**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**
**ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ**

**О.С. Зеленський**
**В.С. Лисенко**

# ОСНОВИ ПРОГРАМУВАННЯ НА МОВІ C#

Навчальний посібник

Кривий Ріг
2023

Навчальний посібник з дисципліни "Основи програмування на мові C#" призначений для вивчення мови C# з використанням платформи .NET. У посібнику дається коротка суть платформи .NET, класична основа мови C# на прикладі консольних додатків. У другій частині підручника розглядається розробка windows-додатків. Мова C# і пов'язане з ним середовище .NET Framework можна назвати найзначнішою з пропонованих в даний час технологій для розробників.

Навчальний посібник адресований студентам, слухачам магістратури, аспірантам, викладачам. Може бути використаний як самовчитель.

*Автори*: Зеленський О.С., Лисенко В.С. – Кривий Ріг: Державний університет економіки і технологій, 2023.-327 с.

*Рецензенти*:

А.І. Купін, д.т.н, професор, завідувач кафедри комп'ютерних систем та мереж, Криворізький національний університет.

І.О. Музика, к.т.н, доцент кафедри комп'ютерних систем та мереж, декан факультету інформаційних технологій Криворізький національний університет.

В.Б. Хоцкіна, к.т.н., доцент кафедри інформатики і прикладного програмного забезпечення, Державний університет економіки і технологій.

*Рекомендовано Вченою радою Державного університету економіки і технологій*

*Протокол № 10 від 30.03.2023 р.*

**ЗМІСТ**

# ВСТУП

Навчальний посібник призначений для поглибленого вивчення студентами об'єктно-орієнтованого програмування на мові C# - однієї з найперспективніших сучасних мов програмування. У даному посібнику дається класична основа мови C# на прикладі консольних додатків.

Окрім конструкцій мови розглядаються основні структури даних, які використовуються при написанні програм, класи бібліотеки, а також рекомендації по стилю і технології програмування. По ключових темах приводяться завдання для лабораторних робіт, кожна з яких містить до двадцяти однотипних варіантів з розрахунку на учбову групу студентів.

Мова C# як засіб навчання програмуванню володіє рядом безперечних переваг. Вона добре організована, строга, більшість її конструкцій логічні і зручні. Розвинені засоби діагностики і редагування коду роблять процес програмування приємним і ефективним. Могутня бібліотека класів платформи .NET бере на себе масу рутинних операцій, що дає можливість вирішувати складніші завдання.

Важливо, що C# є не учбовою, а професійною мовою, призначеною для вирішення широкого спектру завдань, і насамперед - в області створення розподілених застосувань, що швидко розвиваються. Тому базовий курс програмування, побудований на основі мови C#, дозволить студентам швидше стати затребуваними фахівцями - професіоналами.

# РОЗДІЛ 1. ОСНОВИ ПЛАТФОРМИ .NET

## 1.1. Основи платформи .NET

Програма створюється на мові, зрозумілій людині, а комп'ютер вміє виконувати тільки програми, написані на його мові - в машинних кодах. Сукупність засобів, за допомогою яких програми пишуть, коригують, перетворюють в машинні коди, налагоджують і запускають, називають середовищем розробки, або оболонкою. Середовище розробки зазвичай містить:

- текстовий редактор, призначений для введення і коригування тексту програми;
- компілятор, за допомогою якого програма перекладається з мови, на якій вона написана, в машинні коди;
- засоби налагодження і запуску програм;
- загальні бібліотеки;
- довідкову систему та інші елементи.

Під платформою розуміється щось більше, ніж середовище розробки для однієї мови. Платформа .NET включає не тільки середовище розробки для декількох мов програмування, яка називається Visual Studio .NET, але і багато інших засобів, наприклад, механізми підтримки баз даних, електронної пошти та комерції.

В епоху стрімкого розвитку Інтернету - глобальної інформаційної мережі, що об'єднує комп'ютери різних архітектур, найважливішими завданнями при створенні програм стають:

- переносимість - можливість виконання на різних типах комп'ютерів;
- безпека - неможливість несанкціонованих дій;
- надійність - здатність виконувати необхідні функції у визначених умовах;
- середній інтервал між відмовами;
- використання готових компонентів - для прискорення розробки;
- міжмовна взаємодія - можливість застосовувати тимчасово кілька мов програмування.

Платформа .NET дозволяє успішно вирішувати всі ці задачі. Для забезпечення переносимості компілятори, що входять до складу платформи, переводять програму не в машинні коди, а в проміжну мову (Microsoft Intermediate Language, MSIL, або просто IL), яка не містить команд, що залежать від мови, операційної системи і типу комп'ютера. Програма цією мовою виконується не самостійно, а під управлінням системи, яка називається загальномовним середовищем виконання (Common Language Runtime, CLR).

Середовище CLR може бути реалізоване для будь-якої операційної системи. При виконанні програми CLR викликає так званий JIT-компілятор, що переводить код з мови IL в машинні команди конкретного процесора, які негайно виконуються. JIT означає «just in time», що можна перевести як «вчасно», тобто

компілюються тільки ті частини програми, які потрібно виконати в даний момент. Кожна частина програми компілюється один раз і зберігається в кеші для подальшого використання. Схема виконання програми при використанні платформи .NET наведена на рис. 1.1.

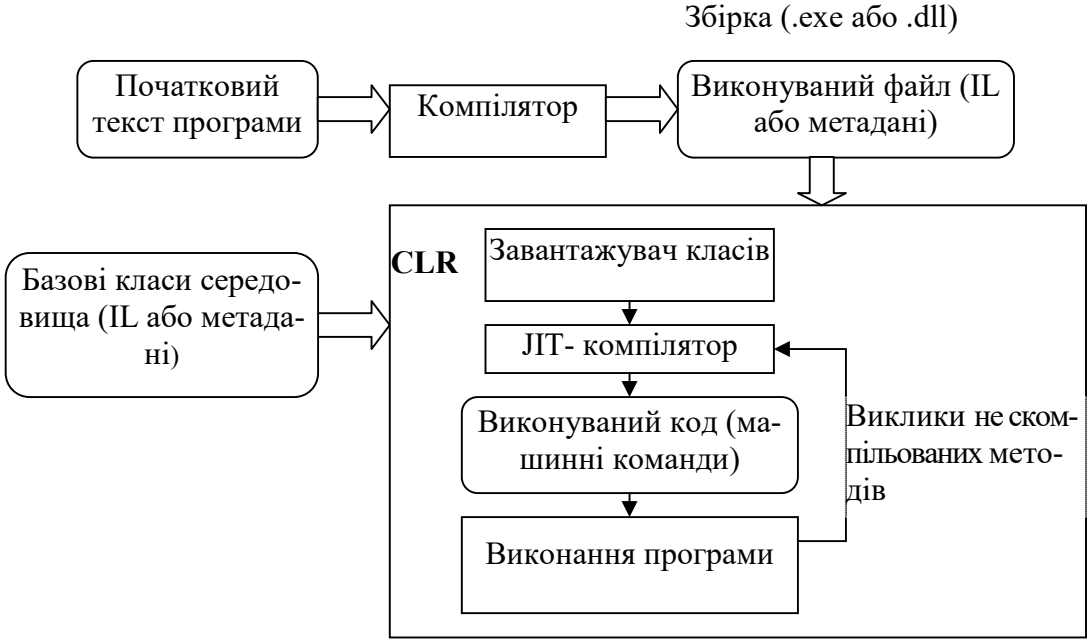



Рис. 1.1. Схема виконання програми в .NET

Компілятор як результат свого виконання створює так звану збірку - файл з розширенням exe або dll, який тримає код на мові IL і метадані. Метадані – це відомості про об'єкти, що використовуються в програмі, а також відомості про саму збірку. Вони дозволяють організувати міжмовну взаємодію, забезпечують безпеку і полегшують розгортання додатків, тобто установку програм на комп'ютерах користувачів. Збірка може складатися з декількох модулів. У будь-якому випадку вона являє собою програму, готову для установки і не вимагає для цього ні додаткової інформації, ні складної послідовності дій. Кожна збірка має унікальне ім'я.

Під час роботи програми CLR стежить за тим, щоб виконувались тільки дозволені операції, здійснює розподіл та очищення пам'яті і обробляє помилки. Це багатократно підвищує безпеку і надійність програм.

Платформа. NET містить величезну бібліотеку класів, які можна використовувати при програмуванні на будь-якій мові. NET. Загальна структура бібліотеки наведена на рис. 1.2. Бібліотека має декілька рівнів. На самому нижньому знаходяться базові класи середовища, які використовуються при створенні будь-якої програми: класи введення-виведення, обробки рядків, управління безпекою, графічного інтерфейсу користувача, зберігання даних і т.п.

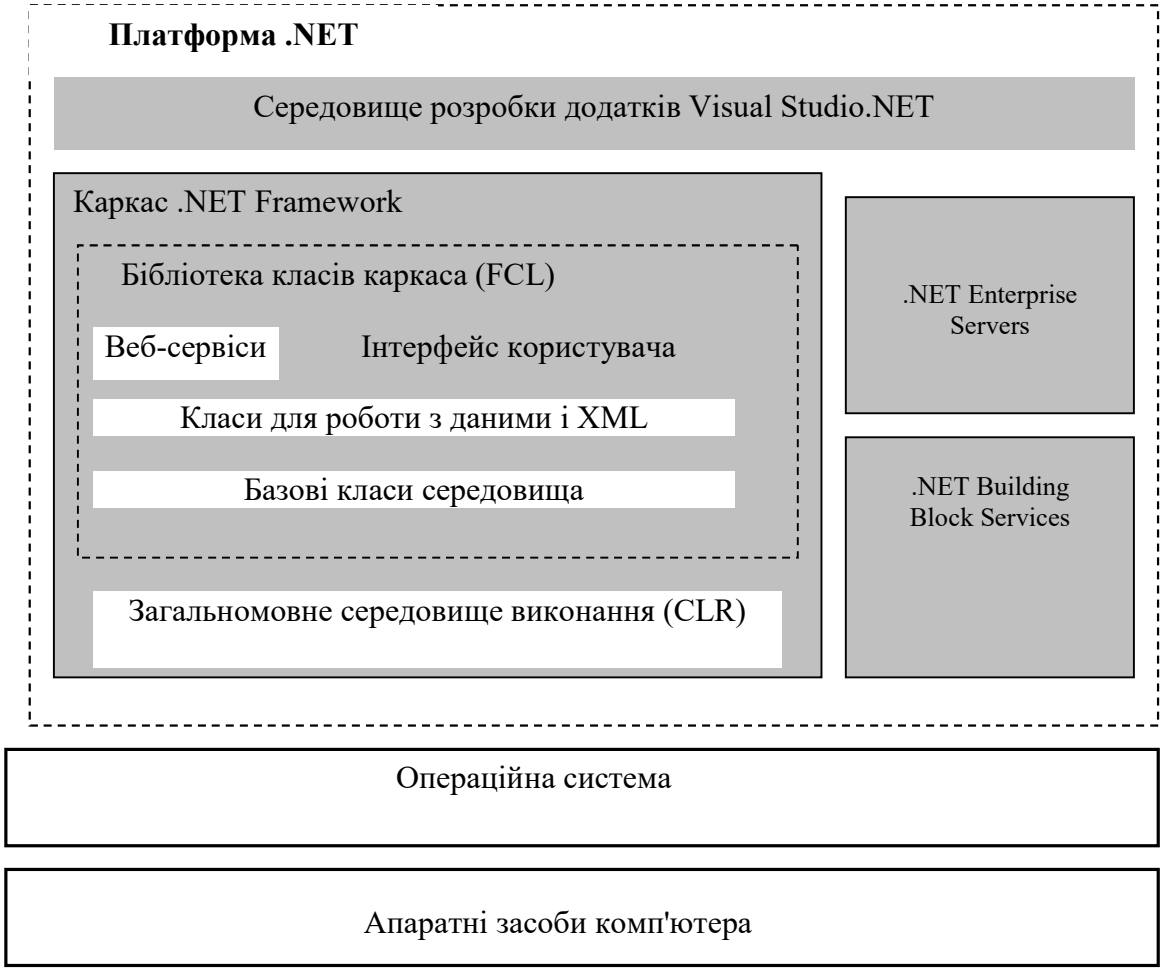



Рис. 1.2. Структура платформи .NET

Над цим шаром знаходиться набір класів, що дозволяє працювати з базами даних і XML. Класи самого верхнього рівня підтримують розробку розподілених застосувань, а також Web і Windows-додатків. Програма може використовувати класи будь-якого рівня.

Докладне вивчення бібліотеки класів .NET необхідне, але і найбільш трудомістке завдання програміста при освоєнні цієї платформи. Бібліотека класів разом з CLR утворюють каркас (Framework), тобто основу платформи. Призначення решти частин платформи ми розглянемо у міру вивчення матеріалу. Платформа .NET розрахована на об'єктно-орієнтовану технологію створення програм, тому перш ніж починати вивчення мови C#, необхідно познайомитися з основними поняттями об'єктно-орієнтованого програмування (ООП).

## 1.2. Загальні відомості об'єктно-орієнтованого програмування

Для тих, хто вивчав мову C++ підрозділи 1.2, 1.3 можна було б пропустити. Проте для узагальнення знань по ООП пропонуємо ці підрозділи прочитати. Принципи ООП найпростіше зрозуміти на прикладі програм моделювання. На реальному світі кожен предмет або процес володіє набором статичних і динамічних характеристик, іншими словами, властивостями і поведінкою. Поведінка

об'єкту залежить від його стану і зовнішніх дій. Наприклад, об'єкт “автомобіль” нікуди не поїде, якщо в баку немає бензину, а якщо повернути кермо, зміниться положення коліс.

Поняття об'єкту в програмі збігається з буденним сенсом цього слова: об'єкт представляється як сукупність даних, що характеризують його стан, і функцій їх обробки, що моделюють його поведінку. Виклик функції на виконання часто називають посилкою  повідомлення об'єкту. Наприклад, виклик функції “повернути кермо” інтерпретується як посилка повідомлення “автомобіль”, поверни кермо!”.

При створенні об'єктно-орієнтованої програми наочна область представляється у вигляді сукупності об'єктів. Виконання програми полягає в тому, що об'єкти обмінюються повідомленнями. Це дозволяє використовувати при програмуванні поняття, що адекватніше відображають наочну область.

При представленні реального об'єкту необхідно виділити  його особливості. Їх список залежить від мети моделювання. Наприклад, об'єкт “щур” з погляду біолога, що вивчає міграції, ветеринара або, скажімо, кухаря матиме абсолютно різні характеристики. Виділення істотних з тієї або іншої точки зору властивостей називається абстрагуванням. Таким чином, програмний об'єкт - це абстракція.

Важливою властивістю об'єкту є його відособленість. Деталі реалізації об'єкту, тобто внутрішні структури даних і алгоритми їх обробки, приховані від користувача об'єкту і недоступні для ненавмисних змін. Об'єкт використовується через його інтерфейс - сукупність правил доступу. Приховування деталей реалізації називається інкапсуляцією (від слова “капсула”). Нічого складного в цьому понятті немає: адже і в звичайному житті ми користуємося об'єктами через їх інтерфейси. Скільки інформації довелося б тримати в голові, якби для перегляду новин треба було знати пристрій телевізора.

Таким чином, об'єкт є “чорним ящиком”, замкнутим по відношенню до зовнішнього світу. Це дозволяє представити програму в укрупненому вигляді - на рівні об'єктів і їх взаємозв'язків, а отже, управляти великим об'ємом інформації і успішно відладити складні програми.

Сказане можна сформулювати коротше і строго: об'єкт - це інкапсульована абстракція з чітко певним інтерфейсом.

Інкапсуляція дозволяє змінити реалізацію об'єкту без модифікації основної частини програми, якщо його інтерфейс залишився тим самим. Простота модифікації є дуже важливим критерієм якості програми, адже будь-який програмний продукт протягом свого життєвого циклу зазнає безліч змін і доповнень.

Крім того, інкапсуляція дозволяє використовувати об'єкт в іншому оточенні і бути упевненою, що він не зіпсує області пам'яті, що не належать йому, а також створювати бібліотеки об'єктів для застосування в багатьох програмах.

Щороку в світі пишеться величезна кількість нових програм, і найважливішого значення набуває можливість багатократного використання коду. Перевага об'єктно-орієнтованого програмування полягає в тому, що для об'єкту можна визначити спадкоємців, що коректують або доповнюють його поведінку.

При цьому немає необхідності не тільки повторювати початковий код батьківського об'єкту, але навіть мати до нього доступ.

Спадкоємство є щонайпотужнішим інструментом ООП і застосовується для наступних взаємозв'язаних цілей:

- виключення з програми фрагментів коду, що повторюються;
- спрощення модифікації програми;
- спрощення створення нових програм на основі тих, що існують.

Крім того, тільки завдяки спадкоємству з'являється можливість використовувати об'єкти, початковий код яких недоступний, але в яких потрібно внести зміни.

Спадкоємство дозволяє створювати ієрархії об'єктів. Ієрархія представляється у вигляді дерева, в якому більш загальні об'єкти розташовуються ближче до кореню, а більш спеціалізовані - на гілках і листках. Спадкоємство полегшує використання бібліотек об'єктів, оскільки програміст може узяти за основу об'єкти, розроблені якимось іншим, і створити спадкоємців з необхідними властивостями.

Об'єкт, на підставі якого будується новий об'єкт, називається батьківським об'єктом, об'єктом-предком, базовим класом, а успадкований від нього об'єкт - нащадком, підкласом, або похідним класом.

ООП дозволяє писати гнучкі, розширювані і читабельні програми. Багато в чому це забезпечується завдяки поліморфізму, під яким розуміється можливість під час виконання програми за допомогою одного і того ж імені виконувати різні дії або звертатися до об'єктів різного типу. Найчастіше поняття поліморфізму пов'язують з механізмом віртуальних методів, який ми розглянемо нижче.

Переваги ООП:

- використання при програмуванні понять, близьких до наочної області;
- можливість успішно управляти великими об'ємами коду завдяки інкапсуляції, тобто приховуванню деталей реалізації об'єктів і спрощенню структури програми;
- можливість багатократного використання коду за рахунок спадкоємства;
- порівняно проста можливість модифікації програм;
- можливість створення і використання бібліотек об'єктів.

Ці переваги особливо явно виявляються при розробці програм великого об'єму і класів програм. Проте ніщо не дається дарма: створення об'єктно-орієнтованої програми є непростим завданням, оскільки вимагає розробки ієрархії об'єктів, а погано спроектована ієрархія може звести до нуля всі переваги об'єктно-орієнтованого підходу. Крім того, ідеї ООП не прості для розуміння і особливо для практичного застосування. Щоб ефективно використовувати готові об'єкти з бібліотек, необхідно освоїти великий об'єм достатньо складної інформації. Безграмотне застосування ООП приводить до створення надмірно складних програм, які неможливо відладити і удосконалити.

## 1.3. Середовище Visual Studio .NET

Середовище розробки Visual Studio .NET надає могутні і зручні засоби написання, коректування, компіляції, відладки і запуску додатків, що використовують .NET-сумісні мови. Корпорація Microsoft включила в платформу засоби розробки для чотирьох мов: C#, VB.NET, C ++ , Java#. Платформа .NET є відкритим середовищем. Це означає, що компілятори для неї можуть поставлятися і сторонніми розробниками. До теперішнього часу розроблені десятки компіляторів для .NET, наприклад, Ada, COBOL, Delphi, Eiffel, Fortran, Lisp, Oberon, Perl і Python.

NET-сумісні мови повинні відповідати вимогам загальномовної специфікації (Common Language Specification, CLS), в якій описується набір загальних для всіх мов характеристик. Це дозволяє використовувати для розробки додатку декілька мов програмування і вести повноцінну міжмовну відладку. Всі програми незалежно від мови використовують одні і ті ж базові класи бібліотеки .NET.

Додаток в процесі розробки називається проектом. Проект об'єднує все необхідне для створення додатку: файли, папки, посилання і інші ресурси. Середовище Visual Studio .NET дозволяє створювати проекти різних типів, наприклад:

- windows-додатки використовують елементи інтерфейсу Windows, включаючи форми, кнопки, прапорці і т.п;
- консольне застосування виконує виведення “на консоль”, тобто у вікно командного процесора;
- бібліотека класів об'єднує класи, які призначені для використання в інших застосуваннях;
- web-додатки - це додатки, доступ до яких виконується через браузер (наприклад, Internet Explorer) і які за запитом формують web-сторінку і відправляють її клієнтові по мережі;
- web-сервіс - компонент, методи якого можуть викликатися через Internet.

Декілька проектів можна об'єднати (solution). Це полегшує сумісну розробку проектів.

## 1.4 Консольні додатки

Середовище Visual Studio .NET працює орієнтовано на створення Windows- і Web- додатків, проте розробники передбачили роботу і з консольними застосуваннями. При запуску консольного застосування операційна система створює так зване консольне вікно, через яке йде все уведення-виведення програми. Зовні це нагадує роботу в операційній системі в режимі командного рядка, коли уведення-виведенням є потік символів.

Консольні застосування щонайкраще підходять для вивчення мови, оскільки в них не використовується множина стандартних об'єктів, необхідних для створення графічного інтерфейсу. У першій частині курсу ми створюватимемо

тільки консольні застосування, щоб зосередити увагу на базових властивостях мови C#. У наступному розділі розглянуті найпростіші дії в середовищі: створення і запуск на виконання консольного застосування на C# . Більш повні відомості, необхідні для роботи в Visual Studio.NET, можна отримати з документації або книг [8] [16]. Більшість прикладів, приведених в посібнику, ілюструють базові можливості C# і розроблялися в Microsoft Visual Studio 2019 (бібліотека .NET Framework 6.0.).

Для створення проекту слід після запуску Visual Studio 2019 в головному меню вибрати команду File ► New ► Project. У лівій частині діалогового вікна New Project, що відкрилося, потрібно вибрати пункт Visual C#, в правій - пункт Console Application. У полі Name можна ввести ім'я проекту, а в полі Location - місце його збереження на диску, якщо задані за умовчанням значення вас не влаштовують. Після натискання на кнопці ОК середовище створить рішення і проект з вказаним ім'ям. Зразковий вид екрану приведений на рис. 1.3.

У верхній частині екрану розташовується головне меню (з розділами File, Edit, View і т.д.) і панелі інструментів. Панелей інструментів в середовищі велика кількість, і якщо включити їх всіх (View ► Toolbars), вони займуть половину екрану.

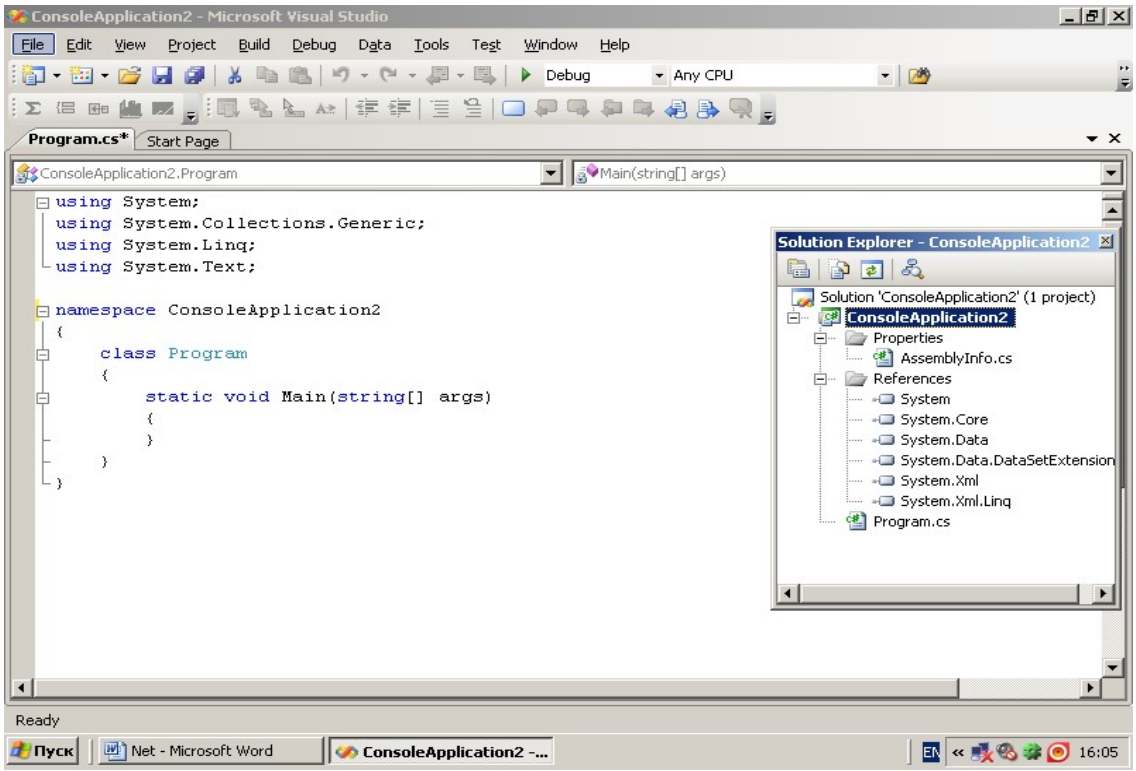



Рис. 1.3. Шаблон проекту при роботі з консоллю

У верхній лівій частині екрану розташовується вікно управління проектом Solution Explorer (якщо воно не відображається, слід скористатися командою View ► Solution Explorer головного меню). У вікні перераховані всі ресурси, що входять в проект: посилання на бібліотеку (System, System.Data, System.XML.), файл з початковим текстом (Program.cs), інформація про збірку

(Assemblyinfo.cs). У цьому ж вікні можна побачити і іншу інформацію, якщо перейти на вкладку Class View у вікні, що з'явилося, можна бачити список всіх класів, що входять в додаток, їх елементів і предків. У міру вивчення матеріалу будуть описані інші можливості середовища Visual Studio .NET.

Основний простір екрану займає вікно редактора, в якому розташовується текст програми, створений середовищем автоматично. Текстом є каркас, в який програміст додає код в міру необхідності. На лістингу 1.1 приведена перша програма. Тут введений один рядок:

```
Console.WriteLine("Первая программа");
```

Решта тексу була сформована автоматично.

Лістинг 1.1.

