового проекту VS утворює папку проекту наступної структури:

Каталоги bin та obj призначені для розташування компільованих і тимчасових файлів.

Основний каталог WindowsApplication1 (назва каталогу відповідатиме назві проекту) призначений винятково для VS.NET і містить інформацію щодо проекту. У процесі розробки в проект можна додавати нові каталоги.

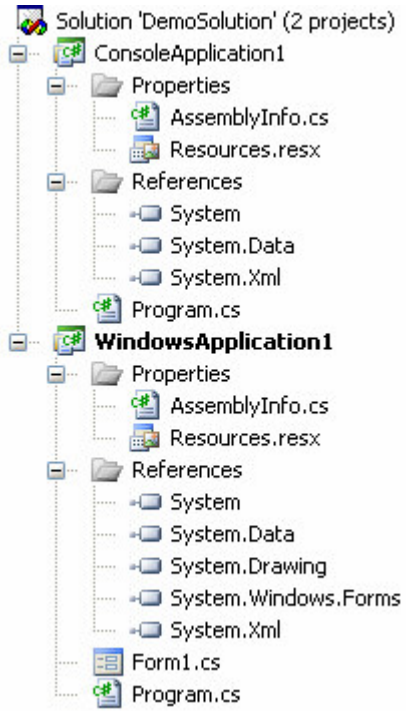
Рішення та проекти

Рішення (*Solution*) – це набір усіх проектів, який утворює програмне забезпечення для поставленої задачі.

Проект (*Project*) – це набір усіх файлів вихідного коду та ресурсів, які компілюються в один складений модуль (assembly). У простіших проектах складений модуль – це один модуль.

Структура рішення відображається у вікні провідника по рішенню (*Solution Explorer*)

Якщо утворити два проекти (наприклад, консольний та для Windows), то Solution Explorer матиме приблизно таку структуру:



Поточний (активний) проект виокремлено жирним шрифтом. Установити поточний проект можна з допомогою контекстного меню на назві проекту: **Set As Startup Project**.

Рішення може містити проекти на різних алгоритмічних мовах, адаптованих до .NET.

Натиснення правої клавіші миші на довільному вузлі активізує контекстне меню, орієнтоване на особливості обраного вузла.

Перегляд та написання коду

Вікна

VS містить великий набір інструментів для розробки проектів. Ці інструменти – вікна – мають декілька режимів розташування та встановлення розмірів:

Режим вікна	**Розташування та розміри**
Плаваючий (*Floating*)	Окреме вікно із заданими користувачем розмірами.
Прикріплений (*Dockable*)	Вікно розташовується усередині деякого іншого вікна-контейнера. Контейнери можна міняти, перетягуючи вікно за його заголовок. Для прикріпленого вікна контейнер утворює сторінку табуляції. Вікно займає усю вільну область контейнера.
Табульований документ (*Tabbed Document*)	Вікно розташоване усередині вікна **Редактора**, утворює відповідну закладку. Займає всю вільну область контейнера або ділить область **Редактора** навпіл (по горизонталі або по вертикалі).

Вікно можна зробити невидимим (*Hide*), що є аналогом закритого. Для активізації невидимого вікна його потрібно знайти у списку меню *View*.

Контейнер можна зробити прихованим (*Auto hide*). У цьому випадку вікно ховається за одним із країв екрана і з'являється при наближенні курсора миші.

Редактор

Редактор VS містить усі стандартні можливості редакторів тексту, форм, ресурсів та інших елементів проекту.

Блоки коду (класи, члени класу, цикли, набори однотипних стрічок та ін.) розглядаються як елементи дерева. Їх можна згорнути або розгорнути, акцентуючи увагу лише на необхідному коді.

З допомогою директив `#region` та `#endregion` можна формувати свої гілки дерева перегляду коду.

Редактор коду використовує технологію IntelliSense. Зокрема, виводиться контекстний список можливих елементів після крапки наприкінці назви класу. Цей список можна також активізувати

комбінацією клавіш Ctrl+Space. Ctrl+Shift+Space активізує список та опис параметрів методів.

Редактор проводить часткову синтаксичну перевірку коду. Синтаксичні помилки підкреслюються хвилястою лінією. При наведенні курсора миші на підкреслене слово VS виводить віконце з описом помилки.

Вікно інструментів

Вікно інструментів (*ToolBox*) містить згруповані за категоріями компоненти .NET, які використовують при розробці застосувань. Компоненти перетягують у програму з допомогою миші.

Можна додавати власні категорії елементів (контекстне меню **Add Tab**). Елементи ActiveX та компоненти COM долучають опцією меню **Customize ToolBox**.

Вікно властивостей

Вікно властивостей (*Properties*) відображає та дає змогу редагувати значення властивостей і подій (*events*) активного (виокремленого) керуючого елемента (компонента). Властивості та події можна впорядкувати за категоріями чи алфавітом. Виокремлений елемент супроводжується коротким описом.

Вікно класів

Вікно класів (*Class View*) дає ієрархічний список просторів імен, класів та об'єктів проекту. Список можна сортувати та групувати за категоріями.

Контекстне меню для активного елемента списку містить опцію Go To **Definition** (F12) – перехід до означення активного елемента в коді програми.

Браузер об'єктів

Браузер об'єктів (*Object Browser*) містить ієрархічний список класів проекту. Причому, на відміну від вікна Class View, передбачає перегляд просторів імен та класів у всіх складених модулях, які використовує проект.

Вікно реалізоване подібно до файлового провідника: ліва панель показує дерево класів, а права – члени класу та опис синтаксису.

Для перегляду COM-об'єктів доцільно використовувати програму OLEVIEW або інтерфейси .NET обгортки над COM-об'єктом, автоматично згенерованої середовищем розробки.

Серверний провідник

Серверний провідник (*Server Explorer*) надає інформацію про комп'ютер: з'єднання з базами даних, служби, Web-служби, запущені процеси, журнали подій та інше. Server Explorer з'єднаний з вікном властивостей Properties: вибір елемента ініціює налаштування вікна Properties на показ властивостей цього елемента.

Розробка проекту

При створенні проекту VS.NET автоматично генерує дві конфігурації: *Debug* та *Release*. Головною відмінністю конфігурації Debug від Release є те, що оптимізація коду не проводиться, а у виконавчі файли додається інформація відлагодження. Очевидно, що відлагоджений продукт повинен розповсюджуватися у конфігурації Release.

Visual Studio дає змогу також утворювати власні конфігурації. Для вибору конфігурації використовують опцію меню **Debug | Set Active Configuration**, а для редагування – **Project | Properties**.

Процес розробки проекту загалом містить етапи проектування, кодування (написання програм) та відлагодження (пошук і виправлення помилок).

Одним з головних інструментів відлагодження є точки переривання (*Break Points*) процесу виконання програми. У VS для точок переривання можна задавати умови (опція меню **Debug | Break Points**), зокрема:

- задати лічильник, за умови досягнення яким заданого значення здійснити переривання;
- здійснити переривання через заданих *n* проходів;

- додати точку переривання для змінної (спрацьовує за умови зміни значення).

Важливим є також дослідження стану об'єктів у довільній точці виконання програми. З цією метою використовують такі вікна:

- **Autos** – відстежує декілька останніх змінних, доступ до яких здійснювався в процесі виконання програми;
- **Locals** – локальні змінні методу, який виконується**;**
- **Watch1, Watch 2, …** - явно задані змінні.

У процесі відлагодження корисним може бути вікно **Call Stack** (стек викликів), яке містить упорядкований перелік методів, які виконуються в поточний момент часу, а також значення аргументів цих методів.

Вікно **Exceptions** (винятки) дає змогу зазначити, які дії виконати при генеруванні обраного винятку. Можна обрати і варіант продовження виконання, і варіант переходу до відлагодження. В останньому випадку виконання програми призупиняється, а відлагоджувач переходить до відповідного оператора `throw`.

Меню **Debug** містить значну кількість опцій активізації інших корисних інструментів відлагодження програми.

ОСНОВИ C#

Проста програма

Розглянемо код:

```
using System;

public class MyConsoleWrite {
 public static int Main() {
   Console.WriteLine("Це є моя C#-
програма!");
   return 0;
 }
}
```

Цей код можна записати в текстовий файл `MyCSharp.cs` та скомпілювати з допомогою компілятора C# (програма csc.exe): `csc.exe MyCSharp.cs`.

У C# кожен оператор закінчується символом „`;`”.

Для об’єднання операторів у блоки використовують фігурні дужки: {…}.

Текст, розташований між наборами символів `/*` та `*/`, є коментарем і компілятором ігнорується. Текст, розташований після символів `//` і до кінця стрічки, також є коментарем.

Весь код програми повинен міститися в класі або в іншому означенні типу. Класи (та інші типи) на платформі .NET організовуються у *простори імен (namespaces)*. У цьому прикладі простір імен не зазначено, отож клас належить неіменованому загальному простору імен.

Директива `using` дає вказівку компілятору, що неописані в поточному просторі імен класи потрібно шукати в поданому списку просторів імен.

Кожен виконуваний файл C# повинен мати точку входу – метод `Main`. Цей метод може не повертати результат (тип `void`), або повертати ціле число (тип `int`). Означення методів у C# таке:

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

Тут квадратні дужки позначають необов'язкові елементи.

Модифікатори використовують для встановлення рівня доступу до методу.

Типи даних

C# підтримує загальну систему типів CTS. Кожен тип CTS є класом і володіє методами, корисними для форматування, серіальності та перетворення типів.

Дані програми зберігаються у *стеку*.

Стек зберігає дані фіксованої довжини, наприклад, цілі значення. Якщо деяка функція *A* активізує функцію *B*, то всі змінні, які є локальними щодо функції *A*, зберігаються у стеку. Після виконання коду функції *B* керування повертається функції *A*. У цьому випадку зі стека зчитуються збережені значення локальних змінних.

Дані змінної довжини зберігаються в динамічно розподіленій пам'яті (*heap allocation*). Динамічну пам'ять використовують також для збереження даних, тривалість життя яких повинна бути довшою за тривалість життя методу, у якому їх означено.

C# ділить свої типи даних на дві категорії залежно від місця зберігання. *Змінні типів за значенням* зберігають свої дані в стеку, а *змінні типів за посиланням* – в динамічній пам'яті.

Операція присвоєння одній змінній за значенням іншої змінної за значенням утворює дві різні копії одних і тих же даних у стеку.

Операція присвоєння одній змінній за посиланням іншої змінної за посиланням спричинює виникнення двох посилань на одну й ту ж ділянку пам'яті, тобто реально дані не дублюються.

У C# базові типи даних, такі, як `bool` і `long`, є типами за значенням. Здебільшого складні типи C#, у тім числі й класи, означені користувачем, є типами за посиланням.

Зауважимо, що на відміну від C++, де тип `struct` є специфічним класом, у C# тип `struct` є типом за значенням.

Типи C#

У C# ціле число можна оголосити з використанням типу CTS:

```
System.Int32 x;
```

Однак це виглядає неприродно для програміста C. Тому у C# означені псевдоніми (*aliases*) типів CTS.

C# має 15 базових типів: 13 типів за значенням та 2 типи (`string` та `object`) за посиланням.

Типи за значенням

C# вимагає, щоб кожна змінна за значенням була явно ініціалізована початковим значенням до того, як її використовуватимуть в операціях.

Наступна таблиця містить перелік типів за значенням.

Тип C#	Тип CTS	Розмір (байт)	Кількість точних розрядів
Цілі типи			
`Sbyte`	`System.SByte`	1	
`Short`	`System.Int16`	2	
`Int`	`System.Int32`	4	
`Long`	`System.Int64`	8	
`Byte`	`System.Byte`	1	
`Ushort`	`System.UInt16`	2	
`Uint`	`System.UInt32`	4	
`Ulong`	`System.UInt64`	8	
Числа з плаваючою крапкою			
`Float`	`System.Single`	4	7
`Double`	`System.Double`	8	15-16
Десятковий тип			
`Decimal`	`System.Decimal`	16	18
Логічний тип			
`Bool`	`System.Boolean`	1	
Символьний тип			
`Char`	`System.Char`	2	

Змінним цілого типу можна присвоювати значення у десятковій та шістнадцятковій системах числення. Остання вимагає префікс `0x`:

```
long x = 0x1ab;
```

Можна використовувати явно типізовані значення:

```
uint ui = 12U;
long l = 12L;
ulong ul = 12UL;
double f = 1.2F;
decimal d = 1.20M;
```

Десятковий тип `decimal` призначений для забезпечення високої точності фінансових операцій.

Змінні логічного типу можуть набувати лише значення `false` або `true`.

Значення змінних символьного типу – це символи Unicode. У коді символи розташовують між одинарними лапками. Наприклад: ‘a’, ‘c’, ‘\u0041’, ‘\x0041’. Останні два значення – це символи, задані своїми номерами. Передбачено також зведення типів: `(char)65`.

Існують також спеціальні символи:

`\'` – одинарна лапка
`\"` - подвійна лапка
`\\` - зворотний слеш
`\o` – null-значення
`\e` – увага
`\b` – повернення назад на 1 символ
`\f` – подання сторінки
`\n` – нова стрічка
`\r` – повернення каретки
`\t` – символ табуляції
`\v` – вертикальна табуляція

Типи за посиланням

Нехай існує деякий клас `myClass`. Розглянемо стрічку коду

```
myClass objMyClass;
```

Ця стрічка формує посилання (тобто резервує `sizeof(int)` байт для розташування адреси) на ще не утворений об’єкт `objMyClass`. Посилання `objMyClass` матиме значення `null`.

У C# утворення екземпляра об'єкта за посиланням вимагає ключового слова `new`:

```
objMyClass = new myClass();
```

Цей код утворює об'єкт у динамічній пам'яті та його адресу записує в `objMyClass`.

C# містить два базових типи за посиланням:

- `object` (тип CTS `System.Object`) – кореневий тип, який успадковують усі інші типи CTS (у тім числі типи за значенням);
- `string` (тип CTS `System.String`) – стрічка символів Unicode.

У C# тип `object` є стартовим типом-предком, від якого беруть початок усі внутрішні та всі визначені користувачем типи. Цей тип реалізує низку базових універсальних методів: `Equals()`, `GetHashCode()`, `GetType()`, `ToString()` та інші. Класам користувача, можливо, доведеться замінити реалізації деяких із цих методів, використовуючи об'єктно-орієнтований принцип перекриття (*overriding*).

При оголошенні змінної за посиланням типу `object` або похідного від `object` класу цій змінній початково надається значення `null` (за умови, що ця змінна є частиною класу). Якщо ж неініціалізована змінна оголошена в межах методу, у програмі виникне помилка на етапі компіляції.

Об'єкт `string` дає змогу зберігати (в динамічній пам'яті) набір символів Unicode. Зауважимо, що коли одну стрічкову змінну присвоїти іншій, то в результаті одержимо два посилання на одну й ту ж стрічку в пам'яті. Якщо ж подальший код вносить зміни в одну із цих стрічок, то утворюється новий об'єкт `string`, водночас інша стрічка залишається без змін.

Стрічкові літерали розташовуються у подвійних лапках "...". Стрічки C# можуть містити ті ж символи, що й тип `char`. Символ „\", якщо він не призначений для означення спеціального символу, подвоюється:

```
string filepath = "C:\\Program Files\\F.cs";
```

Стрічковому літералу може передувати символ `@`, який дає вказівку всі символи трактувати за принципом „як є”:

```
string filepath = @"The file
     C:\Program Files\F.cs is C#-file";
```

Значення цієї стрічки містить також і всі пробіли перед `C:\`.

Складні типи

Структури

Структура – це подібний до класу тип даних за значенням. Оскільки структура розташовується в стеку, вона утворюється більш ефективно, ніж клас. До того ж копіюється простим присвоєнням.

Структура ініціалізується відразу після свого оголошення, а всі поля встановлюються у значення за замовчуванням (`0`, `false`, `null`). Однак компілятор не дає змоги копіювати одну структуру в іншу до її ініціалізації за допомогою ключового слова `new`.

Приклад:

```
public struct Student {
  public string FirstName;
  public string LastName;
  public string Group;
}
Student st1, st2;
st1 = new Student;
st1.FirstName = "Андрій";
st1.LastName = "Адаменко";
st1.Group = "Пма-51";
st2 = st1;
```

У C# структура може виконувати більшість функцій класу (однак не підтримує наслідування реалізацій). Оскільки екземпляри структур розташовані в стеку, доцільно їх використовувати для представлення невеликих об’єктів. Це одна з причин, за якої їх застосовують для реалізації всіх інших типів даних за значенням у платформі .NET.

Переліки

Перелік – це означуваний користувачем цілий тип даних. При оголошенні переліку вказується набір допустимих значень, які можуть набувати екземпляри переліку:

```
public enum Light {
  Green = 0,   Yellow = 1,   Red = 2
}
Light l;
l = Light.Red;
```

Переліки мають тип за значенням.

Класи

Класи – це складені типи даних, які містять члени даних (поля, константи та події) і функції (методи, властивості, оператори). Клас є інкапсуляцією даних і функціональних засобів для доступу та роботи із цими даними. Класи можуть також містити вкладені типи даних.

Інтерфейси

Інтерфейси використовують для означення деякої функціональності: властивості, методи, події та індексатори. Інтерфейси не містять самої реалізації цієї функціональності, отож екземпляр інтерфейсу не може бути створений. Якщо кажуть, що клас підтримує деякий інтерфейс, це означає, що в класі є реалізація всіх оголошених інтерфейсом функцій.

Делегати

Делегати – це типи даних, які посилаються на методи. Вони подібні на вказівники функцій в C++, проте передбачають утворення екземпляра класу та активізацію як статичних методів класу, так і методів конкретного екземпляра класу. Делегати дозволяють під час виконання коду визначити, який метод із заданого набору необхідно активізувати.

Масиви

Масив – це колекція змінних однакового типу, звернення до яких відбувається з використанням загального для всіх імені.

Для оголошення одновимірного масиву використовують такий синтаксис:

```
тип[] ім'я_масиву = new тип[розмір];
```

Наприклад,

```
int[] arrInt;
int[] arrInt = new int[10];
```

Масиви у C# є 0-базованими, тобто перший елемент масиву має індекс 0:

```
arrInt[0] = 100;
```

Для встановлення розміру масиву використовують число, константу або змінну. Установити розмір можна також наданням значень елементам масиву при утворенні масиву:

```
string[] str = new string[]{"1-й", "2-й"};
```

Масиви є класом у C# і володіють певними властивостями та методами. Наприклад,

```
int L = arrInt.Length; //повертає розмірність масиву
int L = arrInt.GetLength(0); //повертає розмірність
        // заданого виміру для багатовимірних масивів
```

До масивів застосовують статичні методи класу `Array`. Наприклад, елементи масиву можна сортувати:

```
Array.Sort(arrInt); //сортувати у порядку зростання
Array.Reverse(arrInt); //змінити напрям сортування
```

Багатовимірні масиви

У C# підтримуються багатовимірні масиви двох типів: прямокутні та ортогональні.

Роботу з прямокутними масивами демонструють такі приклади:

```
int[,] arr2Int = new int[5,10];
string[,] studList = { {"Андрій","Банах"},
                       {"Ірина","Бужська"},
                       {"Семен","Вовк"}
                     };
int[,,] arr3Int = new int[5,10,5];
arr3Int[0,0,0] = 100;
```

В ортогональних масивах кожен вимір може мати свою довжину:

```
int[][] a = new int[3][];
a[0] = new int[5];
a[1] = new int[3];
a[2] = new int[10];
```

Типи внутрішніх масивів не обов'язково збігаються з оголошеним:

```
int [][,] b = new int[3][,];
b[0] = new int[5,2];
```

На відміну від прямокутних масивів кожен індекс в ортогональних масивах виокремлено набором квадратних дужок:

```
a[0][0] = 100;
b[0][0,0] = 100;
```

Перетворення типів

C# дає змогу присвоїти значення змінних одного типу змінним деяких інших типів. Це присвоєння може бути явним або неявним.

Виконати неявні перетворення для цілих типів можна лише тоді, коли перетворюються цілі у більш крупніші цілі або цілі без знака в цілі зі знаком такого ж розміру. В інших випадках компілятор генерує помилку. Наприклад,

```
byte b1 = 1;
byte b2 = 2;
byte b = b1 + b2;   //помилка компіляції
int i = b1 + b2;    //коректне неявне перетворення
```

Довільне ціле можна перетворювати в типи `float`, `double` та `decimal`. Однак можлива втрата точності для перетворень із `int`, `uint`, `long` або `ulong` до `float` та з `long` або `ulong` до `double`.

Дозволеними є також неявні перетворення з типу `float` у тип `double` та з типу `char` до `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` і `decimal`.

Явне перетворення типів використовують з метою уникнення помилки компіляції при неявному перетворенні. Типовий синтаксис:

```
long l = 1000;
int  i = (int)l;
```

Якщо значення, яке явно перетворюється, лежить за межами діапазону значень типу призначення, то помилка не виникає, а результат перетворення залежатиме від конкретного типу. Для перевірки коректності перетворення використовують оператор `checked`:

```
int i = checked((int)l);
```

Якщо значення `l` виходить за межі значень типу `int`, то оператор `checked` згенерує виняткову ситуацію (переповнення).

Типи за значенням допускають перетворення лише у числові типи, типи `enum` і тип `char`.

Не можна безпосередньо перетворити тип `bool` до довільного іншого, і навпаки.

Тип стрічки перетворюється в інші типи (і навпаки) за допомогою відповідних методів .NET. Оскільки кожен клас C# є нащадком класу `object`, то він успадковує (або має власну реалізацію) метод `ToString()`:

```
int i = 1;   string s = i.ToString();
```

Для переводу стрічкового значення у числове або значення типу `bool` використовують метод `Parse`:

```
string s = "1";   int i = Int32.Parse(s);
```

Широкий набір методів перетворення типів надає також клас `System.Convert`.

Упакування та розпакування

Упакування (boxing) та розпакування дає змогу перетворювати типи за значенням у типи за посиланням, і навпаки.

Приклад упакування: `int i = 10;  object obj = i;`

Приклад розпакування: `int j = (int)obj;`

Розпаковувати можна лише змінну, яка попередньо була упакована. Змінна, у яку розпаковують об'єкт, повинна мати достатній розмір для розташування усіх байтів змінної, яку розпаковують.

Змінні та константи

Змінні

Змінні оголошуються у C# із використанням такого синтаксису:

```
[модифікатори] тип_даних ідентифікатор;
```

Якщо в одному виразі оголошуються та ініціалізуються кілька змінних, то всі вони матимуть однаковий тип і модифікатори.

Змінні з однаковими іменами не можуть оголошуватися двічі в одній області видимості.

Ідентифікатори чутливі до регістру символів. Ідентифікатори повинні починатися літерою або символом "_". Ідентифікаторами не можуть бути ключові слова C# (76 слів).

Якщо необхідно використати ключове слово як ідентифікатор, то перед ним ставлять символ `@`. Цей символ дає вказівку компілятору сприймати такі символи як ідентифікатор, а не ключове слово.

Ідентифікатори можна записувати і з допомогою символів Unicode `\uxxxx`. Наприклад, `\u050fidentifier`.

Змінні з однаковими ідентифікаторами можна використовувати за умови їхнього розташування у різних областях видимості:

```
for (int i = 0; i < 10; i++) {…};
for (int i = 0; i < 50; i++) {…};
```

Дві змінні з однаковими ідентифікаторами можна використовувати у спільній області видимості лише у випадку, коли одна змінна – поле класу, а друга – локальна змінна, оголошена в методі класу. Наприклад:

```
public class A {
  private int j = 10;
  public int Method() {
    int j = 20;
    return j + this.j;
  }
}
```

Ключове слово `this` вказує, що елемент (змінна `j`) належить поточному екземпляру класу.

Модифікатори змінних

Модифікатори змінних описують ті чи інші особливості оголошуваних змінних. Перелічимо модифікатори змінних: `internal`, `new`, `private`, `protected`, `public`, `readonly`, `static`, `const`.

Модифікатори застосовують лише до полів (членів класів), однак не до локальних змінних. Змінна може мати декілька модифікаторів.

Модифікатор `new` використовують лише в класах, які є похідними від іншого, поля якого потрібно приховати.

Такі модифікатори утворюють групу модифікаторів доступу:

- `public` – змінна доступна скрізь як поле типу, якому вона належить;
- `internal` – змінна доступна лише поточному складеному модулю;

- `protected` – доступ до змінної відкритий лише з класу, якому вона належить, або з похідних класів;
- `protected internal` – те ж саме, що й `protected`, однак доступ лише з поточного складеного модуля;
- `private` – доступ до змінної лише з класу, у якому її оголошено.

За замовчуванням, поле є членом екземпляра класу. Тобто кожен екземпляр класу має власне поле (виділену ділянку пам'яті) із відповідним ідентифікатором. Якщо до оголошення поля додати модифікатор `static`, то це поле буде єдиним для всіх екземплярів класу.

Модифікатор `readonly` робить змінну доступною лише для читання. Її значення можна встановити при першому оголошенні (для статичних змінних) або в конструкторі для типу, до якого вона належить (для нестатичних змінних).

Константи

Константи – це змінні, описані з використанням модифікатора `const`. Наприклад, `const int a = 10;`

Константи є подібними до статичних полів лише для читання із такими розбіжностями:

- локальні змінні та поля можуть бути оголошені константами;
- константи ініціалізуються в оголошенні;
- значення константи повинно бути обчислюваним під час компіляції, отож константу не можна ініціалізувати виразом із використанням змінної (однак можна використати `readonly` поле);
- константа завжди є статичною.

Операції

C# підтримує такі операції:

Категорія	Операції
Арифметичні	`+ – * / %`
Логічні	`& | ^ ~ && || !`

Конкатенація стрічок	`+`
Інкремент та декремент	`++ --`
Побітовий зсув	`<< >>`
Порівняння	`== != < > <= >=`
Присвоєння	`= += -= *= /= %= |= ^= <<= >>=`
Доступ до членів (для об'єктів)	`.`
Індексування (для масивів та індексаторів)	`[]`
Перетворення типу	`()`
Умовна (тернарна операція)	`?:`
Створення об'єкта	`new`
Інформація про тип	`sizeof is typeof`
Керування винятками переповнення	`checked unchecked`
Розіменування та адресація	`* -> & []`

Зазначимо, що чотири із цих операцій (`sizeof, *, ->, &`) доступні лише в „небезпечному коді" (*unsafe* - код, для якого C# не виконує перевірку безпечності за типом).

Скорочений запис операцій

Операції інкремента та декремента (`++` та `--`) можуть стояти як до, так і після змінної. Вирази `x++` та `++x` еквівалентні виразу `x = x + 1`. Однак префіксний оператор (`++x`) збільшує значення `x` до його використання у виразі. Навпаки, постфіксний оператор (`x++`) збільшує значення `x` після обчислення виразу. Наприклад:

```
bool b1, b2;
int x = 0;
b1 = ++x == 1;    //b1=true
b2 = x++ == 2;    //b2=false
```

Префіксний і постфіксний оператори декремента поводять себе аналогічно, однак зменшують операнд.

Інші скорочені оператори (`+=, -=, *=, /=, %=, |=, ^=, <<=` та `>>=`) вимагають двох операндів. Їх використовують для зміни значення першого операнда шляхом виконання над ним арифметичної, логічної чи бітової операції. Наприклад, оператори `x += y;` та `x = x + y;` є еквівалентними.

Тернарний оператор

Тернарний оператор `?:` є скороченою формою конструкції `if…else`. Його назва вказує на використання трьох операндів. Оператор обчислює умову та повертає одне значення у випадку, якщо умова істинна, та інше - в протилежному випадку. Синтаксис оператора:

умова ? значення_істина : значення_хибність

Наприклад:

```
string s = (x >= 0 ? "Додатне" : "Від'ємне");
```

Оператори checked та unchecked

Використання оператора `checked` ми вже демонстрували, розглядаючи перетворення типів. Зауважимо, що перевірку на переповнення можна увімкнути відразу для всього коду, виконавши компіляцію програми з опцією `/checked`. Власне щодо цього випадку може виникнути потреба відміни перевірки на переповнення деякого коду, для чого й використовують оператор `unchecked`.

Оператор is

Оператор `is` тестує об'єкт на сумісність сумісним із певним типом. Наприклад, можна перевірити, чи сумісна змінна з типом `object`:

```
int i = 1;
string s;
s = (i is object) ? "i має тип object" :
  String.Empty;
```

Зауважимо, що всі типи даних C# успадковують `object`, отож вираз `i is object` дорівнюватиме `true`.

Оператор sizeof

Розмір (у байтах), необхідний для збереження у стеку змінної за значенням, можна визначити з допомогою оператора `sizeof`:

```
int L;
unsafe {
  L = sizeof(double);
}
```

У результаті змінна `L` набуде значення 8.

Зазначимо, що оператор `sizeof` можна використовувати лише в блоках коду, який не є безпечним. За замовчуванням компілятор C# не сприймає такий код. Отож його підтримку необхідно активізувати або використанням опції командної стрічки компілятора `/unsafe`, або встановленням у значення `true` пункту *Allow unsafe code blocks* на сторінці властивостей проекту.

Оператор typeof

Оператор `typeof` повертає об'єкт `Type`, який представляє зазначений тип. Наприклад, `typeof(string)` поверне об'єкт `Type`, який представляє тип `System.String`. Це корисно за умови використання відображення для динамічного отримання інформації про об'єкт.

Пріоритет операцій

Наступна таблиця демонструє порядок дій операцій C#.

Група	**Операції**
	`() . [] x++ x-- new typeof sizeof checked unchecked`
Унарні	`+ - ! ++x --x` та операції приведення типів
Множення / ділення	`* / %`
Додавання / віднімання	`+ -`
Операції побітового зсуву	`<< >>`
Відношення	`< > <= >= is`
Порівняння	`== !=`
Побітовий AND	`&`
Побітовий XOR	`|`
Побітовий OR	`^`
Логічний AND	`&&`
Логічний OR	`||`
Тернарний оператор	`?:`
Присвоєння	`= += -= *= /= %= |= ^= <<= >>=`

Оператори керування

Умовні оператори

C# має два умовні оператори: `if` та `switch`.

Синтаксис оператора `if` такий:

```
if (умова)
  оператор(и)
[else
  оператор(и)]
```

Квадратні дужки позначають необов'язкову частину оператора `if`. *Умова* повинна повертати результат логічного типу: `true` або `false`. Якщо потрібно виконати декілька операторів, то їх об'єднують у блок (розташовують у фігурних дужках {…}).

Оператор `switch…case` призначений для вибору одного з варіантів подальшого виконання програми з декількох. Синтаксис оператора такий:

```
switch (вираз) {
  case константа:
    оператор(и)
    оператор переходу
  [default:
    оператор(и)
    оператор переходу]
}
```

Якщо *вираз* дорівнюватиме значенню однієї з позначок `case`, то виконуватиметься наступний за цією позначкою код. Зауважимо, що цей код не обов'язково оформляти як блок. Однак він повинен завершуватися оператором переходу (зазвичай, це оператор `break`). Єдиний виняток – кілька позначок `case` підряд:

```
switch (s) {
  case "A":
  case "D":
    i = 1;
    break;
  case "C":
```

```
      i = 2;
      break;
    default
      i = 0;
      break;
}
```

Константа може бути константним виразом. Дві позначки `case` не можуть набувати однакового значення.

Цикли

C# реалізує чотири типи циклів.

Цикл `for` має такий синтаксис:

```
for ([ініціалізатор]; [умова]; [ітератор])
  оператор(и)
```

Тут *ініціалізатор* – вираз, який обчислюється до початку виконання циклу (зазвичай, тут ініціалізується локальна змінна, яку використовують як лічильник циклу), *умова* – вираз, який обчислюється перед виконанням кожної ітерації циклу (наприклад, перевірка того, що лічильник циклу менший за деяке значення), *ітератор* – вираз, який виконується після кожної ітерації циклу (наприклад, зміна лічильника циклу).

Кожен (або всі) із цих елементів може бути пропущений.

Цикл `for` є циклом із передумовою. Таким є й цикл `while`:

```
while (умова) оператор(и)
```

Цикл `do…while` є прикладом циклу з постумовою:

```
do оператор(и) while (умова);
```

Оскільки *умова* перевіряється після виконання тіла циклу, то хоча б одна ітерація виконається обов'язково.

Усі розглянуті цикли завершують свої ітерації, якщо *умова* набуде значення `false`.

C# пропонує ще один механізм циклів – `foreach`:

```
foreach (тип ідентифікатор in вираз)
  оператор(и)
```

Тут *тип* – тип ідентифікатора, *ідентифікатор* – змінна циклу, що представляє елемент колекції або масиву, а *вираз* – об'єкт колекції або масив (або вираз над ними).

Цикл `foreach` дає змогу проводити ітерацію по кожному об'єкту в контейнерному класі, який підтримує інтерфейс `IEnumerable`. До контейнерних класів належать масиви C#, класи колекцій у просторі імен `System.Collection` та означені користувачем класи колекцій:

```
int even = 0, odd = 0;
int[] arr = new int [] {0,1,2,5,7,8,11};
  foreach (int i in arr) {
    if (i%2 == 0)
       even++;
    else
       odd++;
   }
```

Зауважимо, що значення об'єкта в колекції всередині циклу `foreach` змінювати не можна.

Оператори переходу

Програма C# може містити позначки – ідентифікатор із двокрапкою. Оператор `goto` дає змогу передати керування стрічці програми з позначкою:

```
goto позначка;
```

Не можна передавати керування у блок коду, за межі класу, а також не можна вийти з блоку `finally`, розташованого після блоків `try…catch`.

Оператор `goto` може використовуватися для переходів усередині оператора `switch`. У цьому випадку синтаксис такий:

```
goto case константа;
goto default;
```

Оператор `break` використовують для виходу з коду позначки в операторі `switch`, а також для виходу із циклів `for`, `foreach`, `while` та `do…while`.

У циклах також можна використовувати оператор `continue`, який перериває поточну ітерацію та ініціює наступну.

Оператор `return` використовують для припинення роботи поточного методу та передачі керування в метод, який його активізував. Якщо метод повертає значення, то `return` повинен повернути значення відповідного типу:

```
return [вираз];
```

Оператор using

Застосування оператора `using` (на відміну від директиви `using`) гарантує, що об'єкти, які інтенсивно використовують ресурси, будуть вивільнені відразу після закінчення роботи з ними. Синтаксис оператора:

```
using (об'єкт) оператор(и)
```

Тут *об'єкт* – це екземпляр класу, який реалізує інтерфейс `IDisposable`. Такі класи повинні містити реалізацію методу `Dispose`, який вивільняє ресурси, зайняті об'єктом класу. Метод `Dispose` викликається відразу після завершення блоку `using`.

Допускається наявність у списку оператора `using` декількох об'єктів:

```
using (Font MyFont = new Font("Arial", 10.0f),
            MyFont2 = new Font("Arial", 10.0f))
  {
  // використання MyFont та MyFont2
  } // компілятор активізує Dispose для MyFont
    // та MyFont2
```

Винятки

Під час виконання програми можливе виникнення помилок: ділення на нуль, вихід за межі діапазону тощо. Такі помилки називають винятком. Код може і явно активізувати виняток. Для коректного опрацювання таких ситуацій використовують блоки `try`, `catch` та `finally`.

У блок `try` розміщають код, який потенційно може спричинити виникнення винятків. Якщо під час виконання якогось

із операторів цього блоку виникає помилка, то такі оператори не виконуються і керування передається на відповідний блок `catch`, розташований за блоком `try`. Якщо ж виняток не виникне, то оператори блоку `catch` не виконуватимуться.

Щоб гарантувати виконання деяких операторів незалежно від того, завершився код винятком чи ні, потрібно оформити блок `finally`.

Типові схеми блоків `try`, `catch` та `finally` такі:

```
try { оператор(и) }
catch { оператор(и) }
finally { оператор(и) }
```

або

```
try { оператор(и) }
catch (тип винятку 1) { оператор(и) }
catch (тип винятку 2) { оператор(и) }
…
finally { оператор(и) }
```

Виняток – це об'єкт класу `System.Exception` або його спеціалізованих нащадків. Наприклад, `catch(Exception e)` перехоплює всі винятки, а `catch(DivideByZeroException e)` – лише ділення на 0. Якщо існує набір конкретних типів винятків і блок `catch(Exception e)`, його розташовують останнім.

Об'єкт `e` містить інформацію про виняток. Можна, наприклад, вивести стрічку `e.ToString()` з повідомленням про помилку або ім'я методу та класу, в якому виникла помилка, за допомогою `e.StackTrace()`.

Досить часто виникає потреба утворити власний об'єкт-виняток і згенерувати виняток. Наступний код демонструє цей процес:

```
Exception myEx = new Exception("myException");
myEx.HelpLink = "Ресурс недоступний.";
myEx.Source = "MyDemoProject";
throw myEx;  //генерування винятку
```

Класи та структури

Алгоритмічна мова C# є цілковито об'єктно-орієнтованою. Програма на C# - це набір класів та маніпулювання ними.

Класи та структури дуже подібні між собою. Головні розбіжності між ними:

- об'єкти класів мають тип даних за посиланням, а структури – за значенням;
- структури не підтримують спадковість;
- для структури не можна оголосити конструктор за замовчуванням;
- для структури не можна оголосити деструктор;
- неможливо використати ініціалізатори для надання значень полям.

Оскільки екземпляри структур розташовані у стеку, доцільно їх використовувати для представлення невеликих об'єктів. Зокрема, клас `Point`, який розглядатимемо нижче, з успіхом можна було б реалізувати як структуру. Вибір класу зумовлено лише бажанням продемонструвати на простій логічній побудові більше понять класу та структури.

Розглянемо поняття класу на такому прикладі:

```
public class Point {
  //статичне поле
  private static uint count = 0;
  //поля
  public double x = 0;
  public double y = 0;
  //властивість лише для читання
  public static uint Count {
    get  {return count;}
  }
  //перекритий метод
  public override string ToString() {
    return "(" + x + "," + y + ")";
  }
  //метод
  public void Set(double x, double y) {
```

```
    this.x = x;
    this.y = y;
   }
  //конструктор без параметрів
  public Point() {
    count++ ;
  }
  //конструктор з параметрами
  public Point(double x, double y) {
    count++ ;
    Set(x,y);
  }
  //конструктор копій
  public Point(Point a) : this(a.x,a.y) {
  }
  //деструктор
  ~Point() {
    count-- ;
  }
```

Зауваження. В .NET Framework 2 з допомогою модифікатора класу `partial` оголошення класу можна розбити на декілька частин, які можна розташувати як в одному, так і в різних файлах.

Члени класу

Дані та функції всередині класу називають членами класу. Дані класу – це *поля*, *константи* та *події*, тобто ті члени класу, які містять дані для класу.

Поле – це довільна змінна, оголошена на рівні класу: `count`, `x`, `y`. Якщо поле оголошене як статичне (`static`), то воно єдине для всього класу і жоден екземпляр об'єкта класу не матиме окремого екземпляра цього поля. У протилежному випадку для кожного екземпляра об'єкта цього класу утворюються власні незалежні екземпляри полів.

Клас `Point` описує точку: кожен екземпляр класу – це точка на площині з координатами `x` та `y`. Статичне поле `count` містить кількість утворених і активних в поточний момент часу екземплярів класу.

Подія – це член класу, який дає змогу об'єкту надіслати повідомлення про початок певної події (зміна значення поля, взаємодія з користувачем і т.п.). Клієнт (код, який використовує об'єкт класу) може містити код обробки події.

Функції класу – це ті члени класу, які забезпечують функціональність для роботи з даними класу. Вони містять *методи*, *властивості*, *конструктори*, *деструктори*, *оператори* та *індексатори*.

Методи

Види функцій класу визначаються синтаксисом оголошення. Синтаксис оголошення методів у C# такий:

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

Модифікатори методу аналогічні модифікаторам змінних. Додатково можна використовувати такі модифікатори:

Модифікатор	Опис
`virtual`	Метод може бути переозначений у дочірньому класі
`abstract`	Віртуальний метод, який визначає сигнатуру методу, однак не містить його реалізації. За наявності абстрактних методів екземпляр класу не може бути утворений
`override`	Метод переозначує успадкований віртуальний або абстрактний метод
`sealed`	Метод переозначує успадкований віртуальний метод і забороняє його переозначення у дочірніх класах. Використовують разом із `override`
`extern`	Метод реалізований поза програмою на іншій мові

Клас `Point` містить перекритий (`override`) метод `ToString` класу `object`.

Для активізації методу потрібно вказати ім'я об'єкта, для якого активізують метод, а після крапки – ім'я методу та список аргументів у дужках:

```
Point a = new Point(1,1);
string s = a.ToString();
```

Для активізації статичного методу потрібно зазначити назву класу, а не ім'я об'єкта. Статичний метод може працювати лише зі статичними полями класу.

Якщо метод активізується всередині класу, то назву класу не використовують.

Аргументи методів можуть передаватися *за посиланням* або *за значенням*.

За замовчуванням параметри передаються за значенням, тобто метод одержує копію значення аргументу. Якщо параметр має тип даних за значенням, то довільні зміни цього параметра всередині методу ніяк не вплинуть на значення аргументу-оригіналу. Якщо ж параметр має тип за посиланням (масив, клас), то метод працюватиме безпосередньо з даними аргументу, оскільки передана копія значення – адреса розташування об'єкта.

Зауважимо, що стрічки не поводять себе як тип за посиланням, отож зміни у стрічці, що відбулися всередині методу, не зачіпають її оригінал.

Якщо потрібно змінні за значенням передати в метод за посиланням, використовують ключове слово `ref` перед типом параметра. Слово `ref` повинно вказуватися і при виклику методу. Довільна модифікація змінної методом викликає відповідні зміни аргументу-оригіналу.

Перед передачею значень методу всі змінні-аргументи повинні бути ініціалізованими. Інакше компілятор C# видасть повідомлення про помилку. Обійти це обмеження можна шляхом використання ключового слова `out` перед типом параметра. Слово `out` необхідно зазначити і при виклику методу. Усередині методу параметрові, позначеному як `out`, необхідно присвоїти значення.

Властивості

Властивості використовують з метою зробити виклик методу подібним на поле. Їх і оголошують подібно до поля. Однак додатково після оголошення у фігурних дужках розташовують блок коду для контролю даних та реалізації потрібної функціональності. Цей блок може мати два методи доступу: аксесор `get` та аксесор `set`.

Клас `Point` має властивість `Count`, яка повертає кількість утворених екземплярів класу. Розширимо опис цієї властивості:

```
public static uint Count {
  get  {return count;}
  set  {if (value >= 0) count = value;}
}
```

Зауважимо, що аксесору `set` неявно передається параметр з іменем `value` такого ж типу, як і властивість. Аналогічно, значення типу властивості повинен повертати аксесор `get`.

Оскільки властивість подібна до поля, то й активізацію її здійснюють подібно:

```
uint cnt = Point.Count;     // аксесор get
Point.Count = cnt;          // аксесор set
```

Якщо властивість містить лише аксесор `get`, то її використовують тільки для читання значення. Якщо властивість містить лише аксесор `set`, то її використовують тільки для запису значення.

Конструктори

Конструктори – це методи класу, які використовуються разом з оператором `new` для утворення об'єкта.

Якщо клас не містить власного конструктора, то компілятор утворить конструктор за замовчуванням (без параметрів).

Конструкторів може бути кілька. Вони відрізнятимуться кількістю і типом параметрів, однак матимуть одну й ту ж назву – ім'я класу. Клас `Point` містить три конструктори. Перший із них збільшує лічильник утворених об'єктів, а другий додатково ініціалізує поля `x` та `y`. Третій конструктор `public Point(Point a)` належить до так званих *конструкторів копій*, оскільки ініціалізує екземпляр класу на основі значень іншого екземпляра цього ж класу.

Для утворення об'єкта можна використати довільний з конструкторів класу, проте лише один.

Конструктор не може повертати значення, отож тип результату не вказують при визначенні конструктора.

Якщо конструктор оголошений як `private`, то його можна використовувати лише всередині класу, а якщо як `protected` – то в класах-нащадках. Обмеження доступу до конструктора може бути корисним, наприклад, для методів `Clone()`, `Copy()` або для аналогічних методів класу, яким потрібно утворювати інші екземпляри цього класу. Очевидно, що клас повинен мати хоча б один конструктор з доступом `public`, якщо передбачається утворення об'єктів цього класу клієнтським кодом.