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

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
     Console.WriteLine("Первая программа");
      }
   }
}
```

Найпростіший спосіб запустити програму - натиснути клавішу F5 (або вибрати в меню команду Debug н Start). Якщо програма написана без помилок, то фраза "*Перша програма*" промайне перед очима в консольному вікні, яке негайно закриється. Це добрий результат, але для того, щоб побачити результат рішення, слід скористатися клавішами Ctrl+F5 (або вибрати в меню команду Debug ► Start Without Debugging).

Для збереження файлу використовується команда File н Save головного меню або кнопка Save на панелі інструментів. Втім, при запуску програми середовище збереже початковий текст самостійно.

Компілятор може виявити в тексті програми синтаксичні помилки. Він повідомляє про це у вікні, розташованому в нижній частині екрану.

Після загальних уявлень про платформу .NET, можна приступити до вивчення мови C#.

У даному розділі дано дуже коротке введення в цікаву і обширну тему - платформу .NET. Для глибшого розуміння механізмів її функціонування рекомендується вивчити додаткову літературу ([5], [19], [26], [27]) і публікації в Інтернеті.

# РОЗДІЛ 2. ОСНОВНІ ПОНЯТТЯ МОВИ

## 2.1. Склад мови

### 2.1.1 Алфавіт і лексеми

У C # використовується кодування символів Unicode. Існує багато різних кодувань символів. Наприклад, в Windows часто використовується кодування ANSI, а конкретно - CP1251. Кожен символ представляється в ній одним байтом (8 біт), тому в цьому кодуванні можна одночасно задати лише 256 символів. У першій половині кодової таблиці знаходяться латинські букви, цифри, знаки арифметичних операцій і інші поширені символи. Другу половину займають символи російського алфавіту. Якщо потрібно представляти символи іншого національного алфавиту (наприклад, албанського), необхідно використовувати іншу кодову таблицю. Кодування Unicode дозволяє представити символи всіх існуючих алфавітів одночасно, що корінним чином поліпшує переносимість текстів. Кожному символу відповідає свій унікальний код. Природно, що при цьому для зберігання кожного символу потрібно більше пам'яті. Перші 128 Unicode-символів відповідають першій частині кодової таблиці ANSI. Алфавіт C # включає:

- літери (латинські і національних алфавітів) і символ підкреслення (_), який вживається поряд з літерами;
- цифри;
- спеціальні символи, наприклад +, *, {, &;
- пробільні символи (пробіл і символи табуляції);
- символи переводу рядка.

З символів складаються більші будівельні блоки: *лексеми*, *директиви* препроцесора і *коментарі.*

*Лексема* це мінімальна самостійна одиниця мови. Існують такі види лексем:

- імена (ідентифікатори);
- ключові слова;
- знаки операцій;
- роздільники;
- літерали (константи).

Лексеми мови програмування аналогічні словам природної мови. Наприклад, лексемами є число 128 (але не його частина 12), ім'я Vasia, ключове слово goto і знак операції додавання +.

*Директиви препроцесора* прийшли в C # з мови C ++. Препроцесор - попередня стадія компіляції, на якій формується вид вихідного тексту програми. Наприклад, за допомогою директив (інструкцій, команд) препроцесора можна включити або виключити з процесу компіляції фрагменти коду. Директиви препроцесора не відіграють в C # такої важливої ролі, як в C + +. Ми розглянемо їх детально в розділі "Директиви препроцесора". *Коментарі* призначені для запису пояснень до програми і формування документації. Правила запису ко-

ментарів описані далі в цьому розділі. З лексем складаються вирази і оператори. *Вираз* задає правило обчислення деякого значення. Наприклад, вираз a + b задає правило обчислення суми двох величин. *Оператор* задає закінчений опис деякої дії, даних або елемента програми. Наприклад: *int a*; - оператор опису цілочисельної змінної *a*.

### 2.1.2. Ідентифікатори і ключові слова

Імена в програмах служать тій же меті, що і імена в світі людей, - щоб звертатися до програмних об'єктів і розрізняти їх, тобто ідентифікувати. Тому імена також називають ідентифікаторами. У ідентифікаторі можуть використовуватися букви, цифри і символ підкреслення. Прописні і рядкові букви розрізняються, наприклад, sysop, Sysop і Sys0p - три різних імені. Першим символом ідентифікатора може бути буква або знак підкреслення, але не цифра. Довжина ідентифікатора не обмежена. Пропуски усередині імен не допускаються. У ідентифікаторах C# дозволяється використовувати окрім латинських букв букви національних алфавітів. Наприклад, Собачка або gg є правильними ідентифікаторами. Більш того, в ідентифікаторах можна застосовувати навіть так звані escape-послідовності Unicode, тобто представляти символ за допомогою його коду в шістнадцятирічному вигляді з префіксом \u, наприклад \u00F2.

Імена даються елементам програми, до яких потрібно звертатися: змінним, типам, константам, методам, міткам і так далі. Ідентифікатор створюється на етапі оголошення змінної (методу, типу і тому подібне), після цього його можна використовувати в подальших операторах програми. При виборі ідентифікатора необхідно мати на увазі наступне:

- ідентифікатор не повинен збігатися з ключовими словами.
- не рекомендується починати ідентифікатори з двох символів підкреслення, оскільки такі імена зарезервовані для службового використання.

Для поліпшення читабельності програми слід давати об'єктам осмислені імена, складені відповідно до певних правил. Зрозумілі і узгоджені між собою імена - основа хорошого стилю програмування. Існує декілька видів так званих нотацій - угод про правила створення імен.

У нотації *Паскаля* кожне слово, що становить ідентифікатор, починається з прописної букви, наприклад, MaxLength, MyFuzzyShoosh. *Угорська нотація* (її запропонував угорець за національністю, співробітник компанії Microsoft) відрізняється від попередньої наявності префікса, відповідного типу величини, наприклад, iMaxLength, lpFfuzzy. Згідно нотації *Camel*, з прописної букви починається кожне слово, що становить ідентифікатор, окрім першого, наприклад, maxLength, myFuzzyShoosh. Ще одна традиція - розділяти слова, складові ім'я, знаками підкреслення: max_length, my_fuzzy_shoosh, при цьому всі складові частини починаються з рядкової букви.

У C# для іменування різних видів програмних об'єктів найчастіше використовуються дві нотації: Паскаля і Camel.

Ключові слова - це зарезервовані ідентифікатори, які мають спеціальне значення для компілятора. Їх можна використовувати тільки в тому сенсі, в якому вони визначені. Список ключових слів C# приведений в таблиці. 2.1.

### 2.1.3. Знаки операцій і роздільники

*Знак операції* - це один або більш символів, які визначають дію над операндами. Усередині знаку операції пропуски не допускаються. Наприклад, у виразі *a* += *b* знак += є знаком операції, *a* і *b* - операндами. Операції діляться на унарні, бінарні і тернарні по кількості операндів, що беруть участь в них. Один і той же знак може інтерпретуватися по-різному в залежності від контексту. Всі знаки операцій, за виключенням [ ], ( ) і ? :, є окремими лексемами.

Таблиця 2.1.

Ключові слова C#

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

*Роздільники* використовуються для розділення або, навпаки, групування елементів. Приклади роздільників: дужки, крапка, кома. Нижче перераховані всі знаки операцій і роздільники, що використовуються в C#:

{} [] () . , : : + - * / % & | ^ ! ~ =

< > ? ++ -- && || << >> == != <= >= += -= *= /= %= &= |= ^= <<= >>= ->

### 2.1.4. Літерали

*Літералами*, або *константами* називають незмінні величини. У C# є *логічні, цілі, дійсні, символьні і рядкові константи*, а також константа *null*. Компі-

лятор, виділивши константу як лексему, відносить її до одного з типів даних по її зовнішньому вигляду. Програміст може задати тип константи і самостійно. Опис і приклади констант кожного типу приведені в таблиці 2.2.

Як видно з таблиці 2.2, *логічних* літералів всього два. Вони широко використовуються як ознаки наявності або відсутності чого-небудь. *Цілі літерали* можуть бути представлені або в десятковій, або в шістнадцятирічній системі числення. *Дійсні літерали* можуть бути представлені тільки в десятковій системі, але в двох формах: з фіксованою точкою і з порядком. *Дійсна* константа з порядком представляється у вигляді мантиси і порядку. Мантиса записується зліва від знаку експоненти (E або e), порядок - праворуч від знаку. Значення константи визначається як множення мантиси і зведеного у вказаний в порядку ступінь числа 10 (наприклад, 1.3e2 = 1,3 • 100 = 130). При записі дійсного числа можуть бути відсутні або ціла частина, або дріб, але не обидві відразу.

Пропуски усередині числа не допускаються. Для відділення цілої частини від дробу використовується не кома, а крапка. Символ E не є знайомим всім з математики число e, а указує, що далі розташовується ступінь, в який потрібно звести число 10. Якщо потрібно сформувати від'ємну цілу або дійсну константу, то перед нею ставиться знак унарної операції зміни знаку (-), наприклад: -218, -022, -0x3C, -4.8, -1e4.

Коли компілятор розпізнає константу, він відводить їй місце в пам'яті відповідно до її вигляду і значення. Якщо по яких-небудь причинах потрібно явним чином задати скільки пам'яті слід відвести під константу, використовуються суфікси, описи яких приведені в таблиці. 2.3.

*Символьна константа* - будь-який символ в кодуванні Unicode. Вони записуються в одній з чотирьох форм:

- "звичайний" символ, що має графічне уявлення (окрім апострофа і символу перекладу рядка), - 'a', 'ю';
- послідовність, що управляє, - '\0', '\n';
- символ у вигляді шістнадцятирічного коду - '\xf', '\ x74';
- символ у вигляді escape-послідовності Unicode - '\u00ff<file://'/ua81b'>.

*Управляючою послідовністю (escape-послідовністю) називають* певний символ, що передує зворотною косою межею. *Управляюча послідовність* інтерпретується як одиночний символ і використовується для уявлення:

- кодів, що не мають графічного зображення (наприклад \n - перехід в початок наступного рядка);
- символів, що мають спеціальне значення в рядкових і символьних літералах, наприклад, апострофа.

У таблиці 2.4 приведені допустимі значення послідовностей. Якщо безпосередньо за символом "\" слідує символ, не передбачений таблицею, виникає помилка компіляції.

Таблиця 2.2.

**Константи в C#**

| Константа | Опис | Приклади |
|---|---|---|
| Логічна | true (істина) або false (неправда) | true false |
| Ціла | *Десяткова*: послідовність десяткових цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), за якою може слідувати суфікс (U, u, L, l, UL, Ul, uL, ul, LU, Lu, lU, lu).<br>*Шістнадцятирічна*: символи 0x, за якими слідують шістнадцятирічні цифри (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F), а за цифрами, у свою чергу, може слідувати суфікс (U, u, L, l, UL, Ul, uL, ul, LU, Lu, lU, lu). | 8 0 199226<br>8u 0Lu 199226L<br>0xA 0x1B8 0x00FF<br>0xAU 0xlB8LU<br>0x00FFl |
| Дійсна | *З фіксованою точкою*:<br>[цифри] [.] [цифри] [суфікс]<br>Суфікс - один з символів F, f, D, d, M,m<br>*З порядком:*<br>[цифри][.][цифри]{ E\|e}[+ \|-] [цифри] [суфікс]<br>Суфікс - один з символів F, f, D, d, M, m | 5.7 .001 35<br>5.7F .00ld<br>5F .00lf 35m<br>0.2E6 .lle+3 5E-10<br>0.2E6D .lle-3<br>5E10 |
| Символьна | Символ, взятий в апострофи | 'A' 'ю' '\0'<br>'\n' '\xF' '\x74'<br>'\uA81B' |
| Рядкова | Послідовність символів, взятих у лапки | "Тут був Vasia"<br>"tЗначення грам = \xF5 \n"<br>"Тут був \u0056\u0061"<br>"C: \\temp\\file1.txt"<br>@"C: \temp\file1.txt |
| Константа null | Посилання, яке не указує ні на який об'єкт | Null |

Таблиця 2.3.

**Суфікси цілих і дійсних констант**

| Суфікси | Значення |
|---|---|
| L, l | Довге ціле (long) |
| U, u | Беззнакове ціле (unsigned) |
| F, f | Дійсна з одинарною точністю (float) |
| D, d | Дійсна з подвійною точністю (double) |
| M, m | Фінансове десяткового типу (decimal) |

Таблиця 2.4.

**Управляючі послідовності у C#**

| Вигляд | Найменування |
|---|---|
| \a | Звуковий сигнал |
| \b | Повернення на крок |
| \f | Переведення сторінки (формату) |
| \n | Переведення рядка |
| \r | Повернення каретки |
| \t | Горизонтальна табуляція |
| \v | Вертикальна табуляція |
| \\ | Зворотна коса лінія |
| \’ | Апостроф |
| \" | Лапки |
| \0 | Нуль-символ |

Символ, представлений у вигляді *шістнадцятирічного* коду, починається з префікса \0x, за яким слідує код символу. Числове значення повинне знаходитися в діапазоні від 0 до FFFF, інакше виникає помилка компіляції.

Escape-послідовності Unicode служать для представлення символу в кодуванні Unicode за допомогою його коду в шістнадцятирічному вигляді з префіксом \u або \U, наприклад \u0041, \UFFFF. Управляючі послідовності можуть використовуватися і в рядкових константах, названих інакше рядковими літералами. Наприклад, якщо потрібно вивести декілька рядків, можна об'єднати їх в один літерал, відокремивши один рядок від іншого символами \n:

“Ніхто не задоволений своєю\n зовнішністю, але кожен задоволений \n своїм розумом”

Цей літерал при виведенні виглядатиме так:

*Ніхто не задоволений своєю*

*зовнішністю, але кожен задоволений*
*своїм розумом*

Інший приклад: якщо усередині рядка потрібно використовувати лапки, перед ними стоїть символ “\”, по якому компілятор відрізняє його від лапок, що обмежують рядок:

*"Видавничий будинок \"Пітер\""*

Як бачите, рядкові літерали з символами, що управляють, декілька втрачають в читабельності, тому в C# введений другий вид літералів - *дослівні літерали* (verbatim strings). Перед цими літералами вказується символ @, який відключає обробку послідовностей, що управляють, і дозволяє отримувати рядки в тому вигляді, в якому вони записані. Наприклад, два приведених вище за літерал в дослівному вигляді виглядають так:

@"Ніхто не задоволений своєю
зовнішністю, але кожен задоволений
своїм розумом"
@"Видавничий будинок "Пітер""

Найчастіше *дослівні літерали* застосовуються в регулярних виразах і при завданні повного шляху файлу, оскільки в ньому присутні символи зворотної косої межі, які в звичайному літералі довелося б представляти за допомогою послідовності, що управляє. Порівняєте два варіанти запису одного і того ж шляху:

"C: \\app\\bin\\debug\\a. exe"
@"C:\app\bin\debug\a.exe"

Рядок може бути порожнім (записується парою суміжних подвійних лапок ""), порожня символьна константа недопустима.

Константа null є значенням, що задається за умовчанням для величин так званих посилальних типів, які ми розглянемо далі в цьому розділі.

### 2.1.5. Коментарі

*Коментарі* призначені для запису пояснень до програми і формування документації. Компілятор коментарі ігнорує. У середині коментаря можна використовувати будь-які символи. У C# є два види коментарів: однорядкові і багаторядкові.

*Однорядковий* коментар починається з двох символів прямої косої межі (//) і закінчується символом переходу на новий рядок, *багаторядковий* полягає між символами-дужками /* і */ і може займати частину рядка, цілий рядок або декілька рядків. Коментарі не вкладаються один в одного: символи // і /* не володіють ніяким спеціальним значенням усередині коментаря.

Крім того, в мові є ще один різновид коментарів, які починаються з трьох підряд символів косої межі, що йдуть (///). Вони призначені для формування документації до програми у форматі XML. Компілятор витягує ці коментарі з програми, перевіряє їх відповідність правилам, записує їх в окремий файл. Правила завдання коментарів цього вигляду розглянемо у розділі 14.

## 2.2. Типи даних

Дані, з якими працює програма, зберігаються в оперативній пам'яті. Природно, що компілятору необхідно точно знати, скільки місця вони займають, як саме закодовані і які дії з ними можна виконувати. Все це задається при описі даних за допомогою типу.

*Тип даних* однозначно визначає:

- *внутрішнє представлення даних*, а, отже*, і множину їх можливих значень;*
- *допустимі дії над даними* (операції і функції).

Наприклад, цілі та дійсні числа, навіть якщо вони займають однаковий об'єм пам'яті, мають абсолютно різні діапазони можливих значень; цілі числа можна множити один на одного, а, наприклад, символи - не можна.

Кожен вираз в програмі має певний тип. Величин, що не мають ніякого типу, не існує. Компілятор використовує інформацію про тип при перевірці допустимості описаних в програмі дій.

Пам'ять, в якій зберігаються дані під час виконання програми, ділиться на дві області: стек (stack) і динамічна область, або хіп (heap). Динамічну область називають іноді “купою”. *Стек* використовується для зберігання величин, пам'ять під яких виділяє компілятор, а в *динамічній області* пам'ять резервується і звільняється під час виконання програми за допомогою спеціальних команд. Основним містом для зберігання даних в C# є хіп.

### 2.2.1. Класифікація типів

Перш ніж перейти до вивчення конкретних типів мови C#, розглянемо їх класифікацію. Всі типи можна розділити на *прості* (не мають внутрішньої структури) і *структуровані* (складаються з елементів інших типів). Типи можна розділити на *вбудовані* (стандартні) і *визначених програмістом* (рис. 2.1). Для даних *статичного* типу пам'ять виділяється у момент оголошення, при цьому її необхідний об'єм відомий. Для даних *динамічного* типу розмір даних у момент оголошення може бути невідомий, і пам'ять під них виділяється за запитом в процесі виконання програми.

Вбудовані типи не вимагають попереднього визначення. Для кожного типу існує ключове слово, яке використовується при описі змінних, констант і т.д. Якщо ж програміст визначає власний тип даних, він описує його характеристики і сам дає йому ім'я, яке потім застосовується точно так, як і імена стандартних типів. Опис власного типу даних повинен включати всю інформацію, необхідну для його використання, а саме внутрішнє уявлення і допустимі дії.

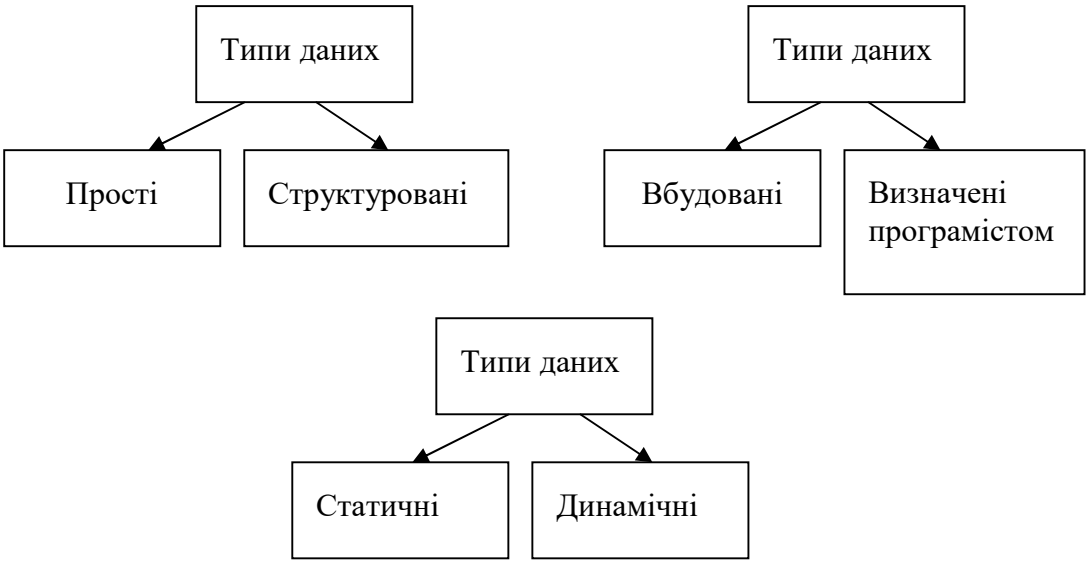



Рис. 2.1. Різні класифікації типів даних C#

Вбудовані типи C# приведені в таблиці. 2.5. Вони однозначно відповідають стандартним класам бібліотеки .NET, визначеним в просторі імен System. Як видно з таблиці, існують декілька варіантів представлення цілих і дійсних величин. Програміст вибирає тип кожної величини, яка використовується в програмі, з урахуванням необхідного йому діапазону і точності представлення даних.

Цілі типи, а також символьний, дійсний і фінансовий типи можна об'єднати під назвою арифметичних типів.

Внутрішнє представлення величини цілого типу - ціле число в двійковому коді. У знакових типах старший біт числа інтерпретується як знаковий (0 додатне число, 1 – від'ємне). Від'ємні числа найчастіше представляються в так званому додатковому коді. Для перетворення числа в додатковий код всі розряди числа, за винятком знакового інвертуються, потім до числа додається одиниця, і знаковому біту теж привласнюється одиниця. Беззнакові типи дозволяють представляти тільки додатні числа, оскільки старший розряд розглядається як частина коду числа.

Для дійсних і фінансового типів в таблиці приведені абсолютні величини мінімальних і максимальних значень. Дійсні типи, або типи даних, з плаваючою крапкою зберігаються в пам'яті комп'ютера інакше, ніж цілочисельні. Внутрішнє представлення дійсного числа складається з двох частин - мантиси і порядку, кожна частина має знак. Довжина мантиси визначає точність числа, а довжина порядку - його діапазон. У першому наближенні це можна уявити собі так: наприклад, для числа $0{,}381 \cdot 10^4$ зберігаються цифри мантиси 381 і порядок 4, для числа $560{,}3 \cdot 10^2$ - мантиса 5603 і порядок 5 (мантиса нормалізується), а число 0,012 представлено як 12 і -1. Звичайно, в даному прикладі не враховані система числення і інші особливості. Всі дійсні типи можуть бути представлені як додатні, так і від'ємні числа.

Таблиця 2.5.

Вбудовані типи C#

| **Ключове слово** | **Тип .NET** | **Діапазон значень** | **Опис** | **Розмір, бітів** |
|---|---|---|---|---|
| boo1 | Boolean | true, false | | |
| sbyte | SByte | От-128 до 127 | Із знаком | 8 |
| byte | Byte | От 0 до 255 | Без знаку | 8 |
| short | Intl6 | От –З2768 до З2767 | Із знаком | 16 |
| ushort | UIntl6 | От 0 до 655З5 | Без знаку | 16 |
| int | Int32 | От $-2^{31}$ до $2^{31} – 1$ | Із знаком | 32 |
| uint | UInt32 | От 0 до $2^{32} – 1$ | Без знаку | 32 |
| long | Int64 | От $-2^{63}$ до $2^{63} – 1$ | Із знаком | 64 |
| ulong | UInt64 | От 0 до $2^{64} – 1$ | Без знаку | 64 |
| char | Char | От U+0000 до U+ffff | Unicode-символ | 16 |
| float | Single | От $1.5•10^{-45}$ до $3.4•10^{38}$ | 7 цифр | 32 |
| double | Double | От $5.0•10^{-324}$ до $1.7•10^{308}$ | 15-16 цифр | 64 |
| decimal | Decimal | От $1.0•10^{-28}$ до $7.9•10^{28}$ | 28-29 цифр | 128 |
| string | String | Довжина обмежена об'ємом доступної пам'яті | Рядок з Unicode-символів | |
| object | Object | Можна зберігати все що завгодно | Загальний предок | |

Найчастіше в програмах використовується тип double, оскільки його діапазон і точність задовольняють більшість потреб. Цей тип мають дійсні літерали і багато стандартних математичних функцій. При однаковій кількості байтів, що відводяться під величини типу float і int, діапазони їх допустимих значень сильно розрізняються із-за внутрішньої форми уявлення.

Те ж саме відноситься до long і double. Тип decimal призначений для грошових обчислень, в яких критичні помилки округлення. Як видно з таблиці 2.5, тип float дозволяє зберігати одночасно всього 7 значущих десяткових цифр, тип double - 15-16. При обчисленнях помилки округлення накопичуються, і при певному поєднанні значень навіть може привести до результату, в якому не буде жодної вірної значущої цифри! Величини типу decimal дозволяють зберігати 28-29 десяткових розрядів.

Тип decimal не відноситься до дійсних типів, у них різне внутрішнє уявлення. Величини грошового типу навіть не можна використовувати в одному виразі з дійсними без явного перетворення типу. Використання величин фінансового типу в одному виразі з цілими допускається.

Будь-який вбудований тип C# відповідає стандартному класу бібліотеки .NET, визначеному в просторі імен System. Скрізь, де використовується ім'я вбудованого типу, його можна замінити ім'ям класу бібліотеки. Це означає, що у вбудованих типів даних C# є методи і поля. З їх допомогою можна, наприклад, набути мінімальних і максимальних значень для цілих, символьних, фінансових і дійсних чисел:

- *double.MaxValue* (или *System.Double.MaxValue*) - максимальне число типу double;
- *uint.MinValue* (или *System.UInt32.MinValue*)- мінімальне число типа uint.

### 2.2.2. Типи літералів

Літерали (константи) теж мають тип. Якщо значення цілого літерала знаходяться усередині діапазону допустимих значень типу int, літерал розглядається як int, інакше він відноситься до найменшого з типів uint, long або ulong, в діапазон значень якого він входить. Дійсні літерали за умовчанням відносяться до типу double.

Наприклад, константа 10 відноситься до типу int (хоча для її зберігання достатньо одного байта), а константа 2147483648 буде визначена як uint. Для явного завдання типу літерала служить суфікс, наприклад, 1.1f, 1UL, 1000m (суфікси показані в таблиці. 2.3). Явне завдання застосовується в основному для зменшення кількості неявних перетворень типу, що виконуються компілятором.

### 2.2.3. Типи-значення і посилальні типи

Найчастіше типи C# розділяють *за способом зберігання* елементів на типи-значення і посилальні типи (рис. 2.2). Елементи типів-значень, або значущих типів (value types), є просто послідовністю бітів в пам'яті, необхідний об'єм якої виділяє компілятор. Іншими словами, величини значущих типів зберігають свої значення безпосередньо. Величина посилального типу зберігає не самі дані, а посилання на них (адреса, по якій розташовані дані). Самі дані зберігаються в хіпові. Не дивлячись на відмінності в способі зберігання, і типи-значення, і посилальні типи є нащадками загального базового класу object.

Рисунок 2.3 ілюструє різницю між величинами значущого і посилального типів. Одні і ті ж дії над ними виконуються по-різному. Розглянемо як приклад перевірку на рівність. Величини значущого типу рівні, якщо рівні їх значення. Величини посилального типу рівні, якщо вони посилаються на одні і ті ж дані (на рисунку *b* і *c* рівні, але *a* не рівне *b* навіть при однакових значеннях). З цього виходить, що якщо змінити значення однієї величини посилального типу, це може відбитися на іншій.

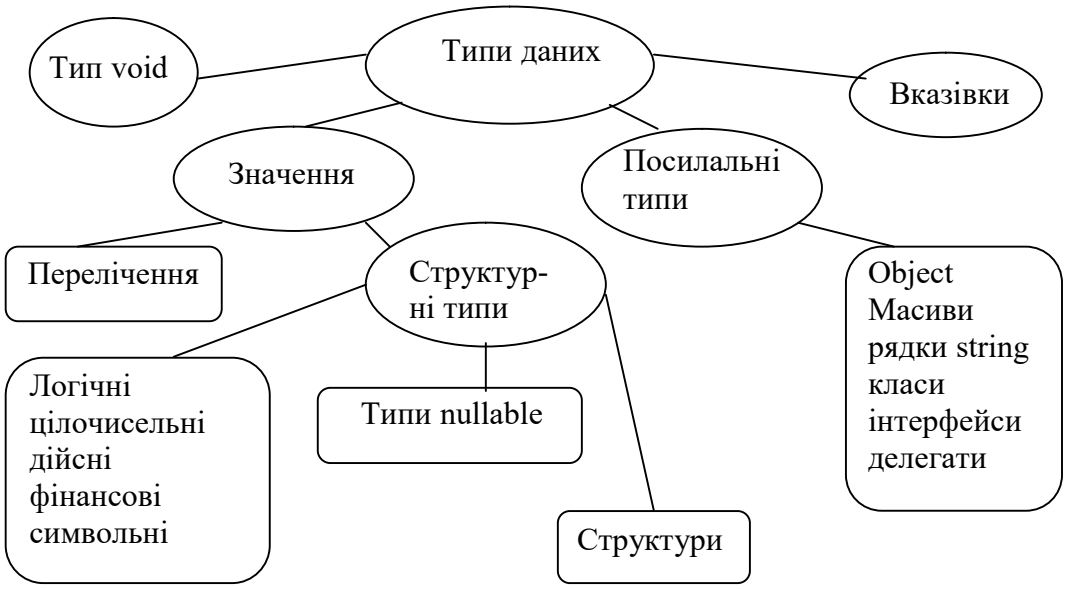



Рис. 2.2. Класифікація типів даних C# за способом зберігання

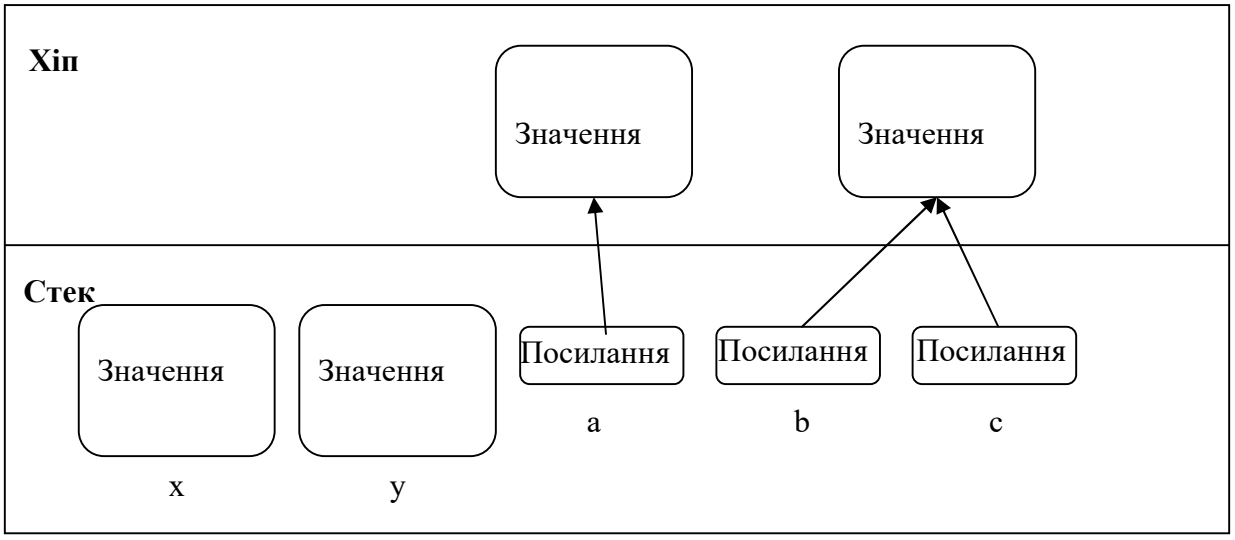



Рис. 2.3. Зберігання в пам'яті величин значущого і посилального типів

Всі значущі типи є простими. По іншій класифікації структури і перелічення відносяться до структурованих типів, що визначаються програмістом. Деталізація типів даних, приведених на рис. 2.2 буде розглянута в подальших розділах.

### 2.2.4. Упаковка і розпаковування

Для того, щоб величини посилального і значущого типів могли використовуватися спільно, необхідно мати можливість перетворення з одного типу в інший. Мова C# забезпечує таку можливість. Перетворення з типу-значення в посилальний тип називається упаковкою (boxing), зворотне перетворення - розпаковуванням (unboxing).

Якщо величина значущого типу використовується в тому місці, де потрібний посилальний тип, автоматично виконується створення проміжної величини посилального типу: створюється посилання, в хіпові виділяється відповідний об'єм пам'яті і туди копіюється значення величини, тобто значення ніби упаковується в об'єкт. При необхідності зворотного перетворення з величини посилального типу «знімається упаковка», і в подальших діях бере участь тільки її значення.

## 2.3. Рекомендації по програмуванню

Поняття, введені в цьому розділі, є базою для всього подальшого матеріалу. На перший погляд, вивчення видів лексем може здаватися зайвим (нехай їх розрізняє компілятор!), проте це абсолютно не так. При розробці програми необхідно розуміти, з яких елементів мови вона складається. Це допомагає і при пошуку помилок, і при зверненні до довідкової системи, і при вивченні нових версій мови. Більш того, вивчення будь-якої нової мови рекомендується починати саме з лексем, які в ній підтримуються.

Поняття типу даних лежить в основі більшості мовних засобів. При вивченні будь-якого типу необхідно розглянути дві речі: його внутрішнє уявлення (а отже, множина можливих значень величин цього типу), а також що можна робити з цими величинами. Типи даних є найважливішими характеристиками мови. Вибір найбільш відповідного типу для представлення даних - одна з необхідних умов створення ефективних програм.

Нові мови і засоби програмування з'являються безперервно, тому програміст вимушений вчитися все життя і повинен уміти:

- грамотно поставити завдання;
- вибрати відповідні мовні засоби;
- вибрати найбільш відповідні для представлення даних структури;
- розробити ефективний алгоритм;
- написати і документувати надійну і таку, що легко модифікується програму;
- забезпечити її вичерпне тестування.

# РОЗДІЛ 3. ЗМІННІ, ІМЕНОВАНІ КОНСТАНТИ, ОПЕРАЦІЇ І ВИРАЗИ

## 3.1. Змінні і іменовані константи

*Змінна* - це іменована область пам'яті, призначена для зберігання даних певного типу. Під час виконання програми значення змінної можна змінювати. Всі змінні, що використовуються в програмі, мають бути описані явним чином. При описі змінною задаються *ім'я* і *тип*. *Ім'я* змінної служить для звернення до області пам'яті, в якій зберігається значення змінної. *Ім'я* повинне відповідати правилам іменування ідентифікаторів C#, відображати сенс величини, що зберігається, і бути легко розпізнаваним. *Тип змінної* вибирається, виходячи з діапазону і необхідної точності представлення даних. Наприклад, немає необхідності заводити змінну дійсного типу для зберігання величини, яка може набувати тільки цілих значень, - хоч би тому, що цілочисельні операції виконуються набагато швидше.

При оголошенні можна привласнити змінній деяке початкове значення, тобто ініціалізувати її, наприклад:

```
int a, b =1;
float x  = 0.1, y = 0.1f:
```

Тут описані:

1. змінна *a* типу *int*, початкове значення якої не привласнюється;
2. змінна *b* типу *int*, її початкове значення дорівнює 1;
3. змінні *x* і y типу *float*, яким привласнені однакові початкові значення 0.1. Різниця між ними полягає в тому, що для ініціалізації змінної *x* спочатку формується константа типу *double* (це тип, що привласнюється за умовчанням літералам з дробовою частиною), а потім вона перетвориться до типу *floa*t; змінній *y* значення 0.1 привласнюється без проміжного перетворення.

При ініціалізації можна використовувати не тільки константу, але і вираз - головне, щоб на момент опису він був обчислюваним, наприклад:

```
int b = 1, a = 100:
int  x = b * a + 25;
```

Ініціалізувати змінну прямо при оголошенні не обов'язково, але перед тим, як її використовувати в обчисленнях, це зробити все одно доведеться, інакше компілятор повідомить про помилку. Рекомендується ініціалізувати змінні при описі.

Втім, іноді цю роботу робить за програміста компілятор, це залежить від місцезнаходження опису змінною. Як вже наголошувалося, програма на C# складається з класів, усередині яких описують методи і дані. Змінні, описані безпосередньо усередині класу, називаються полями класу. Їм автоматично

привласнюється так зване “значення по умовчанню”- як правило, це 0 відповідного типу.

Змінні, описані усередині методу класу, називаються локальними змінними. Їх ініціалізація покладається на програміста. У даному розділі розглядаються тільки локальні змінні простих вбудованих типів даних.

Так звана зона дії змінної, тобто область програми, де можна використовувати змінну, починається в точці її опису і триває до кінця блоку, усередині якого вона описана. Блок - це код, взятий у фігурні дужки. Основне призначення блоку - угрупування операторів. В C# будь-яка змінна описана усередині якого-небудь блоку: класу, методу або блоку усередині методу.

*Ім'я* змінної має бути унікальним в області її дії. Зона дії розповсюджується на вкладені в метод блоки, з цього виходить, що у вкладеному блоці не можна оголосити змінну з таким же ім'ям, як і в тому, що охоплює його, наприклад:

```
class X  // початок опису класу X
      {
            int A;          // поле A класу X
            int B;          // поле B класу X

          void Y( )         // метод Y класу X
            {
            int C;          // локальна змінна C, зона дії - метод Y
            int A;     // локальна змінна A (НЕ конфліктує з полем A)
            {// ============ вкладений блок 1 ============
            int D;          // локальна змінна D, зона дії - блок
            //int A;        Неприпустимо! Помилка компіляції
                  C = B;// привласнення змінній C поля B
                  C = this.A;  // привласнення змінній C поля A
            }// ============ кінець блоку 1 ===============

            {     // ============ вкладений блок 2 ============
            int D;          // локальна змінна D, зона дії - блок
            }     // ============ кінець блоку 2 ===============

            }     //        кінець опису метода Y класу X

      }     // кінець опису класу X
```

У лістингу 3.1 приведений приклад програми, в якій описуються і виводяться на екран локальні змінні.

Лістинг 3.1. Опис змінних

```
using System;
namespace ConsoleApplicationl
{
   class Classl
   {
         static void Main( )
         {
```

```
            int i = 3;
            double y = 4.12;
            decimal d = 600m;
            string s = "Вася";
            Console.Write( "i =  " );       Console.WriteLine( i );
            Console.Write( "y = " );        Console.WriteLine( y );
            Console.Write( "d = " );        Console.WriteLine( d );
            Console.Write( "s  = ");        Console.WriteLine( s );
            }
        }
    }
```

У розглянутому прикладі використовується метод Write. Він робить те ж саме, що і Writeline, але не переводить рядок.

Можна заборонити змінювати значення змінної, задавши при її описі ключове слово *const*, наприклад:

```
const int b = 1;
const float x = 0.1, y = 0.1f;  // const розповсюджується на обидві змінні
```

Такі величини називають константами. Вони застосовуються для того, щоб замість значень констант можна було використовувати в програмі їх імена. Це робить програму зрозумілішою і полегшує внесення до неї змін, оскільки змінити значення досить тільки в одному місці програми.

Поліпшення читабельності відбувається тільки при осмисленому виборі імен констант. У добре написаній програмі взагалі не повинно зустрічатися інших чисел окрім 0 і 1, решта всіх чисел повинна задаватися іменованими константами з іменами, що відображають їх призначення.

Іменовані константи повинні обов'язково ініціалізуватися при описі. При ініціалізації можна використовувати не тільки константу, але і вираз - головне, щоб він був обчислюваним на етапі компіляції, наприклад:

```
const int b = 1, a = 100;
const int x = b * a + 25;
```

## 3.2. Операції і вирази

*Вираз* - це правило обчислення значення. У виразі беруть участь операнди, об'єднані знаками операцій. Операндами простого виразу можуть бути константи, змінні і виклики функцій.

Наприклад, a + 2 - це вираз, в якому + є знаком операції, *a* і *2* -операнди. Пропуски усередині знаку операції, що складається з декількох символів, не допускаються. По кількості операндів операції, що беруть участь в одній операції, діляться на *унарні, бінарні і тернарні*. Операції C# показані в таблиці 3.1, в якій символ *x* показує розташування операнда і не є частиною знаку операції.

Таблиця 3.1.

Операції C#

| Категорія | Знак операції | Назва |
|---|---|---|
| Первинні | . | Доступ до елементу |
| | x() | Виклик методу або делегата |
| | x[] | Доступ до елементу |
| | x ++ | Постфіксний інкремент |
| | x -- | Постфіксний декрмент |
| | new | Виділення пам'яті |
| | typeof | Отримання типу |
| | checked | Код, який перевіряється |
| | unchecked | Код, який не перевіряється |
| Унарні | + | Унарний плюс |
| | - | Унарний мінус |
| | ! | Логічне заперечення |
| | ~ | Порозрядне заперечення |
| | ++x | Префіксний інкремент |
| | --x | Префіксний декремент |
| | (тип)x | Перетворення типу |
| Мультиплікативні (типу множення) | * | Множення |
| | / | Ділення |
| | % | Залишок від ділення |
| Адитивні (типу складання) | + | Складання |
| | - | Віднімання |
| Здвигу | << | Здвиг вліво |
| | >> | Здвиг вправо |
| Відношення і перевірки типу | < | Менше |
| | > | Більше |
| | <= | Менше або рівно |
| | >= | Більше або рівно |
| | is | Перевірка приналежності типу |
| | as | Приведення типу |
| Перевірки на рівність | = = | Рівно |
| | ! = | Не рівно |
| Порозрядні логічні | & | Порозрядна кон'юнкція (І) |
| | ^ | Порозрядне виключення (АБО) |
| | \| | Порязрядна диз'юнкція(АБО) |

Продовження таблиці 3.1

| Категорія | Знак операції | Назва |
|---|---|---|
| Умовні логічні | && | Логічне І |
| | \|\| | Логічне АБО |
| Умовна | ? : | Умовна операція |
| Привласнення | = | Привласнення |
| | *= | Множення з привласненням |
| | /= | Ділення з привласненням |
| | %= | Залишок від ділення з привласненням |
| | += | Складання з привласненням |
| | -= | Віднімання з привласненням |
| | <<= | Зрушення вліво з привласненням |
| | >>= | Зрушення вправо з привласненням |
| | &= | Порозрядне І з привласненням |
| | ^= | Порозрядне виключення АБО з привласненням |
| | \| = | Порозрядне АБО з привласненням |

Операції у виразі виконуються в певному порядку відповідно до пріоритетів, як і в математиці. У таблиці 3.1 операції розташовані по спаданню пріоритетів, рівні пріоритети розділені в таблиці горизонтальними лініями.

Результат обчислення виразу характеризується значенням і типом. Наприклад, нехай *a* і *b* - змінні цілого типу і описані так: *int a = 2, b = 5;*

Тоді вираз *a + b* має значення *7* і тип *int*, а вираз *a = b* має значення, рівне поміщеному в змінну *a*, і тип, співпадаючий з типом цієї змінної.

Якщо в одному виразі є сусідами декілька операцій однакового пріоритету, операції привласнення і умовна операція виконуються справа наліво, останні - зліва направо. Для зміни порядку виконання обчислення використовуються круглі дужки, рівень їх вкладеності практично обмежений.

Наприклад, *a + b + c означає (a + b) + c, a = b = c* означає *a = (b = c)*. Тобто спочатку обчислюється вираз *b = c*, а потім його результат стає правим операндом для операції привласнення змінної *a*.

### 3.2.1. Перетворення вбудованих арифметичних типів-значень

Якщо операнди різного типу і/або операція для цього типу не визначена, перед обчисленнями автоматично виконується перетворення типу згідно правил, що забезпечують приведення коротших типів до довших для охорони значущості і точності. Автоматичне (неявне) перетворення можливе не завжди, а тільки якщо при цьому не може трапитися втрата значущості.

Якщо неявного перетворення з одного типу в інший не існує, програміст може задати явне перетворення типу за допомогою операції (тип) *x*. Явне перетворення розглядається в цьому розділі трохи пізніше.

Арифметичні операції не визначені для коротших, ніж int, типів. Це означає, що якщо у виразі беруть участь тільки величини типів sbyte, byte, short і ushort, перед виконанням операції вони будуть перетворені в int. Таким чином, результат будь-якої арифметичної операції має тип не менше int.

Правила неявного перетворення ілюструє рис. 3.1. Якщо один з операндів має тип, зображений на нижчому рівні, ніж інший, то він буде приводитися до типу іншого операнда за наявності шляху між ними. Якщо шляху немає, виникає помилка компіляції. Якщо шляхів декілька, вибирається найбільш короткий, такий, що не містить пунктирних ліній. Перетворення виконується не послідовно, а безпосередньо з початкового типу в результуючий.

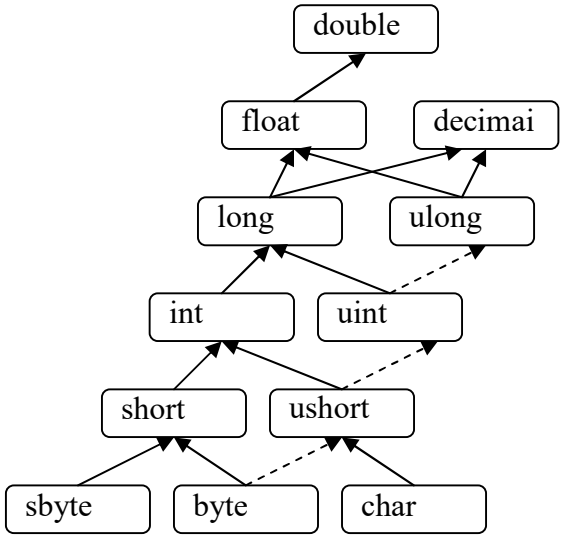



Рис. 3.1. Неявні арифметичні перетворення типів

Перетворення коротших, ніж int, типів виконується при привласненні. Зверніть увагу на те, що неявного перетворення з float і double в decimal не існує.

### 3.2.2. Введення у виключення

При обчисленні виразів можуть виникнути помилки, наприклад, переповнювання, зникнення порядку або ділення на нуль. У C# є механізм, який дозволяє обробляти подібні помилки і таким чином уникати аварійного завершення програми. Він так і називається: *механізм обробки виняткових ситуацій (виключень)*.

Якщо в процесі обчислень виникла помилка, система сигналізує про це за допомогою спеціальної дії, яка називається викиданням (*генеруванням) виключення.* Кожному типу помилки відповідає своє виключення. Оскільки C# - мова об'єктно-орієнтована, виключення є класами, які мають загального предка - клас *Exceptio*n, визначений в просторі імен *System.* Наприклад, при діленні на нуль буде викинуто (згенеровано) виключення з довгим, але зрозумілим ім'ям *DivideByZeroException,* при недоліку пам'яті - виключення

*OutofMemoryException,* при переповнюванні - виключення *OverflowException*. Стандартних виключень дуже багато, проте, програміст може створювати і власні виключення на основі класу Exception.

Програміст може задати спосіб обробки виключення в спеціальному блоці кода, що починається з ключового слова catch ("перехопити"), який буде автоматично виконаний при виникненні відповідної виняткової ситуації. Усередині блоку можна, наприклад, вивести застережливе повідомлення або скорегувати значення величин і продовжити виконання програми. Якщо цей блок не заданий, система виконує дії за умовчанням, які, зазвичай, полягають у виведенні діагностичного повідомлення при нормальному завершенні програми.

Процесом викидання виключень, що виникають при переповнюванні, можна управляти. Для цього служать ключові слова *checked* і *unchecked*. Слово *checked* включає перевірку переповнювання, слово *unchecked* вимикає. При вимкненій перевірці виключення, пов'язані з переповнюванням, не генеруються, а результат операції усікається. Перевірку переповнювання можна реалізувати для окремого виразу або для цілого блоку операторів, наприклад:

```
a = checked (b + c);          // для виразу
unchecked {a = b + c;}        //для блоку операторів
```

Перевірка не розповсюджується на функції, викликані в блоці. Якщо перевірка переповнювання включена, говорять, що обчислення виконуються в контексті, що перевіряється, якщо вимкнена - в тому, що не перевіряє. Перевірку переповнювання вимикають у випадках, коли усікання результату операції необхідне відповідно до алгоритму.

Можна задати перевірку переповнювання у всій програмі за допомогою ключа компілятора /checked, це корисно при відладці програми. Оскільки подібна перевірка декілька уповільнює роботу, в готовій програмі цей режим зазвичай не використовується.

### 3.2.3. Основні операції C#

**Інкремент і декремент**

Операції інкремента (++) і декремента (--), називаються також операціями збільшення і зменшення на одиницю, мають дві форми запису - префіксну, коли знак операції записується перед операндом, і постфіксну. У префіксній формі спочатку змінюється операнд, а потім його значення стає результуючим значенням виразу, а в постфіксній формі значенням виразу є початкове значення операнда, після чого він змінюється. Лістинг 3.2 ілюструє ці операції.

Лістинг 3.2. Операції інкремента і декремента

```
using System;
namespace ConsoleApplicationl
  {class Classl
    {
      static void Main( )
      {
      int x = 3, y = 3;
      Console.Write("Значення префіксного виразу:   ");
```

```
Console.WriteLine(++x);
Console.Write("Значення x після приросту:   ");
Console.WriteLine(x);
Console.Write("Значення постфіксного виразу:   ");
Console.WriteLine(y++);
Console.Write("Значення y після приросту:   ");
Console.WriteLine(y);
}}}
```

Результат роботи програми:

*Значення префіксного виразу: 4*
*Значення x після приросту: 4*
*Значення постфіксного виразу: 3*
*Значення y після приросту: 4*

Стандартні операції інкремента існують для цілих, символьних, дійсних і фінансових величин, а також для перелічень. Операндом може бути змінна, властивість або індексатор (ми розглянемо властивості і індексатори в подальших розділах.

**Операція new**

Операція new служить для створення нового об'єкту. Формат операції:
*new тип ( [ аргументи ] )*

За допомогою цієї операції можна створювати об'єкти як посилальних, так і значущих типів, наприклад:

```
object z = new object();
int i = new int();                    // те ж саме,  що int i  = 0;
```

Об'єкти посилального типу зазвичай формують саме цим способом, а змінні значущого типу частіше створюються так, як описано раніше в розділі "Змінні".

При виконанні операції new спочатку виділяється необхідний об'єм пам'яті (для посилальних типів в хіпові, для значущих - в стеку), а потім викликається так званий конструктор за умовчанням, тобто метод, за допомогою якого ініціалізувався об'єкт. Змінною значущого типу привласнюється значення за умовчанням, яке дорівнює нулю відповідного типу. Для посилальних типів стандартний конструктор ініціалізував значеннями за умовчанням всі поля об'єкту.

Якщо необхідний для зберігання об'єкту об'єм пам'яті виділити не вдалося, генерується виключення *Outofmemoryexception*.

**Операції заперечення**

*Арифметичне заперечення* (унарний мінус -) міняє знак операнда на протилежний. Стандартна операція заперечення визначена для типів *int, long, float, double і decimal.* До величин інших типів її можна застосовувати, якщо для них можливе неявне перетворення до цих типів (див. рис. 3.1). Тип результату відповідає типу операції.

Для значень цілого і фінансового типів результат досягається відніманням

початкового значення з нуля. При цьому може виникнути переповнювання. Чи буде при цьому викинуто виключення, залежить від контексту. Логічне заперечення (!) визначене для типу bool. Результат операції - значення *false*, якщо операнд рівний *true*, і значення *true*, якщо операнд рівний *false*. Порозрядне заперечення (~), часто зване побітовим, інвертує кожен розряд в двійковому представленні операнда типу *int, uint, long* або *ulong*.

Операції заперечення представлені в лістингу 3.3.

Лістинг 3.3. Операції заперечення using System;

```
using System;
namespace Consoleapplicationl
{
    class Classl
    {
        static void Main( )
        {
         sbyte a = 3, b = -63, c = 126;
         bool d = true;
         Console.WriteLine( -a);          // Результат -3
         Console.WriteLine( -c);          // Результат -126
         Console.WriteLine( !d);          // Результат false
         Console.WriteLine( ~a);          // Результат -4
         Console.WriteLine( ~b);          // Результат 62
         Console.WriteLine( ~c);          // Результат -127
        }
    }
}
```

**Явне перетворення типу**

Операція використовується, як і виходить з її назви, для явного перетворення величини з одного типу в інший. Це потрібно у тому випадку, коли неявного перетворення не існує. При перетворенні з довшого типу в коротший можлива втрата інформації, якщо початкове значення виходить за межі діапазону результуючого типу. Формат операції:

*( тип ) вираз*

В даному разі тип - це ім'я того типу, в який здійснюється перетворення, а вираз найчастіше є ім'ям змінної, наприклад:

```
long b = 300;
int   a = (int) b;       // дані не втрачаються
byte d = (byte) a;     // дані втрачаються
```

**Множення, ділення і залишок від ділення**

*Операція множення (*)* повертає результат перемножування двох операндів. Стандартна операція множення визначена для типів *int, uint, long, ulong, float, double і decimal.* До величин інших типів її можна застосовувати, якщо для них можливе неявне перетворення до цих типів (див. рис. 3.1). Тип результату операції дорівнює “найбільшому” з типів операндів, але не менше int. Якщо

обидва операнди цілочисельні або типу *decimal* і результат операції перевищує допустиме значення, то генерується виключення *System OverflowException*. Всі можливі значення для дійсних операндів приведені в таблиці 3.2. Символами *x* і *y* позначені кінцеві додатні значення, символом z - результат операції дійсного множення. Якщо результат дуже великий, він приймається рівним значенню “нескінченність”, якщо дуже малий, він приймається за 0. NAN (not a number) означає, що результат не є числом.

Таблиця 3.2.

Результати дійсного множення

| * | +y | -y | +0 | -0 | +∞ | -∞ | NaN |
|---|---|---|---|---|---|---|---|
| +x | +z | -z | +0 | -0 | +∞ | -∞ | NaN |
| -x | -z | +z | -0 | +0 | -∞ | +∞ | NaN |
| +0 | +0 | -0 | +0 | -0 | NaN | NaN | NaN |
| -0 | -0 | +0 | -0 | +0 | NaN | NaN | NaN |
| +∞ | +∞ | -∞ | NaN | NaN | +∞ | -∞ | NaN |
| -∞ | -∞ | +∞ | NaN | NaN | -∞ | +∞ | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

*Операція ділення (/)* визначена для *типів int, uint, long, ulong, float, double і decimal*. Якщо обидва операнди цілочисельні, результат операції округляється вниз до найближчого цілого числа. Якщо дільник дорівнює 0, генерується виключення *System. DivideByZeroException*.

Якщо хоч би один з операндів дійсний, дробова частина результату ділення не відкидається, а всі можливі значення приведені в таблиці 3.3. Символами *x* і *y* позначені кінцеві додатні значення, символом *z* - результат операції дійсного ділення. Якщо результат дуже великий він приймається рівним значенню “нескінченність”, якщо дуже малий, він приймається за 0.

Таблиця 3.3.

Результати дійсного ділення

| / | +y | -y | +0 | -0 | +∞ | -∞ | NaN |
|---|---|---|---|---|---|---|---|
| +x | +z | -z | +∞ | -∞ | +0 | -0 | NaN |
| -x | -z | +z | -∞ | +∞ | -0 | +0 | NaN |
| +0 | +0 | -0 | NaN | NaN | +0 | -0 | NaN |
| -0 | -0 | +0 | NaN | NaN | -0 | +0 | NaN |
| +∞ | +∞ | -∞ | +∞ | -∞ | NaN | NaN | NaN |
| -∞ | -∞ | +∞ | -∞ | +∞ | NaN | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

Для фінансових величин (тип *decimal*) при діленні на 0 і переповнюванні генеруються відповідні виключення, при зникненні порядку результат рівний 0.

*Операція залишку від ділення (%)* також інтерпретується по-різному для цілих, дійсних і фінансових величин. Якщо обидва операнди цілочисельні, результат операції обчислюється за формулою

$$x - (x / y) * y.$$

Якщо дільник дорівнює нулю, генерується виключення *System.DivideByZeroException*. Тип результату операції дорівнює "найбільшому" з типів операндів, але не менше *int.*

Для фінансових величин (тип *decima*l) при отриманні залишку від ділення на 0 і при переповнюванні генеруються відповідні виключення, при зникненні порядку результат дорівнює 0. Знак результату дорівнює знаку першого операнда. Якщо хоч би один з операндів дійсний, результат операції обчислюється за формулою

$$x - n * y,$$

де n - найбільше ціле, яке менше або дорівнює результату ділення *x* на *y*. Всі можливі комбінації значень операндів приведені в таблиці 3.4. Символами *x* і *y* позначені кінцеві додатні значення, символом *z* - результат операції залишку від ділення.

Таблиця 3.4.

Результати дійсного залишку від ділення

| **%** | **+y** | **-y** | **+0** | **-0** | **+∞** | **-∞** | **NaN** |
|---|---|---|---|---|---|---|---|
| +x | +z | z | NaN | NaN | x | x | NaN |
| -x | -z | -z | NaN | NaN | -x | -x | NaN |
| +0 | +0 | +0 | NaN | NaN | +0 | +0 | NaN |
| -0 | -0 | -0 | NaN | NaN | -0 | -0 | NaN |
| +∞ | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| -∞ | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

Приклад застосування операцій множення, ділення і отримання залишку представлений в лістингу 3.4.

Лістинг 3.4. Операції множення, ділення і отримання залишку

```
using System;
namespace Consoleapplicationl
{
    class Classl
    {
        static void Main( )
        {
        int x = 11, y = 4;
        float z = 4;
        Console.WriteLine(z * y ) ;          // Результат 16
        Console.WriteLine(z * 1e308 );       // Результат "нескінченність"
        Console.WriteLine(x / y );           // Результат 2
        Console. WriteLine(x / z );          // Результат 2.75
```

```
            Console.WriteLine(x % y );              // Результат 3
            Console.WriteLine(1e-324 / 1e-324 );          // Результат NaN
            }
        }
    }
```

Декілька операцій одного пріоритету виконуються зліва направо. Для прикладу розглянемо вираз *2 / x • y*. Ділення і множення мають один і той же пріоритет, тому спочатку 2 ділиться на *x*, а потім результат множиться на *y*. Якщо ж ми хочемо, щоб вираз *x • y* був в знаменнику, слід укласти його в круглі дужки або спочатку поділити чисельник на *x*, а потім на *y*:

2 / (x • y) или 2 / x /y.

**Операції складання і віднімання**

*Операція складання ( + )* повертає суму двох операндів. Стандартна операція складання визначена для *типів int, uint, long, ulong, float*, *double і decimal*. До величин інших типів її можна застосовувати, якщо для них існує неявне перетворення до цих типів (див. рис. 3.1). Тип результату операції дорівнює "найбільшому" з типів операндів, але не менше int. Якщо обидва операнди цілочисельні або типу *decimal* і результат операції дуже великий для уявлення за допомогою заданого типу, генерується виключення *System.OverflowException*.

Можливі значення для дійсних операндів приведені в таблиці 3.5. Якщо результат дуже великий для заданого типу, він приймається рівним значенню "нескінченність", якщо дуже малий, він приймається за 0.

Таблиця 3.5.

Результати дійсного складання

| + | y | +0 | -0 | +∞ | -∞ | NaN |
|---|---|---|---|---|---|---|
| x | z | x | x | +∞ | -∞ | NaN |
| +0 | У | +0 | +0 | +∞ | -∞ | NaN |
| -0 | У | +0 | -0 | +∞ | -∞ | NaN |
| +∞ | +∞ | +∞ | +∞ | +∞ | NaN | NaN |
| -∞ | -∞ | -∞ | -∞ | NaN | -∞ | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN |

*Операція віднімання (- )* повертає різницю двох операндів. Стандартна операція віднімання визначена для типів *int, uint, long, ulong, float, double і decimal*. До величин інших типів її можна застосовувати, якщо для них існує неявне перетворення до цих типів (див. рис. 3.1). Тип результату операції дорівнює "найбільшому" з типів операндів, але не менше int. Якщо обидва операнди цілочисельні або типу *decimal* і результат операції дуже великий то генерується виключення *System.OverflowException*. Всі можливі значення результату віднімання для дійчних операндів приведені в таблиці 3.6. Символами *x* і *y* позначені кінцеві додатні значення, символом *z* - результат операції дійсного від-

німання. Якщо *x* і *y* рівні, результат дорівнює додатному нулю. Якщо результат дуже великий для заданого типу, він приймається рівним значенню “нескінченність” зі знаком, що і *x - y*, якщо дуже малий, він приймається за 0 з тим же знаком, що й *x - y*.

Таблиця 3.6.

Результати дійсного віднімання

| **-** | **y** | **+0** | **-0** | **+∞** | **-∞** | **NaN** |
|---|---|---|---|---|---|---|
| x | z | x | x | -∞ | +∞ | NaN |
| +0 | -y | +0 | +0 | -00 | +∞ | NaN |
| -0 | -y | -0 | +0 | -00 | +∞ | NaN |
| +v | +∞ | +∞ | +∞ | NaN | +∞ | NaN |
| -∞ | -∞ | -∞ | -∞ | -∞ | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN |

**Операції зсуву**

*Операції зсуву (<< і >>)* застосовуються до цілочисельних операндів. Вони зрушують двійкове представлення першого операнда вліво або вправо на кількість двійкових розрядів, задану другим операндом. При зрушенні розряди, що звільнилися вліво (<<) , обнуляються. При зсуву біти, що звільнилися управо (>>), заповнюються нулями, якщо операнд додатний або беззнакового типу. Для від’ємних чисел вони заповнюються одиницею (1). Операції зсуву ніколи не приводять до переповнювання і втрати значущості. Стандартні операції зсуву визначені для типів *int, uint, long і ulong*.

Приклад застосування операцій зсуву представлений в лістингу 3.5.

Лістинг 3.5. Операції зсуву

```
using System;
namespace Consoleapplicationl
{
    class Classl
    {
      static void Main( )
       {
       byte   a = 3, b = 9;
       sbyte c = 9, d = -9;
       Console.WriteLine( a << 1 );          // Результат   6
       Console.WriteLine( a << 2 );          // Результат   12
       Console.WriteLine( b >> 1 );          // Результат   4
       Console.WriteLine( c >> 1 );          // Результат   4
       Console.WriteLine( d >> 1 );          // Результат   -5
       }
    }
}
```

**Операції відношення і перевірки на рівність**

*Операції відношення (<, <=, >, >=, ==, != )* порівнюють перший операнд з другим. Операнди мають бути арифметичного типу. Результат операції - логічного типу, рівний true або false. Правила обчислення результатів приведені в таблиці 3.7.

Таблиця 3.7.

Результати операцій відношення

| Операція | Результат |
|---|---|
| x = =y | true, якщо x рівне y, інакше false |
| x != y | true, якщо x не рівне y, інакше false |
| x<y | true, якщо x менше y, інакше false |
| x<=y | true, якщо x менше або рівно y, інакше false |
| x>=y | true, якщо x більше або рівно y, інакше false |

**Порозрядні логічні операції**

*Порозрядні логічні операції (&, |, ^)* застосовуються до цілочисельних операндів і працюють з їх двійковими уявленнями. При виконанні операцій операнди зіставляються побітно (перший біт першого операнда з першим бітом другого, другий біт першого операнда з другим бітом другого і т. д.). Стандартні операції визначені для *типів int, uint, long і ulong.*

При *порозрядній кон'юнкції*, або порозрядному І (операція позначається &), біт результату дорівнює 1 тільки тоді, коли відповідні біти обох операндів дорівнюють 1.

При *порозрядній диз'юнкції*, або порозрядному АБО (операція позначається |),біт результату дорівнює 1 тільки тоді, коли біт хоч би одного з операндів рівний 1.

При *порозрядному виключаючому* АБО (операція позначається ^), біт результату дорівнює 1 тільки тоді, коли біт тільки одного з операндів рівний 1.

Приклад застосування порозрядних логічних операцій представлений в лістингу 3.6.

Лістинг 3.6. Порозрядні логічні операції