Клас може також мати статичний конструктор без параметрів. Такий конструктор буде використано лише один раз. Його можна використати для ініціалізації статичних змінних. Наприклад,

```
static Point() {
  count = 0; }
```

Статичний конструктор не має модифікатора доступу, оскільки він активізується лише середовищем .NET при завантаженні класу.

Зауважимо, що клас може водночас мати конструктор екземплярів без параметрів і статичний конструктор. Це єдиний випадок, коли може бути два методи класу з однаковими назвами та однаковим списком параметрів.

Конструктор може викликати інший конструктор цього ж класу. Для цього випадку існує спеціальний синтаксис:

```
public Point()
  : this(0,0)
{
 // додатковий код
}
```

Такий синтаксис повідомляє компілятору, що у випадку використання конструктора спочатку потрібно виконати інший конструктор цього класу з переданими йому аргументами після ключового слова `this`, а потім виконати код (якщо є) зазначеного конструктора.

Ключове слово `this` вказує, що використовують елемент поточного класу. Якщо ми використаємо ключове слово `base`, то

дамо вказівку шукати відповідний елемент у базовому класі (клас-предок).

Деструктори

Деструктор класу використовують для виконання дій, необхідних при знищенні екземпляра класу. У нашому прикладі класу `Point` деструктор зменшує лічильник екземплярів класу.

Як і конструктор, деструктор має те ж ім'я, що й клас, однак з префіксом тильда (~): `~Point`. Деструктор не має параметрів та не повертає результат.

Явним чином деструктори у C# не викликають. Виконання коду деструктора ініціюється механізмом прибирання „сміття". Отож не можна передбачити, коли буде виконано код деструктора.

Ініціювати негайне прибирання „сміття" можна з допомогою методу Collect() об'єкта .NET System.GC, який реалізує „прибиральника". Однак цим доцільно користуватися лише у випадку, коли ви впевнені в необхідності такого кроку.

Якщо при знищенні екземпляра класу необхідно негайно звільнити ресурси, зайняті екземпляром класу (наприклад, закрити файл) або надіслати повідомлення іншим об'єктам, доцільно утворити спеціальні методи. Типові назви таких методів – `Close` та `Dispose`. Клієнтський код повинен явно активізувати ці методи. І це є недоліком такого підходу.

Інший варіант звільнення ресурсів – використання оператора `using`. У цьому випадку клас повинен успадковувати інтерфейс `IDisposable`, означений у просторі імен `System`:

```
class Point : IDisposable {
// - - -
  public void Dispose() {
    // код
  }
}
```

Перевантаження операцій

Нехай задано точки $A(x_1,y_1)$ та $B(x_2,y_2)$. Точку $C(x,y)$ назвемо сумою точок *A* та B, якщо $x= x_1+ x_2$, $y= y_1+ y_2$.

Для додавання точок у класі `Point` можна утворити метод. Наприклад:

```
public Point Add(Point p) {//код};
```

Тоді додавання виглядатиме так:

```
C = A.Add(B);
```

Якщо потрібно додати декілька точок, то вираз ускладниться. Значно зручніше використовувати звичний нам синтаксис для простих типів:

```
C = A + B;
```

C# дає змогу перевантажувати операції. Наприклад, до означення класу `Point` можна додати такий код:

```
public static Point operator + (Point p1, Point p2)
{
  return new Point(p1.x + p2.x, p1.y + p2.y);
}
```

Операція оголошується аналогічно методу, за винятком того, що замість імені методу пишуть ключове слово `operator` і знак операції. Тепер, якщо `A`, `B` та `C` мають тип `Point`, то ми можемо записати: `C = A + B;`

Перевантажимо тепер операцію множення на число.

```
public static Point operator * (double a, Point p)
{
  return new Point(a * p.x, a * p.y);
}
public static Point operator * (Point p, double a)
{
  return a*p;
}
```

Для операції множення ми утворили два варіанти перевантаження, щоб компілятор коректно сприймав, наприклад, код `10*A` та `A*10`.

Зауважимо, що перевантаження операцій +, –, * та / використовуються компілятором для реалізації операцій +=, –=, *= та /= відповідно.

У C# є шість операторів порівняння, які утворюють три пари:

== та != > та <= < та >=

C# вимагає перевантаження операцій порівняння тільки парами. Окрім цього, у випадку перевантаження == та != потрібно також перекрити метод `Equals()`, успадкований від `System.Object`.

Наведемо приклад перевантаження операцій порівняння:

```
public static bool operator == (Point a, Point b){
  return a.x == b.x && a.y == b.y ? true : false;
}
public static bool operator != (Point a, Point b){
  return !(a == b);
}
public override bool Equals(object obj) {
  return (obj is Point) && (this == (Point)obj);
}
public override int GetHashCode() {
  return ToString().GetHashCode();
}
```

У цьому коді перекривається метод `GetHashCode` класу `object`. Поки що цей метод не є предметом нашого розгляду. Його наведено з метою уникнення повідомлення від компілятора, що при перекритті методу `Equals` потрібно також перекрити `GetHashCode`.

C# дає змогу перевантажувати лише такі операції:

Категорія	Операції
Арифметичні	+ – * / % ++ --
Бітові	<< >> & | ^ ! ~ true false
Порівняння	== != < >= <= >

Перевантаження методів

Перевантаження методів використовують у тому випадку, коли потрібно, щоб клас виконував деякі дії, але при цьому

існувало кілька способів передачі інформації методу, який виконує завдання.

Ми вже демонстрували перевантаження методів на прикладі конструкторів класу `Point` – одна назва за різних наборів параметрів.

Наведемо ще один приклад – перевантаження методу `ToString`:

```
public string ToString(string format) {
  return String.Format(format,x,y);
}
```

Тепер можна записати код:

```
Point p = new Point(1,1);
string s = p.ToString(); //s = "x=1 y=1"
s = p.ToString("x:{0} y:{1}", x, y);
  //s = "x:1 y:1"
```

Кількість перевантажених методів не обмежена. Тобто клас може містити багато методів з одним іменем, однак вони повинні вирізнятися кількістю, порядком або типом параметрів. Очевидно, що не доцільно давати однакове ім'я методам, які виконують зовсім різні задачі.

Якщо базовий клас уже містить метод із заданим іменем, а кількість, тип і порядок параметрів збігаються, то можливі два варіанта:

- якщо метод віртуальний, його можна перекрити (механізм перекривання розглянуто у розділі „Похідні класи");
- використати модифікатор `new` в оголошенні методу.

Приклад використання об'єктів класу

Розглянемо клас, який реалізує функціональність роботи з масивом точок. Наведемо код початкового варіанта такого класу:

```
public class Points {
  private readonly uint count;
  protected Point[] points;
  public uint Count {
```

```
    get { return count; }
  }
  public override string ToString() {
     string Result = "";
     for (int i = 0; i < count; i++)
       Result += points[i] + " ";
     return Result;
  }
  public Points(uint count) {
    this.count = count;
    points = new Point[count];
    for (int i = 0; i < count; i++)
      points[i] = new Point();
  }
  public Points(Points pts): this(pts.count) {
    for (int i = 0; i < count; i++)
      points[i] = pts.points[i];
  }
}
```

Клас `Points` містить масив точок `points` (елементів типу `Point`). Розмірність масиву задається параметром конструктора і встановлюється при утворенні об'єкта класу:

```
points = new Point[count];
```

Зверніть увагу, що цей код виокремить пам'ять для розташування `count` вказівників на об'єкти `Point`, а не власне об'єктів. Тим більше, що самих об'єктів ще не існує. Їх утворюють такі дві стрічки:

```
for (int i = 0; i < count; i++)
  points[i] = new Point();
```

Властивість `Count` повертає значення розмірності. Оскільки клас містить конструктори, то конструктор за замовчуванням не утворюється. Отож для утворення об'єкта класу можна використати або конструктор з параметром, який задає розмірність, або конструктор копій.

Індексатори

Індексатори дають змогу здійснювати доступ до об'єкта так, ніби він є масивом. Індексатори означуються приблизно так, як властивості – з використанням функцій `get` та `set`. Однак замість імені індексатора використовують ключове слово `this`.

Якщо `ps` – об'єкт типу `Points`, то для доступу до точки з індексом 0 ми повинні використовувати синтаксис: `ps.points[0]`. Значно елегантніше було б застосувати `ps[0]`, однак для цього потрібно додати індексатор.

Щоб оголосити індексатор для класу `Points`, додамо до його опису такий код:

```
public Points this[int i] {
  get {
    if (i >= 0 && i < count)
      return points[i];
    else
      throw new IndexOutOfRangeException(
      "Вихід за допустимий діапазон індексів"+ i);
    }
  set {
    if (i >= 0 && i < count)
      points[i]=value;
    else
      throw new IndexOutOfRangeException(
      "Вихід за допустимий діапазон індексів"+ i);
    }
  }
```

Тепер для змінної `ps` типу `Points` ми можемо використати код:

```
string s = "x0=" + ps[0].x + "  y0=" + ps[0].y;
```

Індексатори не є обмежені одномірними масивами та цілочисельними індексами. Наприклад, допустимим є такий код:

```
    public bool this[int i, string s] {
      get {
        switch (i) {
```

```
        case 0:
          switch (s) {
            case "AA": return true;
            default: return false;
          }
          break;
- - -
```

Для індексаторів можна застосовувати цикли `for`, `do` та `while`, однак не можна написати цикл `foreach`, оскільки він працює лише з колекціями, а не з масивами.

Інтерфейси

Інтерфейс – це список оголошень методів, властивостей, подій та індексаторів. Оголошення інтерфейсу подібне до класу, однак не містить модифікаторів доступу для членів і реалізацій. Інтерфейс не може мати конструкторів. Отож об'єкт інтерфейсу не можна утворити.

Наприклад, інтерфейс `IEnumerator` із простору імен `System.Collections` оголошений так:

```
interface IEnumerator {
  //властивість
  object Current {get;}
  //методи
  bool MoveNext();
  void Reset();
}
```

Кажуть, що клас підтримує інтерфейс, якщо він містить реалізацію усіх оголошень інтерфейсу. Зокрема, клас підтримує інтерфейс `IEnumerator`, якщо він містить реалізацію властивості `Current` і методів `MoveNext` і `Reset`.

За домовленістю назва інтерфейсу починається літерою `I`.

У попередньому пункті ми оголосили індексатор для класу `Points` і зазначили, що до нього не можна застосувати цикл `foreach`. Для того щоб клас `Points` підтримував колекції, він повинен виконати наперед оголошену домовленість: містити метод

з назвою `GetEnumerator`, який повертає об'єкт деякого класу з підтримкою інтерфейсу `IEnumerator`. Це правило формалізується інтерфейсом `IEnumerable`, оголошеним у просторі імен `System.Collections` так:

```
public interface IEnumerable {
  IEnumerator GetEnumerator();
}
```

Перед тим як додати підтримку цього інтерфейсу до класу `Points`, утворимо допоміжний клас `PointsEnum`:

```
class PointsEnum : IEnumerator {
  int location = -1;
  Points points;
  //конструктор класу
  public PointsEnum(Points points) {
    this.points = points;
    location = -1;
  }
  //реалізація членів інтерфейсу IEnumerator
  public object Current {
    get {
     if (location < 0 || location >= points.Count)
        throw new InvalidOperationException(
          "Некоректний індекс");
      return points[location];
    }
  }
  public bool MoveNext() {
   ++location;
   return (location >= points.Count) ? false:true;
  }
  public void Reset() {
    location = -1;
  }
}
```

Код `class PointsEnum : IEnumerator` вказує, що клас підтримує інтерфейс `IEnumerator`, тобто містить реалізацію

його членів. Якщо необхідно, щоб клас підтримував декілька інтерфейсів, то після двокрапки треба перелічити назви цих інтерфейсів, розділені комами. І, відповідно, реалізувати всі члени цих інтерфейсів.

Клас `PointsEnum` працює з об'єктом типу `Points`, який передається параметром конструктора.

Додамо тепер до класу `Points` підтримку інтерфейсу `IEnumerable`:

```
public class Points : IEnumerable {
- - -
  public IEnumerator GetEnumerator() {
    return new PointsEnum(this);
  }
}
```

Код `new PointsEnum(this)` утворює об'єкт типу `PointsEnum`, а метод повертає значення типу `IEnumerator`. Оскільки клас `PointsEnum` підтримує цей інтерфейс, то протиріччя тут не буде.

Тепер компілятор не заперечуватиме проти використання `foreach`:

```
Points q = new Points(5);
string s = "";
foreach (Point p in q)
  s += p.ToString();
```

Зауваження. Компілятор C# для версії .NET Framework 2 роботу щодо утворення допоміжного класу з інтерфейсом `IEnumerator` виконує самостійно. Для підтримки колекції класом `Points` клас `PointsEnum` можна не утворювати, а метод `GetEnumerator` реалізувати приблизно так:

```
public IEnumerator GetEnumerator() {
  for (int i = 0; i < points.Count; i++)
    yield return points[i]
}
```

Інтерфейс може успадковувати один або декілька інших. Наприклад:

```
interface IInterface3 : IInterface1, IInterface2
{
– – –
}
```

У цьому випадку клас, який підтримує інтерфейс `IInterface3`, повинен містити реалізацію всіх членів успадкованих інтерфейсів `IInterface1` та `IInterface2`.

Похідні класи

Клас – це базовий інструмент *об'єктно-орієнтованого програмування* (ООП). Ми розглянули поняття класу та деякі його елементи. Зокрема, *інкапсуляцію* (об'єднання даних і методів їхньої обробки).

У цьому розділі предметом розгляду будуть дві інші характеристики ООП: *успадкування* та *поліморфізм*. Зазначимо, що структури не підтримують успадкування та поліморфізм.

Успадкування класів

Побудуємо клас, який представлятиме геометричний трикутник. Трикутник визначається трьома точками на площині. Використаємо клас `Points`, який дає змогу будувати множину точок і проводити деякі дії над ними.

```
public class Triangle : Points {
  public Triangle(double x1, double y1, double
x2,
          double y2, double x3, double y3)
:base(3)
  {
    points[0].Set(x1, y1);
    points[1].Set(x2, y2);
    points[2].Set(x3, y3);
  }
  public Triangle(Point p1, Point p2, Point p3):
                                          base(3)
  {
```

```
    points[0] = p1;
    points[1] = p2;
    points[2] = p3;
  }
  public override string ToString() {
    return "трикутник " + base.ToString();
  }
}
```

Код `class Triangle:Points` оголошує, що клас `Triangle` успадковує клас `Points`. Це означає, що `Triangle` має всі компоненти класу `Points`: поля `count`, `points`, метод `ToString` та індексатор. Окрім того, оскільки клас `Points` успадковує клас `object` (як і всі типи C#), то клас `Triangle` має також усі компоненти класу `object`.

Класи `object` і `Points` є класами-предками для класу `Triangle`.

Клас `Points` (клас, який зазначено після двокрапки в оголошенні нового класу) називають *базовим* або *батьківським* класом для класу `Triangle`.

Клас `Triangle` називають *похідним* або *дочірнім* для класу `Points`. А для класів `object` і `Points` він буде *нащадком*.

Похідний клас може безпосередньо використовувати всі члени базового класу, якщо вони означені з модифікаторами `protected` або `public`.

Однак похідний клас не наслідує конструкторів базового класу (проте може використати, як це зроблено в нашому прикладі). Єдиний виняток – це конструктор за замовчуванням, який викликається конструктором за замовчуванням похідного класу.

Якщо похідний клас наслідує всі члени предків, то об’єкт цього класу містить підмножину, яку можна розглядати як об’єкт деякого класу-предка. Наприклад:

```
Triangle T = new Triangle(1,1,2,2,3,3);
Points P = (Points)T;
object obj = (object)T;
IEnumerable ienum = (IEnumerable)T;
```

Приховування компонентів

C# дає змогу замінити члени базового класу в нащадках. Розглянемо такі два класи:

```
public class A {
  public int x = 0; }
public class B: A {
  public int x = 1; }
```

Клас `B` успадковує клас `A`, отже – і поле `x`. Однак у класі `B` оголошене нове поле з ідентичним іменем. У цьому випадку компілятор не є упевненим, що він розуміє логіку програміста, отож видасть повідомлення щодо своїх сумнівів. Проте код буде все ж скомпільовано.

Надання новим членам похідного класу імен, вже використаних у базовому – потенційна небезпека помилок. Однак інколи така потреба виникає. У цьому випадку потрібно приховати компонент `x` класу `A`, оголосивши явно компонент `x` класу `B` новим за допомогою ключового слова `new`:

```
public class B: A {
  public new int x = 1; }
```

Для об'єкта класу `B` можемо отримати значення обох компонентів `x`:

```
B b = new B();
int bx = b.x;  //bx набуде значення 1
int ax = ((A)b).x;  //ax набуде значення 0
```

Приховування методів може стати необхідним у випадку конфлікту версій базового класу. Припустимо, що програміст *A* розробив базовий клас `A`, а програміст *B* на основі класу `A` – клас `B`, у який додав новий метод з назвою `M`. Через деякий час *A* дописує в класі `A` новий метод з назвою `M` та публікує нову версію. Після перекомпілювання програми результат виконання програми може бути не таким, як очікував *B*.

Оскільки компілятор C# відстежує такі ситуації, то він видасть відповідне повідомлення. Програміст *B* має два варіанта дій. Якщо він контролює усі класи, породжені від класу `B`, то краще

перейменувати свій метод M. Якщо ж клас B опублікований для використання іншими користувачами, то до оголошення методу M необхідно додати модифікатор new.

Абстрактні методи

Нехай проектується деякий клас A. Вважають, що всі його дочірні класи повинні мати деякий метод M. Однак на рівні класу A недостатньо інформації для змістовного означення цього методу. Якщо існує необхідність присутності методу M у класі A, то цей метод можна оголосити з модифікатором abstract без реалізації. У цьому випадку метод M називатиметься *абстрактним*. Якщо клас містить абстрактні методи, то він повинен також містити модифікатор abstract. Наприклад:

```
abstract public class A {
– – –
  abstract public void M();
}
```

Зауважимо також:

- неможливо утворити об'єкт абстрактного класу;
- неможливо оголосити конструктор абстрактним методом;
- абстрактні класи використовуються для породження інших класів;
- дочірні класи (якщо вони не є також абстрактними) зобов'язані містити реалізацію усіх абстрактних методів, успадкованих від базового класу.

Віртуальні методи

Продовжимо розгляд класу Points. Об'єкт цього класу містить індексовану множину точок. Ці точки можна розглядати як вузли ламаної лінії. Тоді можна ввести поняття довжини та оголосити метод GetLength, який повертає цю довжину.

Похідний від Points клас Triangle успадкує цей метод GetLength. Однак для трикутника довжина – це периметр. А успадкований GetLength не враховує в довжині пряму, що з'єднує останню точку з першою.

Якщо може виникнути потреба у зміні реалізації деякого методу в дочірніх класах, його потрібно оголошувати *віртуальним*. З цією метою використовують модифікатор `virtual`.

Додамо метод `GetLength` до класу `Points`:

```
public class Points : IEnumerable {
  - - -
  public double GetDistance(Point p1, Point p2){
    return Math.Sqrt(
      (p1.x - p2.x) * (p1.x - p2.x) +
      (p1.y - p2.y) * (p1.y - p2.y));
  }
  public virtual double GetLength() {
    double length = 0;
    for (int i = 0; i < count - 1; i++)
     length +=
GetDistance(Points[i],Points[i+1]);
    return length;
  }
}
```

Метод `GetLength` тут оголошено віртуальним, оскільки поняття довжини може змінюватися в класах-нащадках. А от відстань між точками навряд чи потребуватиме переозначення. Тому метод `GetDistance` не описано як віртуальний.

Перекривання віртуальних методів

Ми вже розглядали перекривання віртуальних методів у класі `Point`, де перекривається успадкований від `object` віртуальний метод `ToString`:

```
public override string ToString()
```

У свою чергу в класі `Points` з таким самим синтаксисом перекривається успадкований уже від `Point` віртуальний метод `ToString`.

Щоб перекрити віртуальний метод, означений у базовому класі, необхідно в похідному класі повторити оголошення методу, але модифікатор `virtual` замінити на модифікатор `override`.

Очевидно, що потрібно також написати нову реалізацію методу. Наприклад:

```
public class Triangle : Points {
  – – –
  public override double GetLength() {
   return base.GetLength() +
    GetDistance(points[Count-1], points[0]);
  }
}
```

C# має модифікатор `sealed`, який використовують в парі з `override` і який дає вказівку заборонити перекривання методу в дочірніх класах – *запечатує*.

Поліморфізм

Механізм віртуальних функцій реалізує концепцію поліморфізму об'єктно-орієнтованого програмування.

Розглянемо такі два класи:

```
public class A {
  public string Method() { return "A.Method"; }
  public virtual string VirtualMethod() {
    return "A.VirtualMethod"; }
}
public class B: A {
  public new string Method() { return "B.Method"; }
  public override string VirtualMethod() {
    return "B.VirtualMethod"; }
}
```

Клас `B` приховує успадкований метод `Method`, оголосивши новий з ідентичним іменем. А віртуальний метод `VirtualMethod` клас `A` перекриває.

Оголосимо змінні:

```
A a = new A();
B b = new B();
A x = b;
```

Змінні a та b містять адреси утворених екземплярів, відповідно, класів A та B. Змінна x містить адресу того ж об'єкта, що й b, але має тип класу A. Наступний код демонструє особливості віртуальних функцій:

```
string s;
s = a.Method(); //"A.Method"
s = b.Method(); //"B.Method"
s = x.Method(); //"A.Method"
s = a.VirtualMethod(); //"A.VirtualMethod"
s = b.VirtualMethod(); //"B.VirtualMethod"
s = x.VirtualMethod(); //"B.VirtualMethod"
s = ((A)b).VirtualMethod();
                        //"B.VirtualMethod"
```

Якщо метод не є віртуальним, компілятор використовує той тип, який змінна мала при оголошенні. У нашому випадку x має тип A. Отож код x.Method() викличе метод класу A, хоча реально x є посиланням на об'єкт класу B.