```
using System;
namespace Consoleapplicationl
{
    class Classl
    {
      static void Main( )
       {
           Console.WriteLine(6 & 5);   // Результат 4
           Console.WriteLine(6 | 5 );    // Результат 7
           Console.WriteLine(6 ^ 5 );   // Результат 3
       }
    }
}
```

**Умовні логічні операції**

*Умовні логічні операції І (&&)* і *АБО (||)* найчастіше використовуються з операндами логічного типу. Результатом логічної операції є *true* або *false*. Результат операції логічне І має значення *true*, якщо обидва операнди мають значення true. Результат операції логічне *АБО* має значення true якщо хоч би один з операндів має значення true.

Якщо значення першого операнда досить, щоб визначити результат операціїі, другий операнд не обчислюється. Наприклад, якщо перший операнд операції *І* рівний *false*, результатом операції буде *false* незалежно від значення другого операнда, тому він не обчислюється.

Приклад застосування умовних логічних операцій представлений в лістингу 3.7.

Лістинг 3.7. Умовні логічні операції

```
using System;
namespace Consoleapplicationl
{
    class Classl
    {
      static void Main( )
       {
        Console.WriteLine( true && true );    // Результат true
        Console.WriteLine( true && false );   // Результат false
        Console.WriteLine( true || true );    // Результат true
        Console.WriteLine( true || false );   // Результат true
       }
    }
}
```

**Умовна операція**

*Умовна операція (? :)* має формат:

**операнд_1 ? операнд_2 : операнд_3.**

Перший операнд - вираз, для якого існує неявне перетворення до логічного типу. Якщо результат обчислення першого операнда рівний true, то результатом умовної операції буде значення другого операнда, інакше - третього операнда. Обчислюється завжди або другий операнд, або третій. Приклад застосування умовної операції представлений в лістингу 3.8.

Лістинг 3.8. Умовна операція

```
using System;
namespace Consoleapplicationl
{
    class Classl
    {
      static void Main( )
       {
```

```
            int a = 11, b = 4;
            int max = b > a ? b : a;
            Console.WriteLine( max );            // Результат 11
        }
    }
}
```

**Операції привласнення**

*Операції привласнення* ( =, + =, -=, *= і т. д.) задають нове значення змінної. Ці операції можуть використовуватися в програмі як закінчені оператори.

Формат операції *простого привласнення* (=):

Змінна = Вираз

Механізм виконання операції привласнення такий: обчислюється вираз і його результат заноситься в пам'ять за адресою, яка визначається ім'ям змінної, операцією, що знаходиться зліва від знаку. Те, що раніше зберігалося в цій області пам'яті, природно, втрачається. Схематично це корисно уявити собі так:

Змінна ← Вираз

Приклади операторів привласнення:

```
a = b + c / 2;
b = a;
a = b;
x = 1;
x = x + 0.5;
```

Для правого операнда операції привласнення повинно існувати неявне перетворення до типу лівого операнда. Наприклад, вираз цілого типу можна привласнити дійсній змінній, тому що цілі числа є підмножиною дійсних, і інформація при такому привласненні не втрачається:

дійсна змінна = цілий_вираз;

Результатом операції привласнення є значення, записане в лівий операнд. Тип результату збігається з типом лівого операнда.

У складних операціях привласнення ( + =, * =, /= і т. п.) при обчисленні виразу, що стоїть в правій частині, використовується значення з лівої частини. Наприклад, при складанні з привласненням до другого операнда додається перший, і результат записується в перший операнд, тобто вираз a += b є компактнішим записом виразу a = a + b.

Результатом операції складного привласнення є значення, яке записане в лівий операнд.

Операції привласнення правоасоціативні, тобто виконуються справа наліво, на відміну від більшості інших операцій (a = b = c означає a = (b = c) ).

## 3.3. Лінійні програми
## (програмування лінійних обчислювальних процесів)

Лінійною називається програма, всі оператори якої виконуються послідовно в тому порядку, в якому вони записані. Простим прикладом лінійної програми є програма розрахунку по заданій формулі. Вона складається з трьох етапів: введення початкових даних, обчислення за формулою і виведення результатів.

### 3.3.1. Просте введення-виведення даних

Будь-яка програма при введенні початкових даних і виведенні результатів взаємодіє із зовнішніми пристроями. Сукупність стандартних пристроїв введення і виведення, тобто клавіатури і екрану, називається консоллю. У мові C# немає операторів введення і виведення. Замість них для обміну із зовнішніми пристроями застосовуються стандартні об'єкти. Для роботи з консоллю в C# застосовується клас *Console*, визначений в просторі імен *System*. Методи цього класу *Write* і *Writeline* вже використовувалися в наших программах. Розглянемо ці методи детальніше на прикладі лістингу 3.9.

Лістинг 3.9. Методи виведення

```
using System;
namespace ConsoleApplicationl
{ class Classl
  { static void Main()
    {
    int i = 3;
    double y = 4.12;
    decimal d = 600m;
    string s = "Вася";
    Console.WriteLine( "i = " + i ) ;                    //1
    Console.WriteLine( "y = {0} \nd = {1}", y, d );     // 2
    Console.WriteLine( "s = " + s ) ;                    //3
    }
  }
}
```

Результат роботи програми:

```
i = 3
y = 4,12
d = 600
s = Вася
```

На лістингу 3.9 в рядках 1 і 3 показано виведення пояснень і значень змінних. Коли метод *Writeline* викликаний з одним параметром, він може бути числом, символом або рядком. Якщо потрібно вивести в рядку не одну, а дві величини: текстове пояснення і значення змінної, необхідно їх “склеїти” в один рядок за допомогою операції “+”. Перед об'єднанням рядка з числом треба перетворити число з його внутрішньої форми уявлення в послідовність символів, тобто в рядок. Перетворення в рядок визначене у всіх стандартних класах C# - для цього служить метод ToString(). В даному випадку він виконується неявно, але можна викликати його і явним чином:

```
Console.WriteLine( "i = " + i.ToString( ));
```

Оператор 2 ілюструє виведення формату. В цьому випадку використовується інший варіант методу *Writeline*, який містить більше одного параметра. Першим параметром передається рядковий літерал, що містить окрім звичайних символів і послідовностей, що управляють, *параметри* у фігурних дужках. Параметри нумеруються з нуля, перед виведенням вони замінюються значеннями відповідних змінних в списку виведення: нульовий параметр замінюється значенням першої змінної (у даному прикладі - *y*), перший параметр - другою змінною (у даному прикладі - *d*) і так далі

***Для кожного параметра можна задати ширину поля виведення і формат виведення. Це буде розглянуто нижче.***

З послідовностей, що управляють, найчастіше використовуються символи переведення рядка (\n) і горизонтальної табуляції (\t).

Розглянемо прості способи введення з клавіатури. У класі *Console* визначені методи введення рядка і окремого символу, але немає методів, які дозволяють безпосередньо зчитувати з клавіатури числа. Введення числових даних виконується в два етапи: символи, що є числом, вводяться з клавіатури в рядкову змінну. Потім виконується перетворення з рядка в змінну відповідного типу. Перетворення можна виконати або за допомогою спеціального класу *Convert*, визначеного в просторі імен *System*, або за допомогою методу *Parse*, наявного в кожному стандартному арифметичному класі. У лістингу 3.10 використовуються обидва способи.

Лістінг 3.10. Методи введення

```
using System;
namespace ConsoleApplicationl
{
   class Classl
    {
       static void Main()
       {
         Console.WriteLine("Введите строку");
```

```
            string s = Console.ReadLine( );                          // 1
            Console.WriteLine( "s = " + s );
            Console.WriteLine("Введите символ" );
            char c = (char)Console.Read( );                          // 2
            Console.ReadLine( );                                     // 3
            Console.WriteLine( "c = " + c );
            string buf;                                  // рядок - буфер дляввведення чисел
            Console.WriteLine( "Введите целое число" );
            buf = Console.ReadLine( );
            int i  = Convert.ToInt32( buf );                         // 4
            Console.WriteLine( i );
            Console.WriteLine( "Введите вещественное число" );
            buf = Console.ReadLine( );
            double x = Convert.ToDouble( buf );                      // 5
            Console.WriteLine( x );
            Console.WriteLine( "Введите вещественное число" );
            buf = Console.ReadLine( );
            double y = double.Parse( buf );                          // б
            Console.WriteLine( y );
            Console.WriteLine( "Введите вещественное число" );
            buf = Console.ReadLine( );
            decimal z = decimal.Parse( buf );                        // 7
            Console.WriteLine( z );
            }
        }
    }
```

### 3.3.2. Математичні функції - клас Math

У виразах часто використовуються математичні функції. Вони реалізовані в класі Math, визначеному в просторі імен *System*. За допомогою методів цього класу можна обчислити:

- тригонометричні функції: *Sin, Cos, Tan*;
- зворотні тригонометричні функції: *Asin, Acos, Atan, Atan2*;
- гіперболічні функції*: Tanh, Sinh, Cosh*;
- експоненту і логарифмічні функції*: Exp, Log, Log10*;
- модуль (абсолютну величину), квадратний корінь, знак: *Abs, Sqrt, Sign*;
- округлення: *Ceiling, Floor, Round*;
- мінімум, максимум: *Min, Max*;
- ступінь, залишок: *Pow, Ieeereminder*;
- повне множення двох цілих величин*: Bigmul*;
- ділення і залишок від ділення: *Divrem*.

Крім того, у класу є два корисні поля: число $\pi$ і число *e*. Опис методів і полів приведений в таблиці 3.8.

Таблиця 3.8.

Основні поля і статичні методи класу *Math*

| Ім'я | Опис | Результат | Пояснення |
|---|---|---|---|
| Abs | Модуль | Перевантажений | \|x\| записується як Abs(x) |
| Acos | Арккосинус | Double | Acos(double x) |
| Asin | Арксинус | Double | Asin(double x) |
| Atan | Арктангенс | Double | Atan(double x) |
| Atan2 | Арктангенс | Double | Atan2(double x, double y) - кут, тангенс якого є ділення y на x |
| BigMul | Множення | Long | BigMul (int x, int y) |
| Ceiling | Округлення до більшого цілого | Double | Ceiling(double x) |
| Cos | Косинус | Double | Cos(double x) |
| Cosh | Гіперболічний косинус | Double | Cosh(double x) |
| DivRem | Ділення і залишок | Перевантажений | DivRem(x, y, rem) |
| e | Підстава натурального логарифма (число e) | Double | 2,71828182845905 |
| Exp | Експонента | Double | Exp(x) |
| Floor | Округлення до меншого цілого | Double | Floor(double x) |
| IEEERemainder | Залишок від ділення | Double | IEEERemainder (double x, double y) |
| Log | Натуральний логарифм | Double | Log(x) |
| Log 10 | Десятковий логарифм | Double | Log10(x) |
| Max | Максимум з двох чисел | Перевантажений | Max(x, y) |
| Min | Мінімум з двох чисел | Перевантажений | Min(x, y) |
| π | Значение числа π | Double | 3,14159265358979 |
| Pow | Піднесення до ступеня | Double | Pow(x, y) |
| Round | Округлення | Перевантажений | Round (3.1) дасть в результаті 3, Round (3.8) дасть в результаті 4 |
| Sign | Знак числа | Int | Аргументи перевантажені |
| Sin | Синус | Double | Sin(double x) |
| Sinh | Гіперболічний синус | Double | Sinh(double x) |
| Sqrt | Квадратний корінь | Double | Sqrt(x) |
| Tan | Тангенс | Double | Tan(double x) |
| Tanh | Гіперболічний тангенс | Double | Tanh(double x) |

У лістингу 3.11 приведений приклад застосування двох методів класу *Math*.

Лістинг 3.11. Застосування методів класу *Math*

```
using System;
namespace ConsoleApplicationl
{
    class Classl
  {
  static void Main()
    {
          Console.Write("Введите x:");
          string buf = Console.ReadLine( );
          double x = double.Parse( buf );
          Console.WriteLine("Значение sin = " + Math.Sin(x) );
          Console.Write("Введите y:");
          buf = Console.ReadLine();
          double y = double.Parse( buf );
          Console.WriteLine("Максимум : " +  Math.Max(x, y) );
  }
  }
}
```

Як приклад розглянемо програму розрахунку по формулі

$$y = \sqrt{\pi \cdot x} - e^{0,2\sqrt{\alpha}} + 2tg2\alpha + 1,6 \cdot 10^3 \cdot \log_{10} x^2 .$$

Лістинг 3.12. Програма розрахунку по заданій формулі

```
using System;
namespace ConsoleApplicationl
{
    class Classl
    {
       static void Main()
       {
           string buf;
           Console.WriteLine( "Введите x" );
           buf = Console.ReadLine();
           double x = Convert.ToDouble( buf );
           Console.WriteLine("Введите alfa" );
           buf = Console.ReadLine();
           double a = double.Parse( buf );
           double y = Math.Sqrt(Math.PI * x ) - Math.Exp(0.2*Math.Sqrt(a))+
           2*Math.Tan(2*a) + 1.6e3 * Math.Log10( Math.Pow(x,2));
           Console.WriteLine("Для x = {0} и alfa = {1}", x, a );
           Console.WriteLine("Результат = " + y );
       }
    }
}
```

# РОЗДІЛ 4. ОПЕРАТОРИ

## 4.1. Вирази, блоки

Будь-який вираз, що завершується крапкою з комою, розглядається як *оператор*. Окремим випадком виразу є порожній оператор “;”

Приклади:

```
i++;              //  виконується операція інкремента
a *= b + c;       //  виконується множення з привласненням
fun(i, k);        //  виконується виклик функції
while( true );    //  цикл з порожнього оператора (нескінченний)
```

*Блок* - це послідовність описів і операторів, що виділяються фігурними дужками.

## 4.2. Оператори розгалуження

Для програмування розгалужених обчислювальних процесів використовуються оператори *if* і *switch*, які залежно від конкретних значень початкових даних забезпечують виконання різних послідовностей операторів. Оператор *if* забезпечує передачу управління на одну з двох гілок обчислень, а оператор *switch* - на одну з довільного числа гілок.

### 4.2.1. Умовний оператор if

Формат оператора:

if ( логическое_выражение) оператор_1 [ else оператор_2 ]

Спочатку обчислюється логічний вираз. Якщо він має значення *true*, виконується перший оператор, інакше - другий. Після цього управління передається на оператора, наступного за умовним. Гілка *else* може бути відсутньою. Якщо в якій-небудь гілці потрібно виконати декілька операторів, їх необхідно укласти в блок, інакше компілятор не зможе зрозуміти, де закінчується розгалуження. Блок може містити будь-яких операторів, зокрема описи і інших умовних операторів (але не може складатися з одних описів). Необхідно враховувати, що змінна, описана в блоці, поза блоком не існує.

Приклади умовних операторів:

```
if (a< 0 ) b = 1;                                              // 1
if (a< b && (a > d || a= = 0 ) ) b++ ; else{ b*=a; a = 0;}     // 2
if (a<b) if ( a < c) m = a ; else m = c;
else       if ( b < c) m = b; else m = c;                      // 3
if (b> 0) max = b; else max = a;                               // 4
```

У прикладі 1 відсутня гілка *else.* Подібна конструкція реалізує пропуск оператора, оскільки привласнення або виконується, або пропускається залежно від виконання умови.

Якщо потрібно перевірити декілька умов, їх об'єднують знаками логічних умовних операцій. Наприклад, вираз в прикладі 2 буде істинний в тому випадку, якщо виконається одночасно умова *a<b* і одна з умов в дужках. Якщо опустити внутрішні дужки, буде виконано спочатку логічне *І* потім - *АБО*. У 3 прикладі обчислюється найменше значення з трьох змінних. У прикладі 4 обчислюється найбільше значення з двох змінних. В даному випадку простіше і наочніше використовувати наступну умовну операцію:

max = a > b ? a : b

Як приклад підрахуємо кількість очок після пострілу по мішені, зображеній на рис. 4.1.

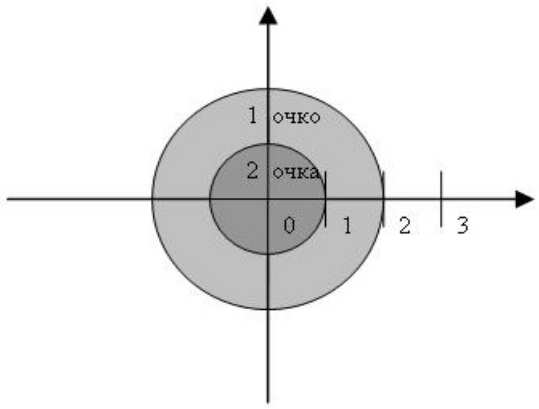



Рис. 4.1. Мішень

Тип змінних вибирається, виходячи з їх призначення. Координати пострілу не можна представити цілими величинами, оскільки це приведе до втрати точності результату, а лічильник очок не має сенсу описувати як дійсний. Програма приведена в лістингу 4.1.

Лістінг 4.1 . Постріл по мішені

```
using System;
namespace ConsoleApplicationl
{
class Classl
    {
        static void Main()
         {
           double x, y , temp;
           int kol;

           Console.Write("Введите x : ");
           x = Convert.ToDouble(Console.ReadLine());
```

```
            Console.Write("Введите y : ");
            y = Convert.ToDouble(Console.ReadLine());

            temp = x * x + y * y;
            kol = 0;
            if ( temp < 4 ) kol = 1;
            if ( temp < 1 ) kol = 2;
            Console.WriteLine( "Результат =  {0}  очков", kol );
        }
    }
}
```

Слід уникати перевірки дійсних величин на рівність, замість цього краще порівнювати модуль їх різниці з деяким малим числом. Це пов'язано з погрішністю представлення дійсних значень в пам'яті:

```
float a, b;
if ( a = = b ) ...                          // не рекомендується!
if ( Math.Abs(a - b) < 1e-6 )               // надійно
```

### 4.2.1. Умовний оператор switch

Формат оператора:

```
switch ( вираз )
{
    case константний_вираз_1:  [ список_операторів_1 ]
    case константний_вираз_2:  [ список_операторів_2 ]
    ...
    case константний_вираз_n:  [ список_операторів_n ]
    [default: оператори ]
}
```

Виконання оператора починається з обчислення виразу. Тип виразу найчастіше цілочисельний (включаючи char) або рядковий. Потім управління передається першому операторові із списку, поміченому константним виразом, значення якого збіглося з обчисленим.

Всі константні вирази мають неявно приводитися до типу виразу в дужках. Якщо збігу не відбулося, виконуються оператори, розташовані після слова default (а при його відсутності управління передається наступному за switch операторові).

Після оператора *case* наявність блоку не обов'язкова. Кожен блок (гілка перемикача) повинен закінчуватись явним оператором break, goto або return:

- оператор break виконує вихід з оператора switch;
- оператор goto виконує перехід на вказану після нього мітку;
- оператор return виконує вихід з функції, в тілі якої він записаний.

У лістингу 4.2. приведений приклад калькулятора на чотири прості дії.

Лістинг 4.2. Калькулятор на чотири дії

```
using System;
namespace ConsoleApplicationl
{class Classl
    {static void Main( )
        {
          string buf;
          double a, b, res;
          Console.WriteLine("Введите первый операнд:");
          buf = Console.ReadLine( );
          a = double. Parse( buf );
          Console.WriteLine("Введите знак операции");
          char op = (char)Console.Read( );
          Console.ReadLine();
          Console.WriteLine("Введите  второй операнд:");
          buf = Console.ReadLine( );
          b = double.Parse(buf);
          bool ok = true;
          switch (op)
            {
          case '+' : res = a + b; break;
          case '-'  : res = a - b; break;
          case '*' : res = a * b; break;
          case '/'  : res = a / b; break;
          default  : res = double.NaN;
          ok = false;
          break;
            }
          if  (ok)  Console.WriteLine(  "Результат:  "  +  res );
          else      Console.WriteLine(   "Недопустимая операция" );
        }
    }
}
```

Хоча наявність гілки default і не обов'язкова, рекомендується завжди обробляти випадок, коли значення виразу не збігається ні з однією з констант. Це полегшує пошук помилок при відладці програми.

У лістингу 4.3. приведений приклад використання оператора переходу goto.

Лістинг 4.3. Використання операторa goto

```
using System;
class SwitchTest
{
   static void Main()
   {
      Console.WriteLine(" Введіть 1, 2 або 3 ");
      string s = Console.ReadLine();
      int n = int.Parse(s);
      int cost = 0;
      switch(n)
      {
```

```
                    case 1:
                       cost += 25;
                       break;
                    case 2:
                       cost += 25;
                       goto case 1;
                    case 3:
                       cost += 50;
                       goto case 1;
                    default:
                       Console.WriteLine("Натиснута не та клавіша");
                       break;
     }
            if (cost != 0)
            Console.WriteLine("Значення: {0}", cost);
            else
            Console.WriteLine("Натиснута не та клавіша");

  } }
```

При значенні змінної n = 2, cost = 50;

В лістингу 4.4 показаний приклад без використання операторів переходу.

Лістинг 4.4.

```
using System;
class SwitchTest
{
   static void Main()
   {
       Console.WriteLine(" Введіть число ");
       string s = Console.ReadLine();
       int n = int.Parse(s);
      switch(n)
      {
         case 1:
         case 2:
         case 3:
            Console.WriteLine("Введено 1, 2, або 3");
            break;
         default:
         Console.WriteLine("Натиснута не та клавіша");
         break;
      }
   }
}
```

При $n > 0$ і $n < 4$ буде повідомлення “Введене 1, 2, або 3”.

## 4.3. Оператори циклу

Оператори циклу використовуються для обчислень, що повторюються багато разів. У C# цикли реалізуються за допомогою операторів: *while*, *do*, *for*, *foreach*. Кожен з них складається з певної послідовності операторів.

Блок, за ради виконання якого і організовується цикл, називається тілом циклу. Решта операторів служить для управління процесом повторення обчислень: це початкові установки, перевірка умови продовження циклу і модифікація параметра циклу. Один прохід циклу називається ітерацією.

Початкові установки служать для того, щоб до входу в цикл задати значення змінних, які в нім використовуються.

Перевірка умови продовження циклу виконується на кожній ітерації або до тіла циклу, або після тіла циклу. Параметром циклу називається змінна, яка використовується при перевірці умови продовження циклу і примусово змінюється на кожній ітерації, причому, як правило, на одну і ту ж величину. Якщо параметр циклу цілочисельний, він називається лічильником циклу. Кількість повторень такого циклу можна визначити заздалегідь. Зазвичай такі цикли називаються арифметичними. Якщо заздалегідь не відома кількість ітерацій, цикл називають ітеративним і він завершується за заданої умови в тілі циклу. Можливе примусове завершення як поточної ітерації, так і циклу в цілому. Для цього служать оператори *break*, *continue*, *return*. Передавати управління ззовні всередину циклу забороняється - виникає помилка компіляції.

### 4.3.1. Цикл з передумовою while

Формат :

while ( вираз) оператор

Вираз має бути логічного типу. Наприклад, це може бути операція відношення або просто логічна змінна. Якщо результат обчислення виразу рівний true, виконується простий або складений оператор (блок). Ці дії повторюються до того моменту, поки результатом виразу не стане значення false. Після закінчення циклу управління передається на наступного за ним оператора.

Вираз обчислюється перед кожною ітерацією циклу. Якщо при першій перевірці вираз рівний false, цикл не виконається жодного разу.

Як приклад розглянемо програму, що виводить таблицю аргументів $x$ і значень $y$ за наступних умов:

$$y = \begin{cases} t, & x < 0 \\ tx, 0 & x < 10 \\ 2t & x \geq 10 \end{cases}.$$

Назвемо початкове значення аргументу Xn, кінцеве значення аргументу Xk, крок зміни аргументу dx і параметр t. Всі величини дійсні. Програма повинна виводити таблицю, що складається з двох стовпців: значень аргументу і відповідних ним значень функції. Текст програми приведений в лістингу 4.5

Лістинг 4.5. Використання циклу while

```
using System;
namespace ConsoleApplicationl
{class Classl
    {static void Main( )
        {
          double Xn = -2, Xk = 12, dX = 2, t = 2, y;
          Console.WriteLine("|     x      |      y     |");
          double x = Xn;
          while (x <= Xk)
          {
            y = t;
            if (x >= 0 && x < 10) y = t * x;
            if (x >= 10) y = 2 * t;
            Console.WriteLine("| {0,6}       |  {1,6}      |", x, y);
            x += dX;
          }
        }
    }
}
```

Поширеним прийомом програмування є організація нескінченного циклу із заголовком while (true) і примусовим виходом з тіла циклу; по виконанню якої-небудь умови за допомогою операторів передачі управління. У лістингу 4.6 приведений приклад використання нескінченного циклу для організації меню програми.

Лістинг 4.6. Організація меню

```
using System;
namespace ConsoleApplicationl
{class Classl
    {static void Main( )
        {
            string buf;
            while  ( true )
            {
              Console.WriteLine(  "1   -  пункт_1,  2 -  пункт_2,  3 -  выход" );
              buf  = Console.ReadLine( );
              switch ( buf )
              {
                case "1":
                Console.WriteLine(   "пункт_1" );
                 break;
                case "2":
                Console.WriteLine(   "пункт_2" );
                 break;
                case "3": return;
                default:   Console.WriteLine(  "Повторите ввод" );
                break;
              }
            }
        } }}
```

На лістингу 4.7 приводиться використання операторів передачі управління *break* і *continue* на спрощеному прикладі, в якому виконується підрахунок суми цілих додатних чисел більше 10. Цикл припиняється при введенні від'ємного числа.

Лістинг 4.7 Використання операторів передачі управління

```
using System;
namespace ConsoleApplicationl
{class Classl
    {static void Main( )
        {
            string s;
            int sum = 0, n;
            while  ( true )
            {
                Console.WriteLine(" Введите число ");
                 s = Console.ReadLine();
                 n = int.Parse(s);
                 if (n < 0) break;
                 if (n <  11) continue;
                 sum+=n;
            }

           Console.WriteLine( "Сумма чисел = " + sum);
           }
        }
}
```

### 4.3.2. Цикл з постумовою do

Формат :

do оператор while вираз

Спочатку виконується простий або складений оператор, який створює тіло циклу, а потім обчислюється вираз (він повинен мати тип bool). Якщо вираз істинний, тіло циклу виконується ще раз і перевірка повторюється. Цикл завершується, коли вираз стане рівним false або в тілі циклу буде виконаний якийнебудь оператор передачі управління.

Цей вид циклу застосовується в тих випадках, коли тіло циклу необхідно обов'язково виконати хоч би один раз, наприклад, якщо в циклі вводяться дані і виконується їх перевірка. Якщо ж такої необхідності немає, переважно користуватися циклом з передумовою. Приклад перевірки введення приведений в лістингу 4.8

Лістинг 4.8 Перевірка введення

```
using System;
namespace ConsoleApplicationl
{  class Classl
    {     static void Main( )
        {
              char answer;
```

```
                    do
                    {
                        Console.WriteLine("Введите y(n) ?" );
                        answer =  (char)  Console.Read ( );
                        Console.ReadLine( );
                    } while  ( !(answer  == 'y'  || answer  == 'n'  ));
        }
    }
}
```

Розглянемо ще один приклад застосування циклу з постумовою - програму, що визначає корінь рівняння cos(x) методом ділення навпіл з точністю до 0,0001. Це ітераційний цикл, оскільки невідома кількість ітерацій. Результатом рішення є число, що є коренем рівняння. Суть методу половинного ділення полягає в наступному: задається інтервал, в якому є тільки один корінь (отже, на кінцях цього інтервалу функція має значення різних знаків). Обчислюється значення функції в середині цього інтервалу. Якщо воно того ж знаку, що і значення на лівому кінці інтервалу, значить, корінь знаходиться в правій половині інтервалу, інакше - в лівій. Процес повторюється для знайденої половини інтервалу до тих пір, поки значення функції буде менше заданої точності. Програма приведена у лістингу 4.9

Лістинг 4.9. Обчислення кореня заданого рівняння

```
using System;
namespace ConsoleApplication4
{
   class Program
   {
      static void Main(string[] args)
      {
        double x,  nach , left = 0,   right = 3;
        nach = left;
          do
          {
             x = (left + right) / 2;
             if (Math.Cos(nach) * Math.Cos(x) > 0) left = x;
             else right = x;
          } while (Math.Abs(Math.Cos(x)) >= 1E-4);

          Console.WriteLine( "Корень равен " + x );

      }
   }
}
```

### 4.3.3. Цикл з параметром for

Цикл з параметром *for* має наступний формат:

**for ( ініціалізація; вираз; модифікації ) оператор;**

*Ініціалізація* служить для оголошення величин, використовуваних в циклі, і привласнення ним початкових значень. У цій частині можна записати декілька операторів, розділених комою, наприклад:
for (int i = 0, j = 20; ...

Зоною дії змінних, оголошених в частині ініціалізації циклу, є цикл. Ініціалізація виконується один раз на початку виконання циклу.

*Вираз* типу bool визначає умову виконання циклу: якщо його результат рівний true, цикл виконується. Інакше виконується вихід з циклу. Цикл з параметром реалізований як цикл з передумовою.

*Модифікації* виконуються після кожної ітерації циклу і служать зазвичай для зміни параметрів циклу. У частині модифікацій можна записати декілька операторів через кому, наприклад:

for ( int i = 0, j = 20; i< 5 && j > 10; i ++ , j-- ) ...

Простим або складеним *оператором* є тіло циклу. Будь-яка з частин оператора for може бути відсутня але крапки з комою треба залишити на своїх місцях.

Для прикладу обчислимо суму чисел від 1 до 100:

```
int s = 0;
for ( int i = 1 , i <= 100; i ++ )s += i;
```

На лістингу 4.10 показаний приклад виведення значень аргументу і функцій з використанням оператора *for* ( на лістингу 4.5. цей же приклад був реалізований з використанням оператора while).

Лістинг 4.10. Використання оператора for

```
using System;
namespace ConsoleApplicationl
{
 class Classl
 {
  static void Main()
  {
   double Xn = -2, Xk = 12, dX = 2, t = 2, y;
   Console.WriteLine( " |     x      |      y     |" );
   for (double x = Xn; x <= Xk; x += dX)
```

```
            {
              y = t;
              if (x >= 0 && x < 10) y = t * x;
              if (x >= 10) y = 2 * t;
              Console.WriteLine(" | {0,6}       |  {1,6}      |", x, y);
            }
          }
        }
      }
```

Як бачите, в цьому варіанті програми все управління цілком зосереджене в його заголовку. Це робить програму зрозуміліше. Крім того, зоною дії службової змінної *x* є цикл, а не вся функція, як це було в лістингу 4.5, що переважно, оскільки змінна *x* поза циклом не потрібна.

Будь-який цикл *while* може бути приведений до еквівалентного йому циклу *for* і навпаки.

### 4.3.4. Цикл перебору foreach

Цикл *foreach* використовується для перегляду всіх об'єктів з деякої групи даних, наприклад масиву, списку або іншого контейнера. Він буде розглянутий нижче, коли у нас з'явиться в ньому необхідність.

### 4.3.5. Рекомендації по вибору оператора циклу

Оператори циклу взаємозамінні, але можна привести деякі рекомендації по вибору якнайкращого у кожному конкретному випадку.

Оператора *do while* зазвичай використовують, коли цикл потрібно обов'язково виконати хоч би раз, наприклад, якщо в циклі проводиться введення даних.

Оператором *while* зручніше користуватися в тих випадках, коли або число ітерації заздалегідь невідоме, або очевидних параметрів циклу немає, або модифікацію параметрів зручніше записувати не в кінці тіла циклу. Оператора *foreach* застосовують для перегляду елементів різних колекцій об'єктів.

Оператор *for* використовуэться переважно в останіх випадках. Однозначно - для організації циклів з лічильниками, тобто з цілочисельними змінними, які змінюють своє значення при кожному проході циклу регулярним чином (наприклад, збільшуються на 1).

## 4.4. Обробка виняткових ситуацій

Виняткова ситуація, або виключення, - це виникнення аварійної події, яка може породжуватися некоректним використанням апаратури або неправильною роботою програми, наприклад діленням на нуль або переповнюванням. Зазвичай ці події приводять до завершення програми з системним повідомленням

про помилку. C# дає програмістові можливість відновити працездатність програми і продовжити її виконання.

Виключення C# не підтримують обробку асинхронних подій, таких як помилки устаткування або переривання, наприклад натиснення клавіш Ctrl+C. Механізм виключень призначений тільки для подій, які можуть відбутися в результаті роботи самої програми і указуються явним чином. Виключення виникають тоді, коли деяка частина програми не змогла зробити те, що від неї було потрібно. При цьому інша частина програми може спробувати зробити щонебудь інше.

Виключення дозволяють логічно розділити обчислювальний процес на дві частини - виявлення аварійної ситуації і її обробка. Це важливо не тільки для кращої структуризації програми. Виключення генерує або середовище виконання, або програміст за допомогою оператора *throw*. У таблиці 4.1 приведені найбільш часто використовувані стандартні виключення, що генеруються середовищем. Вони визначені в просторі імен *System*. Всі вони є нащадками класу *Exception*, а точніше, нащадками його нащадка *SystemException*. Виключення виявляються і обробляються в операторові *try*.

Таблиця 4.1.

Часто використовувані стандартні виключення.

| Ім'я | Опис |
|---|---|
| ArithmeticException | Помилка в арифметичних операціях або перетвореннях (є предком Dividebyzeroexception і Overflowexception) |
| ArrayTypeMismatchException | Спроба збереження в масиві елементу несумісного типу |
| DivideByZeroException | Спроба ділення на нуль |
| FormatException | Спроба передати в метод аргумент невірного формату |
| IndexOutOfRangeException | Індекс масиву виходить за межі діапазону |
| InvalidCastException | Помилка перетворення типу |
| OutOfMemoryException | Недостатньо пам'яті для створення нового об'єкту |
| OverFlowException | Переповнювання при виконанні арифметичних операцій |
| StackOverFlowException | Переповнення стека |

### 4.4.1. Оператор try

Оператор *try* містить три частини:

- *контрольований блок* - складений оператор, що передує ключовим словом try. У контрольований блок включаються потенційно небезпечні оператори програми;

- *один або декілька обробників виключень* - блоків *catch*, в яких описується, як обробляються помилки різних типів;
- *блок завершення finally* виконується незалежно від того, виникла помилка в контрольованому блоці чи ні.

Синтаксис оператора *try*:

**try блок [ блоки catch ] [ блок finally ]**

Бути відсутніми можуть або блоки *catch*, або блок *finally*, але не обидва одночасно.

Розглянемо, яким чином реалізується обробка виняткових ситуацій.

1. Обробка виключення починається з появи помилки. Функція або операція, в якій виникла помилка, генерує виключення. Як правило, виключення генерується не безпосередньо в блоці *try*, а у функціях, прямо або побічно в нього вкладених.
2. Виконання поточного блоку припиняється, відшукується відповідний обробник виключення, і йому передається управління.
3. Виконується блок *final1y*, якщо він присутній (цей блок виконується і в тому випадку, якщо помилка не виникла).
4. Якщо обробник не знайдений, викликається стандартний обробник виключення. Його дії залежать від конфігурації середовища. Зазвичай він виводить на екран вікно з інформацією про виключення і завершує поточний процес.

Подібні вікна не призначені для користувачів програми, тому всі виключення, які можуть виникнути в програмі, мають бути перехоплені і оброблені.

*Обробники виключень* повинні розташовуватися безпосередньо за блоком *try*. Вони починаються з ключового слова *catch*, за яким в дужках слідує тип оброблюваного виключення. Можна записати один або декілька обробників відповідно до типів оброблюваних виключень. Блоки *catch* є видимими в тому порядку, в якому вони записані, поки не буде знайдений відповідний типу викинутого виключення.

Синтаксис обробників нагадує визначення функції з одним параметром - типом виключення. Існують три форми запису:

```
catch( тип ім'я ) { ... /* тіло обробника */ }
catch( тип )      {... /* тіло обробника */ }
catch             { ... /* тіло обробника */ }
```

Перша форма застосовується, коли ім'я параметра використовується в тілі обробника для виконання яких-небудь дій, наприклад виведення інформації про виключення.

Друга форма не припускає використання інформації про виключення, грає роль тільки його тип.

Третя форма застосовується для перехоплення всіх виключень. Оскільки обробники є видимими в тому порядку, в якому вони записані, обробник третього типу (він може бути тільки один) слід поміщати після всіх останніх. Приклад:

```
try
{
        //  Контрольований блок
}

catch (OvertlowException e )
{
        //Обробка виключення класу OverflowException (переповнення)
}
catch (DivideByZeroExceptfon)
{
        //Обробка виключення класу DivideByZeroException (ділення на 0)
}
catch {
        //Обробка усіх інших виключень
}
```

Якщо виключення в контрольованому блоці не виникло, всі обробники пропускаються. У будь-якому випадку, відбулося виключення чи ні, управління передається в блок завершення finally (якщо він існує), а потім - першому операторові, що знаходиться безпосередньо за оператором *try*. У завершуючому блоці зазвичай записуються оператори, яких необхідно виконати незалежно від того, виникло виключення чи ні, наприклад, закриття файлів, з якими виконувалася робота в контрольованому блоці, або виведення інформації.

У лістингу 4.11 приведена програма, що обчислює силу струму по заданій напрузі і опору. Оскільки вірогідність невірного набору дійсного числа досить висока, оператор введення включений в контрольований блок.

**Виключення, пов'язане з діленням на нуль, для дійсних значень виникнути не може, тому не перевіряється. При діленні на нуль буде виданий результат “нескінченність”.**

Лістинг 4.11. Використання виключень для перевірки введення

```
using System;
namespace ConsoleApplicationl
{class Classl
    {static void Main( )
        {
            string buf;
            double u, i, r;
```

```
            try
            {
            Console.WriteLine(   "Введите  напряжение:" );
            buf = Console.ReadLine( );
            u = double.Parse( buf );
            Console.WriteLine( "Введите сопротивление:" );
            buf = Console.ReadLine();
            r = double. Parse(buf );
            i = u / r;
            Console.WriteLine(  "Сила тока -  " + i );
            }

            catch  (   FormatException )
            {
              Console.WriteLine(   "Неверный формат  ввода!" );
            }
            catch           // общий случай
            {
               Console.WriteLine(  "Неопознанное исключение" );
            }
        }
    }
}
```

Оператори try можуть багато разів вкладатися один в одного. Виключення, яке виникло у внутрішньому блоці *try* і не було перехоплене відповідним блоком *catch*, передається на верхній рівень, де продовжується пошук відповідного обробника. Цей процес називається розповсюдженням виключення. Розповсюдження виключень надає програмістові цікаві можливості. Наприклад, якщо на внутрішньому рівні недостатньо інформації для того, щоб провести повну обробку помилки, можна виконати часткову обробку і згенерувати виключення повторно, щоб воно було оброблене на верхньому рівні. Генерація виключення виконується за допомогою оператора throw.

### 4.4.2. Оператор throw

До цих пір ми розглядали виключення, які генерує середовище виконання C#, але це може зробити і сам програміст. Для генерації виключення використовується оператор throw з параметром, що визначає вид виключення. Параметр має бути об'єктом, породженим від стандартного класу System.Exception. Цей об'єкт використовується для передачі інформації про виключення його обробникові.

Оператор throw викликається або з параметром, або без нього:

**throw [ вираз ];**

Форма без параметра застосовується тільки усередині блоку *catch* для повторної генерації виключення. Тип виразу, що стоїть після *throw*, визначає тип виключення, наприклад:

throw new DivideByZeroException( );

Тут після слова throw записаний вираз, що створює об'єкт стандартного класу “помилка при діленні на 0” за допомогою операції new. При генерації виключення виконання поточного блоку припиняється і відбувається пошук відповідного обробника з передачею йому управління. Обробник вважається знайденим, якщо тип об'єкту, вказаного після throw, або той же що заданий в параметрі catch, або є похідним від нього.

Розглянемо приклад генерації виключень на лістингу 4.12.

Лістинг 4.12. Генерація виключень

```
using System;
namespace ConsoleApplication1
{
   class Test
   {
      static void F()
      {
         try
         {
            G();
         }
         catch (Exception e)
         {
            Console.WriteLine("Exception in F:   " + e.Message);
            e = new Exception("F");
            throw e;                          // повторна  генерація виключення
         }
      }
      static void G()
      {
         throw new Exception("G");       //моделювання виняткових  ситуацій
      }

      static void Main()
      {
         try
         {
            F();
         }
         catch (Exception e)
         {
          Console.WriteLine("Exception in Main:" + e.Message);
         }
      }
   }
}
```

У методі F виконується проміжна обробка виключення, яка полягає в тому, що на консоль виводиться поле Message перехопленого об'єкту e (про елементи класу Exception розповідається в наступному розділі). Після цього виключення генерується заново. Не дивлячись на те що в обробнику виключення створюється новий об'єкт класу Exception із зміненим рядком інформації,що передається у виключенні, викидається не цей об'єкт, а той, який був перехоплений обробником, тому результат роботи програми наступний:

```
Exception in F: G
Exception in Main: G
```

Замінимо оператор *throw* таким оператором:

```
throw e;
```

В цьому випадку в обробнику буде викинуто виключення, створене в попередньому операторові, і виведення програми зміниться:

```
Exception in F: G
Exception in Main: F
```

### 4.4.3. Клас Exception

Клас *Exception* містить декілька корисних властивостей, за допомогою яких можна отримати інформацію про виключення. Вони перераховані в таблиці 4.2.

Таблиця 4.2

Властивості класу System.Exception

| **Властивість** | **Опис** |
|---|---|
| HelpLink | URL файла справки з описом помилки |
| Message | Текстовий опис помилки. Встановлюється при створенні об'єкту. Властивість доступна тільки для читання |
| Source | Ім'я об'єкту або додатку, яке згенерувало помилку |
| StacкTrace | Послідовність викликів, які привели до виникнення помилки. Властивість доступна тільки для читання |
| InnerExcepti on | Містить посилання на виключення, що послужило причиною генерації поточного виключення |
| TargetSite | Метод, що викинув виключення |

### 4.4.4. Оператори checked и unchecked

Процесом генерації виключень, що виникають при переповнюванні, можна управляти за допомогою ключових слів *checked* і *unchecked*, які використовуються як операції, якщо вони використовуються у виразах, і як оператори, якщо вони стоять перед блоком, наприклад:

```
a =  checked  (b + c);        // для   виразу   (перевірка включена)
unchecked
 {        // для  блоку операторів   (перевірка вимкнена)
          a = b + c:
 }
```

Перевірка не розповсюджується на функції, викликані в блоці.

### 4.5. Рекомендації по програмуванню

Головна мета, до якої потрібно прагнути, - отримати зрозумілу програму якомога простішої структури. Всі технології програмування направлені на досягнення саме цієї мети, оскільки тільки так можна добитися надійності програми і легкості її модифікації.

Створення програми треба починати з визначення її початкових даних і результатів. При цьому замислюються не тільки про сенс кожної величини, але і про те, яку множину значень вона може приймати. Відповідно до цього вибираються типи змінних.

Наступний крок - записати алгоритм рішення задачі на природній мові або із застосуванням узагальнених блок-схем. Другий варіант ефективніший. Якщо розробник не може сформулювати алгоритм, велика вірогідність того, що він погано продуманий. При цьому не потрібно детально описувати алгоритм на рівні окремих операторів. Опис алгоритму дозволяє детальніше осмислити завдання, знайти на ранній стадії деякі помилки, розбити програму на логічну послідовність блоків, а також забезпечити коментарі до програми.

При кодуванні програми необхідно пам'ятати про принципи структурного програмування: програма повинна складатися з чіткої послідовності блоків - базових конструкцій, кожна з яких має один вхід і один вихід. Конструкції можуть вкладатися, але не перетинатися.

Програма має бути “прозора”. Якщо яку-небудь дію можна запрограмувати різними способами, то перевага повинна віддаватися не найбільш компактному і навіть не найбільш ефективному, а зрозумілішому. Особливо це важливо тоді, коли пишуть програму одні, а супроводжують інші, що є широко поширеною практикою. «Непрозоре» програмування може спричинити величезні витрати, пов'язані з пошуком помилок при відладці.

Для запису кожного фрагмента алгоритму необхідно використовувати найбільш відповідні засоби мови. Наприклад, галуження на декілька напрямів за значенням цілої або рядкової змінної ефектніше записати за допомогою одного оператора *switch*, а не декількох операторів *if*. Для перегляду масиву краще користуватися циклом *for* або *foreach*. Оператор goto застосовують дуже рідко, наприклад, в операторі вибору switch або для примусового виходу з декількох вкладених циклів, а в більшості інших ситуацій краще використовувати інші засоби мови, такі як break або return.

Для організації циклів користуйтеся найбільш відповідним оператором. Цикл *do* застосовується тільки в тих випадках, коли тіло у будь-якому випадку потрібно буде виконати хоч би один раз, наприклад, при перевірці введення. При використанні циклів треба прагнути об'єднувати ініціалізацію, перевірку умови виходу і приріст в одному місці. При записі ітеративних циклів (у яких для перевірки умови виходу використовуються співвідношення змінних, що формуються в тілі циклу), необхідно передбачати аварійний вихід після досягнення заздалегідь заданої максимальної кількості ітерацій. Це підвищує надійність програми.

Безглуздо використовувати перевірку на рівність true або false:

```
bool busy;
...
if ( busy = = true )    { ... } // погано! Краще if ( busy )
if ( busy = = false )  { ... } // погано!  Краще if ( !busy )
```

Слід уникати зайвих перевірок умов. Наприклад:

```
if ( a < b ) c = 1;
else if ( a > b ) c = 2;
else if ( a == b ) c = 3;
```

Замість цих операторів можна написати:

```
if       ( a < b )  c = 1;
else if ( a > b )  c = 2;
else                    c = 3;
```

Або навіть так:

```
c = 3;
if ( a < b ) c = 1;
if ( a > b ) c = 2;
```

Якщо перша гілка оператора if забезпечує передачу управління, використовувати *гілку else* немає необхідності:

```
if ( i > 0 ) break;
// тут i <= 0
```

В деяких випадках умовна операція краще умовного оператора:

```
if ( z = = 0 ) i = j; else i = k;      // краще так: i = z = = 0 ? j  :  k;
```

*Необхідно передбачати друк повідомлень або генерацію виключення в тих точках програми, куди управління при нормальній роботі програми передаватися не повинно*. Саме це повідомлення ви з великою вірогідністю отримаєте при першому ж запуску програми. Наприклад, корисно, якщо оператор switch має гілку default, що реалізовує обробку ситуації за умовчанням, якщо в ній перераховані всі можливі значення перемикача.

*У програмі корисно передбачати реакцію на невірні вхідні параметри*. Це може бути генерація виключення (переважно), друк повідомлення або формування ознаки результату з його подальшим аналізом. *Повідомлення про помилку має бути інформативним і* підказувати користувачеві, як її виправити. Наприклад, при введенні невірного значення в повідомленні має бути вказаний допустимий діапазон.

Укладайте потенційно небезпечні фрагменти програми в блок *try*, що перевіряється, і обробляйте хоч би виключення типу *Exception*, а краще - всі виключення, які можуть в нім виникнути, окремо.

*Після написання програму слід ретельно відредагувати* - прибрати непотрібні фрагменти, згрупувати описи, оптимізувати перевірки умов і цикли, перевірити, чи оптимальне розбиття на методи, і так далі. З першого разу без помарок хороший текст не напишеш, будь то твір, стаття або програма. Проте не слід намагатися оптимізувати все, що “попадається під руку”, оскільки головний принцип програміста той же, що і лікаря: “Не нашкодь!”.

Підходити до написання програми потрібно так, щоб її можна було у будь-який момент передати іншому програмістові. Корисно дати почитати свою програму кому-небудь з друзів або колег (а ще краще - ворогів або заздрісників) і в тих місцях, які вони не зможуть зрозуміти без усних коментарів внести їх прямо до тексту програми.

Навіть якщо супроводжуючим програмістів є автор програми, знатися через рік на погано документованому тексті сумнівне задоволення. Подальші поради стосуються коментарів і форматування тексту програми, що є невід'ємною частиною процесу програмування.

Програма, якщо вона використовується, живе не один рік, потреба в якихось її нових властивостях виявляється відразу ж після введення в експлуатацію, і супровід програми займає значно більше часу, чим її написання. Основна частина документації повинна знаходитися в тексті програми. Хороші коментарі написати майже так само складно, як і хорошу програму. Коментарі мають бути правильними пропозиціями без скорочень і з розділовими знаками.

Якщо коментар до фрагмента програми займає декілька рядків, розмістити його краще перед фрагментом, чим справа від нього, і вирівняти по вертикалі.

Вкладені блоки повинні мати відступ в 3-5 символів, причому блоки одного рівня вкладеності мають бути вирівняні по вертикалі. Форматуйте текст по стовпцях скрізь, де це можливо, це робить програму набагато зрозумілішою:

```
string but      = "qwerty";
double ex       = 3.1234;
int      number       = 12;
byte     z       =  0;
…
if ( done  ) Console.WriteLine(   "Сума ряду  -   " + y );
else         Console.WriteLine(   "Ряд розходиться" );:
…
if( x>= 0 && x < 10 )       y = t * x;
else if( x >= 10  )         y = 2 * t;
else                        y = x;
```

Для розділення методів і інших логічно закінчених фрагментів користуйтеся порожніми рядками або коментарем вигляду

```
//---------------------------------------------------------------------------
```

Позначайте кінець довгого складеного оператора, наприклад:

```
while  ( true )
{
    while ( x < y )
    {
        for(i=0; i<10;  + + i )
        {
              for( j = 0; j<10; + + j )
              {
                    // дві сторінки коду
              }     // end for ( j = 0;  j < 10; ++ j )
        }           // end for ( i = 0; i < 10; ++ i )
    }               // end while ( x < y )
}                   // end while  ( true )
```

Конструкції мови C# в основному сприяють хорошому стилю програмування, проте, і на цій мові можна написати заплутану, ненадійну, неприваблиу програму, яку простіше переписати заново, чим внести до неї необхідні зміни. Тільки постійні тренування, самоконтроль і прагнення до вдосконалення допоможуть вам освоїти хороший стиль, без якого неможливо стати кваліфікованим програмістом.

# РОЗДІЛ 5. КЛАСИ: ОСНОВНІ ПОНЯТТЯ

Всі програми, які приведені в роботі, складалися з одного класу і єдиним методом *Main*. Починаючи з цього розділу, розглядається вивчення створення і використання класів.

Клас є типом даних, визначуваним користувачем. Він має бути однією логічною суттю, наприклад, бути моделлю реального об'єкту або процесу. Елементами класу є дані і функції, призначені для їх обробки.

Опис класу містить ключове слово *class*, за яким слідує його ім'я, а далі у фігурних дужках - тіло класу, тобто список його елементів. Крім того, для класу можна задати його базові класи (предки) і ряд необов'язкових атрибутів і специфікаторів, що визначають різні характеристики класу:

**[ атрибути ] [ специфікатори ] class ім'я_класу [ : предки ]**
**тіло класу**

Як бачите, обов'язковими є тільки ключове слово *class*, а також ім'я і тіло класу. Ім'я класу задається програмістом за загальними правилами C#. Тіло класу - це список описів його елементів, взятий у фігурні дужки. Список може бути порожнім, якщо клас не містить жодного елементу. Таким чином, простий опис класу може виглядати так:

```
class  Demo {}
```

Необов'язкові атрибути задають додаткову інформацію про клас. Оскільки ми вивчаємо поки основні поняття про клас, то відкладемо розгляд атрибутів до розділу 12.

Специфікатори визначають властивості класу, а також доступність класу для інших елементів програми. Можливі значення специфікаторів перераховані в таблиці 5.1.

Таблиця 5.1

Специфікатори класу

| № | Специфікатор | Опис |
|---|---|---|
| 1 | new | Використовується для вкладених класів. Задає новий опис класу замість успадкованого від предка. Застосовується в ієрархіях об'єктів |
| 2 | public | Доступ не обмежений |
| 3 | protected | Використовується для вкладених класів. Доступ тільки з елементів даного класу і похідних класів |
| 4 | internal | Доступ тільки з даної програми (збірки) |
| 5 | protected internal | Доступ тільки з даного класу і похідних класів або з даної програми (збірки) |
| 6 | private | Використовується для вкладених класів. Доступ тільки з елементів класу, усередині якого описаний даний клас |

Продовження таблиці 5.1

| № | Специфікатор | Опис |
|---|---|---|
| 7 | abstract | Абстрактний клас. Застосовується в ієрархіях об'єктів |
| 8 | sealed | Безплідний клас. Застосовується в ієрархіях об'єктів |
| 9 | static | Статичний клас |

Клас можна описувати безпосередньо усередині простору імен або усередині іншого класу. У останньому випадку клас називається вкладеним. Залежно від місця опису класу деякі з цих специфікаторів можуть бути заборонені.

Специфікатори 2-6 називаються специфікаторами доступу. Вони визначають, звідки можна безпосередньо звертатися до даного класу. Специфікатори доступу можуть бути присутніми в описі тільки у варіантах, приведених в таблиці, а також можуть комбінуватися з рештою специфікаторів.

У цьому розділі ми вивчатимемо класи, які описуються в просторі імен безпосередньо (тобто не вкладені класи). Для таких класів допускаються тільки два специфікатори: *public* і *internal*, за умовчанням, тобто якщо жоден специфікатор доступу не вказаний, мається на увазі специфікатор *internal*.

Клас є узагальненим поняттям, що визначає характеристики і поведінку деякої множини конкретних об'єктів цього класу, званих екземплярами, або об'єктами, класу.

Об'єкти створюються явним або неявним чином, тобто або програмістом, або системою. Програміст створить екземпляр класу за допомогою операції *new*, наприклад:

```
Demo a = new Demo ( );          // створення екземпляра класуDemo
Demo b = new Demo ( );          // створення іншого екземпляра класуDemo
```

Як ви пам'ятаєте, клас відноситься до посилальних типів даних, пам'ять під яких виділяється. Таким чином, змінні *x* і *y* зберігають не самі об'єкти, а посилання на об'єкти, тобто їх адреси. Якщо достатній для зберігання об'єкту об'єм пам'яті виділити не вдалося, операція *new* генерує виключення *Outofmemory-exception*. Рекомендується передбачати обробку цього виключення в програмах, що працюють з об'єктами великого об'єму.

Для кожного об'єкту при його створенні в пам'яті виділяється окрема область, в якій зберігаються його дані. Крім того, в класі можуть бути присутніми статичні елементи, які існують в єдиному екземплярі для всіх об'єктів класу. Часто статичні дані називають даними класу, а останні - даними екземпляра.

Функціональні елементи класу не тиражуються, тобто завжди зберігаються в єдиному екземплярі. Для роботи з даними класу використовуються методи класу (статичні методи), для роботи з даними екземпляра - методи екземпляра, або просто методи.

До цих пір ми використовували в програмах тільки один вид функціональних елементів класу - методи. Поля і методи є основними елементами класу. Крім того, в класі можна задавати цілу гамму інших елементів, приведених на рис. 5.1.

Опис елементів класу:

• Константи класу зберігають незмінні значення, пов'язані з класом.

• Поля містять дані класу.

• Методи реалізують обчислення або інші дії, що виконуються класом або екземпляром.

• Властивості визначають характеристики класу в сукупності із способами їх завдання і отримання, тобто методами запису і читання.

• Конструктори реалізують дії з ініціалізації екземплярів або класу в цілому.

• Деструктори визначають дії, які необхідно виконати до того, як об'єкт буде знищений.

• Індексатори забезпечують можливість доступу до елементів класу по їх порядковому номеру.

• Операції задають дії з об'єктами за допомогою знаків операцій.

• Події визначають повідомлення, які може генерувати клас.

• Типи - це типи даних, внутрішні по відношенню до класу.

Перші п'ять видів елементів класу ми розглянемо в цьому розділі, а останні - в подальших. Але перш ніж почати вивчення, необхідно поговорити про привласнення і порівняння об'єктів

## 5.1. Привласнення і порівняння об'єктів

Механізм виконання привласнення один і той же для величин будь-якого типу, як посилального, так і значущого, проте результати розрізняються. При наданні значення копіюється значення, а при привласненні посилання - посилання, тому після привласнення одного об'єкту іншому ми отримуємо два посилання, вказуючи на одну і ту ж область пам'яті (рис. 5.2).

Рисунок ілюструє ситуацію, коли було створено три об'єкти, *a, b* і *c*, а потім виконано привласнення *b = c.* Старе значення *b* стає недоступним і очищається складальником сміття. З цього виходить, якщо змінити значення однієї величини посилального типу, це може відбитися на іншій (в даному випадку, якщо змінити об'єкт через посилання *c*, об'єкт *b* також змінить своє значення).

Аналогічна ситуація з операцією перевірки на рівність. Величини значущого типу рівні, якщо рівні їх значення. Величини посилального типу рівні, якщо вони посилаються на одні і ті ж дані (на рисунку об'єкти *b* і *c* рівні, але *a не* рівне *b* навіть при рівності їх значень або якщо вони обидві рівні null).

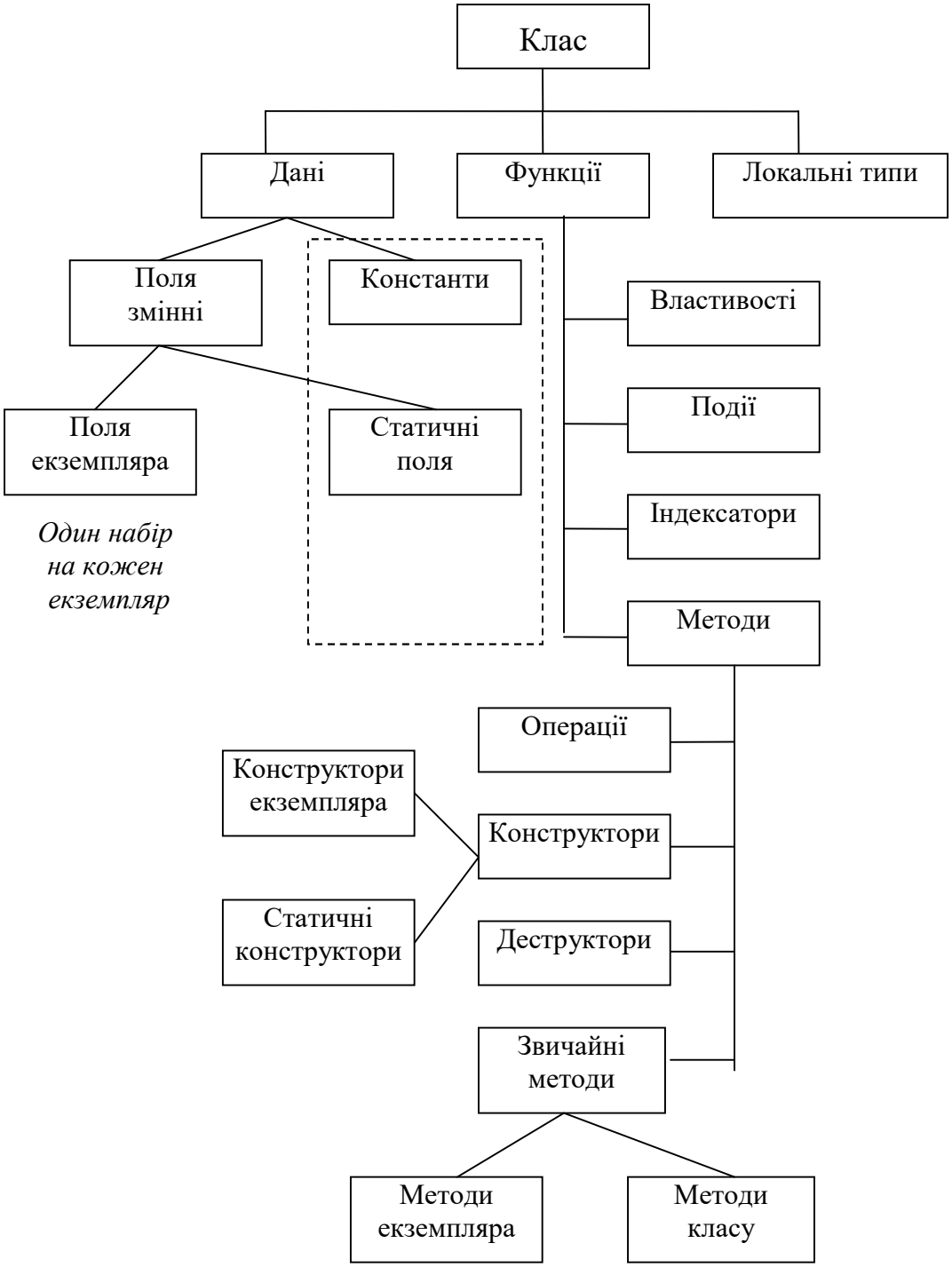



Рис. 5.1. Склад класу

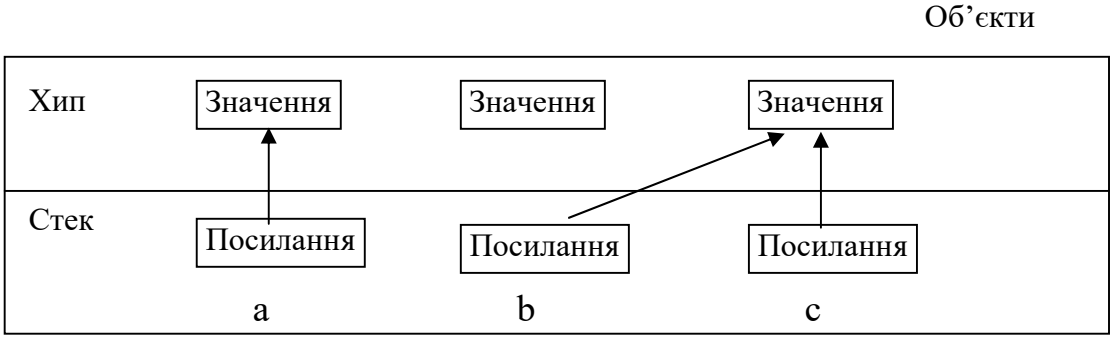



Рис. 5.2. Привласнення об'єктів

## 5.2. Дані: поля і константи

Дані, що містяться в класі, можуть бути змінними або Змінні, описані в класі, називаються полями класу.

При описі елементів класу можна також указувати атрибути і специфікатори, які задають різні характеристики елементів. Синтаксис опису елементу даних приведений нижче:

[ атрибути ] [ специфікатори ] [ const ] тип ім'я [ = начальне_значення ]

Атрибути будуть розглянуті в розділі 12, а можливі специфікатори полів і констант перераховані в таблиці 5.2. Для констант можна використовувати тільки специфікатори 1- 6.

Таблиця 5.2.

Специфікатори полів і констант класу

| № | Специфікатор | Опис |
|---|---|---|
| 1 | new | Новий опис поля, що приховує успадкований елемент класу |
| 2 | public | Доступ до елементу не обмежений |
| 3 | protected | Доступ тільки з даного класу і похідних класів |
| 4 | internal | Доступ тільки з даної збірки |
| 5 | protected internal | Доступ тільки з даного класу і похідних класів і з даної збірки |
| 6 | private | Доступ тільки з даного класу |
| 7 | static | Одне поле для всіх екземплярів класу |
| 8 | readonly | Поле доступне тільки для читання |
| 9 | volatile | Поле може змінюватися іншим процесом або системою |

За умовчанням елементи класу вважаються за закриті ( *private* ) . Для полів класу цей вид доступу є переважним, оскільки поля визначають внутрішню будову класу, яка має бути прихована від користувача. Всі методи класу мають безпосередній доступ до його закритих полів.

Поля, описані із специфікатором *static*, а також константи існують в єдиному екземплярі для всіх об'єктів класу, тому до них звертаються не через ім'я екземпляра, а через ім'я класу. Якщо клас містить тільки статичні елементи, екземпляр класу створювати не потрібно.

Звернення до поля класу виконується за допомогою операції доступу (крапка). Праворуч від крапки задається ім'я поля, зліва - ім'я екземпляра для звичайних полів або ім'я класу для статичних. У лістингу 5.1 приведені приклад простого класу Demo і два способи звернення до його полів.

Лістинг 5.1. Клас Demo, що містить поля і константу