Якщо метод є віртуальним, компілятор згенерує код, який під час виконання перевірятиме, куди насправді вказує посилання, і використовуватиме методи відповідного класу. Хоча x має тип A, викликається метод VirtualMethod класу B. Окрім того, навіть явне приведення типу до A ситуацію не змінює.

Використаємо описану властивість поліморфізму для означення функції, яка повертає довжину об'єкта класу Points або Triangle.

```
public double PointsLength(Points points) {
  return points.GetLength();
}
```

Оскільки метод GetLength є віртуальним у класах Points та Triangle, то функція PointsLength повертатиме коректні значення довжини для об'єктів різних типів:

```
Points Ps = new Points(3);
Ps[0].Set(0, 0);
Ps[1].Set(0, 3);
Ps[2].Set(4, 0);
```

```
Triangle T = new Triangle(Ps[0], Ps[1],
Ps[2]);
double pl = PointsLength(Ps); // pl = 8
double tl = PointsLength(T); // tl = 12
```

ДОДАТКОВІ МОЖЛИВОСТІ C#

Вказівники

Незахищений код

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 | Properties | Build | 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 = &complex;
```

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 значення 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 `k` є довільним числовим типом, то вираз `p[k]` завжди інтерпретується як `*(p+k)`. Наприклад, останню стрічку коду можна записати так:

```
pDoubles[k] = 5;
```

Зазначимо, що, на відміну від звичайних масивів, ця стрічка не ініціює виняток, якщо `k` 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 невірні результати. Недарма такий код необхідно свідомо оголосити небезпечним.

Делегати

Утворення та використання делегатів

Делегати подібні до вказівників на функції. Їх можна використати для виклику різноманітних функцій з однаковою сигнатурою під час виконання програми. Сигнатура функції – це список типів параметрів і результату. Синтаксис оголошення делегатів подібний до оголошення функції з доданим ключовим словом `delegate`:

[*модифікатори*] `delegate` *тип_результату* НазваДелегата([*параметри*])

Для прикладу, схематично розглянемо клас:

```
public class ClassA {
  public static double M1(int i) {…;}
  public double M2(int i) {…;}
}
```

Методи M1 та M2 мають однакову сигнатуру. Опишемо делегата для цих функцій:

```
public delegate double DelegateM(int i);
```

Тепер наведемо приклад використання делегата:

```
DelegateM delegateM = new DelegateM(ClassA.M1);
double m1 = delegateM(10);
ClassA A = new ClassA();
delegateM = new DelegateM(A.M2);
double m2 = delegateM(10);
```

Зверніть увагу на схожість делегата на вказівник. У першій стрічці делегату надається адреса статичного методу M1 (назві методу передує назва класу). Тому в другій стрічці активізація delegateM(10) ініціює власне виклик методу ClassA.M1.

Далі код delegateM = new DelegateM(A.M2) надає делегату адресу методу M2 об'єкта A (назві методу передує назва об'єкта). Тому в останній стрічці активізація delegateM(10) ініціює виклик методу A.M2.

Зауважимо, що методи, на які посилається делегат, не обов'язково повинні належати одному класу.

Багатоадресні делегати

З допомогою делегата можна викликати декілька методів. При цьому на делегата та методи накладається додаткове обмеження: і методи, і делегат повинні повертати тип void.

Для прикладу розглянемо такий код:

```
public delegate void DelegateM(int i);
public class ClassA {
  public static void M1(int i) {…;}
  public void M2(int i) {…;}
}
// тут деякий код
ClassA A = new ClassA();
DelegateM delegateM = new
                    DelegateM(ClassA.M1);
delegateM += new DelegateM(A.M2);
```

```
delegateM(10);
```

Як бачимо, об'єкти-делегати можна додавати, а також віднімати. Тобто застосовувати до них оператори +, –, +=, –=.

Виклик делегата `delegateM(10)` ініціює послідовні виклики всіх належних йому методів (`ClassA.M1` та `A.M2`), параметром яких у нашому випадку буде число 10.

Простий приклад використання делегатів

У попередніх прикладах проілюстровано механізм утворення та використання делегатів. Однак не зовсім очевидна доцільність використання делегатів, оскільки з тим самим результатом можна було б обмежитися простим викликом двох методів `ClassA.M1(10)` та `A.M2(10)`.

Наведемо простий приклад корисності використання делегатів.

Наша задача полягає в написанні функції, яка повертає максимальний елемент із заданого одновимірного масиву. Причому бажано передбачити використання цієї функції для масивів довільного типу. Алгоритм пошуку максимального елемента простий: пройти всі індекси масиву, і на кожній ітерації шляхом порівняння між поточним елементом і досягнутим рекордом обирати новий рекорд. Однак тут виникає проблема: ми не знаємо нічого про тип елементів, отож не вміємо їх порівнювати.

Вирішимо цю проблему таким способом. Оскільки клієнтському коду відомо, з яким типом даних він хоче працювати, він може передати цю інформацію (правило порівняння елементів) функції пошуку максимуму через делегата. Означимо цього делегата так:

```
public delegate bool Compare(object obj1,
                             object obj2);
```

Необхідно, щоб функція, на яку посилається делегат, повертала значення `true`, якщо перший аргумент більший за другий, і `false` - у протилежному випадку.

Метод `Max` матиме таку реалізацію:

```
static public object Max(
             object[] objs, Compare cmp) {
  if (objs.Length == 0)
    return null;
  else {
    object record = objs[0];
    for (int i = 1; i < objs.Length; i++)
      if (cmp(objs[i], record))
        record = objs[i];
    return record;
  }
}
```

Раніше нами означено демонстраційний клас множини точок `Points`. Вважатимемо більшим той об'єкт `Points`, у якого довжина є більшою. Додамо до класу `Points` метод порівняння:

```
public static bool PointsCompare(
                    object l, object r) {
  Points L = (Points)l;
  Points R = (Points)r;
  return L.GetLength() > R.GetLength() ?
                                  true : false;
}
```

Тепер усе готово для пошуку найбільшого об'єкта класу `Points` методом `Max`:

```
int k = 10;
Points[] points = new Points[k];
//тут утворюються та ініціалізуються
//об'єкти масиву points
//…
//готуємо параметр для методу Max
Compare delegateCmp= new
Compare(Points.PointsCompare);
//і одержуємо максимальний елемент
Points Ps = (Points)Max(points, delegateCmp);
```

Події

Події дають змогу одному об'єкту інформувати інші про те, що щось відбулося. Наприклад, при натисканні клавіші на клавіатурі або миші Windows *генерує* подію, інформацію про яку при бажанні можуть отримати зацікавлені об'єкти.

Оголошення та генерування події

Подію оголошують з допомогою ключового слова `event`:

[*модифікатори*] `event` *ім'я-класу-делегата* НазваПодії

За домовленістю імена подій розпочинаються префіксом `On`: `OnClick`, `OnMouseDown` і т.п. При генеруванні події потрібно активізувати метод, який представляє делегат.

Наведемо простий приклад генерування події виходу значення цілочисельної змінної за межі деякого інтервалу.

Означимо клас делегата:

```
public delegate void RangeOutHandler(
  object sender, RangeOutEventArgs e )
```

Усі делегати, які активізують код обробки подій, повинні повертати значення типу `void` та приймати два параметри. Перший параметр має тип `object` і представляє об'єкт, який згенерував подію. Другий параметр – це об'єкт класу, успадкованого від класу `System.EventArgs`. В означенні делегата ми використали клас `RangeOutEventArgs`, який оголосимо так:

```
public class RangeOutEventArgs: EventArgs {
  private string message;
  public RangeOutEventArgs(string message) {
    this.message = message;
  }
  public string Message {
    get { return message;}
  }
}
```

Тепер означимо клас, який може генерувати події:

```
public class RangeControl {
  private int value;
  private int left;
  private int right;
  //оголошення класу делегата
  public delegate void RangeOutHandler(
    object sender, RangeOutEventArgs e);
  //оголошення події OnRangeOut
  public event RangeOutHandler OnRangeOut;
  //конструктор класу
  public RangeControl(int value, int left, int right)
  {
    this.left = left;
    this.right = right;
    Value = value;
  }
  //властивість для встановлення значення value
  public int Value {
    set {
      this.value = value;
      if (value < left || value > right)
        //генеруємо подію виходу за межі діапазону
        OnRangeOut(this,
          new RangeOutEventArgs("Вихід за межі!"));
    }
  }
}
```

В оголошенні класу означені делегат `RangeOutHandler` та подія `OnRangeOut`. Властивість `Value` контролює значення для змінної `value` на предмет виходу за межі діапазону [`left,right`]. Якщо відбувся вихід за межі діапазону, то утворюємо об'єкт класу `RangeOutEventArgs`, який разом з об'єктом класу `RangeControl` передаємо як параметр події `OnRangeOut`.

Зауважимо, що подія `OnRangeOut` має тип класу-делегата `RangeOutHandler`, який відповідає обмеженням багатоадресних делегатів (тип результату – `void`). Тобто делегат може

представляти декілька методів. Це дає змогу клієнтському коду реєструвати необмежену кількість методів обробки події `OnRangeOut`.

Обробка подій

Генерування подій – не така розповсюджена практика, як перехоплення та обробка повідомлень. Що стосується користувацького інтерфейсу, то Microsoft вже написала всі генератори подій, які можуть вам знадобитися (вони розміщені у просторі імен `Windows.Forms`).

Щоб отримати подію у клієнтському коді, достатньо лише повідомити про це екземпляр класу `RangeControl` та передати йому інформацію про обробника цієї події:

```
//ця функція буде обробляти подію
protected void UserHandler(
  object sender, RangeOutEventArgs e) {
  MessageBox.Show(e.Message);
}
//тут деякий код
//утворюємо екземпляр класу RangeControl
RangeControl rc = new RangeControl(1, 0, 5);
//додаємо новий метод для обробки події
rc.OnRangeOut += new
  RangeControl.RangeOutHandler(UserHandler);
//ініціюємо генерування події
rc.Value = 6;
```

У результаті виконання цього коду буде згенерована подія `OnRangeOut`, і функція `UserHandler` виведе на екран діалогову форму з повідомленням „Вихід за межі”.

Загальні типи

Загальні типи (*generics*) дають змогу при оголошенні класів, структур, методів, інтерфейсів і делегатів не вказувати конкретні типи параметрів. Тип параметра визначить компілятор самостійно в момент оголошення змінної.

Загальний тип подібний до *шаблону* (*template*) C++, однак має деякі обмеження.

Утворимо клас з використанням загальних типів.

```
class CGeneric <TYPE1, TYPE2> {
 public TYPE1 Field1;
 public TYPE2 Field2;
 public CGeneric (TYPE1 Field1, TYPE2 Field2){
   this.Field1 = Field1;
   this.Field2 = Field2;
 }
 //інші члени класу
}
```

Після назви класу розміщено перелік загальних типів, які використовує клас. При оголошенні змінної типу цього класу замість загальних потрібно вказати конкретні типи. Наприклад:

```
CGeneric<int,int> intVar =
        new CGeneric<int,int>(0,1);
CGeneric<bool,bool> boolVar =
      new CGeneric< bool,bool >(true,false);
CGeneric<string,float> mixedVar =
      new CGeneric<string,float>("Pi",3.14);
```

Компілятор читає конкретні типи та підставляє їх замість загальних. Наприклад, змінна `intVar` буде екземпляром класу `CGeneric`, де поля `Field1` та `Field2` мають тип `int`. У цьому випадку можна написати, наприклад, такий код:

```
int sum = intVar.Field1 + intVar.Field2;
```

Однак помилковим буде код

```
float sum = mixedVar.Field1+mixedVar.Field2;
```

оскільки операція додавання стрічки до числа неозначена.

У межах класу конкретний тип невідомий. Отож потрібно бути обережним з використанням специфічної функціональності конкретних типів і складанням виразів.

Загальні типи корисні за необхідності утворення класів з ідентичною функціональністю, проте різними типами даних. Як приклад, утворимо простий клас, який вміє шукати мінімум і максимум двох значень типу (з означеною операцією порівняння).

```
class CMinMax <TYPE> {
  public TYPE Field1;
  public TYPE Field2;
  public CMinMax (TYPE Field1, TYPE Field2){
    this.Field1 = Field1;
    this.Field2 = Field2;
  }
  public TYPE Min{
    get { return Field1 < Field2 ?
                          Field1 : Field2; }
  }
  public TYPE Max{
    get { return Field1 < Field2 ?
                          Field2 : Field1; }
  }
}
```

Тепер використаємо клас `CMinMax`:

```
//оголошення
CMinMax<int> iVar = new CMinMax<int>(0,1);
CMinMax<float> fVar = new
                  CMinMax<float>(0.5,1.71);
CMinMax<string> sVar = new
                CMinMax<string>("Ab","Cd");
//використання
int i = iVar.Min; // i = 0
float f = fVar.Max; // f = 1.71
string s = sVar.Min; // s = "Ab"
```

Очевидно, що для пошуку мінімуму-максимуму двох значень простіше використати безпосередньо оператор `?` або статичні функції `Min` та `Max` класу `Math`. Наведений приклад демонструє лише принцип використання загальних типів.

Загальні типи можуть бути корисними і в тих випадках, де задля узагальнення деякої функціональності використовують тип `object`, а в кожному конкретному випадку він явним чином приводиться до потрібного типу. Оскільки компілятор замість загального типу підставляє заданий, то явного приведення типів можна уникнути. Це дає змогу писати продуктивніші програми. Окрім цього, на противагу використанню типу `object` з наступним приведенням типу, для загальних типів компілятор здійснює їхню перевірку у момент компіляції. Це зменшує можливість виникнення помилок виконання.

Параметризованими можна оголошувати і структури, інтерфейси та делегати:

```
//структура
public struct SGeneric <TYPE> {
  public TYPE Field;
}
//інтерфейс
public interface IGeneric <TYPE> {
  public TYPE Method();
}
//оголошення делегата
public delegate RESULTTYPE
DGeneric<RESULTTYPE,PARAMTYPE>
                          (PARAMTYPE aParam);
//використання делегата
DGeneric<double,int> dgen =
       new DGeneric<double,int>(ClassA.M1);
double m1 = dgen(10);
```

Примітка. Загальні типи введені в .NET Framework 2.

Директиви препроцесора C#

C# містить команди, які впливають на процес компіляції, проте ніколи не транслюються у виконуваний код. Вони називаються *директивами препроцесора* і починаються з символу #.

Наступна таблиця містить перелік і зміст директив препроцесора.

Директива	Приклад	Зміст
`#define`	`#define DEBUG`	Повідомляє компілятору, що існує символ із зазначеним іменем.
`#undef`	`#undef DEBUG`	Повідомляє компілятору, що вже не існує символ із зазначеним іменем.

`#if`
`#elif`
`#else`
`#endif`
	`#if DEBUG`
	`//деякі дії`
	`#elif WIN64`
	`//деякі дії`
	`#else`
	`//деякі дії`
	`#endif`
		Ці директиви дають вказівку компілятору: компілювати деяку частину коду чи ні. Директива `#elif` означає `else if`. Директиви `#elif` та `#else` можуть бути пропущеними. .NET надає альтернативу `#if` із використанням атрибута `Conditional`.

`#warning`	`#warning "не забути вилучити попередження"`	Компілятор виводить користувачу текст, розташований після директиви. Компіляція буде продовжена.

`#error`	`#if DEBUG && RELEASE`
	`#error "одночасно не може бути"`
	`#endif`
		Компілятор виводить користувачу текст, розташований після директиви. Компіляція буде припинена.

`#region`
`#endregion`
	`#region RegionName`
	`//код`
	`#endregion`
		Редактори тексту можуть використати ці директиви для кращого розташування коду на екрані, у тім числі згорнути/розгорнути цей код.

`#line`	`#line 100 "File.cs"`
	`//деякі дії`
	`#line default`
		Використовують для зміни інформації про ім'я файла та номер стрічки, яка виводиться компілятором у попередженнях і повідомленнях про помилки.

ЕЛЕМЕНТИ БІБЛІОТЕКИ КЛАСІВ .NET

Бібліотека класів .NET містить тисячі тисяч класів, інтерфейсів і переліків. Немає жодної змоги зробити вичерпний опис цих класів навіть у досить об'ємних публікаціях. Отож в процесі програмування в середовищі .NET необхідно мати електронний довідник.

У цьому розділі наведемо короткий огляд деяких методів окремих класів .NET, які здебільшого використовуються Windows-програмами.

Простори імен

Класи, інтерфейси та переліки в бібліотеці .NET згруповані у *простори імен*. У кожному з них зібрані засоби для виконання певних дій: інтерфейс застосувань Windows, робота з текстом, система безпеки та інших. Простори імен можуть містити інші простори імен, тобто мають деревовидну структуру.

Наведемо список деяких найкорисніших просторів імен:

Простір імен	**Опис**
`Microsoft.CSharp`	Функції компілювання та генерування коду мови C#.
`Microsoft.Win32`	Підтримка подій операційної системи та системного реєстра.
`System`	Кореневий простір імен для більшості класів бібліотеки. Містить також типи даних і базові класи.
`System.CodeDom`	Представляє структуру документа з вихідним кодом.
`System.Collections`	Словники, кеш-таблиці, черги, стеки.
`System.ComponentModel`	Реалізує поведінку компонент і керуючих елементів під час виконання та під час розробки програми.
`System.Configuration`	Програмний доступ до файлів `.config`.
`System.Data`	Класи, які реалізують технологію ADO.NET.
`System.Diagnostics`	Підтримка взаємодії з журналом подій, системними процесами та лічильниками продуктивності.
`System.Directory-Services`	Підтримка взаємодії зі службою Active Directoryю
`System.Drawing`	Класи, які реалізують графічні можливості

	інтерфейсу GDI+.
System.Enterprise-Services	Підтримка взаємодії зі службами COM+.
System.Globalization	Підтримка національних стандартів.
System.IO	Ввід та вивід для потоків і файлів.
System.Messaging	Підтримка черг подій.
System.Net	Підтримка мережевих протоколів.
System.Reflection	Доступ до метаданих, розташованих у застосуваннях .NET.
System.Resources	Утворення та керування ресурсами.
System.Runtime	Підтримка різноманітних додаткових можливостей загального оточення мов (CLR).
System.Security	Система безпеки середовища CLR.
System.ServiceProcess	Утворення та управління сервісами Windows.
System.Text	Підтримка роботи з текстом і стрічками.
System.Threading	Підтримка багатопотоковості.
System.Timers	Реалізація таймерів.
System.Web	Підтримка роботи через Web-браузер, технології ASP.NET та web-сервісів.
System.Windows.Forms	Користувацький інтерфейс застосувань Windows.
System.Xml	Підтримка мови XML та XML-стандартів.

Елементи простору імен можуть бути означені в різних файлах складених модулів.

Просторам імен можна надавати псевдоніми:

```
using Forms = System.Windows.Forms;
```

Глобальний простір імен має псевдонім `global`.

Щоб звернутися до простору імен з використанням його псевдоніма, використовують синтаксис `alias::`. Наприклад:

```
Forms::Application.ExitThread();
```

Можлива ситуація, коли використовують різні складені модулі з однаковими просторами імен. Наприклад, назву `StringLib` використовують для позначення простору імен у складених модулях `StringLibrary.dll` та `Str.dll`:

Складений модуль `StringLibrary.dll`

```
namespace StringLib {
  public class StrFunctions {
    //функції роботи зі стрічками
  }
```

```
}
```

Складений модуль `Str.dll`

```
namespace StringLib {
  public class StrFunctions {
    //функції роботи зі стрічками
  }
}
```

При використанні цих двох складених модулів може виникнути конфлікт імен. Щоб уникнути такої неоднозначності, необхідно використати *зовнішні аліаси*:

```
extern alias StringLibrary;
extern alias Str;
// деякий код
StringLibrary.StringLib.StrFunctions.Method();
Str.StringLib.StrFunctions.Method();
```

Примітка. Специфікатори до псевдонімів і зовнішні аліаси доступні у .NET Framework версії 2.

Універсальний базовий клас Object

Усі класи .NET є похідними від `System.Object`. Цей клас містить сім методів:

Метод	**Зміст**
`ToString`	Віртуальний метод, який повертає стрічкове представлення об'єкта. Зазвичай, перекривається класами-нащадками.
`GetHashCode`	Віртуальний метод, який повертає *кеш* (*hash*) об'єкта для ефективного пошуку екземплярів класів у довідниках або кеш-таблицях (`HashTable`).
`Equals`	Метод має дві версії: статичну та віртуальну. Перевіряє об'єкти на рівність.
`ReferenceEquals`	Перевіряє на рівність два посилання
`GetType`	Повертає об'єкт класу `System.Type` (або нащадка цього класу) з інформацією про тип даних об'єкта.
`MemberwiseClone`	Утворює коротку копію об'єкта: копіює в новий об'єкт дані типів за значенням і повертає посилання на цей об'єкт.
`Finalize`	Метод виконує роль деструктора. Викликається прибиральником „сміття" і може бути використаний для звільнення ресурсів.

Детальніше зупинимося на методах порівняння.

Метод `ReferenceEquals` має сигнатуру

```
public static bool ReferenceEquals(
                  object objA, object objB);
```

і перевіряє, чи дві змінні (за посиланням) посилаються на один і той же екземпляр класу, чи ні. Якщо `objA` та `objB` містять посилання на одну адресу або обидва мають значення `null`, то метод повертає результат `true`.

Віртуальну версію

```
public virtual bool Equals(object obj);
```

методу `Equals` використовують для конкретного екземпляра. Його реалізація в `System.Object` порівнює посилання. Проте цей метод призначений для випадку, коли потрібно перекрити його з метою порівняння даних об'єктів. Ми це вже робили в демонстраційному класі `Point`.

Статична версія