```
using System;
namespace ConsoleApplication1
{
    class Demo
    {
      public int a = 1;                          // поле даних
      public const double c = 1.66;              // константа
      public static string s = "Demo";           // статичне поле класу
      double y;                                  // закрите поле даних
    }

    class Classl
    {
       static void Main( )
        {
           Demo x = new Demo( );                 // створення екземпляра класу Demo
           Console . WriteLine ( x.a );          // x.a - звернення до поля класу
           Console . WriteLine ( Demo.c );       // Demo.c - звернення до константи
           Console . WriteLine ( Demo.s );       // звернення до статичного поля
        }
    }
}
```

Поле *y* вивести на екран аналогічним чином не вдасться: воно є закритим, тобто недоступно ззовні ( з класу Classl). Оскільки значення цьому полю явним чином не привласнене, середовище привласнює йому значення нуль.

Всі поля спочатку автоматично ініціалізувалися нулем відповідного типу (наприклад, полям типу *int* привласнюється *0*, а посиланням на об'єкти - значення *nu11*) . Після цього полю привласнюється значення, задане при його явній ініціалізації. Завдання початкових значень для статичних полів виконується при ініціалізації класу, а звичайних - при створенні екземпляра.

Поля із специфікатором *readonly* призначені тільки для читання. Встановити значення такого поля можна або при його описі, або в конструкторі (конструктори розглядаються далі в цьому розділі).

## 5.3. Методи

Метод - це функціональний елемент класу, який реалізує обчислення або інші дії, що виконуються класом або екземпляром. Методи визначають поведінку класу.

Методом є закінчений фрагмент коду, до якого можна звернутися по імені. Він описується один раз, а викликатися може стільки разів, скільки необхідно. Один і той же метод може обробляти різні дані, передані йому як аргументів.

**Синтаксис методу:**

**[ атрибути ] [ специфікатори ] тип імя_метода ( [ параметри ] )**

**тіло методу**

Розглянемо основні елементи опису методу. Перший рядок є заголовком методу. Тілом методу є блок - послідовність операторів у фігурних дужках.

При описі методів можна використовувати специфікатори 1-7 з таблиці 5.2, що мають ті ж самі значення, що і для полів, а також специфікатори *virtual, sealed, override, abstract і extern*, які будуть розглянуті в міру необхідності. Найчастіше для методів задається специфікатор доступу *public*, адже методи складають інтерфейс класу - те, з чим працює користувач, тому вони мають бути доступні.

Статичні (*static*) методи, або методи класу, можна викликати, не створюючи екземпляр об'єкту. Саме таким чином використовується метод *Main*.

Параметри використовуються для обміну інформацією з методом. Параметр є локальною змінною, яка при виклику методу набуває значення відповідного аргументу. Зона дії параметра - весь метод.

Наприклад, щоб обчислити значення синуса для дійсної величини *x*, ми передаємо її як аргумент в метод *Sin* класу *Math*, а щоб вивести значення цієї змінної на екран, ми передаємо її у метод *Writeline* класу *Console:*

```
double x = 0.1;
double y = Math.Sin(x);
Console.WriteLine(x);
```

При цьому метод *Sin* повертає в точку свого виклику дійсне значення синуса, яке привласнюється змінній *y*, а метод *Writeline* нічого не повертає.

*Параметри,* що описуються в заголовку методу, визначають множину *аргументів*, які можна передавати в метод. Список аргументів при виклику начебто накладається на список параметрів, тому вони повинні відповідати один одному. Правила відповідності детально розглядаються в наступних розділах.

Для кожного параметра повинні задаватися його тип і ім'я. Наприклад, заголовок методу Sin виглядає таким чином:

```
public static double Sin( double a ):
```

Ім’я методу укупі з кількістю, типами і специфікаторами його параметрів є сигнатурою методу. У класі не повинно бути методів з однаковими сигнатурами.

У лістингу 5.2 в клас *Demo* додані методи установки і набуття значення поля *y.* Крім того, статичне поле *s* закрите, тобто визначено за умовчанням як private, а для його отримання використовується метод *Gets*.

Як видно з лістингу 5.2, методи класу мають безпосередній доступ до його закритих полів. Метод, описаний із специфікатором static, повинен звертатися тільки до статичних полів класу. Зверніть увагу на те, що статичний метод викликається через ім'я класу, а звичайний через ім'я екземпляра. При виклику методу з іншого методу того ж класу ім'я класу/екземпляра можна не указувати.

Лістинг 5.2. Прості методи

```
using System;
namespace ConsoleApplicationl
{
    class Demo
    {
        public int a = 1 ;
        public const double c = 1.66;
        static string s = "Demo";
        double y;

        public double Gety( )                    // метод отримання поля y
        {
          return y;
        }

        public void Sety( double y_ )            // метод установки поля y
        {
            y = y_;
        }

        public static string Gets ( )            // метод отримання поля s
        {
           return s;
        }
    }

    class Classl
    {
        static void Main( )
        {
        Demo x = new Demo( );
        x . Sety ( 0.12 );                       // виклик метода установки поля y
        Console.WriteLine (x.Gety( ));           // виклик метода отримання поля y
        Console.WriteLine (Demo.Gets( ));        // виклик метода отримання поля s
        }
    }
}
```

### 5.3.1. Параметри методів

Розглянемо детальніше, яким чином метод обмінюється інформацією з кодом, що викликав його. При виклику методу виконуються наступні дії:

1. Обчислюються вирази, що стоять на місці аргументів.

2. Виділяється пам'ять під параметри методу відповідно до їх типу.

3. Кожному з параметрів зіставляється відповідний аргумент (аргументи ніби накладаються на параметри і заміщають їх).

4. Виконується тіло методу.

5. Якщо метод повертає значення, воно передається в точку виклику; якщо метод має тип void, управління передається на оператора, наступного після виклику.

При цьому перевіряється відповідність типів аргументів і параметрів і при необхідності виконується їх перетворення. При невідповідності типів видається діагностичне повідомлення. Лістинг 5.3 ілюструє цей процес.

Лістинг 5.3. Передача параметрів методу

```
using System;
namespace ConsoleApplicationl
{
    class Class1
    {
          static int Max(int a, int b)
          {
             if (a>b) return a;
             return b;
          }

          static void Main( )
          {
            int a = 2, b = 4;
            int x = Max( a, b );                    //    виклик методу Max
            Console.WriteLine(x);                   //    результат: 4
            short tl =3,   t2 = 4;
            int y = Max( tl , t2 );                 //    виклик методу Max
            Console.WriteLine(y);                   //    результат: 4

            int z = Max( a + tl, tl / 2 * b);       //    виклик методу Max
            Console.WriteLine(z );                  //    результат: 5
            }
    }
}
```

У класі описаний метод Max, який вибирає найбільше з двох переданих йому значень. Параметри описані як *a* і *b*. У методі Main виконуються три виклики Max. В результаті першого виклику методу Max передаються два аргументи того ж типу, що і параметри, в другому виклику - аргументи сумісного типу, в третьому - вирази.

Головна вимога при передачі параметрів полягає в тому, що аргументи при виклику методу повинні записуватися в тому ж порядку, що і в заголовку методу, і повинне існувати неявне перетворення типу кожного аргументу до типу відповідного параметра.

Кількість аргументів повинна відповідати кількості параметрів. На імена ніяких обмежень не накладаються: імена аргументів можуть як збігатися, так і не збігатися з іменами параметрів.

Існують два способи передачі параметрів: за значенням і по посиланню.

При передачі за значенням метод отримує копії значень аргументів, і оператори методу працюють з цими копіями. Доступу до початкових значень аргументів у методу немає, а отже, немає і можливості їх змінити.

При передачі по посиланню (за адресою) метод отримує копії адреси аргументів, він здійснює доступ до елементів пам'яті за цими адресами і може змінювати початкові значення аргументів, модифікуючи параметри.

Для обміну даними між функціями, що викликаються, передбачено чотири параметри:

- параметри-значення;
- параметри-посилання - описуються за допомогою ключового слова ref;
- вихідні параметри - описуються за допомогою ключового слова out;
- параметри-масиви - описуються за допомогою ключового слова params.

Ключове слово стоїть перед описом типу параметра. Якщо воно відсутнє, параметр вважається за параметр-значення. Параметр-масив може бути тільки один і повинен знаходитися останнім в списку, наприклад:

*public int Calculate(  int a, ref int b,  out int c,  params int[] d  ) ...*

Параметри-масиви будуть розглянуті в розділі 7, а зараз розглянемо решту типів параметрів.

### 5.3.2. Параметри-значення

*Параметр-значення* описується в заголовку методу таким чином:

**тип ім’я**

Приклад заголовка методу, що має один параметр-значення цілого типу:

void P( int x )

Ім'я параметра може бути довільним. Параметр *x* є локальною змінною, яка набуває свого значення із функції при виклику методу. У метод передається копія значення аргументу.

Механізм передачі наступний: з елементу пам'яті, в якій зберігається змінна,  береться її значення і копіюється в спеціальну область пам'яті ( область параметрів). Метод працює з цією копією, отже, доступу до осередку, де зберігається сама змінна, не має. Після закінчення роботи методу область параметрів звільняється. Таким чином, для параметрів-значень використовується, як ви здогадалися, передача за значенням. Зрозуміло, що цей спосіб годиться тільки для величин, які не повинні змінитися після виконання методу, тобто для його початкових даних.

При виклику методу на місці параметра, переданого за значенням, може знаходитися вираз, а також, звичайно, його окремі випадки - змінна або константа. Повинне існувати неявне перетворення типу виразу до типу параметра.

Наприклад, нехай в функції описані змінні і їм до виклику методу привласнені значення:

```
int      x = 1;
sbyte    c = 1;
ushort   y = 1;
```

Тоді наступні виклики методу P будуть синтаксично правильними:
P( x );  P( c );  P( y );  P( 200 );  P( x / 4 + 1 );

### 5.3.3. Параметри-посилання

Якщо метод повинен повертати більш за одну величину потрібно змінити значення яких-небудь переданих в нього величин. У цих випадках використовуються параметри-посилання.

Ознакою параметра-посилання є ключове слово *ref* :

*ref тип ім'я*

Приклад заголовка методу, що має один параметр-посилання цілого типу:
void  P(  ref int x )

При виклику методу в область параметрів копіюється не значення аргументу, а його адреса, і метод через нього має доступ до осередку, в якому зберігається аргумент. Таким чином, параметри-посилання передаються за адресою (частіше вживається термін «передача по посиланню»). Метод працює безпосередньо із змінною функції і, отже, може її змінити, тому якщо в методі потрібно змінити значення параметрів, вони повинні передаватися тільки по посиланню.

Проілюструємо передачу параметрів-значень і параметрів-посилань на прикладі лістингу 5.4.

Лістинг 5.4. Параметри - значення і параметри - посилання

```
using System;
namespace ConsoleApplicationl
{
    class Class1
    {
        static void P(  int a,  ref int b )
        {
          a = 44; b = 33;
          Console.WriteLine(  "усередині методу {0}  {1}",  a,  b );
        }

        static void Main( )
        {
          int a = 2,  b = 4;
          Console.WriteLine(  "до виклику{0}  {1}",  a,  b );
          P( a, ref  b );
          Console.WriteLine(  "після виклику  {0}  {1}",  a,  b );
        }
    }
}
```

Результат роботи цієї програми:
до виклику 2 4
усередині методу 44 33
після виклику 2 33

Як бачите, значення змінної *a* у функції *Main* не змінилося, оскільки змінна передавалася за значенням, а значення змінної *b* змінилося тому, що вона була передана по посиланню. Декілька інша картина вийде, якщо передавати в метод не величини значущих типів, а екземпляри класів, тобто величини посилальних типів. Як ви пам'ятаєте, змінна-об'єкт насправді зберігає посилання на дані, розташовані в динамічній пам'яті, і саме це посилання передається в метод або за адресою, або за значенням. У обох випадках метод отримує в своє розпорядження фактичну адресу даних і, отже, може їх змінити.

Для простоти можна вважати, що об'єкти завжди передаються по посиланню. Різниця між передачею об'єктів за значенням і по посиланню полягає в тому, що в останньому випадку можна змінити саме посилання, тобто після виклику методу воно може указувати на інший об'єкт.

### 5.3.4. Вихідні параметри

Досить часто виникає необхідність в методах, які формують декілька величин, наприклад, якщо в методі створюються об'єкти або ініціалізуються ресурси. В цьому випадку стає незручним обмеження параметрів-посилань: необхідність надання значення аргументу до виклику методу. Це обмеження знімає специфікатор *out*. Параметру, що має цей специфікатор, має бути обов'язково привласнене значення усередині методу, компілятор за цим стежить. Зате в коді можна обмежитися описом змінною без ініціалізації.

Змінимо опис другого параметра в лістингу 5.4 так, щоб він став вихідним (лістинг 5.5).

При виклику методу перед відповідним параметром теж указується ключове слово *out.*

Лістинг 5.5. Вихідні параметри

```
using System;
namespace ConsoleApplicationl
{
    class Class1
    {
        static void P(   int a,   out int b )
        {
          a = 44; b = 33;
          Console.WriteLine(   "усередині  методу  {0}   {1}",   a,   b );
        }

        static void Main( )
        {
```

```
            int a = 2,  b;
            P( a, out  b );
            Console.WriteLine(   "після виклику  {0}   {1}",   a,   b );
        }
    }
}
```

Результат роботи цієї програми:

усередині методу  44  33
після виклику  2  33

Рекомендується, в списку параметрів записуйте спочатку всі вхідні параметри, потім всі посилання і вихідні параметри. Давайте параметрам імена, за якими можна отримати уявлення про їх призначення.

## 5.4. Ключове слово this

Кожен об'єкт містить свій екземпляр полів класу. Методи знаходяться в пам'яті в єдиному екземплярі і використовуються всіма об'єктами спільно, тому необхідно забезпечити роботу методів нестатичних екземплярів з полями саме того об'єкту, для якого вони були викликані. Для цього в будь-який нестатичний метод автоматично передається прихований параметр *this*, в якому зберігається посилання на екземпляр, що викликав функцію.

У явному вигляді параметр *this* застосовується для того, щоб повернути з методу посилання на об'єкт, що викликав, а також для ідентифікації поля у випадку, якщо його ім'я збігається з ім'ям параметра методу, наприклад:

```
class Demo
{
    double y;
    public Demo T( )                        // метод повертає посилання на екземпляр
    {
        return this;
    }
    public void Sety ( double y )
    {
        this.y = y;                         // полю y привласнюється значення параметра
    }
}
```

## 5.5. Конструктори

Конструктор призначений для ініціалізації об'єкту. Він викликається автоматично при створенні об'єкту класу за допомогою операції new. Ім'я конструктора збігається з ім'ям класу. Нижче перераховані властивості конструкторів:

- Конструктор не повертає значення, навіть типу void.
- Клас може мати декілька конструкторів з різними параметрами для різних видів ініціалізації.
- Якщо програміст не вказав жодного конструктора або якісь поля не

ініціалізовані, полям значущих типів привласнюється нуль, полям посилальних типів - значення null.

- Конструктор, що викликається без параметрів, називається конструктором за умовчанням.

До цих пір ми задавали початкові значення полів класу при описі класу (лістинг 5.1). Це зручно у тому випадку, коли для всіх екземплярів класу початкові значення деякого поля однакові. Якщо ж при створенні об'єктів потрібно привласнювати полю різні значення, це слід робити в конструкторі. У лістингу 5.6 в клас *Demo* доданий конструктор, а поля зроблені закритими (непотрібні в даний момент елементи відсутні). У програмі створюються два об'єкти з різними значеннями полів.

Лістинг 5.6. Клас з конструктором

```
using System;
namespace ConsoleApplicationl
{
    class Demo
    {
        public int a = 1 ;
        public const double c = 1.66;
        double y;

        public Demo(  int a,   double y )       // конструктор з параметрами
        {
          this.a = a;
          this.y = y;
        }

        public double Gety( )                          // метод отримання поля y
                {
          return y;
        }


    }

    class Classl
    {
        static void Main( )
        {
          Demo a = new Demo( 300,  0.002);      //  виклик конструктора
          Console.WriteLine (a.Gety( ));        // результат: 0,002
          Demo b = new Demo(1,5.71 );           //  виклик конструктора
          Console.WriteLine( b.Gety( ) );       // результат: 5,71
        }
    }
}
```

Часто буває зручно задати в класі декілька конструкторів, щоб забезпечити можливість ініціалізації об'єктів різними способами. Приклад:

```
class Demo
{
      public Demo(  int a  )                                    // конструктор 1
      {
        this.a = a;
        this.y = 0.002;
      }

      public Demo( double y )                                   // конструктор 2
      {
        this.a = 1;
        this.y = y;
      }
…
}
…
        Demo x = new Demo( 300  );                          // виклик конструктора 1
        Demo y = new Demo( 5.71  );                         // виклик конструктора 2
```

Всі конструктори повинні мати різні сигнатури.

Якщо один з конструкторів виконує які-небудь дії, а інший повинен робити те ж саме плюс ще що-небудь, зручно викликати перший конструктор з другого. Для цього використовується вже відоме вам ключове слово *this* в іншому контексті, наприклад:

```
class Demo
{
      public Demo( int a  )       // конструктор 1
      {
          this.a = a;
      }
      public Demo(  int a,  double y ) : this(a)          // виклик конструктора 1
      {
          this.y = y;
      }
      …
}
```

Конструкція, що знаходиться після двокрапки, називається ініціалізатором, тобто тим кодом, який виконується до початку виконання тіла конструктора.

Як ви пам'ятаєте, всі класи в C# мають загального предка - клас object. Конструктор будь-якого класу, якщо не вказаний ініціалізатор, автоматично викликає конструктор свого предка. Це можна зробити і явним чином за допомогою ключового слова base, що позначає конструктор базового класу. Таким чином, перший конструктор з попереднього прикладу можна записати і так:

```
public Demo( int a  ) :  base( )                    //  конструктор 1
{
    this.a = a;
}
```

Конструктор базового класу викликається явним чином в тих випадках, коли йому потрібно передати параметри.

До цих пір мова йшла про “звичайні ” конструктори, або конструктори екземпляра. Існує другий тип конструкторів - статичні конструктори, або конструктори класу. Конструктор екземпляра ініціалізував дані екземпляра, конструктор класу - дані класу.

У класі, що складається тільки із статичних елементів (полів і констант), описувати статичний конструктор не обов'язково, початкові значення полів зручніше задати при їх описі.

В лістингу 5.7 приведений приклад статичного класу.

Лістинг 5.7. Статичний клас

```
using System;
namespace  ConsoleApplicationl
{
      static class D
      {
          static int a = 200;
          static double b = 0.002;

          public   static  void  Print ( )
          {
            Console.WriteLine("a="+a);
            Console.WriteLine("b="+b);
          }
      }

      class Classl
      {
          static void Main( )
          {
            D.Print( );
          }
      }
}
```

Як приклад, на якому демонструватиметься робота різними елементами класу, створимо клас *Monster*, моделюючий персонаж комп'ютерної гри. Для цього потрібно задати його властивості (наприклад, кількість щупальців, силу або наявність гранатомета) і поведінку. Опис класу *Monster* приводиться на лістингу 5.8.

Лістинг 5.8. Клас Monster

```
using System;
namespace ConsoleApplication1
{
    class Monster
    {
        public Monster( )
        {
          this.name     = "Noname";
          this.health   = 100;
          this.ammo     = 100;
        }

        public  Monster(string name) : this( )
        {
          this.name = name;
        }

        public Monster(  int health, int ammo, string name )
        {
          this.name  = name;
          this.health = health;
          this.ammo = ammo;
        }

        public string GetName( )
        {
          return name;
        }

        public  int GetHealth( )
        {
          return health;
        }

        public int GetAmmo( )
        {
          return ammo;
        }

        public   void Passport( )
        {
          Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
           name,  health,  ammo );
        }
        string name;              // закриті поля
        int health, ammo;
    }

    class Classl
    {
        static void Main()
```

```
        {
            Monster X = new Monster( );
            X.Passport( );
            Monster Vasia = new Monster("Vasia");
            Vasia.Passport( );
            Monster Masha = new Monster(200, 200, "Masha" );
            Masha.Passport( );
        }
    }
}
```

Результат роботи програми:

| | |
|---|---|
| Monster Noname | health = 100 ammo = 100 |
| Monster Vasia | health = 100 ammo = 100 |
| Monster Masha | health = 200 ammo = 200 |

У класі три закриті поля (name, health і ammo), чотири методи (Getname, Gethealth, Getammo і Passport) і три конструктори.

## 5.6. Властивості

Властивості служать для організації доступу до полів класу. Як правило, властивість пов'язана із закритим полем класу і визначає методи його отримання і установки. Синтаксис властивості:

**[ атрибути ] [ специфікатори ] тип ім’я_властивості**
{
**[ get код_доступа ]**
**[ set код_доступа ]**
}

Значення специфікаторів для властивостей і методів аналогічні. Частіше за всю властивість оголошуються як відкриті (із специфікатором public), оскільки вони входять в інтерфейс об'єкту.

*Код доступу* є блоками операторів, які виконуються при отриманні (get) або установці (*set*) властивості. Може бути відсутньою або частина *get*, або *set*, але не обидві одночасно.

Якщо відсутня частина *set*, властивість доступна тільки для читання (*read-only*), якщо відсутня частина *get*, властивість доступна тільки для запису (*write-only*).

Введена зручна можливість задавати різні рівні доступу для частин *get* і *set*. Наприклад, в багатьох класах виникає потреба забезпечити необмежений доступ для читання і обмежений - для запису.

Специфікатори доступу для окремої частини повинні задавати або такий же, або більш обмежений доступ, ніж специфікатор доступу для властивості в цілому. Наприклад, якщо властивість описана як *public,* її частини можуть мати будь-який специфікатор доступу, а якщо властивість має доступ *protected*

*internal*, її частини можуть оголошуватися як *internal*, *protected* або *private*. Синтаксис властивості має вигляд:

**[ атрибути ]  [ специфікатори ]  тип ім'я_властивості**
{
**[ [ атрибути ] [ специфікатори ] get код_ доступу ]**
**[ [ атрибути ] [ специфікатори ] set код_доступу ]**
}

Приклад опису властивостей:

```
public class Button: Control
{
    private string caption;        //  закрите поле, з яким пов'язана властивість
    public string Caption         // властивість
    {
        get  {                                        // спосіб отримання властивості
               return caption;
             }

        set {                                        // спосіб установки властивості
               if (caption  != value)
               caption = value;
            }
    }
…
}
```

Двокрапка між іменами Button і Control в заголовку класу Button означає, що клас Button є похідним від класу Control.

При зверненні до властивості автоматично викликаються вказані в нім методи читання і установки.

Синтаксично читання і запис властивості виглядають майже як методи. Метод *get* повинен містити оператора *return*, що повертає вираз, для типу якого повинне існувати неявне перетворення до типу властивості. У методі *set* використовується параметр із стандартним ім'ям *value,* який містить встановлюване значення.

Взагалі кажучи, властивість може і не зв'язуватися з полем. Фактично, воно описує один або два методи, які здійснюють деякі дії над даними того ж типу, що і властивість. На відміну від відкритих полів, властивості *забезпечують розділення між внутрішнім станом об'єкту і його інтерфейсом* і, таким чином, спрощують внесення змін до класу.

За допомогою властивостей можна відкласти ініціалізацію поля до того моменту, коли його фактично буде потрібно, наприклад:

```
class A
{
    private static ComplexObject x;    // закрите поле
    public static ComplexObject X;     // властивість
```

```
    {
        get
        {
          if (x = = null)
          x = new ComplexObject(); // створення об’єкта при 1-му зверненні
          return x;
        }
    }
...
}
```

Додамо в клас *Monster*, описаний в лістингу 5.8, властивості, що дозволяють працювати із закритими полями цього класу. Властивість *Name* зробимо доступною тільки для читання, оскільки ім'я об'єкту задається в конструкторі і його зміна не передбачена, у властивостях *Health* і *Ammo* введемо перевірку на додатність встановлюваної величини.

Лістинг 5.9. Клас *Monster* з властивостями

```
using System;
namespace ConsoleApplication1
{
    class Monster
    {
        public Monster( )
        {
          this.name    = "Noname";
          this.health   = 100;
          this.ammo    = 100;
        }

        public  Monster(string name) : this( )
        {
          this.name = name;
        }

        public Monster(  int health, int ammo, string name )
        {
          this.name  = name;
          this.health = health;
          this.ammo = ammo;
        }

        public int Health          // властивість Health пов'язана з полем health
        {
            get
            {
              return health;
            }
            set
            {
              if (value > 0) health = value;
```

```
            else          health = 0;
        }
    }

    public int  Ammo         // властивість Ammo пов'язана з полем ammo
    {
        get
        {
          return ammo;
        }
        set
        {
          if (value > 0) ammo = value;
          else          ammo = 0;
        }
    }

    public string Name       // властивість Name пов'язана з полем name
    {
        get
        {
          return name;
        }
    }
    public   void Passport( )
    {
      Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
       name,  health,  ammo );
    }
    string name;              // закриті поля
    int health, ammo;
}
class Classl
{
    static void Main()
    {
        Monster Masha = new Monster(200,200, "Masha");
        Masha.Passport ( );
        --Masha. Health;
        Masha.Ammo += 100;
        Masha.Passport ( );
    }
}
}
```

Результат роботи програми:

```
Monster Masha       health=200    ammo = 200
Monster Masha       health=199    ammo = 300
```

## 5.7. Рекомендації по програмуванню

При створенні класу, тобто нового типу даних, слід добре продумати його інтерфейс. Інтерфейс добре спроектованого класу інтуїтивно ясний.

Поля переважно робити закритими (*private)*. Це дає можливість згодом змінити реалізацію класу без змін в його інтерфейсі, а також регулювати доступ до полів класу за допомогою набору що надаються користувачеві властивостей і методів. Важливо пам'ятати, що поля класу вводяться тільки для того, щоб реалізувати характеристики класу, представлені в його інтерфейсі за допомогою властивостей і методів.

Не потрібно розширювати інтерфейс класу без необхідності. Збільшення кількості методів утрудняє розуміння класу користувачем. У ідеалі інтерфейс має бути повним, тобто надавати можливість виконувати будь-які розумні дії з класом, і одночасно мінімально необхідним - без дублювання і перетину можливостей методів.

Методи визначають поведінку класу. Кожен метод класу повинен вирішувати тільки одну задачу (не треба об'єднувати два короткі незалежні фрагменти коду в один метод). Розмір методу може варіюватися в широких межах, все залежить від того, які функції він виконує.

Якщо метод реалізує складні дії, слід розбити його на послідовність кроків, і кожен крок оформити у вигляді допоміжної функції. Якщо деякі дії зустрічаються в коді, хоч би двічі, їх також потрібно оформити у вигляді окремої функції.

Переважно, щоб кожна функція обчислювала рівно один результат, проте це не завжди виправдано. Якщо величина обчислюється усередині функції і повертається з неї через список параметрів, необхідно використовувати перед відповідним параметром ключове слово *out*. Якщо параметр значущого типу може змінити свою величину усередині функції, перед ним ставиться ключове слово *ref*. Величини посилального типу завжди передаються за адресою і, отже, можуть змінити усередині функції своє значення.

Необхідно прагнути до максимального скорочення зони дії кожної змінної, тобто до реалізації принципу інкапсуляції. Це спрощує відладку програми. Поля, що характеризують клас в цілому, тобто що мають одне і те ж значення для всіх екземплярів, слід описувати як статичні. Всі літерали, пов'язані з класом (числові і рядкові константи), описуються як поля-константи з іменами, що відображають їх сенс.

# РОЗДІЛ 6. МАСИВИ І РЯДКИ

Елементи масиву мають одне і те ж ім'я, а розрізняються порядковим номером (індексом). Це дозволяє компактно записувати множину операцій за допомогою циклів.

Масив відноситься до посилальних типів даних, тобто розташовується в динамічній області пам'яті, тому створення масиву починається з виділення пам'яті під його елементи. Елементами масиву можуть бути величини як значущих, так і посилальних типів (зокрема масиви). Масив значущих типів зберігає значення, масив посилальних типів - посилання на елементи. Всім елементам при створенні масиву привласнюються значення за умовчанням: нулі для значущих типів і null - для посилальних.

На рис. 6.1 представлений масив, що складається з п'яти елементів будь-якого значущого типу, наприклад int або double, а рис. 6.2 ілюструє організацію масиву з елементів посилального типу.

Ось, наприклад, як виглядають оператори створення масиву з 10 цілих чисел і масиву з 100 рядків:

```
int[]    w = new int [10];
string[] z = new string [100];
```

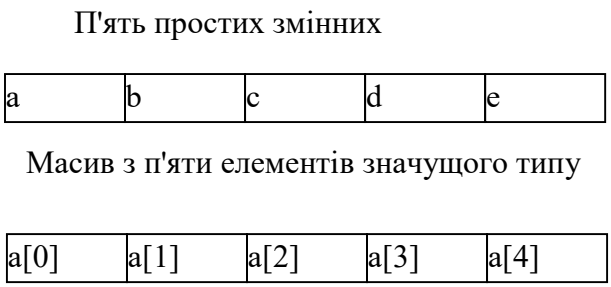



Рис. 6.1. Прості змінні і масив з елементів значущого типу

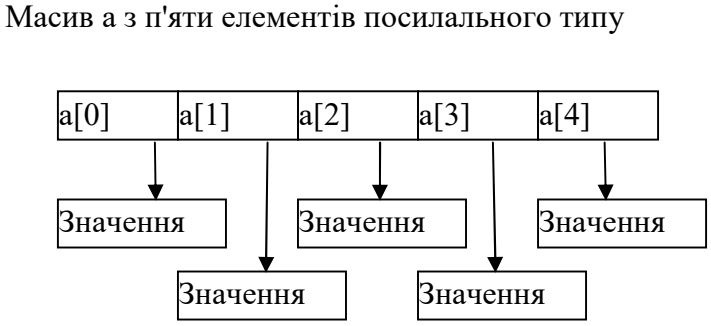



Рис. 6.2. Масив з елементів посилального типу

У першому операторі описаний масив *w* типу int[]. Операція *new* виділяє пам'ять під 10 цілих елементів, і вони заповнюються нулями.

У другому операторі описаний масив *z* типу string[]. Операція *new* виділяє пам'ять під 100 посилань на рядки, і ці посилання заповнюються значенням *null*. Кількість елементів в масиві (розмірність) не є частиною його типу, ця кількість задається при виділенні пам'яті і не може бути змінена згодом. Розмірність може задаватися не тільки константою, але і виразом. Результат обчислення цього виразу має бути додатним, а його тип повинен мати неявне перетворення до int, uint, long або ulong.

Приклад розмірності масиву, заданої виразом:

```
short n = 10;
string[] z = new string[n + 1];
```

Елементи масиву нумеруються з нуля, тому максимальний номер елементу завжди на одиницю менше розмірності (наприклад, в описаному вище масиві w елементи мають індекси від 0 до 9). Для звернення до елементу масиву після імені масиву указується номер елементу в квадратних дужках, наприклад:

```
w[4]    z[i]
```

З елементом масиву можна робити все, що допустимо для змінних того ж типу. При роботі з масивом автоматично виконується контроль виходу за його межі: якщо значення індексу виходить за межі масиву, генерується виключення IndexOutOfRangeException.

Масиви одного типу можна привласнювати один одному. При цьому відбувається привласнення посилань, а не елементів, як і для будь-якого іншого об'єкту посилального типу, наприклад:

```
int[] a = new int[10];
int[] b = a;     // b і a указують на один і той же масив
```

Всі масиви в C# мають загальний базовий клас Array, визначений в просторі імен System. У нім є декілька корисних методів, що спрощують роботу з масивами

У C# існують три різновиди масивів: одновимірні, прямокутні і ступінчасті.

## 6.1. Одновимірні масиви

Одновимірні масиви використовуються в програмах найчастіше. Варіанти опису масиву:

```
тип[]          ім'я;
тип[]          ім'я = new тип [ розмірність ];
тип[]          ім'я = {список_ініціалізаторів };
тип[]          ім'я = новий тип [] { список ініціалізаторів };
тип[]   ім'я = new тип [ розмірність ] { список ініціалізаторів };
```

Приклади описів (один приклад для кожного варіанту опису):

```
int[] a;  // 1     елементів немає
int[] b     new  int[4];                    // 2   елементи дорівнюють 0
int[] c = { 61, 2, 5, -9 };                 // 3   new мається на увазі
int[] d = new int[] { 61, 2, 5, -9};        // 4   розмірність обчислюється
int[] e = new int[4] { 61, 2, 5, -9};       // 5   надмірний опис
```

Тут описано п'ять масивів. Відмінність першого оператора від останніх полягає в тому, що в нім, фактично, описано тільки посилання на масив, а пам'ять під елементи масиву не виділена. Якщо список ініціалізації не заданий, розмірність може бути не тільки константою, але і виразом типу, що приводиться до цілого.

У кожному з решти масивів по чотири елементи цілого типу. Як видно з операторів 3-5, масив при описі можна ініціалізувати. Якщо при цьому не задана розмірність (оператор 3), кількість елементів обчислюється по кількості ініціалізованих значень. Для полів об'єктів і локальних змінних можна не вказувати операцію new, вона буде виконана за умовчанням (оператор 2). Якщо присутня і розмірність, і список ініціалізаторів, розмірність має бути константою (оператор 4).

Якщо кількість ініціалізованих значень не збігається з розмірністю, виникає помилка компіляції.

Як приклад розглянемо програму, яка визначає суму і кількість від’ємних елементів, а також максимальний елемент масиву, що складається з 6 цілочисельних елементів (лістинг 6.1).

Лістинг 6.1 . Робота з одновимірним масивом

```
using System;
namespace ConsoleApplication1
{
    class Classl
    {
        static void Main( )
        {
            const int n = 6;
            int i;
            int[] a = new int[n] {3,  12,  5,  -9,  8,  -4};
            Console.WriteLine("Початковий масив:");
            for (i = 0;  i<n; i ++)
            Console.Write("\t" + a[i ]);
            Console.WriteLine( );

            long sum = 0;              // сума від’ємних елементів
            int num = 0;               // кількість від’ємних елементів
            for (i = 0; i< n; i ++)
                if (a[i] < 0 )
                {
                    sum += a[i];
                    num ++;
                }
```

```
            Console.WriteLine( "Сума від'ємних елементів= " + sum );
            Console.WriteLine("Кількість від'ємних елементів = " + num);

            int max = a[0];       // максимальний елемент
            for (i =1; i < n; i ++)
            if ( a[i ] > max ) max = a[i];
            Console.WriteLine("Максимальний елемент = " + max );
        }
    }
}
```

## 6.2. Прямокутні масиви

Прямокутний масив має більше одного вимірювання. Найчастіше в програмах використовуються двовимірні масиви. Варіанти опису двовимірного масиву:

```
тип[,]  ім'я;
тип[,]  ім'я = new тип [ розм _1, розм_2 ];
тип[,]  ім'я = { список ініціалізаторів };
тип[,]  ім'я = new тип [,]  { список ініціалізаторів };
тип[,]  ім'я = new тип [ розм_1, розм_2 ] {список ініціалізаторів};
```

Приклади описів (один приклад для кожного варіанту опису):

```
int [ , ] a;       //1     елементів немає
int [ , ] b = new int [2,3];   //2     елементи дорівнюють 0
int [ , ] c = {{1, 2, 3},  {4, 5, 6}};        //3     new мається на увазі
int [ , ] d = new int[ , ] {{1, 2, 3}, {4, 5. 6}};  //4     розмірність обчислюється
int [ , ] e = new int[2,3] {{1, 2, 3}, {4, 5, 6}};//5     надмірний опис
```

Якщо список ініціалізації не заданий, розмірності можуть бути не тільки константами, але і виразами типу, що приводиться до цілого. До елементу двовимірного масиву звертаються, указуючи номери рядка і стовпця, на перетині яких він розташований, наприклад:

```
a[1 , 4]        b[i , j ]              b[j, i]
```

Як приклад розглянемо програму на лістингу 6.2, яка для цілочисельної матриці визначає середнє арифметичне її елементів і кількість додатних елементів в кожному рядку.

Лістинг 6.2. Робота з двовимірним масивом

```
using System;
namespace ConsoleApplication1
{
```

```
class Classl
{
    static void Main( )
    {

        const int n = 3, m = 4;
        int i,j;
        int[,] a = new int[n, m]
        {
            { 2,-2, 8, 9},
            {- 4,- 5, 6,- 2},
            { 7, 0, 1, 1}
        };
        Console.WriteLine( "Початковий масив:");
          for (i = 0; i < n; i ++)
          {
            for (j = 0; j < m; j ++)
            Console.Write( "\t" + a[i, j]);
            Console.WriteLine( );
            }

        double sum = 0;
        int pel;

        for (i = 0; i < n; i ++)
        {
        pel = 0;
            for (j = 0; j < m; j ++)
            {
                sum += a[i, j] ;
                if ( a[i, j] > 0 )pel ++;
            }
            Console.WriteLine( "У рядку {0} {1} додат. елементів ", i, pel);
        }
      Console.WriteLine("Серед. ариф. значення елементів = " +sum/m/n);
    }
}
}
```

## 6.3. Ступінчасті масиви

У ступінчастих масивах кількість елементів в різних рядках може розрізнятися. У пам'яті ступінчастий масив зберігається інакше, ніж прямокутний: у вигляді декількох внутрішніх масивів, кожен з яких має свій розмір. Крім того, виділяється окрема область пам'яті для зберігання посилань на кожен з внутрішніх масивів.

Опис ступінчастого масиву:
тип[][] ім’я;

Під кожен з масивів, складових ступінчастий масив, пам'ять потрібно виділяти явним чином, наприклад:

```
int[][] a = new int[3][]; // виділення    пам'яті  під посилання на три рядки
a[0] = new int[5];    // виділення        пам'яті  під 0-й рядок  (5 елементів)
a[1] = new int[3];    // виділення        пам'яті  під 1-й рядок  (3 елементи)
a[2] = new int[4];    // виділення        пам'яті  під 2-й рядок  (4 елементи)
```

Тут a[0],a[1] і a[2] – це окремі масиви, до яких можна звертатися по імені. Інший спосіб виділення пам'яті:

```
int[][ ] a = { new int [5], new int[3], new int[4] };
```

До елементу ступінчастого масиву звертаються, указуючи кожну розмірність в своїх квадратних дужках, наприклад:

```
a[1][2]          a[i][j]          a[j][i]
```

У останньому випадку використання ступінчастих масивів не відрізняється від використання прямокутних. Невирівняні масиви зручно застосовувати, наприклад, для роботи з трикутними матрицями великого об'єму.

## 6.4. Клас System.Array

Раніше вже розповідалося, що всі масиви в C# побудовані на основі базового класу Array, який містить корисні для програміста властивості і методи, частина з яких перераховані в таблиці 6.1.

Таблиця 6.1

Основні елементи класу Array

| Елемент | Вигляд | Опис |
|---|---|---|
| Length | Властивість | Кількість елементів масиву (по всіх размерностям) |
| Rank | Властивість | Кількість размерностей масиву |
| BinarySearch | Статичний метод | Двійковий пошук у відсортованому масиві |
| Clear | Статичний метод | Привласнення елементам масиву значень за умовчанням |
| Copy | Статичний метод | Копіювання заданого діапазону елементів одного масиву в інший масив |
| CopyTo | Метод | Копіювання всіх елементів поточного одновимірного масиву в інший одновимірний масив |
| GetValue | Метод | Набуття значення елементу масиву |
| IndexOf | Статичний метод | Пошук першого входження елементу в одновимірний масив |
| LastIndexOf | Статичний метод | Пошук останнього входження елементу в одновимірний масив |
| Reverse | Статичний метод | Зміна порядку проходження елементів на зворотний |
| SetValue | Метод | Установка значення елементу масиву |
| Sort | Статичний метод | Впорядковування елементів одновимірного масиву |

Властивість *Length* дозволяє реалізовувати алгоритми, які працюватимуть з масивами різної довжини або, наприклад, із ступінчастим масивом. У лістингу 6.3 продемонстровано застосування елементів класу *Array* при роботі з одновимірним масивом.

Лістинг 6.3. Використання методів класу *Array* з одновимірним масивом

```
using System;
namespace ConsoleApplication1
{
    class Classl
    {
        static void Main( )
        {
            int[] a ={24, 50, 18, 3, 16, 7, 9, -1};
            PrintArray(  "Початковий масив:", a );
            Console.WriteLine( Array.IndexOf( a, 18) );
            Array.Sort(a);
            PrintArray( "Впорядкований масив:", a );
            Console.WriteLine(Array.BinarySearch(a, 18) );
        }
        public static void PrintArray( string header, int[] a )
        {
            Console.WriteLine(header);
            for ( int i = 0; i < a.Length; i ++)
                  Console.Write( "\t" + a[i] );
            Console.WriteLine( );
        }
    }
}
```

Методи *Sort, IndexOf* і *BinarySearch* є статичними, тому до них звертаються через ім'я класу, а не екземпляра, і передають в них ім'я масиву. Двійковий пошук можна застосовувати тільки для впорядкованих масивів. Він виконується набагато швидше, ніж лінійний пошук, реалізований в методі *IndexOf*. У лістингу пошук елементу, що має значення 18, виконується обома цими способами.

У класі *Class1* описаний допоміжний статичний метод *PrintArray*, призначений для виведення масиву на екран. У нього передаються два параметри: рядок заголовка header і масив. Кількість елементів масиву визначається усередині методу за допомогою властивості *Length*. Цей метод можна використовувати для виведення будь-якого цілочисельного одновимірного масиву.

Результат роботи програми:

```
Початковий масив:
24      50      18      3       16      -7      9       -1
2

Впорядкований масив:
-7      -1      3       9       16      18      24      50
5
```

Для того, щоб застосовувати метод PrintArray до масивів, що складаються з елементів іншого типу, можна описати його другий параметр як Array. Правда, при цьому значення елементу масиву доведеться набувати за допомогою методу GetValue, оскільки доступ по індексу для класу Array не передбачений. Узагальнений метод виведення масиву виглядає так:

```
public static void PrintArray(string header, Array a )
{
    Console.WriteLine( header );
    for ( int i = 0; i < a.Length; i ++)
        Console.Write( "\t" + a.GetValue(i) );
    Console.WriteLine( );
}
```

У лістингу 6.4 продемонстровано застосування елементів класу Array при роботі із ступінчастим масивом.

Лістинг 6.4. Використання методів класу *Array* зі ступінчастим масивом

```
using System;
namespace ConsoleApplication1
{
    class Classl
    {
        static void Main( )
        {
            int[][] a = new int[3][];
            a[0] = new int [5] {24, 50, 18, 3, 16};
            a[1] = new int [3]  {7, 9,-1};
            a[2] = new int [4] {6, 15, 3, 1 };
            Console.WriteLine("Початковий масив:");

            for ( int i = 0; i < a.Length; i ++)
            {
                for ( int j = 0; j < a[i].Length; j ++)
                    Console.Write("\t"  + a[i][j]);
                Console.WriteLine();
            }
            Console.WriteLine(Array.IndexOf(a[0], 18));
        }
    }
}
```

Зверніть увагу на те, як усередині циклу по рядках визнається довжина кожного масиву. Результат роботи програми:

```
Початковий масив:
    24      50      18      3       16
    7       9       -1
    6       15      3       1
2
```

## 6.5. Клас Random

При відлагодженні програм, що використовують масиви, зручно мати можливість генерувати початкові дані, задані випадковим чином. У бібліотеці C# на цей випадок є клас Random, визначений в просторі імен System.

Для отримання псевдовипадкової послідовності чисел необхідно спочатку створити екземпляр класу за допомогою конструктора, наприклад:

```
Random a = new Random( );    // 1
Random b = new Random (1);  // 2
```

Є два види конструктора: конструктор без параметрів (оператор 1) використовує початкове значення генератора, обчислене на основі поточного часу. В цьому випадку кожного разу створюється унікальна послідовність. Конструктор з параметром типу int (оператор 2) задає початкове значення генератора, що забезпечує можливість отримання однакових послідовностей чисел.

Для набуття чергового значення серії користуються методами, перерахованими в таблиці 6.2.

Таблиця 6.2

Основні методи класу System.Random

| Назва | Опис |
|---|---|
| Next ( ) | Повертає ціле додатне число у всьому додатному діапазоні типу int |
| Next (макс) | Повертає ціле додатне число в діапазоні [0, макс] |
| Next (мин, макс) | Повертає ціле додатне число в діапазоні [мін, макс] |
| NextBytes (массив) | Повертає масив чисел в діапазоні [0, 255] |
| NextDouble ( ) | Повертає дійсне додатне число в діапазоні [0,1] |

Приклад застосування методів приведений в лістингу 6.5.

Лістинг 6.5. Робота з генератором псевдовипадкових чисел

```
using System;
namespace ConsoleApplication1
{
   class Classl
   {
      static void Main()
      {
         Random a = new Random();
         Random b = new Random(1);
         const int n = 10;
         Console.WriteLine("\n Діапазон [0,1] : ");
         for (int i = 0; i < n; i ++)
         Console.Write(" {0,6:0.##} ", a.NextDouble());
         Console.WriteLine();
         Console.WriteLine("Діапазон [0,1000]: \n");
         for (int i = 0; i < n; i ++)
```

```
            Console.Write("  " + b.Next(1000));
        Console.WriteLine("\n Діапазон [-10, 10 ]:");
        for (int i = 0; i < n; i ++)
            Console.Write(" " + a.Next(-10, 10));
        Console.WriteLine("\n Масив [0, 255]:");
        byte[] mas = new byte[n];
        a.NextBytes(mas);
        for (int i = 0; i < n; i ++)
        Console.Write(" " + mas[i]);
        Console.WriteLine();
        Console.ReadLine();
      }
    }
}
```

Результат роботи програми:
Діапазон [0, 1]:
0,02 0,40 0,24 0,55 0,92 0,84 0,90 0,78 0,78 0,74
Діапазон [0, 1000]:
248 110 467 771 657 432 354 943 101 642
Діапазон [-10, 10]:
-8 9 -6 -10 7 4 9 -5 -2 -1
Масив [0, 255]:
181 105 60 50 70 77 9 28 133 150

Більш ускладнений приклад роботи з масивом приведений в лістингу 6.6 та 6.7.

Лістинг 6.6. Сортування масиву

```
using System;

namespace examp8
{
  class Program
  {
    static void Main(string[] args)
    {
      // Кількість елементів
      uint n = 0;
      Console.Write("Введіть кількість елементів масиву: ");
      try
      {
        // Вводимо кількість елементів з клавіатури
        n = Convert.ToUInt32(Console.ReadLine());
      }
      catch (OverflowException ex)
      {
        // У разі помилкового введення (переповнювання)
        Console.WriteLine(ex.Message + " Use default size (=10)");
      }
```

```
        catch (FormatException ex)
        {
         // У разі помилкового введення
        Console.WriteLine(ex.Message+"Use default size (=10)");
        }
        // Якщо все погано, то створимо масив з 10 елементів
        if (n == 0)
           n = 10;
        // Створення масиву
        int[] ar = new int[n];
        // Ініціалізація генератора випадкових чисел
        Random rand = new Random();
        for (int i = 0; i < n; i++)
        {
           // Заповнюємо масив випадковими числами
           // от -500 до 500
           ar[i] = rand.Next(-500, 500);
           Console.WriteLine("ar[{0}] = {1,8}", i, ar[i]);
        }
        Console.WriteLine();
        for (int i = 0; i < n - 1; i++)
           for (int j = i + 1; j < n; j++)
           {
              if (ar[i] > ar[j])
              {
                 int t = ar[i];
                 ar[i] = ar[j];
                 ar[j] = t;
              }
           }
        Console.WriteLine("Масив після сортування:");
        for (int i = 0; i < n; i++)
           Console.WriteLine("ar[{0}] = {1,8}", i, ar[i]);
        Console.WriteLine();
        Console.Read();
     }
  }
}
```

Лістинг 6.7. Сортування матриці по стовпцях в двовимірному масиві

```
using System;

namespace examp9
{
   class Program
   {
      // Сортування матриці по стовпцях
      static int[,] Sort_Column(int[,] mtr)
      {
         // число строк
         int M = mtr.GetLength(0);
         // число стовпців
```

```
    int N = mtr.GetLength(1);

    // Створення масиву
    int[,] t_ar = new int[M, N];
    int[,] s_ar = new int[N, 2];

    Console.WriteLine("Сума елементів по стовпцях:");
    //Находим сумму
    for (int i = 0; i < N; i++)
    {
        s_ar[i, 0] = 0;
        s_ar[i, 1] = i;
        for (int j = 0; j < M; j++)
        {
            s_ar[i, 0] += mtr[j, i];
        }
        Console.Write("{0,8}", s_ar[i, 0]);
    }
    Console.WriteLine();

    for (int i = 0; i < N - 1; i++)
        for (int j = i + 1; j < N; j++)
            if (s_ar[i, 0] > s_ar[j, 0])
            {
                int t = s_ar[i, 0];
                s_ar[i, 0] = s_ar[j, 0];
                s_ar[j, 0] = t;
                t = s_ar[i, 1];
                s_ar[i, 1] = s_ar[j, 1];
                s_ar[j, 1] = t;
            }

    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            t_ar[i, j] = mtr[i, s_ar[j, 1]];
    return t_ar;
}

// Виведення матриці
static void Print(int[,] mtr)
{
    // число строк
    int M = mtr.GetLength(0);
    // число стовпців
    int N = mtr.GetLength(1);

    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {
            Console.Write("{0,8}", mtr[i, j]);
        }
```

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

    // Ініціалізація матриці випадковими значеннями
    static void Init(int[,] mtr)
    {
        // Ініціалізація генератора випадкових чисел
        Random rand = new Random();

        // Визначення розмірності матриці
        int M = mtr.GetLength(0);
        int N = mtr.GetLength(1);

        for (int i = 0; i < M; i++)
        {
            for (int j = 0; j < N; j++)
            {
                // Забиваємо масив випадковими числами
                // от -500 до 500
                mtr[i, j] = rand.Next(-500, 500);
            }
        }
    }

    static void Main(string[] args)
    {
        // Розмірності матриці
        uint M = 0, N = 0;

        Console.WriteLine("Введіть розмірності матриці: ");
        try
        {
            // Вводимо розмірності матриці з клавіатури
            Console.Write("Строк: ");
            M = Convert.ToUInt32(Console.ReadLine());
            Console.Write("Столбцов: ");
            N = Convert.ToUInt32(Console.ReadLine());
        }
        catch (OverflowException ex)
        {
            // У разі помилкового введення (переповнювання)
            Console.WriteLine(ex.Message + " Use default size (=10x10)");
        }
        catch (FormatException ex)
        {
            // У разі помилкового введення
            Console.WriteLine(ex.Message + " Use default size (=10x10)");
        }

        // Якщо все погано, то створимо масив з 10x10 елементів
```

```
            if (M == 0 || N == 0)
               M = N = 10;

            // Створення масиву
            int[,] ar = new int[M, N];

            Console.WriteLine("Матрица:");
            // Ініціалізація
            Init(ar);
            // Виведення
            Print(ar);

            // Сортування
            int[,] t_ar = Sort_Column(ar);

            Console.WriteLine();
            Console.WriteLine("Відсортована матриця:");
            // Виведення матриці
            Print(t_ar);

        }
    }
}
```

## 6.6. Оператор foreach

Оператор *foreach* застосовується для перебору елементів в організованій групі даних. Масив є саме такою групою. Зручність цього циклу полягає в тому, що не потрібно визначати кількість елементів в групі і виконувати їх перебір по індексу. Синтаксис оператора:

foreach (тип ім'я in вираз ) тіло_циклу

Ім'я задає локальну по відношенню до циклу змінну, яка по черзі набуватиме всіх значень з масиву. В якості *виразу* найчастіше застосовується ім'я масиву. У простому або складеному операторові, що є тілом циклу, виконуються дії із змінною циклу. Тип змінної повинен відповідати типу елементу масиву.

Наприклад, заданий масив:
int[] a = {24, 50, 18, 3, 16, -7, 9, -1 };

Виведення цього масиву на екран за допомогою оператора foreach виглядає таким чином:
foreach (int x in a ) Console.WriteLine( x );

У лістингу 6.8 вирішується те ж завдання, що і в лістингу 6.1, але з використанням циклу foreach. Зверніть увагу на те, наскільки зрозуміліше стала програма.

Лістинг 6.8. Робота з одновимірним масивом з використанням циклу foreach

```
using System;
namespace ConsoleApplication1
{
    class Classl
    {
        static void Main( )
        {
            const int n = 6;
            int[] a = new int[n] {3,  12,  5,  -9,  8,  -4};
            Console.WriteLine("Исходный  массив:");

            foreach ( int elem in a )
            Console.Write( "\t" + elem );
            Console.WriteLine( );

            long sum = 0;               // сума від'ємних елементів
            int num = 0;                // кількість від'ємних елементів
            foreach (int elem in a )
              if ( elem < 0 )
              {
              sum += elem;
              num ++;
              }

            Console.WriteLine( "Сума від'ємних елементів= " + sum );
            Console.WriteLine("Кількість від'ємних елементів = " + num);
            int max = a[0];       // максимальний елемент
            foreach ( int elem in a )
                  if ( elem > max ) max = elem;
            Console.WriteLine("Максимальний елемент = " + max );
        }
    }
}
```

Обмеженням оператора foreach є те, що з його допомогою можна тільки переглядати значення в групі даних, але не змінювати їх.

## 6.7. Масиви об'єктів

При створенні масиву, що складається з елементів посилального типу, пам'ять виділяється тільки під посилання на елементи, а самі елементи необхідно розмістити в хіпові явним чином. Як приклад створимо масив з об'єктів деякого класу Monster:

```
using System;
namespace ConsoleApplication1
{
   class Monster
   {

      string name; // закриті поля
      int health, ammo;

      public Monster()
      {
         this.health = 100;
         this.ammo = 100;
         this.name = "Noname";
      }
      public Monster(string name)
         : this()
      {
         this.name = name;
      }
      public Monster(int health, int ammo, string name)
      {
         this.health = health;
         this.ammo = ammo;
         this.name = name;
      }
      public int Health
      {
         get
         {
            return health;
         }
         set
         {
            if (value > 0) health = value;
            else health = 0;
         }
      }
      public int Ammo
      {
         get
         {
            return ammo;
         }
         set
         {
            if (value > 0) ammo = value;
            else
               ammo = 0;
         }
      }
      public string Name
```

```
        {
            get
            {
                return name;
            }
        }
        public void Passport()
        {
            Console.WriteLine("Monster {0} \t health = {1} ammo = {2}",
             name, health, ammo);
        }
    }
    class Classl
    {
        static void Main()
        {
            Random rnd = new Random();
            const int n = 5;
            Monster[] stado = new Monster[n]; // 1
            for ( int i = 0; i < n; i ++) // 2
            {
             stado[i ] = new Monster( rnd.Next( 1, 100 ),
             rnd.Next( 1, 200 ) , "Crazy" + i .ToString() );
            }
            foreach ( Monster x in stado ) x.Passport(); // 3
        }
    }
}
```

Результат роботи програми:

```
Monster     Crazy0     health =   18     ammo =   94
Monster     Crazyl     health =   85     ammo =   75
Monster     Crazy2     health =   13     ammo =   6
Monster     Crazy3     health =   51     ammo =   104
Monster     Crazy4     health =   68     ammo =   114
```

У операторові 1 виділяється п'ять елементів пам'яті під посилання на екземпляри класу *Monster*. Ці посилання заповнюються значенням *null*. У циклі 2 створюються п'ять об'єктів. Цикл 3 демонструє зручність застосування оператора *foreach* для роботи з масивом.