```
public static bool Equals(
                  object objA, object objB);
```

методу `Equals` працює аналогічно віртуальній. Метод перевіряє, чи не становить `null` одне чи обидва переданих йому посилання. Якщо так, то повертає значення `false` (одне посилання `null`) або `true` (обидва `null`). Якщо ж обидва посилання змістовні, то викликається віртуальна версія `Equals`.

Операція порівняння == порівнює посилання. Якщо потрібно порівнювати дані, то цю операцію необхідно перекрити (як це зроблено в класі `Point`).

Розглянуті методи порівняння застосовуються до типів за посиланням. Типи за значенням також є похідними від `object`. Не безпосередньо, а через клас `System.ValueType`. І у цьому класі вже перекритий метод `Equals`. Цей метод перевіряє вже не рівність посилань, а співпадіння даних у всіх полях структури.

Зазначимо, що метод `ReferenceEquals` для типів за значенням завжди повертає `false`.

Стрічки

Клас String

Ключове слово C# `string` насправді посилається на базовий клас .NET `System.String`. Наведемо деякі методи цього класу:

Метод	Зміст
`Compare`	Порівнює зміст стрічок з урахуванням регіональних налаштувань
`CompareOrdinal`	Порівнює зміст стрічок без урахування регіональних налаштувань
`Format`	Повертає стрічку, відформатовану відповідно до переданого формату
`IndexOf`	Повертає індекс (починаючи з 0) першого входження підстрічки у стрічці
`IndexOfAny`	Повертає індекс першого входження у стрічці довільного символу із заданого масиву символів
`LastIndexOf`	Повертає індекс останнього входження підстрічки у стрічці
`LastIndexOfAny`	Повертає індекс останнього входження у стрічці довільного символу із заданого масиву символів
`PadLeft`	Вирівнює стрічку, додаючи декілька заданих символів на початок стрічки
`PadRight`	Вирівнює стрічку, додаючи декілька заданих символів наприкінці стрічки
`Replace`	Замінює деякий символ або підстрічку на інший символ або під стрічку
`Split`	Розбиває стрічку на масив стрічок, використовуючи заданий символ для визначення меж підстрічок
`SubString`	Повертає підстрічку заданої довжини, починаючи з зазначеної позиції у стрічці
`ToLower`	Повертає копію стрічки, у якій всі символи переведені у нижній регістр
`ToUpper`	Повертає копію стрічки, у якій всі символи переведені у верхній регістр
`Trim`	Вилучає пробіли на початку й наприкінці стрічки
`TrimEnd`	Вилучає пробіли наприкінці стрічки
`TrimStart`	Вилучає пробіли на початку стрічки

Клас StringBuilder

Тип `string` ефективно зберігає стрічки. Якщо ми одній стрічковій змінній присвоїмо значення іншої, то насправді обидві змінні будуть посилатися на одну і ту ж ділянку пам’яті. Але при

внесенні змін в одну з них виділиться нова ділянка пам'яті. Якщо ми спробуємо у циклі замінити кожен символ стрічки на якийсь інший, то таке копіювання у нові ділянки пам'яті відбудеться певну кількість разів, залежно від довжини стрічки.

Отож для маніпуляції зі стрічками доцільно використовувати інший клас представлення стрічок – `System.Text.StringBuilder`. Цей клас має декілька конструкторів:

```
StringBuilder sb1 = new
                StringBuilder("стрічка", 10);
StringBuilder sb1 = new StringBuilder(10);
StringBuilder sb1 = new
                StringBuilder("стрічка");
```

Перший конструктор виділяє пам'ять для розташування 10-ти символів та записує текст "стрічка". Другий – лише виділяє пам'ять. Третій – виділяє пам'ять за замовчуванням (зазвичай, удвічі більшу за потрібну) та записує текст.

Властивість `Length` повертає кількість значущих символів стрічки, а властивість `Capacity` – кількість зарезервованих символів.

Перелічимо основні методи класу `StringBuilder`:

Метод	Зміст
`Append`	Додає стрічку до поточної стрічки
`AppendFormat`	Додає стрічку, отриману з допомогою специфікатора формату
`Insert`	Вставляє підстрічку у поточну стрічку
`Remove`	Вилучає символи зі стрічки
`Replace`	Замінює деякий символ або підстрічку на інший символ або під стрічку
`ToString`	Повертає стрічку, приведену до об'єкта `String`

Дата та час

Структури дати та часу

Для роботи з інформацією про дату та час використовують структури `System.DateTime` і `System.TimeSpan`.

Структура `DateTime` має вісім перевантажених конструкторів, які надають зручний синтаксис для утворення екземплярів структури. Наприклад:

```
int year = 2005;
int month = 5;
int day = 8;
DateTime d1 = new DateTime(year, month, day);
int hour = 21;
int minute = 35;
int second = 20;
int milisecond = 120;
DateTime d2 = new DateTime(year, month, day,
  hour, minute, second, milisecond);
```

Конструктору `DateTime` можна передати (останнім параметром) об'єкт класу `System.Globalization.Calendar`. Існує декілька нащадків цього класу: `GregorianCalendar`, `JulianCalendar` та інші.

Структура `TimeSpan` призначена для представлення часових інтервалів. Ось приклади утворення екземплярів структури:

```
int hours = 10;
int minutes = 15;
int seconds = 30;
TimeSpan ts1 = new TimeSpan(hours, minutes,
                                        seconds);
int days = 20;
TimeSpan ts2 = new TimeSpan(days,10,15,30);
long ticks = 776890;
TimeSpan ts3 = new TimeSpan(ticks);
```

Останній конструктор утворює часовий проміжок тривалістю 776890 *тіків* (1 тік містить 100 наносекунд).

Властивості та методи структури DateTime

Структура `DateTime` має три статичні властивості, які відображають поточну дату та час, установлені на комп'ютері:

Властивість	**Зміст**
`Now`	повертає дату та час
`Today`	повертає дату (час – 0:00:00)
`UtcNow`	повертає дату та час за Гринвічем

Наприклад,

```
DateTime today = DateTime.Today;
```

Решта властивостей відображає дату та час конкретного екземпляра структури:

Властивість	Зміст
`Date`	повертає дату (час – 0:00:00)
`Day`	повертає день місяця
`DayOfWeek`	повертає день тижня
`DayOfYear`	повертає номер дня в році
`Hour`	повертає годину
`Minute`	повертає хвилини
`Second`	повертає секунди
`TimeOfDay`	повертає екземпляр типу TimeSpan, який означає час, що минув від півночі
`Year`	повертає рік

Структура `DateTime` містить низку статичних методів. Зокрема:

Метод	Зміст
`DaysInMonth`	повертає кількість днів у заданому місяці заданого року
`IsLeapYear`	повертає `true`, якщо вказаний рік є високосним
`Parse`, `ParseExact`	перетворюють стрічкове значення дати та часу в екземпляр структури `DateTime`

Нестатичні методи маніпулюють значеннями дати та часу конкретного екземпляра і дозволяють додавати або віднімати часові інтервали, перетворювати дату та час у стрічкове представлення та інше.

Структура містить перевантажений метод `ToString()`, якому можна передавати стрічковий параметр з описом формату стрічки, у яку потрібно перетворити екземпляр структури. Існує значна кількість можливих форматів перетворення. Наприклад:

```
DateTime dt = new DateTime();
dt = DateTime.Now;
string s;
s = dt.ToString("dd MMMM yyyy");
                              //08 травня 2005
s = dt.ToString("dddd, d-MMM-yy");
                              //неділя, 8-тра-05
```

```
s = dt.ToString("d.MM.yy H:mm:ss");
                              //8.05.05 9:08:20
```

Окрім цього, символи форматування використовують для надання різноманітних вбудованих форматів:

```
s = dt.ToString("d"); //08.05.2005
s = dt.ToString("G"); //08.05.2005 1:25:38
```

Для ознайомлення зі всіма компонентами форматної стрічки для `DateTime` і вбудованих форматів дати та часу потрібно звернутися до довідкової системи.

Властивості та методи структури TimeSpan

Структура `TimeSpan` має 11 загальнодоступних властивостей. Властивості `Days`, `Hours`, `Minutes`, `Seconds`, `Milliseconds` та `Ticks` повертають ціле значення відповідно до кількості днів, годин, хвилин, секунд, мілісекунд і тіків в екземплярі структури. Властивості `TotalDays`, `TotalHours`, `TotalMinutes`, `TotalSeconds`, `TotalMilliseconds` повертають значення типу `double`, яке представляє тривалість екземпляра структури у відповідних одиницях.

Статичні методи `FromDays`, `FromHours`, `FromMinutes`, `FromSeconds`, `FromMilliseconds`, `FromTicks` дають змогу утворити новий екземпляр структури. Наприклад, код

```
TimeSpan tm = TimeSpan.FromDays(2.5);
```

повертає інтервал у два з половиною дні.

Метод `Parse` використовують для перетворення стрічкового представлення проміжку часу в еквівалентний екземпляр структури `TimeSpan`. Методу передається стрічковий параметр такого формату:

```
[-][d.]hh:mm:ss[.ff]
```

де *d* – кількість днів, *hh* – годин, *mm* – хвилин, *ss* – секунд, *ff* – часток секунди. Знак „мінус” задає від’ємний інтервал. Елементи в квадратних дужках не є обов’язкові.

Наступний код утворює екземпляр структури `TimeSpan` тривалістю два дні п'ять годин десять хвилин і 25,8750225 секунд:

```
TimeSpan tm = TimeSpan.Parse("2.5:10:25.8750225");
```

Арифметичні операції над датами та часом

Над екземплярами структури `TimeSpan` можна виконувати операції додавання та віднімання:

```
TimeSpan tm1 = TimeSpan.FromDays(2);
TimeSpan tm2 = TimeSpan.FromDays(1);
tm1 += tm2;  //tm1 = 3 дні
tm2 -= tm1;  //tm2 = -2 дні
```

Ці ж дії можна виконати методами `Add` і `Substract`.

Операції множення часових інтервалів і множення часового інтервалу на число не означені.

Над екземплярами структур `DateTime` та `TimeSpan` можна виконувати операції додавання та віднімання з операндами таких типів:

Операція	Операнд 1	Операнд 2	Тип результату
+	DateTime	TimeSpan	DateTime
-	DateTime	TimeSpan	DateTime
-	DateTime	DateTime	TimeSpan

Наприклад:

```
DateTime dt1 = new DateTime(2005,1,1);
dt1 += TimeSpan.FromHours(45);
                    // 02.01.2005 21:00:00
DateTime dt2 = new DateTime(2006,1,1);
TimeSpan tm1 = dt2 - dt1; // 363.03:00:00
```

Колекції

У просторі імен `System.Collections` міститься кілька класів, в екземплярах яких можна зберігати множину елементів, кількість яких може змінюватися під час виконання програми. Окрім цього, можна організувати доступ до елементів з використанням індексів, які не є цілими числами.

Динамічний масив

Динамічний масив може змінювати свій розмір у процесі додавання нових елементів. Динамічний масив реалізований у класі `ArrayList`.

Наступний код містить приклади використання конструкторів класу `ArrayList`:

```
ArrayList ar = new ArrayList();
ArrayList ar10 = new ArrayList(10);
ArrayList arar = new ArrayList(ar);
```

Перший конструктор утворює масив, місткість якого 0. Місткість – це максимальна кількість елементів, яку може зберігати масив без перебудови.

Другий конструктор утворює масив, місткість якого 10. Окрім конструктора, задавати (і отримувати) місткість можна з допомогою властивості `Capacity`.

Третій конструктор утворює масив `arar`, який є копією масиву `ar`.

Для додавання елементів в об'єкт `ArrayList` використовують метод `Add`:

```
ar.Add("Це елемент динамічного масиву");
```

Нумерація елементів починається з нуля. Кількість елементів у масиві відображає властивість `Count`:

```
for (int i=0; i < ar.Count; i++)
  Console.WriteLine(ar[i].ToString());
```

Якщо місткість масиву дорівнює кількості доданих елементів, то при додаванні нового елемента масив перебудовується з метою збільшення місткості. Цей процес не можна назвати ефективним, тому краще зарезервувати достатню місткість масиву. Загалом, швидкодія роботи динамічних масивів у декілька разів менша порівняно зі звичайними масивами. Отож при нагоді краще використовувати звичайні масиви.

Елементи масиву можуть мати довільний тип, оскільки вони зберігаються як об'єкти класу `System.Object`.

Для додавання або вставляння елементів використовують методи

```
int  Add(value),
void Insert(index, value),
void AddRange(collection),
void InsertRange(index, collection),
void SetRange(index, collection).
```

Наприклад:

```
ar.Add(new Point(1, 1));
ar.Insert(0,new Point(2, 1));
string[] strs = {"елементи", "для",
"Range"};
ar.AddRange(strs);
ar.SetRange(3,strs);
```

Метод `Add` додає елемент наприкінці списку. Метод `Insert` вставляє елемент у середину списку, посуваючи всі елементи з індексами, не меншими за *index*.

Методи `AddRange` та `InsertRange` використовують для додавання або вставляння діапазону елементів. Цей діапазон задається колекцією (наприклад, масивом).

Метод `SetRange` не вставляє діапазон елементів, а замінює ними інші, починаючи із заданого індексу.

Для вилучення елементів з динамічного масиву використовують методи

```
void RemoveAt(index),
void Remove(value),
void RemoveRange(startIndex, collection).
```

Пошук елементів у динамічному масиві здійснюють такими методами:

```
bool Contains(value),
int IndexOf(value [,startIndex [, count]]),
int LastIndexOf(value [,startIndex [, count]]),
int BinarySearch(value [, comparer]),
int BinarySearch(index,count,value,comparer).
```

Метод `Contains` повертає `true`, якщо динамічний масив містить хоча б один елемент зі значенням *value*.

Методи `IndexOf` і `LastIndexOf` повертають позицію, відповідно, першого та останнього елемента зі значенням *`value`*. Необов'язкові параметри *`startIndex`* і *`count`* локалізують область пошуку.

Метод `BinarySearch` застосовують для пошуку елемента *`value`* у відсортованому масиві. Параметри *`index`* та *`count`* локалізують область пошуку. Параметр *`comparer`* задає правило порівняння об'єктів. Об'єкт *`comparer`* – це екземпляр деякого класу, який підтримує інтерфейс `IComparer` і реалізує метод `IComparer.Compare`. Об'єкт *`comparer`* можна використовувати й у методах сортування:

```
void Sort([comparer]),
void Sort(index, count, comparer),
void Reverse([index, count]).
```

Метод `Sort` впорядковує елементи динамічного масиву за заданим об'єктом *`comparer`* правилом порівняння. Якщо параметр *`comparer`* відсутній або має значення `null`, то числа та дати впорядковуються за зростанням, а символи – за абеткою в порядку зростання. Метод `Reverse` використовують для зміни порядку елементів на протилежний.

Як і всі колекції, клас `ArrayList` реалізує інтерфейс `IEnumerable`, отож для поелементної ітерації екземпляра цього класу можна використовувати цикл `foreach`.

Бітовий масив

Бітовий масив – це масив булевих значень. Кожне булеве значення у такому масиві представляється єдиним бітом (0 або 1). Збереження одного біта вимагає менше пам'яті, ніж збереження значення типу `bool`. Як і інші колекції, бітовий масив є динамічним.

Бітовий масив реалізований у класі `BitArray`, який має декілька конструкторів:

```
BitArray (length [,defaultValue]),
BitArray (values),
BitArray (bits).
```

Тут *length* – місткість масиву, а *defaultValue* – значення, яким потрібно заповнити елементи масиву при утворенні.

Параметр *values* може бути масивом елементів типу `bool`, `byte` або `int`. Відповідний конструктор утворить бітовий масив з місткістю, яка відповідає довжині масиву-аргумента. Елементи набудуть значення `false`, якщо відповідний елемент аргумента містить значення 0 або `false`. У протилежному випадку ці значення будуть `true`.

Останній конструктор – це конструктор копій. Тут *bits* – бітовий масив.

Як і для звичайного масиву, доступ до елементів бітового масиву здійснюється з допомогою індексів:

```
BitArray bitAr = new BitArray(10);
bitAr[0] = false;
```

Кількість елементів у масиві показує властивість `Count`. Значення цієї властивості завжди збігається зі значенням властивості `Length`. Відмінність між ними в тому, що `Count` є властивістю лише для читання.

Клас `BitArray` має декілька корисних методів.

Методи `And`, `Or` та `Xor` виконують, відповідно, операції І, АБО та виключне І між елементами поточного бітового масиву та відповідними елементами бітового масиву-аргумента.

Метод `Not` інвертує всі значення елементів масиву.

Метод `CopyTo` копіює елементи з бітового масиву в одномірний масив.

Кеш-таблиця

Кеш-таблиця дає змогу зберігати пари *ключ - значення*. Ключ у кеш-таблиці можна використовувати для пошуку відповідного йому значення. Як ключ, так і значення можуть бути довільного типу.

Кеш-таблиця реалізована в класі `Hashtable`. Цей клас належить до множини *словників*, тобто класів, які підтримують інтерфейс `IDictionary`.

Клас `Hashtable` містить одинадцять конструкторів. Найпростіші з них мають сигнатуру

```
Hashtable ([capacity [,loadFactor]]);
```

Розглянемо простий приклад:

```
Hashtable ht = new Hashtable(7);
ht.Add("Point_1", new Point());
Point p = (Point)ht["Point_1"];
```

Конструктор `Hashtable(`*`capacity`*`)` використовують найчастіше. Словники найефективніше працюють, коли їх місткість *`capacity`* є простим числом. Це спричинено специфікою алгоритмів, які використовуються у словниках.

Додають об'єкти в `Hashtable` з допомогою методу `Add`:

```
void Add(key, value);
```

де обидва параметри мають тип `object`. Зауважимо, що словники не мають індексів таких, як масиви. Тому методів типу `Insert` також мати не можуть.

Доступ до значення здійснюється з допомогою ключа у квадратних дужках. Оскільки значення повертається як об'єкт типу `object`, потрібне явне приведення типу: `(Point)ht["Point_1"]`.

Якщо ключем слугує клас користувача, то в ньому має бути реалізовано алгоритм *кеш*. Ця реалізація, розташована в перекритому методі `GetHashCode`, передбачає виконання таких умов:

- алгоритм повинен бути швидким;
- якщо `A.Equals(B)` дорівнює `true`, то методи `A.GetHashCode()` та `B.GetHashCode()` повинні завжди повертати ідентичний кеш;
- в ідеалі алгоритм повинен видавати значення, рівномірно розподілені між `int.MinValue` та `int.MaxValue`.

Остання вимога спричинена потребою розташування поруч об'єктів, для яких кеш дає ідентичний індекс. У цьому випадку бажано, щоб місткість словника була значно більшою за кількість

розташованих у ньому елементів. Пропорція між заповненою та незаповненою частинами таблиці характеризується коефіцієнтом завантаження - *loadFactor*. За меншого максимального завантаження ефективніше працюватиме кеш-таблиця, проте пам'яті вона займатиме більше.

Простий приклад перекриття методу `GetHashCode` ми вже навели, розглядаючи перевантаження операцій на прикладі демонстраційного класу `Point`, де неявно використали кеш Microsoft для стрічкового типу:

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

Властивість `Count` повертає кількість елементів, які зберігаються в таблиці.

Властивість `Keys` повертає колекцію (інтерфейс `ICollection`), яка містить ключі таблиці.

Властивість `Values` повертає колекцію, яка містить значення таблиці.

Серед методів класу `Hashtable` зазначимо такі:

Метод	Зміст
`Add`	Додає елемент із заданим ключем і значенням
`Clear`	Вилучає всі елементи з таблиці
`Clone`	Утворює копію таблиці
`Contains`	Визначають, чи містить таблиця заданий ключ
`ContainsKey`	
`ContainsValue`	Визначає, чи містить таблиця задане значення
`Remove`	Вилучає з таблиці елемент із заданим ключем

Відсортований список

Відсортований список – це комбінація динамічного масиву та кеш-таблиці. Його можна розглядати як кеш-таблицю з функціональністю індексування. Однак це не повнофункціональна кеш-таблиця, оскільки список не підтримує інтерфейси серіалізації.

Відсортований список реалізований у класі `SortedList`. Клас містить шість конструкторів, з яких найчастіше використовують такий:

```
SortedList ([capacity]);
```

Властивості і більшість методів класу `SortedList` такі ж, як і в класу `Hashtable`. Зі специфічних методів зазначимо такі:

Метод	Зміст
`GetByIndex`	Повертає значення на заданій позиції у списку
`GetKey`	Повертає ключ на заданій позиції у списку
`GetKeyList`	Повертає інтерфейс `IList` зі всіма ключами списку
`GetValueList`	Повертає інтерфейс `IList` зі всіма значеннями списку
`IndexOfKey`	Повертає номер позиції заданого ключа у списку
`IndexOfValue`	Повертає номер позиції заданого значення у списку
`RemoveAt`	Вилучає елемент на заданій позиції у списку
`SetByIndex`	Заміняє значення на заданій позиції у списку

Черга

Доступ до елементів *черги* здійснюється за принципом „першим увійшов – першим вийшов” (FIFO – firs in, first out).

Черга реалізована в класі `Queue`. Клас містить такі конструктори:

```
Queue ([capacity [, growFactor]]);
Queue (collection);
```

За замовчуванням місткість черги *`capacity`* дорівнює 32, а фактор збільшення *`growFactor`* – 2.0. Фактор збільшення задає множник, на який потрібно помножити місткість, якщо існуючої недостатньо. Друга версія конструктора використовує для утворення черги колекцію – екземпляр класу з підтримкою інтерфейсу `ICollection`.

Елементи черги можуть мати довільний тип.

Властивість `Count` повертає кількість елементів у черзі.

Із методів класу `Queue` зазначимо такі:

Метод	Зміст
`Clear`	Вилучає всі елементи з черги
`Contains`	Визначає, чи містить черга заданий елемент
`CopyTo`	Копіює елементи черги в одновимірний масив

`Dequeue`	Повертає значення першого елемента черги та вилучає його з черги
`Enqueue`	Додає елемент наприкінці черги
`Peek`	Повертає значення першого елемента черги, однак не вилучає його з черги
`ToArray`	Копіює елементи черги в масив

Стек

Доступ до елементів *стека* здійснюється за принципом „останнім увійшов – першим вийшов" (LIFO – last in, first out).

Стек реалізований у класі `Stack`. Клас налічує такі конструктори:

```
Stack ([capacity]);
Stack (collection);
```

За замовчуванням місткість стеку *`capacity`* дорівнює 10. Друга версія конструктора для утворення стека використовує колекцію.

Елементи стека можуть мати довільний тип.

Властивість `Count` повертає кількість елементів у стеку.

Із методів класу `Stack` зазначимо такі:

Метод	**Зміст**
`Clear`	Вилучає всі елементи зі стека
`Contains`	Визначає, чи містить стек заданий елемент
`CopyTo`	Копіює елементи стека в одновимірний масив
`Peek`	Повертає елемент на вершині стека, проте не вилучає його зі стека
`Pop`	Повертає елемент на вершині стека та вилучає його зі стека
`Push`	Додає елемент на вершину стека
`ToArray`	Копіює елементи стека в масив

Робота з файловою системою

Робота з файлами

Простір імен `System.IO` містить понад 50 класів, інтерфейсів і переліків, які дають змогу працювати з файлами та каталогами (папками). Додатково простір імен `System.Windows.Forms` містить декілька діалогових форм вибору файлів.

При роботі з файлами доцільно детально ознайомитися з функціональними можливостями елементів `System.IO`, оскільки Microsoft передбачив дуже широкий набір інструментів маніпулювання файловою системою. Ми ж розглянемо лише декілька з них.

Для поодиноких операцій над файлами використовують клас `File`. Він містить лише статичні методи, отож утворювати екземпляр класу нема необхідності.

Серед загальнодоступних методів класу `File` зазначимо такі:

Метод	Зміст
`AppendText`	Додає текст в існуючий файл
`Copy`	Копіює файл
`Create`	Утворює файл
`CreateText`	Утворює файл і відкриває його для запису тексту
`Delete`	Вилучає файл
`Exists`	Перевіряє, чи існує файл
`GetAttributes`	Повертає атрибути файла
`Move`	Переміщає файл
`Open`	Відкриває файл
`OpenRead`	Відкриває файл для читання
`OpenText`	Відкриває файл для читання тексту
`OpenWrite`	Відкриває файл для запису
`SetAttributes`	Встановлює атрибути файла

Клас `File` містить також кілька методів для повернення або встановлення дати утворення файла, останнього доступу, останнього запису.

Наведемо простий приклад використання класу `File`:

```
if (openDlg.ShowDialog()==DialogResult.OK){
  string s = openDlg.FileName;
  FileAttributes fa = File.GetAttributes(s);
  if ((fa & FileAttributes.ReadOnly) ==
      FileAttributes.ReadOnly)
    MessageBox.Show("Файл " + s +
      " є лише для читання";
}
```

Цей код активізує діалогову форму вибору файла та записує у змінну `fa` атрибути файла, які є набором бітових прапорців.

Якщо цей набір у позиції ознаки `ReadOnly` містить ненульовий біт, то виводиться повідомлення.

Якщо потрібно виконати декілька операцій над файлом, то значно ефективнішим буде використання класу `FileInfo`. Для роботи з файлом з допомогою цього класу потрібно утворити об'єкт. Єдиний конструктор класу має просту сигнатуру:

```
public FileInfo(string fileName);
```

Клас `FileInfo` має такі загальнодоступні властивості:

Властивість	Зміст
`Attributes`	Атрибути файла
`CreationTime`	Час утворення файла
`Directory`	Каталог (тип `DirectoryInfo`), у якому розташований файл
`DirectoryName`	Назва каталогу, у якому розташований файл
`Exists`	Повертає `true`, якщо файл існує
`Extension`	Розширення файла
`FullName`	Повний шлях до файла та назва файла
`LastAccessTime`	Час останнього доступу до файла
`LastWriteTime`	Час останньої модифікації файла
`Length`	Розмір файла
`Name`	Назва файла

Серед загальнодоступних методів класу `FileInfo` зазначимо такі:

Метод	Зміст
`AppendText`	Додає текст у файл
`CopyTo`	Копіює файл
`Create`	Утворює файл
`CreateText`	Утворює текстовий файл
`Delete`	Вилучає файл
`MoveTo`	Переміщує файл
`Open`	Відкриває файл
`OpenText`	Відкриває файл для читання тексту
`OpenWrite`	Відкриває файл для запису

Робота з каталогами

Аналогічно до файлів, для роботи з каталогами бібліотека .NET містить клас `Directory` зі статичними методами та клас

`DirectoryInfo`, екземпляр якого використовують за необхідності здійснення декількох операцій з каталогом.

Серед методів класу `Directory` зазначимо такі:

Метод	Зміст
`CreateDirectory`	Утворює новий каталог
`Delete`	Знищує каталог
`Exists`	Визначає, чи існує каталог
`GetCurrentDirectory`	Повертає поточний каталог
`GetDirectories`	Повертає імена всіх підкаталогів у каталозі
`GetFiles`	Повертає імена всіх файлів у каталозі
`GetFileSystemEntries`	Повертає імена всіх файлів і підкаталогів
`GetLogicalDrivers`	Повертає список логічних дисків
`GetParent`	Повертає назву батьківського каталогу
`Move`	Переміщує каталог
`SetCurrentDirectory`	Встановлює поточний каталог

Клас `Directory` містить також декілька методів для повернення або встановлення дати утворення файла, останнього доступу, останнього запису.

Властивості класу `DirectoryInfo` подібні до властивостей класу `FileInfo`. З додаткових властивостей Зазначимо `Root` (повертає кореневу частину шляху) та `Parent` (повертає об'єкт, який є батьківським каталогом).

Методи класу `DirectoryInfo` подібні до методів класу `Directory`.

Моніторинг файлової системи

Бібліотека .NET дає змогу програмі вести моніторинг змін у файловій системі. Ця можливість реалізована в класі `FileSystemWatcher`, який відстежує такі події:

Подія	Зміст
`Changed`	Модифікація файла або каталогу
`Created`	Утворення файла або каталогу
`Deleted`	Вилучення файла або каталогу
`Error`	Переповнення внутрішнього буфера
`Renamed`	Перейменування файла або каталогу

Щоб опрацювати одну із цих подій, необхідно утворити відповідну функцію та долучити її до списку обробників події.

Наприклад, для опрацювання події внесення змін у файли можна написати:

```
public static void OnChanged(
          object sender, FileSystemEventArgs e){
 MessageBox.Show(e.FullPath+": "+e.ChangeType);
}
```

Сигнатура методу повинна відповідати сигнатурі делегата `FileSystemEventHandler`. Клас `FileSystemEventArgs` має три властивості, які характеризують подію: `Name` або `FullPath` містять назву чи повний шлях зміненого файла (каталогу), а `ChangeType` – ознаку змін.

Наведемо тепер фрагмент коду, який демонструє використання класу `FileSystemWatcher`:

```
FileSystemWatcher fsw = new
           FileSystemWatcher("C:\\");
fsw.Filter = "*.txt";
fsw.Changed += new
           FileSystemEventHandler(OnChanged);
fsw.NotifyFilter =
  NotifyFilters.LastAccess|NotifyFilters.LastWrite;
fsw.EnableRaisingEvents = true;
```

Код утворює об'єкт класу `FileSystemWatcher` і дає вказівку цьому об'єкту відстежувати зміни на диску C: у файлах із розширенням .txt. Потім додає обробник подій, який клас `FileSystemWatcher` активізує при отриманні повідомлення від операційної системи про внесення змін у txt-файли, зазначає, які події відстежувати, та вмикає моніторинг.

Перелічимо властивості класу `FileSystemWatcher`:

Властивість	**Зміст**
`EnableRaisingEvents`	Увімкнення/вимкнення моніторингу
`Filter`	Фільтр файлів для відстеження
`IncludeSubdirectories`	Ознака моніторингу дочірніх підкаталогів
`InternalBufferSize`	Розмір внутрішнього системного буфера
`NotifyFilter`	Список подій для моніторингу
`Path`	Шлях, по якому відстежують зміни
`SynchronizingObject`	Керування потоком виконання, який відстежує зміни

Властивість `NotifyFilter` – це набір прапорців з перерахунку `NotifyFilters`:

Ознака	Значення	Моніторинг змін
`Attributes`	4	Атрибутів файлів
`CreationTime`	64	Часу створення файлів і каталогів
`DirectoryName`	2	Імен каталогів
`FileName`	1	Імен файлів
`LastAccess`	32	Часу останнього доступу
`LastWrite`	16	Часу останнього запису
`Security`	256	Атрибутів безпеки
`Size`	8	Розміру

Потоки введення-виведення

Обмін інформацією передбачає наявність двох суб'єктів, які є джерелом та адресатом цієї інформації. Одним із суб'єктів обміну інформацією є програма. Якщо інформація передається від програми, то кажуть, що *інформація виводиться*. І навпаки, якщо програма отримує інформацію, то маємо справу з *введенням інформації*.

Суб'єкт обміну інформацією, який може її зберігати, називають *сховищем даних*. Це може бути файл, оперативна пам'ять, мережеве з'єднання, URL та інше.

Механізм, який реалізує процес обміну інформацією, називають *потоком введення/виведення*. При утворенні потоку йому вказують сховище даних, з яким він працюватиме.

Бібліотека .NET чітко розділяє сховища даних і потоки. Власне тому розглянуті у попередньому розділі класи роботи з файлами дають змогу широко маніпулювати файлами загалом, однак майже зовсім не містять функцій зчитування та запису даних.

Клас Stream

Усі потоки .NET є екземплярами класів, похідних від базового абстрактного класу `Stream`. Цей клас має п'ять загальнодоступних властивостей:

Властивість	Зміст
CanRead	Повертає ознаку доступу до читання даних
CanSeek	Повертає ознаку коректності переходу на задану позицію в потоці
CanWrite	Повертає ознаку доступу для запису даних
Length	Повертає довжину потоку в байтах
Position	Повертає поточну позицію у потоці

Властивості CanRead, CanSeek і CanWrite повертають значення true або false. Доступ до читання або запису визначається при утворенні потоку. Всі властивості (окрім Position) мають доступ лише для читання.

Клас Stream містить одне статичне поле: Null. Це поле повертає потік, з яким не пов'язане сховище даних. При читанні з потоку Stream.Null ніякі дані не повертаються, а при записі в потік – не зберігаються.

Перелічимо загальнодоступні методи класу Stream:

Метод	Зміст
BeginRead	Починає асинхронну операцію зчитування
BeginWrite	Починає асинхронну операцію записування
Close	Закриває потік і звільняє зайняті ним ресурси
EndRead	Очікує завершення асинхронної операції зчитування
EndWrite	Очікує завершення асинхронної операції записування
Flush	Записує всі дані з внутрішнього буфера у сховище даних
Read	Зчитує послідовність даних
ReadByte	Зчитує один байт
Seek	Установлює позицію в потоці
SetLength	Установлює довжину потоку
Write	Записує послідовність байт
WriteByte	Записує один байт

Клас FileStream

Потоковий клас FileStream призначений для читання та запису як текстових, так і двійкових даних у довільний файл. Зазначимо, що існують спеціалізовані класи для деяких типів файлів, які є дещо ефективнішими порівняно з FileStream. Зокрема, нижче ми розглянемо класи роботи з текстовими файлами.

Клас FileStream має дев'ять конструкторів. Чотири з них використовують дескриптор файла Windows у стилі старого API.

Ми їх не розглядатимемо. Решта конструкторів опишемо наступною схемою:

```
public FileStream( string path, FileMode mode
    [,FileAccess access
    [,FileShare share
    [,int bufferSize
    [,bool useAsync ]]]]);
```

Параметр *path* задає повне ім'я файла.

Параметр *mode* визначає режим файла і може набувати одне із значень переліку `FileMode`: `Append` (додати), `Create` (утворити), `CreateNew` (утворити новий), `Open` (відкрити), `OpenOrCreate` (відкрити або утворити), `Truncate` (обрізати). Зауважимо, що некоректне значення *mode* може породжувати виняток (наприклад, `Open` для неіснуючого файла).

Параметр *access* доступу до файла може набувати одне зі значень переліку `FileAccess`: `Read` (читання), `ReadWrite` (читання та запис), `Write` (запис). Очевидно, що при утворенні потоку з атрибутом `FileAccess.Read` запис у файл буде недоступним.

Параметр *share* визначає доступ до файла з інших потоків. Він може набувати одне зі значень переліку `FileShare`: `None` (нема), `Read` (читання), `ReadWrite` (читання та запис), `Write` (запис). Якщо потік успішно утворений, то інші потоки можуть працювати з цим файлом відповідно до значення параметра *share* зазначеного потоку.

Параметр *bufferSize* задає бажаний розмір у байтах внутрішнього буфера читання. Параметр *useAsync* визначає, чи буде файл відкрито асинхронно.

Наведемо приклади утворення файлових потоків:

```
FileStream fs1 = new FileStream(
      @"C:\Temp\File1.cs", FileMode.Create);
FileStream fs2 = new
   FileStream(@"C:\Temp\File2.cs",
      FileMode.Create, FileAccess.Write);
```

```
FileStream fs3 = new
   FileStream(@"C:\Temp\File3.cs",
      FileMode.Create, FileAccess.Read,
      FileShare.None, 2048, true);
```

Перший конструктор утворює потік і файл `File1.cs` з доступом „читання-запис” та надає іншим потокам доступ на читання. Другий конструктор утворює файл `File2.cs` з доступом лише для запису та надає іншим потокам доступ на читання. Третій конструктор утворює файл `File3.cs` для асинхронного доступу лише для читання, без права доступу іншим потокам, а також установлює розмір буфера.

Клас `FileStream`, окрім успадкованих від `Stream`, має такі властивості:

Властивість	Зміст
`Handle`	Дескриптор операційної системи для відкритого файла
`IsAsync`	Повертає ознаку асинхронності доступу
`Name`	Повертає ім’я об’єкта `FileStream`

Клас `FileStream` має два додаткових методи: `Lock` та `Unlock`. Метод `Lock` забороняє доступ до файла іншим потокам, а `Unlock` скасовує цю заборону.

Асинхронне введення-виведення

За синхронних операцій введення-виведення робота програми зупиняється до завершення цих операцій. Якщо програма працює з великими файлами або з мережевими потоками, то складається враження, що вона „зависла”.

За асинхронного введення-виведення виділяється окремий потік виконання. Це дає змогу програмі працювати, не очікуючи завершення операції введення-виведення. Зазначимо, що асинхронні методи не можна активізувати у випадку, якщо код, наступний за операцією введення-виведення, використовує результат цієї операції.

Щоб надати можливість повідомити програму про завершення асинхронної операції, список параметрів асинхронних методів містить параметр типу `AsyncCallback`. Цей параметр дає змогу означити функцію зворотного виклику, що передається

як делегат. Коли асинхронна операція завершилася, то викликається функція зворотного виклику, в якій можна працювати з результатами операції.

Делегат `AsyncCallback` означений так:

```
public delegate void
          AsyncCallback(IAsyncResult ar);
```

Оголосимо потокову змінну та функцію зворотного виклику:

```
public FileStream fs;
public void OnReadFinished(IAsyncResult asyncResult)
{
  int ReadCount = fs.EndRead(asyncResult);
  MessageBox.Show("Прочитано "+ReadCount+" байт.");
}
```

Наведемо тепер приклад використання асинхронних методів:

```
//пам'ять для читання
byte[] buffer = new byte[100000];
//відкриття файла
fs = new FileStream(@"c:\temp\file1.cs",
           FileMode.Open, FileAccess.Read,
           FileShare.None, 2048, true);
//присвоєння делегату функції зворотного виклику
AsyncCallback acb = new
          AsyncCallback(OnReadFinished);
//асинхронне читання
fs.BeginRead(buffer, 0, 100000, acb, null);
// наступний код, який виконуватиметься без
// очікування на завершення операції читання
```

Як тільки операцію асинхронного введення буде завершено, активізується метод `OnReadFinished`.

Зауважимо: якщо файл відкритий для синхронного доступу, то асинхронні методи працюватимуть також синхронно.

Клас MemoryStream

Клас `MemoryStream` призначений для обміну даними між програмою та областю пам'яті, яка є в цьому випадку сховищем даних.

Клас має декілька конструкторів. Найінформативніший з них такий:

```
public MemoryStream(
  [byte[] buffer[, int index[, int count
  [, bool writable[, bool publiclyVisible]]]]]);
```

З цього синтаксису бачимо, що конструктор може не мати аргументів або мати лише декілька з них.

Параметр *`buffer`* задає масив байтів, який слугуватиме сховищем даних. Якщо аргумент відсутній, то утворюється динамічний буфер з нульовим розміром.

Параметр *`index`* зазначає індекс у масиві *`buffer`*, з якого починається сховище даних. За замовчуванням параметр дорівнює нулю.

Параметр *`count`* зазначає кількість байт в масиві *`buffer`*, виділених для сховища даних.

Параметр *`writable`* задає значення властивості `CanWrite`.

Параметр *`publiclyVisible`* визначає, чи можна отримати масив (тобто сховище даних) з допомогою методу `GetBuffer`.

Наступний код демонструє використання класу `MemoryStream` для копіювання файла в пам'ять:

```
FileStream fs =
File.OpenRead(@"c:\temp\file.cs");
MemoryStream ms = new MemoryStream();
ms.SetLength(fs.Length);
fs.Read(ms.GetBuffer(), 0, (int)fs.Length);
ms.Flush();
fs.Close();
```

Клас `MemoryStream`, окрім успадкованих від `Stream`, має додаткову властивість `Capacity`. Ця властивість повертає або встановлює кількість байт, виділених для потоку.

Клас `MemoryStream` має також два додаткових методи: `GetBuffer` повертає (якщо дозволено) внутрішній буфер, а `WriteTo` записує дані потоку в інший потік.

Клас NetworkStream

Клас `NetworkStream` отримує та відсилає через мережу байти процесу, який працює на другому кінці мережевого з'єднання. Цей клас розташований у просторі імен `System.Net.Sockets`.

Для утворення потоку необхідно задати мережеве з'єднання на основі сокета. Сокет – це об'єкт класу `Socket` або успадкованих від `Socket` класів: `TcpListener`, `TcpClient` та інших.

Для означення сокета потрібно задати адресу *хоста* (host). Ця адреса може бути числовою адресою TCP/IP або іменем хоста, яке операційна система може перетворити в числову адресу. Список таких імен хостів можна отримати з допомогою статичних методів класу `Dns`, розташованого у просторі імен `System.Net`. Ім'я хоста *`localhost`* представляє комп'ютер, на якому запущено програму.

Другим елементом, який означує сокет, є номер порту (*port*). Порти з номерами від 0 до 49151 вже призначені конкретним службам. Отож у власних програмах доцільно використовувати порти з номерами від 49152 до 65536.

Узагальнений конструктор класу `NetworkStream` має вигляд:

```
public NetworkStream( Socket socket,
  [, FileAccess access[, bool ownsSocket]]);
```

Параметр *`socket`* задає сокет і є обов'язковим.

Параметр *`access`* визначає доступ до сокета і може набувати одне зі значень переліку `FileAccess`.

Параметр *`ownsSocket`* визначає, чи потік буде власником сокета. Якщо так, то сокет буде знищуватися при завершенні роботи потоку.

Деякі класи сокетів (наприклад, `TcpClient`) містять вбудовані мережеві потоки, які можна отримати методом `GetStream`.

Клас `NetworkStream`, окрім успадкованих від `Stream`, має додаткову властивість `Capacity`, яка повертає або встановлює кількість байт, виділених для потоку.

Клас `NetworkStream` має також два додаткових методи: `GetBuffer` повертає внутрішній буфер, а `WriteTo` записує дані потоку в інший потік.

Наступний код демонструє процес введення даних з деякого мережевого сервера, тобто програма діє як клієнт:

```
byte[] buffer = new byte[2048];
TcpClient socket = new
  TcpClient("localhost", 65535);
NetworkStream ns = socket.GetStream();
int count = ns.Read(buffer, 0, 2048);
ns.Close();
socket.Close();
```

У цьому прикладі ми не утворювали екземпляр потоку, а використали для читання даних уже існуючий потік сокета. Метод `Read` робить спробу прочитати й розмістити в `buffer` перших 2048 байт, викладених в порті з номером 65535 комп'ютера, заданого хостом. Якщо на цьому комп'ютері відсутня програма (сервер), яка викладає дані на порт 65535, то виконання наведеного коду завершиться винятком `SocketException`.

Наступний код демонструє процес викладання даних на деякий порт, тобто програма діє як сервер:

```
string msg="Зв'язок із сервером встановлений";
TcpListener tcpl = new TcpListener(65535);
tcpl.Start();
Socket sckt = tcpl.AcceptSocket();
if (sckt.Connected) {
 NetworkStream ns= new NetworkStream(sckt);
 byte[] bytes= Encoding.ASCII.GetBytes(msg);
 ns.Write(bytes, 0, bytes.Length);
```

```
        ns.Flush();
        ns.Close();
        sckt.Close();
    }
```

Метод `tcpl.Start` дає команду почати прослуховування порту в очікуванні нового з'єднання по мережі. Метод `AcceptSocket` очікує реального підключення клієнта. Коли підключення відбулося, метод повертає об'єкт `sckt` класу `Socket`, який використовують для пересилання даних.

Інші потокові класи

З метою підвищення ефективності коду в .NET реалізовано успадкований від `Stream` клас `BufferedStream`, який додає рівень буферизації до іншого потоку для операцій читання та запису. Цей клас не може бути успадкованим. При використанні класу `BufferedStream` операційна система надає дані блоками, які забезпечують найкращу продуктивність. Окрім того, клас може зберігати в пам'яті результати декількох операцій запису, і записати їх у сховище даних лише тоді, коли цей запис буде найефективнішим.

Простір імен `System.Security.Cryptography` має ще один успадкований від `Stream` клас – `CryptoStream`. Цей клас є потоковою реалізацією криптографії. Його можна додати до існуючого потоку з метою шифрування інформації.

Введення-виведення типізованих даних

Розглянуті вище потокові класи є зручними для переміщення даних як послідовності байтів. Однак вони недостатньо ефективні для операцій над даними. Наприклад, щоб отримати з файла ціле число типу `int`, необхідно прочитати 4 байти в масив байтів, який після цього потрібно інтерпретувати як ціле.

Щоб забезпечити більш структурований доступ до даних із потоків, у просторі `System.IO` розташовано кілька класів для зчитування та запису даних.

Введення-виведення двійкових даних

Клас `BinaryWriter` призначений для запису значення типізованої змінної в потік. Конструктор класу має такий синтаксис:

```
BinaryWriter([Stream output,
              [Encoding encoding]]);
```

Параметр *`output`* зазначає потік, з яким працюватиме клас. Параметр *`encoding`* дає змогу під час виводу символьних даних в потік конвертувати їх в інші кодові формати.

Наступний код демонструє використання класу `BinaryWriter`:

```
FileStream fs=File.Create(@"c:\temp\file.dat");
BinaryWriter bw = new BinaryWriter(fs);
bw.Write((float) 3.14);
int i = 10;
bw.Write(i);
string s = "стрічка";
bw.Write(s);
bw.Flush();
bw.Close();
```

Клас `BinaryWriter` має єдину властивість `BaseStream`, яка повертає потік, в який пише об'єкт класу.

Перелічимо загальнодоступні методи класу `BinaryWriter`:

Метод	Зміст
`Close`	Закриває об'єкт класу та відповідний йому потік
`Flush`	Записує всі дані з буфера у сховище даних потоку
`Seek`	Установлює позицію в потоці
`Write`	Записує типізоване значення в потік

Зазначимо, що метод `Write` сильно перевантажений, оскільки підтримує запис більшості основних типів даних.

Клас `BinaryReader` призначений для читання даних з потоку в типізовані змінні. Конструктор класу має такий синтаксис:

```
BinaryReader(Stream input, [Encoding encoding]);
```

Параметр *input* визначає потік, з яким працюватиме клас. Необов'язковий параметр *encoding* дає змогу у процесі читання символьних даних з потоку конвертувати їх в інші кодові формати.

Наступний код демонструє використання класу BinaryReader:

```
FileStream fs =
    File.OpenRead(@"c:\temp\file.dat");
BinaryReader br = new BinaryReader(fs);
float f = br.ReadSingle();
int i = br.ReadInt32();
string s = br.ReadString();
br.Close();
```

Клас `BinaryReader` має єдину властивість `BaseStream`, яка повертає потік, з якого читає об'єкт класу.

Призначення наступних методів класу очевидне з назви: `Close`, `ReadBoolean`, `ReadByte`, `ReadBytes`, `ReadChar`, `ReadChars`, `ReadDecimal`, `ReadDouble`, `ReadInt16`, `ReadInt32`, `ReadInt64`, `ReadSByte`, `ReadSingle`, `ReadString`, `ReadUInt16`, `ReadUInt32`, `ReadUInt64`. Окрім перелічених клас має ще два методи: `PeekChar` повертає наступний символ, не змінюючи позиції в потоці, а `Read` повертає символ і змінює позицію.

Читання та запис тексту

Класи `FileStream`, `BinaryReader` та `BinaryWriter` використовують для читання та запису текстових файлів. Однак, з метою підвищення ефективності та зручності програмування, доцільно використовувати для роботи з текстом спеціальні класи, які є нащадками абстрактних класів `TextReader` та `TextWriter`. Методи цих класів вміють читати та записувати стрічку, тобто набір символів, який завершується комбінацією „повернення каретки – переведення стрічки" (`(char)13(char)10`). Окрім цього, класи автоматично розпізнають і підтримують задане у файлі кодування символів (ASCII, Unicode, UTF7, UTF8).

Наведемо деякі загальнодоступні методи класу `TextReader`:

Метод	Зміст
`Close`	Закриває об'єкт і звільняє зайняті ним ресурси
`Peek`	Повертає наступний символ без зміни позиції
`Read`	Читає символи з відповідного об'єктові потоку
`ReadBlock`	Читає буфер з відповідного об'єктові потоку
`ReadLine`	Читає стрічку символів з відповідного об'єктові потоку
`ReadToEnd`	Читає символи від поточної позиції до кінця потоку

Клас `TextWriter` має такі методи:

Метод	Зміст
`Close`	Закриває об'єкт і звільняє зайняті ним ресурси
`Flush`	Записує всі дані з буфера у сховище даних
`Write`	Записує символи у відповідний об'єктові потік
`WriteLine`	Записує стрічку у відповідний об'єктові потік

Клас `TextWriter` має також властивості:

Властивість	Зміст
`Encoding`	Повертає поточне кодування тексту
`FormatProvider`	Повертає об'єкт із урахуванням культурозалежних аспектів форматування
`NewLine`	Повертає або встановлює послідовність символів, які означають кінець стрічки тексту

Реалізацією класів `TextReader` та `TextWriter` для роботи з потоками є класи `StreamReader` та `StreamWriter`.

Узагальнена схема конструктора класу `StreamReader` має такий вигляд:

```
public StreamReader(
  Stream stream | string path
  [, Encoding encoding
  [, bool detectEncodingFromByteOrderMarks
  [, int bufferSize]]]);
```

Зазначимо, що схема не охоплює синтаксис усіх конструкторів класу. Режим і тип доступу не використовують, оскільки клас призначений лише для читання.

Клас `StreamReader` може читати дані з файла (параметр *`path`*) або з іншого потоку (параметр *`stream`*).

Параметр *`encoding`* задає метод кодування.

Параметр *detectEncodingFromByteOrderMarks* зазначає, чи шукати ознаку кодування символів у файлі (потоці).

Параметр *bufferSize* задає мінімальний розмір буфера для читання.

Екземпляр класу `StreamReader` можна також отримати з інших класів. Наприклад, екземпляр цього класу повертають методи `OpenText` і `CreateText` класу `FileInfo`.

Клас `StreamWriter` працює практично так само, як і `StreamReader`, тільки використовується для запису у файл або інший потік. Конструктори класу `StreamWriter` використовують ті ж параметри, що й конструктори класу `StreamReader`.

Наступний код демонструє використання класів `StreamReader` і `StreamWriter`:

```
FileStream fs = File.Create(@"e:\File.txt");
//Запис у файл
StreamWriter sw = new StreamWriter(fs);
sw.WriteLine("Записано класом StreamWriter.");
sw.Flush();
sw.Close();
//Читання з файла
StreamReader sr = new
    StreamReader(@"e:\File.txt");
MessageBox.Show(sr.ReadLine());
sr.Close();
```

Класи `StringReader` і `StringWriter` також є дочірніми для класів `TextReader` і `TextWriter`. Сховищем даних для цих класів є стрічка. Класи `StringReader` і `StringWriter` мають такі ж методи та властивості, як і класи `StreamReader` - `StreamWriter`. Додатково клас `StreamWriter` має метод `GetStringBuilder`, який повертає об'єкт класу `StringBuilder`.

Модифікуємо попередній код під використання класів `StringReader` і `StringWriter`:

```
StringBuilder sb = new StringBuilder();
//Запис у стрічку
```

```
StreamWriter sw = new StreamWriter(sb);
sw.WriteLine("Записано класом StreamWriter.");
sw.Close();
//Читання зі стрічки
StreamReader sr = new
    StreamReader(sb.ToString());
MessageBox.Show(sr.ReadToEnd());
sr.Close();
```

Серіалізація

Серіалізація – це процес перетворення даних об'єкта у формат, який можна зберігати або транспортувати. Серіалізація дає змогу записати об'єкт у сховище даних з допомогою потоку. Зворотний процес – відновлення об'єкта з допомогою потоку зі сховища даних – називають *десеріалізацією*.

Серіалізувати можна лише об'єкти класів, які перед своїм оголошенням містять атрибут `[Serializable]`.

.NET надає дві технології серіалізації.

Бінарна (двійкова) серіалізація передбачає збереження точної копії об'єкта. Це може бути корисним для відтворення об'єкта в різних сеансах роботи програми, для передавання між різними програмами та мережею. З метою відтворення бінарно серіалізованого об'єкта програма повинна знати його тип. У випадку бінарної серіалізації об'єкт-десеріалізатор залежний від версії модуля, у якому серіалізовано стан відповідного об'єкта.

Наступна схема демонструє процес бінарної серіалізації-десеріалізації.

```
//клас Sample будемо серіалізувати
[Serializable]
public class Sample {
  public string Title;
}
//фрагмент коду серіалізації
Sample obj = new Sample();
obj.Title = "Об'єкт класу Sample";
FileStream writer = new
 FileStream("c:\\Sample.dat", FileMode.Create);
```

```
BinaryFormatter bf = new BinaryFormatter();
//бінарна серіалізація
bf.Serialize(writer, obj);
writer.Close();
//фрагмент коду десеріалізації
FileStream reader = new
 FileStream("c:\\Sample.dat", FileMode.Open);
Sample newobj =
  (Sample)bf.Deserialize(reader);
```

Клас `BinaryFormatter` оголошений у просторі імен `System.Runtime.Serialization.Formatters.Binary`.

XML-серіалізація дає змогу зберегти лише загальнодоступні властивості та поля. Здебільшого цього достатньо для смислового відтворення об'єкта. Оскільки XML – відкритий стандарт, XML-серіалізація є хорошим засобом спільного використання даних у мережі. У випадку XML-серіалізації проблема версійності складеного модуля, яким виконано серіалізацію стану об'єкта, не виникає. Проте необхідно зважати на суттєве зростання пакетів серіалізованої інформації (до об'єму даних додається суттєвий об'єм XML-розмітки).

XML-серіалізація може бути виконана, наприклад, наступним кодом:

```
XmlSerializer ser = new
    XmlSerializer(typeof(Sample));
TextWriter writer = new
    StreamWriter("c:\\Sample.xml");
ser.Serialize(writer, obj);
writer.Close();
```

Клас `XmlSerializer` оголошений у просторі імен `System. Xml.Serialization`.

Різновидом-XML серіалізації є SOAP серіалізація, яка використовує відкритий стандарт SOAP. Для її реалізації можна використати спеціалізований клас `SoapFormatter`, оголошений у просторі імен `System.Runtime.Serialization. Formatters.Soap`.

ДОДАТКИ

Складені модулі

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).

Концепція

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 замовчуванням 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.
- Задання стійкого імені. У файлі `AssemblyInfo.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`.

Атрибути складених модулів

Атрибути складеного модуля використовують для опису властивостей модуля.

Перелічимо основні атрибути складених модулів.

Атрибут	Зміст
Атрибути ідентифікації	
`AssemblyCultureAttribute`	Визначає культури, які підтримуються модулем
`AssemblyFlagsAttribute`	Атрибут-перелік визначає, які типи співіснування декількох версій підтримує модуль
`AssemblyVersionAttribute`	Версія модуля
Інформаційні атрибути	
`AssemblyCompanyAttribute`	Назва компанії, яка створила модуль
`AssemblyCopirightAttribute`	Авторські права на модуль
`AssemblyFileVersion-Attribute`	Зручна для читання інформація про версію модуля
`AssemblyInfomational-VersionAttribute`	Додаткова інформація про версію
`AssemblyProductAttribute`	Ім.'я продукту, якому належить модуль
`AssemblyTrademarkAttribute`	Інформація про торгову марку, яка представляє модуль
Атрибути маніфеста	
`AssemblyDefaultAlias-Attribute`	Альтернативне ім'я модуля. Використовують, якщо повне ім'я модуля є ідентифікатором GUID
`AssemblyDescription-Attribute`	Короткий опис модуля
`AssemblyTitleAttribute`	Зручне для читання ім'я модуля. Може містити пробіли
Атрибути стійких імен	
`AssemblyDelaySignAttribute`	Ознака використання відкладеного підпису
`AssemblyKeyFileAttribute`	Ім'я файла, який містить інформацію про криптографічний ключ
`AssemblyKeyNameAttribute`	Ім'я контейнера, який містить інформацію про криптографічний ключ

.NET передбачає утворення власних атрибутів.

Значення атрибутів можна змінювати програмно.

В імені атрибута підстрічку `Attribute` можна не зазначати. Наприклад, терміни `AssemblyKeyNameAttribute` та `AssemblyKeyName` є синонімами.

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 VS.NET 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. Простий модуль також має маніфест, однак всередині маніфеста відсутня позиція `.assembly`.

Наведемо деякі приклади використання компілятора `csc`.

Код	**Зміст**
`csc.exe File.cs`	Компіляція файла `File.cs` утворює складений модуль `File.exe`
`csc.exe /t:library File.cs`	Компіляція файла `File.cs` утворює складений модуль `File.dll`
`csc.exe /out:MyFile.exe File.cs`	Компіляція файла `File.cs` утворює складений модуль `MyFile.exe`
`csc.exe /optimize /out:MyFile.exe *.cs`	Компіляція усіх C#-файлів у поточному каталозі з оптимізацією утворює складений модуль `MyFile.exe`
`csc.exe /t:module /out:MyModule.dll File.sc`	Компіляція файла `File.cs` утворює простий модуль `MyModule.dll`
`csc.exe /t:library /addmodule: MyModule.dll File.cs`	Компіляція файла `File.cs` утворює складений модуль `File.dll`, до якого додається модуль `MyModule.dll`
`csc.exe /t:library /r: ExternalAsm.dll File.cs`	Компіляція файла `File.cs` утворює складений модуль `File.dll`, до якого додається інформація про використання зовнішнього модуля `ExternalAsm.dll`

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 `al`. Наприклад, команда

```
al.exe M1.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).

Метадані та механізми відображення

Поряд із програмним кодом, який представляє логіку виконання програми, складені модулі .NET містять додаткову інформацію – метадані. Метадані складаються з атрибутів. У цьому розділі розглянемо роботу з вбудованими атрибутами, створення власних атрибутів та механізм отримання інформації про складений модуль і типи, які він містить, у процесі виконання.

.NET Атрибути

Атрибут – це клас, породжений від абстрактного базового класу `System.Attribute`. Атрибут може містити як інформацію стану, так і функціональність. Він може асоціюватися (залежно від типу) з методом, класом чи складеним модулем загалом.

До одного елемента програми можна застосувати декілька атрибутів. При цьому атрибути можна розмістити в середині однієї пари квадратних дужок, розділюючи їх комами.

Оскільки всі атрибути є предками класу `Attribute`, доцільно перелічити його методи:

Метод	Зміст
`GetCustomAttribute`	Повертає атрибут заданого класу та заданого типу
`GetCustomAttributes`	Повертає масив атрибутів заданого класу

IsDefaultAttribute	Повертає true, якщо екземпляр є атрибутом за замовчуванням для класу
IsDefined	Показує, чи застосовано заданий атрибут до заданого класу
Match	Повертає true, якщо два атрибути збігаються

Клас Attribute має також властивість TypeId, яка при реалізації в класі-нащадку повертає унікальний ідентифікатор атрибута.

Середовище .NET містить тисячі атрибутів. Однак найкориснішими є лише декілька. Розглянемо деякі з них.

Атрибут System.ObsoleteAttribute призначений для позначення застарілих фрагментів коду та виведення пропозиції про їхню заміну:

```
[Obsolete("Метод RandomDbl має дещо
  ефективніший аналог newRandomDbl", false)]
public static double RandomDbl {
  Random r = new Random();
  return r.NextDouble();
}
```

Нагадаємо, що підстрічку Attribute у назвах атрибутів можна не писати.

Атрибут Obsolete приймає два параметри. Перший – стрічкове повідомлення про застарілий фрагмент коду. Другий – логічне значення. Якщо це false (значення за замовчуванням), то компілятор видасть попередження про застарілий код, однак дасть змогу успішно завершити процес компіляції. Якщо ж true, то компіляція завершиться помилкою.

Атрибут System.Diagnostics.ConditionalAttribute дає змогу керувати умовною компіляцією. Цей атрибут можна використовувати з довільним методом, який повертає значення типу void. Якщо означеной відповідний символ, то компілятор не буде компілювати цей метод. А також не компілюватиме і всі оператори, які містять звертання до методу. Щоб досягнути цього результату з допомогою директив препроцесора #if…#then, їх необхідно розташувати у всіх відповідних місцях коду.

Атрибут `DllImportAttribute` використовують для доступу до функцій Windows API або інших функцій, реалізованих в DLL. Цей атрибут оголошений у просторі імен `System.Runtime.InteropServices`. Наведемо приклад використання атрибута:

```
[DllImport("KERNEL32.DLL",
                    EntryPoint="MoveFileW"]
public static extern bool MoveFile(
                   string src, string dst);
```

Функція API `MoveFile` означена у файлі `Kernel.dll`, отож ім'я цього файла присутнє в аргументах атрибута.

Ще один важливий атрибут `SerializableAttribute` (простір імен `System.Runtime.Serialization`) ми розглянули в розділі **Серіалізація**.

Користувацькі атрибути

.NET передбачає утворення власних атрибутів. Клас, який реалізовує новий атрибут, повинен успадковувати клас `System.Attribute`. Наприклад:

```
public class MyAttribute: Attribute {
  string sNote;
  public string Note {
    get {return sNote;}
  }
  public MyAttribute(string Note) {
    sNote = Note;
  }
}
```

Тепер можна атрибут `MyAttribute` застосувати до потрібних елементів:

```
[MyAttribute("Категорія A")]
public class Class1 {   }
[MyAttribute("Категорія B")]
public class Class2 {   }
```

Значення атрибутів будуть розміщені в коді MSIL програми. Ці значення можна переглянути з допомогою дизасемблера `ILDasm.exe`. Вони також є доступними для довільної аплікації, яка може прочитати MSIL-структуру.

Користувацький атрибут за замовчуванням можна застосувати до довільного елемента коду. Однак .NET дає змогу обмежити область застосування за допомогою атрибута `AttributeUsage`, який має такий синтаксис:

```
[AttributeUsage( validon,
   AllowMultiple = allowmultiple,
   Inherited = inherited )]
```

Тут *`validon`* – позиційний параметр, який визначає область застосування користувацького атрибута комбінацією прапорців `AttributeTargets`. Наприклад:

```
[AttributeUsage(AttributeTargets.Method |
  AttributeTargets.Delegate]
```

Іменований параметр *`allowmultiple`* типу `bool` визначає, чи можна атрибут використовувати багаторазово. За замовчуванням він має значення `false`.

Іменований параметр *`inherited`* типу `bool` визначає, чи значення атрибута успадковуватимуть дочірні класи. За замовчуванням він також має значення `false`.

Для читання значень атрибутів під час виконання програми можна використати статичний метод `GetCustomAttribute` класу `Attribute`:

```
MyAttribute a =
  (MyAttribute)Attribute.GetCustomAttribute(
    typeof(Class1),typeof(MyAttribute));
```

Метод `GetCustomAttribute` сильно перевантажений. У наведеному коді метод приймає два параметри. Перший – це тип елемента, атрибут якого шукаємо. Другий – це тип шуканого атрибута. Метод повертає значення типу `Attribute`, яке необхідно явно привести до потрібного типу. Якщо для

зазначеного елемента не знайдено атрибут заданого типу, то метод поверне значення `null`.

Якщо атрибутів заданого типу декілька, то для читання значень атрибутів можна використати статичний метод `GetCustomAttributes` класу `Attribute`. Наприклад:

```
object[] attrs =
  Attribute.GetCustomAttributes(typeof(Class1));
foreach (object attr in attrs) {
     //використання атрибута attr
}
```

Відображення

Відображення – це механізм отримання під час виконання програми інформації про складений модуль і типи, які в ньому розташовані. Поряд з терміном відображення часто використовують термін *рефлексія* (Reflection).

.NET платформа має широкі можливості доступу до інформації про тип того чи іншого об'єкта програми, а також генерування нового програмного коду в процесі виконання існуючого. Інструменти для роботи з метаданими в режимі виконання програми зосереджені у просторі імен `System.Reflection`. Цей простір імен містить чимало класів, інтерфейсів, структур, делегатів і переліків. Ми ж перелічимо лише деякі з класів:

Клас	Опис
`Assembly`	Представляє окремий складений модуль
`ConstructorInfo`	Представляє конструктор класу
`EventInfo`	Представляє подію класу
`FieldInfo`	Представляє поле класу
`MemberInfo`	Представляє член класу
`MethodBase`	Представляє метод або конструктор
`MethodInfo`	Представляє метод класу
`Module`	Представляє окремий модуль коду
`ParameterInfo`	Представляє параметр компонента
`PropertyInfo`	Представляє властивість класу

Ще один важливий для рефлексії клас `Type`, який представляє тип загалом, розташований у просторі імен `System`.

Розглянемо код:

```
Assembly asm = Assembly.GetExecutingAssembly();
//перебір всіх типів, оголошених у модулі
string s = "";
foreach(Type t in asm.GetTypes()) {
  s += t.Name + "\n";
  s += "  Поля\n";
  foreach(FieldInfo f in t.GetFields()) {
    s += "    " + f.Name + "\n";
  }
  s += "  Методи\n";
  foreach(MethodInfo m in t.GetMethods()) {
    s += "    " + m.Name + "\n";
  }
  s += "  Властивості\n";
  foreach(PropertyInfo p in t.GetProperties()) {
    s += "    " + p.Name + "\n";
  }
}
```

В результаті виконання коду стрічка `s` міститиме список типів у складеному модулі, їхніх полів, методів і властивостей.

Статичний метод `GetExecutingAssembly` класу `Assembly` повертає складений модуль (об'єкт типу `Assembly`), що містить код, активний в даний момент часу.

Метод `GetTypes` об'єкта `Assembly` повертає колекцію усіх типів, оголошених у складеному модулі. Ця колекція містить об'єкти типу `Type`.

Методи `GetFields`, `GetMethods` і `GetProperties` класу `Type` повертають колекції об'єктів з інформацією про поля, методи та властивості типу. Ці об'єкти мають тип, відповідно, `FieldInfo`, `MethodInfo` і `PropertyInfo`. Подібно для обраного типу можна отримати також інформацію про конструктори (метод `GetConstructors`, об'єкти типу `ConstructorInfo`), події (`GetEvents`, `EventInfo`), інтерфейси (`GetInterfaces`, `Type`)), члени типу (`GetMembers`, `MemberInfo`) та багато іншого.

Метод `GetCustomAttribute` класу `Assembly` можна використати для пошуку позначених класів або членів класів. Наприклад,

```
foreach(MethodInfo m in t.GetMethods()) {
  MyAttribute a = (MyAttribute) Attribute.
    GetCustomAttribute(m, typeof(MyAttribute));
  if(m != null) {
    //метод позначений атрибутом MyAttribute
    //виконуємо якісь дії
  }
}
```

З цією ж метою можна використати і метод `GetCustomAttributes`, успадкований класом `Type` від базового класу `MemberInfo`:

```
foreach(Type t in asm.GetTypes()) {
  MemberInfo[] myMembers = t.GetMembers();
  for(int i = 0; i < myMembers.Length; i++){
   object[] myAttributes =
      myMembers[i].GetCustomAttributes(true);
   if(myAttributes.Length > 0){
     for(int j=0; j<myAttributes.Length; j++)
      //якісь дії з аналізу та використання атрибута
    }
  }
}
```

З допомогою класів простору `System.Reflection` ми можемо під час виконання програми отримати не лише перелік складених модулів, типів і членів класів, а й інформацію про розмірності масивів, параметри методів, ієрархію типів, модифікатори та багато чого іншого.

Використання рефлексії для пізнього зв'язування

Механізм відображення дає змогу аплікації виконати код, який стане їй відомим лише під час виконання.

Припустимо, що аплікація, залежно від специфіки роботи користувача, повинна використовувати різні версії деякого метода. Одна з можливих схем реалізації такої структури програми є такою:

- для кожної версії розробляють відповідний складений модуль;
- потрібний клас і метод позначають атрибутами;
- у конфігураційному файлі аплікації зазначають ідентифікатори складеного модуля;
- з конфігураційного файла зчитують ідентифікатори складеного модуля та відповідний модуль завантажують у процес (за допомогою статичних методів `Load` чи `LoadFrom` класу `Assembly`);
- здійснюють пошук потрібного методу за атрибутами, як описано в попередньому розділі;
- запускають метод на виконання.

Конфігураційний файл аплікації (або інші джерела інформації) може містити назву складеного модуля, класу та методу. Тоді нема потреби у використанні атрибутів.

Припустимо, що модуль `Sample.Assembly.dll` у просторі імен `ClassLibrary1` містить клас `Class1`. Один з методів цього класу має сигнатуру

```
public int Method1(int aCount);
```

Наведемо код, який демонструє запуск методу `Method1` на виконання.

```
Assembly asm;
asm = Assembly.LoadFrom
  ("c:\\Sample.Assembly.dll");
Type t = asm.GetType("ClassLibrary1.Class1");
MethodInfo m;
m = t.GetMethod("Method1");
object[] args = new object[1];
args[0] = 5;
object obj = Activator.CreateInstance(t);
int i = (int)m.Invoke(obj, args);
```

```
MessageBox.Show (i.ToString());
```

Метод `LoadFrom` завантажує модуль. Метод `GetType` повертає об'єкт типу `Type` з описом типу, заданого стрічковим параметром. Зазначимо, що цей параметр повинен містити повну назву типу. У прикладі – це назва простору імен і назва класу.

Метод `GetMethod` повертає об'єкт класу `MethodInfo` з описом методу `Method1` класу `Class1`. Цей об'єкт дає змогу отримати всю загальнодоступну інформацію про метод.

Оскільки сигнатура методу нам відома, ми відразу розпочинаємо підготовку даних для виклику методу: утворюємо та ініціалізуємо масив аргументів. У складніших випадках доцільно використати клас `ParameterInfo`.

Щоб викликати нестатичний метод класу, потрібен екземпляр класу. Такий екземпляр утворюємо з допомогою методу `CreateInstance` класу `Activator`. Цей клас призначений для утворення екземплярів типів і містить лише статичні методи:

Метод	Опис
`CreateComInstanceFrom`	За іменем файла утворює екземпляр COM-типу
`CreateInstance`	Утворює екземпляр заданого типу
`CreateInstanceFrom`	Утворює екземпляр заданого типу, оголошеного в заданому файлі
`GetObject`	Утворює проксі-об'єкт для активованого віддаленого об'єкта

З цього переліку бачимо, що в наведеному вище прикладі можна було б скоротити код, використавши метод `CreateInstanceFrom` замість `CreateInstanceFrom`.

Після утворення екземпляра класу `Class1` можемо активізувати його методи. Однак при пізньому зв'язуванні компілятор не має інформації про клас `Class1`. Отож звичний синтаксис `obj.Method1` виклику методу не підходить. Викликати метод можна опосередковано, використовуючи метод `Invoke` класу `MethodBase` (або класів-нащадків). Один з варіантів цього методу має сигнатуру

```
public object Invoke(
  object obj,
  object[] parameters);
```

Параметр *obj* визначає об'єкт, для якого активізують метод (представлений екземпляром MethodBase), а параметр *parameters* задає список аргументів. Метод Invoke повертає результат типу object, тому необхідне явне зведення до потрібного типу:

```
(int)m.Invoke(obj, args);
```

Конфігурація в .NET

Конфігураційні файли

Конфігураційні файли дають змогу змінювати системні налаштування без повторної компіляції аплікації.

Конфігураційний файл – це текстовий файл, який містить XML-елементи (*теги*). Змістовна частина файла повинна бути розміщена всередині пари тегів <configuration> і </configuration>. Наступний код демонструє спрощений конфігураційний файл:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
 <appsettings>
  <add key="Application Name" value="MyApplication"/>
 </appsettings>
 <runtime>
  <gcConcurrent enabled="enabled" />
 </runtime>
</configuration>
```

Приклад містить парні теги (<appsettings> і </appsettings>, <runtime> і </runtime>). Ці теги (а також довільні інші парні теги) можуть містити набори стрічок конфігурації. Якщо тег не містить інші теги, то закриваючий тег може бути відсутнім.

Регістр символів у конфігураційних файлах є істотним.

Простір імен System.Configuration містить декілька класів, призначених для роботи з конфігураційними файлами. Зокрема:

Клас	Опис
`AppSettingsReader`	Читання значень секції `appsettings`
`ConfigurationException`	Обробка помилок конфігурування системи
`ConfigurationSettings`	Доступ до конфігураційної секції у файлі
`DictionarySectionHandler`	Читання пар ключ-значення в заданій секції

Наприклад, для наведеного вище коду метод ConfigurationSettings.AppSettings["Application Name"] поверне значення "MyApplication".

У .NET Framework існує кілька типів конфігураційних файлів: комп'ютера, аплікації, системи безпеки.

У *конфігураційному файлі комп'ютера* `machine.config` зберігаються основні налаштування системи (вбудованих віддалених каналів зв'язку, ASP.NET, зв'язки складених модулів та інше). Конфігураційний файл комп'ютера, зазвичай, розташований у каталозі <install path>\CONFIG, де <install path> – папка інсталяції системи .NET Framework.

Конфігураційний файл аплікації містить налаштування конкретної прикладної програми. Для Windows-аплікації він має таку ж назву, як і exe-файл з доданим розширенням `.config`. Наприклад, WindowsApp.exe.config. Конфігураційний файл аплікації розташовують в каталозі аплікації.

Система конфігурації переглядає конфігураційні файли аплікацій після конфігураційного файла комп'ютера.

У *конфігураційних файлах безпеки* визначається набір правил, які виконує CLR для забезпечення захисту комп'ютера від несанкціонованого коду. Ці файли безпеки визначаються трьома рівнями політик системи безпеки:

- конфігурація політики підприємства (`<install path>\CONFIG\enterprise.config`);
- конфігурація політики машини (`<install path>\CONFIG\security.config`);
- конфігурація політики користувача (`<USERPROFILE>\application data\Microsoft\ CLR security config\vxx.xx\security.config`).

Конфігураційні файли безпеки захищені системою безпеки Windows.

Приклад конфігураційного файла аплікації

Опишемо дві типові задачі керування аплікацією в .NET.

Перескеровування версій складених модулів. Припустимо, що деяка клієнтська програма `Client.exe` використовує загальнодоступний складений модуль `SharedAssembly` версії 1.0. З часом розробник програмного забезпечення випускає нову версію 1.1, в якій виправлено деякі помилки попередньої версії. Необхідно існуючій клієнтській програмі дати вказівку на використання нової версії модуля `SharedAssembly`.

Конфігурування каталогів. Інколи потрібно зробити доступними розподілені і складені модулі, проте не розміщувати їх у глобальний кеш. Існує два способи означення каталогу для складеного модуля: з допомогою елемента `codeBase` в конфігураційному файлі та з допомогою *зондування*. Перший спосіб доступний лише для розподілених складених модулів, а другий – і для приватних, і для розподілених.

Для розв'язування цих задач можна використати такий конфігураційний файл аплікації `Client.exe.config`:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns=
           "urn:schemas-microsoft-com:asm.v1"
      <dependendAssembly xmlns="">
      <assemblyIdentity name="SharedAssembly"
         publicKeyToken="7bc6357263f5e6b7" />
        <bindingRedirect oldVersion="1.0"
                         newVersion="1.1" />
        <codeBase version="1.1"
                    href="file:C:\MyAsm" />
      </dependendAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

З допомогою тега `<runtime>` конфігурують налаштування середовища виконання. Умови генерування модуля розміщують

всередині тега <assemblyBinding>, який, у свою чергу, містить тег опису залежностей <dependendAssembly>. Таких тегів опису залежностей може бути кілька, і кожен з них повинен містити тег <assemblyIdentity> ідентифікації модуля.

Задачу перескеровування версії складеного модуля вирішує тег <bindingRedirect>, який дає вказівку замість версії 1.0 використовувати версію 1.1. Значення oldVersion може бути також діапазоном версій, наприклад, 1.0.11.4–1.0.17.56.

Тег <codeBase> вирішує задачу конфігурування каталогу. Атрибут version визначає, яка версія складеного модуля має завантажуватися з каталогу, заданого атрибутом href.

Інструментальні засоби конфігурації .NET

Конфігураційні файли можна редагувати як звичайні текстові файли. Однак ця процедура передбачає наявність специфічних знань і досвіду.

.NET Framework має декілька спеціалізованих інструментальних засобів конфігурування.

Інструментальний засіб mscorcfg.msc вбудований у консоль Microsoft Management Console (MMC) і може бути запущений командою mmc mscorcfg.msc.

З допомогою mscorcfg.msc можна виконувати такі задачі:

- конфігурація та управління складеними модулями (Assembly Cash, Configured Assemblies);
- установка віддалених налаштувань (Remoting Services Properties);
- конфігурація та управління безпекою (Runtime Security Policy);
- управління аплікацією (Applications).

Утиліта політики захисту доступу до коду Caspol.exe призначена для:

- адміністрування комп'ютера та рівнів політики;
- читання та зміна політики;
- перевірки повноважень, наданих складеному модулю.

Утиліта `Caspol.exe` може запускатися з такими атрибутами:

Атрибут	Дія
`-addfulltrust assembly.dll`	Додає складений модуль, який реалізує клієнтський об'єкт безпеки, до списку повністю довірених складених модулів для конкретного рівня політики
`-addgroup`	Додає нову групу коду до ієрархії груп коду
`-listgroups`	Надає інформацію про доступ до груп коду
`-chggroup`	Вносить зміни в умови членства у групі коду
`-addpset`	Додає новий набір повноважень до політики
`-listdescription`	Виводить ієрархічний список груп коду
`-resolvegroup assembly.dll`	Виводить список груп коду, членом яких є складений модуль
`-resolveperm assembly.dll`	Надає повноваження групі коду складеного модуля
`-user -listgroups`	Виводить список груп коду на рівні користувача
`-Enterprise -listgroups`	Виводить список груп коду на рівні підприємства
`-security off`	Забороняє .NET-безпеку
`-security on`	Дозволяє .NET-безпеку

Взаємодія з COM

Імпорт бібліотеки типів

COM-компоненти та .NET-компоненти орієнтовані на різні стандарти компіляції: двійковий стандарт COM та CLS-стандарт. Для взаємодії з моделлю COM бібліотека .NET містить низку програмних технологій та утиліт. Ці програми можна використовувати для генерування компонент *проксі* (*proxi*, проміжні програми), які розпізнаються як стандартами компіляції COM, так і .NET.

Щоб додати посилання на DLL COM, можна скористатися діалоговим вікном References інструментального середовища VS. У цьому випадку VS утворює компонент проксі .NET для DLL COM і копію DLL COM розташовує в каталог проекту .NET. Якщо імпортована DLL у своєму відкритому інтерфейсі посилається на інші DLL COM, то для кожної з них утворюється свій проксі та своя копія DLL.

Проксі описує DLL COM і є для неї делегатом, який пересилає виклики від клієнта .NET через служби COM до DLL COM. У термінології .NET такий проксі називають RCW (*Runtime Callable Wrapper* – оболонка часу виконання).

Значно більші можливості від діалогового вікна References для утворення проксі надає утиліта `TblImp.exe` (*Type Library Importer*), яка є частиною технології *COM Interop* – складової .NET Framework. Утиліта активізується командною стрічкою і повинна містити аргумент *`tlbFile`* – назву файла, який містить бібліотеку типів COM. Утиліта використовує низку опцій, опис яких можна отримати з командної стрічки такими командами:

```
tblimp.exe /help
tblimp.exe /?
```

Зокрема, опція `/out` *`filename`* дає змогу задати ім'я файла проксі:

```
TblImp.exe comlib.dll /out: comlibRCW.dll
```

Утворений файл виводу (RCW) потрібно розмістити в каталог проекту. За замовчуванням класи бібліотеки будуть розміщені у просторі імен з назвою файла RCW. Для перегляду вмістимого RCW можна використати утиліту дизасемблювання `ILDasm.exe` (IL Disassembler).

Припустимо, що імпортована бібліотека має клас `classCom` з методом `Add()`, який приймає два параметри та повертає їхню суму:

```
// comlib.dll
(classCom)
public int Add(int X, int Y) {
  return X+Y;
}
```

Наведемо схематичний приклад виклику методу:

```
using comlibRCW;
comlibRCW.classCom objcom = new
                        comlibRCW.classCom();
int k;
int x = 2;
```

```
int y = 3;
k = objcom.Add(ref x, ref y);
```

Зауважимо, що наявність RCW не замінює наявність COM-компонента. Тобто COM-компонент повинен бути зареєстрований у службах COM та відповідно налаштований утилітами Windows (`regsrv32.exe`, `dcomcnfg.exe`).

Пізнє зв'язування з компонентами COM

Технологія взаємодії з COM, яка передбачає утворення RCW, використовує *раннє зв'язування* (*early binding*). У цьому випадку компілятор використовує бібліотеку типів компонента, щоб вбудувати адреси методів і властивостей компонента в виконувану клієнтську програму.

Пізнє зв'язування (*late binding*) передбачає, що програма отримає адреси властивостей або методів компонента у момент їхнього виклику. Механізм визначення цих адрес не є особливо швидким, отож використання пізнього зв'язування знижує продуктивність програми. Проте пізнє зв'язування дає змогу будувати значно гнучкіші програмні комплекси. Зокрема, можна використати поліморфізм COM компонент.

Для виклику методів об'єктів COM необхідно виконати такі дії:

- з допомогою статичного метода `GetTypeFromProgID` класу `Type` (простір імен `System.RunTime.InteropServices`) утворити об'єкт `Type` (отримати інтерфейс `IDispatch`);
- використати цей об'єкт `Type` для утворення об'єкта COM з допомогою статичного методу `CreateInstance` класу `Activator`;
- утворити масив аргументів;
- використати метод `InvokeMember` об'єкта `Type` для виклику методів об'єкта COM.

Код реалізації набуде приблизно такого вигляду:

```
using System;
using System.Reflection;
Type objcomtype =
```

```
    Type.GetTypeFromProgID("ComObjectName");
object objcom =
    Activator.CreateInstance(objcomtype);
object[] args = {2,3};
object obj;
obj = objcomtype.InvokeMember("Add",
    bindingFlags.InvokeMethod,
    null, objcom, args);
```

Імпорт елементів керування ActiveX

Для імпорту елементів керування ActiveX в .NET можна використати утиліту `AxImp.exe` (ActiveX Importer). Загальна схема активізації утиліти така:

```
aximp.exe [options]{file.dll | file.ocx}
```

Обов'язковий аргумент *`file.dll`* (або *`file.ocx`*) задає абсолютний або відносний шлях до файла ActiveX, який необхідно імпортувати.

Результатом роботи утиліти будуть два файли: проксі для модуля та елемент керування Windows (в термінах .NET). Наприклад, наступна команда генерує файли `MediaPlayer.dll` та `AxMediaPlayer.dll` для елемента керування Media Player:

```
aximp c:\winnt\system32\msdxm.ocx
```

З допомогою опцій утиліти `AxImp` назви файлів можна задавати.

Проксі `MediaPlayer.dll` дає змогу робити посилання на елемент ActiveX так, ніби він є звичайним неграфічним компонентом COM.

Елемент керування Windows `AxMediaPlayer.dll` дає змогу використовувати графічний аспект імпортованого елемента ActiveX у проектах Forms Windows .NET.

Для розташування зображення імпортованого елемента ActiveX на панелі керування IDE VS.NET необхідно вибрати його на сторінці .NET Framework Components діалогового вікна Customize ToolBox.

Використання компонент .NET у COM

Для підготовки компонента .NET до використання службами COM застосовують утиліту `RegAsm.exe` (Register Assembly). Ця утиліта вводить інформацію про тип компонента .NET у системний реєстр.

Загальна схема виклику утиліти:

```
regasm.exe assemblyFile [options]
```

Аргумент утиліти *`assemblyFile`* задає складений модуль, який реєструють служби COM.

Наведемо можливий сценарій підготовки компонента .NET до використання службами COM на прикладі класу `Points`:

```
namespace Points {
  using System;
  public class Points {
    // оголошення та реалізація класу
  }
}
```

Цей сценарій налічує такі кроки:

- утворення стійкого імені складеного модуля з допомогою утиліти `sn`: `sn.exe –k points.snk`
- утворення файла AssemblyInfo.cs, який містить код
  ```
  using System.Reflection;
  [assembly: AssemblyKeyFile("points.snk")]
  ```
- компіляція файла AssemblyInfo.cs:
  ```
  csc.exe /t:module /out:AssemblyInfo.dll
       AssemblyInfo.sc
  ```
- компіляція файла Points.cs:
  ```
  csc.exe /t:library
    /addmodule:AssemblyInfo.dll Points.sc
  ```
- розташування результуючої dll у глобальний кеш
  ```
  gacutil.exe /i Points.dll
  ```
- реєстрація Points.dll у службах COM:
  ```
  regasm.exe Points.dll
  ```

Тепер можна виконати пізнє зв'язування зі складеним модулем .NET через служби COM. Наприклад, на Visual Basic це можна зробити так:

```
Option Explicit
Dim objPoints
Set objPoints = CreateObject("Points.Points")
Call MsgBox(objPoints.ToString())
```

Описаний механізм дає змогу утворити з пізнім зв'язуванням об'єкт .NET службами COM.

Можливо отримати значно ефективніше раннє зв'язування з компонентами .NET, якщо використовувати бібліотеку типів COM (файл `*.tlb`). Таку бібліотеку типів утворює утиліта експорту `TlbExp.exe` (*Type Library Exporter*).

Загальна схема виклику утиліти:

```
tlbexp.exe assemblyName [options]
```

Аргумент утиліти *`assemblyName`* задає складений модуль, бібліотека типів якого має експортуватися. Якщо опції утиліти відсутні, то утвориться файл із назвою складеного модуля та розширенням `.tlb`.

Зауважимо, що не можна експортувати бібліотеку типів складеного модуля, імпортованого утилітою `TlbImp`. Це обмеження не стосується складених модулів, які мають посилання на імпортовані утилітою `TlbImp` модулі.

На відміну від `RegAsm`, утиліта `TlbImp` лише утворює бібліотеку типів, однак не реєструє її.

СПИСОК ЛІТЕРАТУРИ

1. Бишоп Дж., Хорспул Н. C# в кратком изложении. – М.: Бином. Лаборатория знаний, 2005. – 472с.
2. Петцольд Ч. Программирование в тональности C#. – М.: Русская Редакция, 2004. – 512с.
3. Прайс Д., Гандэрлой М. Visual C# .NET. Полное руководство. – К.: ВЕК+, СПб.: КОРОНА принт, К.: НТИ, М.: Энтроп, 2004. – 960с.
4. Робинсон С., Корнес О., Глинн Д. и др. C# для профессионалов. Том.1,2. – М.: ЛОРИ, 2003. – 1002с.
5. Троелсен Э. C# и платформа .NET. Библиотека программиста. – СПб.: Питер, 2004. – 796с.
6. Чакраборти А., Кранти Ю., Сандху Р.Д. Microsoft .NET Framework. Разработка профессиональных проектов.– СПб.: БХВ-Петербург, 2003. – 896с.
7. Шилд Г. Полный справочник по C#. – М.: Вильямс, 2004. – 752с.
8. Понамарев В. Программирование на C++/C# в Visual Studio .NET 2003. - СПб.: БХВ-Петербург, 2004. – 352с.
9. Barker J., Palmer G. Beginning C# Objects: From Concepts to Code. – Apress, 2004. – 848pp.
10. Drayton P., Albahari B., Neward T. C # in a Nutshell, Second Edition. – O'Reilly, 2003. – 928pp.
11. Gunnerson E., Wienholt N. A Programmer's Introduction to C# 2.0, Third Edition. – Apress, 2005. – 568pp.
12. Liberty J. Learning C#. - O'Reilly, 2002. – 368pp.
13. Sells C. Windows Forms Programming in C#. – Addison-Wesley Professional, 2004. - 681pp.
14. Weller D., Lobao A. S., Hatton E. Beginning .NET Game Programming in C#. – Apress, 2004. – 440pp.

З М І С Т

Навчальне видання

Голуб Богдан Михайлович

C#. Концепція та синтаксис

Навчальний посібник

Редактор *І.М.Лоїк*

Технічний редактор *С.З.Сеник*

Підписано до друку 00.03.2006. Формат 60x84 / 16.
Папір друк. № 3. Друк на різограф. Гарнітура Times New Roman.
Умовн. друк. арк. 7,9. Обл.-вид. арк. 8,2. Тираж прим. Зам.

Видавничий центр Львівського національного університету імені Івана Франка. 79000, м. Львів, вул. Дорошенка, 41.

Голуб Б.М.

Г-62 C#. Концепція та синтаксис. Навч. посібник. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2005. – 136 с.

У посібнику подано опис основних конструкцій алгоритмічної мови C#: типи даних, змінні та константи, операції, оператори керування, класи, вказівники, делегати, події. Значну увагу приділено архітектурі .NET. Розглянуто основні класи базової бібліотеки .NET, складені модулі, метадані та механізми відображення, конфігураційні файли, взаємодію з COM. Описано принципи розробки проектів в середовищі програмування Visual Studio .NET.

Для студентів факультету прикладної математики та інформатики, а також усіх бажаючих навчитися програмувати алгоритмічною мовою C# у середовищі .NET.

ББК 3973.2-018.1я73-1 C#