## 6.8. Символи і рядки

Обробка текстової інформації є, ймовірно, одним з найпоширеніших завдань в сучасному програмуванні, і C# надає для його вирішення широкий набір засобів: окремі символи, масиви символів, змінні і незмінні рядки, а також регулярні вирази.

### 6.8.1. Символи

Символьний тип char призначений для зберігання символів в кодуванні Unicode. Символьний тип відноситься до вбудованих типів даних C# і відповідає стандартному класу Char бібліотеки .NET з простору імен System. У цьому класі визначені статичні методи, які дозволяють задати вигляд і категорію символу, а також перетворити символ у верхній, нижній регістр, або в число. Основні методи приведені в таблиці 6.3.

Таблиця 6.3

Основні методи класу System.Char

| Метод | Опис |
|---|---|
| GetNumericValue | Повертає числове значення символу, якщо він є цифрою, інакше -1 |
| GetUnicodeCategory | Повертає категорію Unicode-символ |
| IsControl | Повертає true, якщо символ є таким, що управляє |
| IsDigit | Повертає true, якщо символ є десятковою цифрою |
| IsLetter | Повертає true, якщо символ є буквою |
| IsLetterOrDigit | Повертає true, якщо символ є буквою або цифрою |
| IsLower | Повертає true, якщо символ в нижньому регістрі |
| IsNumber | Повертає true, якщо символ є числом |
| IsPunctuation | Повертає true, якщо символ є розділовим знаком |
| IsSeparator | Повертає true, якщо символ є роздільником |
| IsUpper | Повертає true, якщо символ у верхньому регістрі |
| IsWhiteSpace | Повертає true, якщо символ є пробільним (пропуск, переведення рядка та повертання каретки) |
| Parse | Перетворить рядок в символ (рядок повинен складатися з одного символу) |
| ToLower | Перетворить символ в нижній регістр |
| ToUpper | Перетворить символ у верхній регістр |
| MaxValue, MinValue | Повертають символи з максимальним і мінімальними кодами (ці символи не мають видимого представлення) |

У лістингу 6.9 продемонстровано використання цих методів.

Лістинг 6.9. Використання методів класу System.Char

```
using System;
namespace exam35
{
   class Classl
   {
      static void Main()
      {
         try
         {
            char a, b = 'B', c = '\x63', d = '\u0032';                    // 1
            Console.WriteLine(" {0} {1} {2}", b, c, d);
```

```
            Console.WriteLine(" {0}  {1}  {2}",
        char.ToLower(b), char.ToUpper(c), char.GetNumericValue(d));
            do                                                                    // 2
            {
               Console.Write("Введіть символ: ");
               a = char.Parse(Console.ReadLine());
               Console.WriteLine("Введений символ {0}, його код - {1} ", a, (int)a);
               if (char.IsLetter(a))Console.WriteLine("Буква");
               if (char.IsUpper(a))Console.WriteLine("Верхній рег.");
               if (char.IsLower(a))Console.WriteLine("Нижній рег.");
               if (char.IsControl(a))Console.WriteLine("Керуючий");
               if (char.IsNumber(a))Console.WriteLine("Число");
               if (char.IsPunctuation(a)) Console.WriteLine("Роздільник");
            } while (a != 'q');
         }

         catch (Exception a)
         {
            Console.WriteLine(a.Message);
            Console.WriteLine("Виникло виключення");
            return;
         }
      }
   }
}
```

У операторові 1 описані три символьних змінних. Вони ініціалізувалися символьними літералами в різних формах представлення. Далі виконуються виведення і перетворення символів.

У циклі 2 аналізується символ, що вводиться з клавіатури. Можна вводити і символи, що управляють, використовуючи поєднання клавіші Ctrl з латинськими буквами. При введенні використаний метод Parse, що перетворює рядок, який повинен містити єдиний символ, в символ типу char. Оскільки вводиться рядок, введення кожного символу слід завершувати натисненням клавіші Enter. Цикл виконується, поки користувач не введе символ q.

Виведення символу супроводжується його кодом в десятковому вигляді. Для виведення коду використовується явне перетворення до цілого типу. Явне перетворення з символів в рядки і назад в C# не існує, неявним же чином будь-який об'єкт, у тому числі і символ, може бути перетворений в рядок, наприклад:

```
string s =  'к' +   'і' +   'т';   // результат - рядок "кіт"
```

При введенні і перетворенні можуть виникати виняткові ситуації, наприклад, якщо користувач введе порожній рядок. Для “м’якого” завершення програми передбачена обробка виключень.

### 6.8.2. Масиви символів

Масив символів, як і масив будь-якого іншого типу, побудований на основі базового класу Array, деякі властивості і методи якого були перераховані в таблиці 6.1. Застосування цих методів дозволяє ефективно вирішувати деякі завдання. Простий приклад приведений в лістингу 6.10.

Лістинг 6.10. Робота з масивом символів

```
using System;
namespace ConsoleApplicationi
{
   class Classl
   {
      static void Main()
      {
         char[] a = { 'm', 'a', 's', 's', 'i', 'v' };                    //1
         char[] b = "Привіт я програміст".ToCharArray();                 //2
         PrintArray("Початковий масив a:", a);
         int pos = Array.IndexOf(a, 'm');
         a[pos] = 'M';
         PrintArray("Змінений масив a:", a);
         PrintArray("Початковий масив b:", b);
         Array.Reverse(b);
         PrintArray("Змінений масив b:", b);
      }
      public static void PrintArray(string header, Array a)
      {
         Console.WriteLine(header);
         foreach (object x in a) Console.Write(x);
         Console.WriteLine("\n");
      }
   }
}
```

Результат роботи програми:
Початковий масив a:
massiv
Змінений массив a:
Massiv
Початковий массив b:
Привіт я програміст
Змінений масив b:
тсімаргорп я тівирП

Символьний масив можна ініціалізувати, або безпосередньо задаючи його елементи (оператор 1), або застосовуючи метод ToCharArray класу string, який розбиває початковий рядок на окремі символи (оператор 2).

### 6.8.3. Рядки типу string

Тип string, призначений для роботи з рядками символів в кодуванні Unicode, є вбудованим типом C#. Йому відповідає базовий клас System.String бібліотеки .NET.

Створити рядок можна декількома способами:

```
string s;          // ініціалізація відкладена
string t =  "qqq";      // ініціалізація рядковим літералом
string u = new string(‘ ’,20);       // конструктор створює рядок з 20 пропусків
char[] a = {'0','0','0'};          // масив для ініціалізації рядка
string v = new string( a );  // створення з масиву символів
```

Для рядків визначені наступні операції:

привласнення ( = );
перевірка на рівність ( = = );
перевірка на нерівність (! = ) ;
звернення по індексу ([]) ;
зчеплення(конкатенація) рядків ( + ).

Не дивлячись на те що рядки є посилальним типом даних, на рівність і нерівність перевіряються не посилання, а значення рядків. Рядки рівні, якщо мають однакову кількість символів і збігаються по символам.

Звертатися до окремого елементу рядка по індексу можна тільки для набуття значення, але не для його зміни. Це пов'язано з тим, що рядки типу *string* відносяться до так званих незмінних типів даних. Методи, що змінюють вміст рядка, насправді створюють нову копію рядка. Невживані “старі” копії автоматично видаляються складальником сміття.

У класі System.String передбачено використання методів, полів і властивостей, що дозволяють виконувати з рядками практично будь-які дії. Основні елементи класу приведені в таблиці 6.4.

Таблиця 6.4.

Основні елементи класу System.String

| Назва | Вигляд | Опис |
|---|---|---|
| Compare | Статичний метод | Порівняння двох рядків в лексикографічному (алфавітному) порядку. Різні реалізації методу дозволяють порівнювати рядки і підрядки з урахуванням і без урахування регістра і особливостей національного представлення дат і так далі |
| CompareOrdinal | Статичний метод | Порівняння двох рядків по кодах символів. Різні реалізації методу дозволяють порівнювати рядки і підрядки |
| CompareTo | Метод | Порівняння поточного екземпляру рядка з іншим рядком |

Продовження таблиці 6.4

| Назва | Вигляд | Опис |
|---|---|---|
| Concat | Статичний метод | Конкатенація рядків. Метод допускає зчеплення довільного числа рядків |
| Copy | Статичний метод | Створення копії рядка |
| Empty | Статичне поле | Порожній рядок (тільки для читання) |
| Format | Статичний метод | Форматування відповідно до заданих специфікаторів формату (див. далі) |
| IndexOf, IndexOfAny, LastIndexOf, LastIndexOfAny | Методи | Визначення індексів першого і останнього входження заданого підрядка або будь-якого символу із заданого набору |
| Insert | Метод | Вставка підрядка в задану позицію |
| Intern, IsInterned | Статичні методи | Повертає посилання на рядок, якщо такий вже існує. Якщо рядка немає, Intern додає рядок у внутрішній пул, IsIntern повертає null |
| Join | Статичний метод | Злиття масиву рядків в єдиний рядок. Між елементами масиву вставляються роздільники (див.далі) |
| Length | Властивість | Довжина рядка (кількість символів) |
| PadLeft, PadRight | Методи | Вирівнювання рядка по лівому або правому краю шляхом вставки потрібного числа пропусків на початку або в кінці рядка |
| Remove | Метод | Видалення підрядка із заданої позиції |
| Replace | Метод | Заміна всіх входжень заданого підрядка або символу новим підрядком або символом |
| Split | Метод | Розділяє рядок на елементи, використовуючи задані роздільники. Результати поміщаються в масив рядків |
| StartsWith, EndsWith | Методи | Повертає true або false залежно від того, починається або закінчується рядок заданим підрядком |
| Substring | Метод | Виділення підрядка, починаючи із заданої позиції |
| ToCharArray | Метод | Перетворення рядка в масив символів |
| ToLower, ToUpper | Методи | Перетворення символів рядка до нижнього або верхнього регістра |
| Trim, TrimStart, TrimEnd | Методи | Видалення пропусків початку і кінця рядка |

Приклад застосування методів приведений в лістингу 6.11.

Лістинг 6.11. Робота з рядками типу string

```
using System;
namespace ConsoleApplication1
{
   class Classl
   {
      static void Main()
      {
            string s = "прекрасна королева Ізольда";
            Console.WriteLine(s);
```

```
            string sub = s.Substring(3).Remove(11, 2);                    //1
            Console.WriteLine(sub);

            string[] mas = s.Split(' ');                                  //2
            string joined = string.Join("!", mas);
            Console.WriteLine(joined);
            Console.WriteLine("Введіть рядок");
            string x = Console.ReadLine();                                //3
            Console.WriteLine("Ви ввели рядок " + x);
            double a = 12.234;
            int b = 29;

           Console.WriteLine(" a = {0,6:C} b = {1,2:X}", a, b);           //4
           Console.WriteLine(" a = {0,6:0.#} b = {1,5:0.# ' руб. '}", a, b);  //5
           Console.WriteLine(" a = {0,6:F3} b = {1:D3}", a, b);           //6
        }
    }
}
```

Результат роботи програми:

```
прекрасна королева Ізольда
красна корова Ізольда
прекрасна! королева! Ізольда
Введіть рядок
не хочу!
Ви ввели рядок не хочу!
a = 12,23р.   b = 1D
a = 12,2     b =  29 руб
a = 12,234 b =  029
```

У операторі 1 виконуються два послідовні виклики методів: метод *Substring* повертає підрядок рядка *s*, який містить символи початкового рядка, починаючи з третього. Для цього підрядка викликається метод *Remove*, що видаляє з нього два символи, починаючи з 11-го. Результат роботи методу привласнюється змінною *sub*.

Аргументом методу *Split* (оператор 2) є роздільник, в даному випадку - символ пропуску. Метод розділяє рядок на окремі слова, які заносяться в масив рядків *mas*. Статичний метод *Join* (він викликається через ім'я класу) об'єднує елементи масиву *mas* в один рядок, вставляючи між кожною парою слів рядок "!". Оператор 3 нагадує вам про те, як вводити рядки з клавіатури.

### 6.8.4. Форматування рядків

У операторові 4 з лістингу 6.8 неявно застосовується метод Format, який замінює всі входження параметрів у фігурних дужках значеннями відповідних змінних із списку виведення. Після номера параметра можна задати мінімальну ширину поля виведення, а також вказати специфікатор формату, який визначає форму представлення значення, що виводиться.

У загальному вигляді параметр задається таким чином:

**{n [,m[:специфікатор_ формату]]}**

Тут *n* -номер параметра. Параметри нумеруються з нуля, нульовий параметр замінюється значенням першої змінної із списку виведення, перший параметр - другою змінною і так далі. Параметр *m* визначає мінімальну ширину поля, яке відводиться під значення, що виводиться. Якщо числу, що виводиться, достатньо меншої кількості позицій, невживані позиції заповнюються пропусками. Якщо числу потрібно більше позицій, параметр ігнорується. Специфікатор формату визначає формат виведення значення. Наприклад, специфікатор C (*Currency*) означає, що параметр повинен форматуватися як валюта з урахуванням національних особливостей представлення, а специфікатор X (*Hexadecimal*) задає шістнадцятирічну форму представлення значення, що виводиться.

У операторі 5 використовуються так звані призначені для користувача шаблони форматування. Якщо придивитися, в них немає нічого складного: після двокрапки задається вид значення, що виводиться, причому на місці кожного символу може бути # або 0. Якщо вказаний знак #, на цьому місці буде виведена цифра числа, якщо вона не дорівнює нулю. Якщо вказаний 0, буде виведена будь-яка цифра, у тому числі і 0. У таблиці 6.5 приведені приклади шаблонів і результатів виведення.

Таблиця 6.5.

Приклади застосування призначених для користувача шаблонів форматування

| Число | Шаблон | Вид |
|---|---|---|
| 1,243 | 00.00 | 01,24 |
| 1,243 | #.## | 1,24 |
| 0,1 | 00.00 | 00,10 |
| 0,1 | #.## | ,1 |

Призначений для користувача шаблон може також містити текст, який в загальному випадку береться в апострофи.

### 6.8.5. Рядки типу StringBuilder

Можливості, що надаються класом *string*, широкі, проте вимога незмінності його об'єктів може виявитися незручною. В цьому випадку для роботи з рядками застосовується клас *StringBuilder*, визначений в просторі імен System.Text і що дозволяє змінювати значення своїх екземплярів.

При створенні екземпляра необхідно використовувати операцію new і конструктор, наприклад:

```
StringBuilder   a=new StringBuilder();                  // 1
StringBuilder   b=new StringBuilder("qwerty");          // 2
StringBuilder   c=new StringBuilder(100);               // 3
StringBuilder   d=new StringBuilder("qwerty", 100);     // 4
StringBuilder   e=new StringBuilder("qwerty",1, 3, 100);  // 5
```

У конструкторі класу указуються два види параметрів: рядок, що ініціалізував, або підрядок і об'єм пам'яті, що відводиться під екземпляр (ємність буфера). Один або обидва параметри можуть бути відсутніми, в цьому випадку використовуються їх значення за умовчанням.

Якщо застосовується конструктор без параметрів (оператор 1), створюється порожній рядок розміру, заданого за умовчанням (16 байт). Інші види конструкторів задають об'єм пам'яті, рядку, що виділяється, і/або її початкове значення. Наприклад, в операторі 5 об'єкт ініціалізувався підрядком завдовжки 3 символи, починаючи з першого (підрядок "wer"). Основні елементи класу *StringBuilder* приведені в таблиці 6.6.

Таблиця 6.6

Основні елементи класу *System.Text.StringBuilder*

| Назва | Вид | Опис |
|---|---|---|
| Append | Метод | Додавання в кінець рядка. Різні варіанти методу дозволяють додавати в рядок величини будь-яких вбудованих типів, масиви символів, рядка і підрядка типу *string* |
| AppendFormat | Метод | Додавання форматованого рядка в кінець рядка |
| Capacity | Властивість | Отримання або установка ємності буфера. Якщо встановлюване значення менше поточної довжини рядка або більше максимального, генерується виключення *ArgumentOutOfRangeException* |
| Insert | Метод | Вставка підрядка в задану позицію |
| Length | Властивість | Довжина рядка (кількість символів) |
| MaxCapacity | Властивість | Максимальний розмір буфера |
| Remove | Метод | Видалення підрядка із заданої позиції |
| Replace | Метод | Заміна всіх входжень заданого підрядка або символу новим підрядком або символом |
| ToString | Метод | Перетворення в рядок типу *string* |

Приклад застосування методів приведений в лістингу 6.12.

Лістинг 6.12. Робота з рядками типу StringBuilder

```
using System;
using System.Text;
namespace ConsoleApplication1
{
   class examp38
   {
      static void Main()
      {
         Console.Write("Введіть зарплату: ");
         double salary = double.Parse(Console.ReadLine());
         StringBuilder a = new StringBuilder();
```

```
            a.Append("зарплата ");
            a.AppendFormat("{0,6:C} - в рік {1,6:C}", salary, salary * 12);
            Console.WriteLine(a);
            a.Replace("р.","тис.$");

            Console.WriteLine("А краще було б: " + a);
        }
    }
}
```

Результат роботи програми:

Введіть зарплату: 3500

зарплата 3 500.00р. - в рік 42 000,00р.

А краще було б: зарплата 3 500,00тис.$ - в рік 42 000.00тис.$

Ємність буфера не відповідає кількості символів в рядку і може збільшуватися в процесі роботи програми як в результаті прямих вказівок програміста, так і унаслідок виконання методів зміни рядка, якщо рядок в результаті перевищує поточний розмір буфера. Програміст може зменшити розмір буфера за допомогою властивості *Capacity*, щоб не позичати зайву пам'ять.

## 6.9. Рекомендації з програмування

Використовуйте для зберігання даних масив, якщо кількість однотипних елементів, які потрібно обробити у вашій програмі, відома або, принаймні, відома максимальна кількість таких елементів. У останньому випадку пам'ять під масив виділяється «по максимуму», а фактична кількість елементів зберігається в окремій змінній, яка обчислюється в програмі.

При роботі з масивом потрібно обов'язково передбачати обробку виключення IndexOutOfRangeException, якщо індекс для звернення до масиву обчислюється в програмі по формулах, а не задається за допомогою констант або лічильників циклів for.

Якщо кількість елементів, що обробляються програмою, може бути довільною, зручніше використовувати не масив, а інші структури даних, колекції, що наприклад параметризуються, які розглядаються в розділі 13.

При роботі з рядками необхідно враховувати, що в C# рядок типу string є незмінним типом даних, тобто будь-яка операція зміни рядка насправді повертає її копію. Для зміни рядків використовується тип StringBuilder. Перш ніж описувати в програмі яку-небудь дію з рядками, корисно подивитися, чи немає в списку елементів використовуваного класу відповідних методів і властивостей.

# РОЗДІЛ 7. КЛАСИ: ПОДРОБИЦІ

У цьому розділі ми продовжимо розглядати елементи класів. Спочатку ми розглянемо додаткові можливості методів, які не описані в розділі 5, а потім перейдемо до нових елементів класу - індексаторів, операцій і деструкторам.

## 7.1. Перевантаження методів

Компілятор визначає який саме метод потрібно викликати за типом фактичних параметрів. Цей процес називається *дозволом* (*resolution)* перевантаження. Тип повертаємого методом значення в дозволі не бере участь. Механізм дозволу заснований на достатньо складному наборі правил, сенс яких зводиться до того, щоб використовувати метод з найбільш відповідними аргументами і видати повідомлення, якщо такий не знайдеться. Допустимо, є чотири варіанти методу, що визначає найбільше значення:

```
// Повертає найбільше з двох цілих:
int max(int a, int b )
// Повертає найбільше з трьох цілих:
int max(int a, int b, int e)
// Повертає найбільше з першого параметра і довжини другого:
int max (int a, string b)
// Повертає найбільше з другого параметра і довжини першого:
int max (string b, int a)
…
…
Console.WriteLine( max (1,2));
Console.WriteLine( max (1,2,3));
Console.WriteLine( max (1,"2"));
Console.WriteLine( max ("1",2));
```

При виклику методу *max* компілятор вибирає варіант методу, відповідний типу переданих в метод аргументів (у приведеному прикладі будуть послідовно викликані всі чотири варіанти методу). Якщо точної відповідності не знайдено, виконуються неявні перетворення типів відповідно до загальних правил, наприклад, *bool* і *char* в *int*, *float* в *double* і тому подібне. Якщо перетворення неможливе, видається повідомлення про помилку. Якщо відповідність на одному і тому ж етапі може бути отримане більш ніж одним способом, вибирається “кращий з варіантів, тобто варіант, що містить менші кількість перетворень. Якщо існує декілька варіантів, з яких неможливо вибрати кращий, видається повідомлення про помилку.

Перевантажені методи мають одне ім'я, але повинні розрізнятися параметрами, точніше, їх типами і способами передачі (out або ref). Наприклад, методи, заголовки яких приведені нижче, мають різні сигнатури і вважаються перевантаженими:

```
int max( int a, int b )
int max( int a, ref int b)
```

Перевантаження методів є проявом поліморфізму, однієї з основних властивостей *ООП*. Програмістові набагато зручніше пам'ятати одне ім'я методу і використовувати його для роботи з різними типами даних, а рішення про те, який варіант методу викликати, покласти на компілятор. Цей принцип широко використовується в класах бібліотеки *.NET*. Наприклад, в стандартному класі *Console* метод *Writeline* перевантажений 19 разів для виведення величин різних типів.

## 7.2. Рекурсивні методи

*Рекурсивним* називається метод, який викликає сам себе. Така рекурсія називається *прямою*. Існує ще *непряма рекурсія*, коли два або більш за метод викликають один одного. Якщо метод викликає себе, в стеку створюється копія значень його параметрів, як і при виклику звичайного методу, після чого управління передається першому виконуваному операторові методу. При повторному виклику цей процес повторюється.

Ясно, що для завершення обчислень кожен рекурсивний метод повинен містити хоч би одну нерекурсивну гілку алгоритму, що закінчується оператором повернення. При завершенні методу відповідна частина стека звільняється і управління передається методу, виконання якого продовжується з точки, наступної за рекурсивним викликом.

Класичним прикладом рекурсивної функції є функція обчислення факторіалу (це не означає, що факторіал слід обчислювати саме так). Для того, щоб набути значення факторіалу числа n, потрібно помножити на *n* факторіал числа (*n - 1*). Відомо також, що 0 != 1 і 1 != 1:

```
long fact (long n )
{
if ( n = = 0 || n = = 1  ) return 1;  // нерекурсивна гілка
return (n* fact(n -1));        // рекурсивна гілка
}
```

Те ж саме можна записати коротше:

```
long fact (long n )
{
return ( n >1 ) ? n * fact(n -1):1;
}
```

Рекурсивні методи найчастіше застосовують для компактної реалізації рекурсивних алгоритмів. Будь-який рекурсивний метод можна реалізувати без застосування рекурсії, для цього програміст повинен забезпечити зберігання всіх необхідних даних самостійно. До переваг рекурсії можна віднести компактність запису, до недоліків - витрата часу і пам'яті на повторні виклики методу і передачу йому копій параметрів. Головний недолік - переповнення стека.

## 7.3. Методи із змінною кількістю аргументів

Іноді буває зручно створити метод, в який можна передавати різну кількість аргументів. Мова C# надає таку можливість за допомогою ключового слова params. Параметр, помічений цим ключовим словом, розміщується в списку параметрів останнім і позначає масив заданого типу невизначеної довжини, наприклад:

```
public int Calculate( int a, out int c, params int[] d)
...
```

У цей метод можна передати три і більше параметрів. Усередині методу до параметрів, починаючи з третього, звертаються як до звичайних елементів масиву. Кількість елементів масиву отримується за допомогою властивості *Length*. Як приклад розглянемо метод обчислення середнього значення елементів масиву (лістинг 7.1).

Лістинг 7.1. Методи із змінною кількістю аргументів

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

namespace examp40
{
   class Program
   {
      static long fact(long n)
      {
         if (n == 0 || n == 1) return 1;
         return (n * fact(n - 1));
      }
      public static double Average(params int[] a)
      {
         if (a.Length == 0)
            throw new Exception
               ("Недостатньо аргументів в методі");
         double av = 0;
         foreach (int elem in a) av += elem;
         return av / a.Length;
      }
      static void Main(string[] args)
      {
         int[] a = { 10, 20, 30 };
         int[] b = { -11, -4, 12, 14, 32, -1, 28 };
         long d;
         try
         {
         // ---------------------------------
```

```
            Console.Write(" ВВедите N  ");
            d =   Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("   " + fact(d));
            // --------------------------------

            Console.WriteLine(Average(a));
            Console.WriteLine(Average(b));
            short z=1,e=12;
            byte v=107;
            Console.WriteLine(Average(z,e,v));
            Console.WriteLine(Average());
            }

               catch(Exception e)
               {
               Console.WriteLine(e.Message);
               return;
               }

        }
    }
}
```

Результат роботи програми:

10
20
40
Недостатньо аргументів в методі

## 7.4. Метод Main

Метод, якому передається управління після запуску програми, повинен мати ім'я *Main* і бути статичним. Він може приймати параметри із зовнішнього оточення і повертати значення в середовище, що викликало. Передбачається два варіанти методу - з параметрами і без параметрів:

```
// без параметрів:
static тип Main() { ... }
static void Main() { ... }
// з параметрами:
static тип Main ( string[] args ){/*...*/}
static void Main( string[] args ){/*...*/}
```

Параметри, що розділяються пропусками, задаються при запуску програми з командного рядка після імені виконуваного файлу програми. Вони передаються в масив args.

Якщо метод повертає значення, воно має бути цілого типу, якщо не повертає, він повинен описуватися як void. В цьому випадку оператор повернення з Main можна не вказувати, а середовище, що викликало, автоматично набуде

нульового значення, що означає успішне завершення. Ненульове значення зазвичай означає аварійне завершення, наприклад:

```
static  int Main( string[] args )
{
if (... /* все пропало */ ) return 1;

if (.../* абсолютно все пропало */)  return 100;
}
```

Повертаєме значення аналізується в командному файлі, з якого запускається програма. Зазвичай це робиться для того, щоб можна було прийняти рішення, чи виконувати командний файл далі. У лістингу 7.2 наводиться приклад методу Main, який виводить свої аргументи і чекає натиснення будь-якої клавіші.

Лістинг 7.2. Параметри методу Main

```
using System;

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
         foreach (string arg in args) Console.WriteLine(arg);
         Console.Read();
       }
   }
}
```

Нехай виконуваний файл програми має ім'я Consoleapplication1.exe і викликається з командного рядка:

d:\cs\consoleapplicationl\bin\debug\consoleapplicationl.exe one two three

Тоді на екран буде виведено:

one
two
three

Для запуску програми з командного рядка можна скористатися командою **Выполнить** меню **Пуск**

## 7.5. Індексатори

Індексатором є різновид властивості. Якщо у класу є приховане поле, що є масивом, то за допомогою індексатора можна звернутися до елементу цього масиву, використовуючи ім'я об'єкту і номер елементу масиву в квадратних дужках. Іншими словами, індексатор - це індекс для об'єктів.

Синтаксис індексатора аналогічний синтаксису властивості:

```
атрибути специфікатори тип this [ список_параметрів]
{
get код_доступа
set код_доступа
}
```

В даному випадку квадратні дужки є елементом синтаксису, а не вказівкою на необов'язковість конструкції.

*Атрибути* ми розглянемо пізніше, в розділі 12, а специфікатори аналогічні специфікаторам властивостей і методів. Індексатори найчастіше оголошуються із специфікатором *public*, оскільки вони входять в інтерфейс об'єкту. Атрибути і специфікатори можуть бути відсутніми.

*Код доступу* є блоком операторів, які виконуються при отриманні (*get*) або установці значення (*set*) елементу масиву. Може бути відсутньою або частина *get*, або *set*, але не обидві одночасно. Якщо відсутня частина *set*, індексатор доступний тільки для читання (*read-only*), якщо відсутня частина *get*, індексатор доступний тільки для запису (*write-only*).

*Список параметрів* містить одне або декілька описів індексів, по яких виконується доступ до елементу. Найчастіше використовується один індекс цілого типу. Індексатори в основному застосовуються для створення спеціалізованих масивів, на роботу з якими накладаються які-небудь обмеження. У лістингу 7.3 створений клас-масив, елементи якого повинні знаходитися в діапазоні [0, 100]. Крім того, при доступі до елементу перевіряється, чи не вийшов індекс за допустимі межі.

Лістинг 7.3. Використання індексаторів

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

namespace examp42
{
    class SafeArray
   {
      public SafeArray(int size)
      {
       a=new int[size];
       length=size;
      }
         public int Length          // властивість розмірність
         {
           get{return length;}
         }
      public int this[int i]          //індексатор
      {
```

```
            get
            {
              if(i>=0&&i<length) return a[i];
              else
              {
              error=true;
              return 0;
              }
            }
            set
            {
               if (i >= 0 && i < length
                  && value >= 0 && value <= 100) a[i] = value;
               else error=true;
            }
          }

      public bool error=false;       //прихована ознака помилки
      int []a;          //закритий масив
      int length;     //закрита розмірність
     }

        class Class1
        {
           static void Main()
           {
           int n=60;
           SafeArray sa=new SafeArray(n);
           for(int i=0;i<n;++i)
           {
           sa[i]=i*2;          // 1 використання індексатора
           Console.Write("  "+ sa[i]);       // 2 використання індексатора
           }
           if(sa.error)Console.Write("Error");
           }
      }
}
```

З лістингу видно, що індексатори описуються аналогічно властивостям. Завдяки застосуванню індексаторів з об'єктом, що містить в собі масив, можна працювати так само, як зі звичайним масивом. Якщо звернення до об'єкту зустрічається в лівій частині оператора привласнення (оператор 1), автоматично викликається метод *get*. Якщо звернення виконується у складі виразу (оператор 2), викликається метод *set*.

У класі *Safearray* прийнята наступна стратегія обробки помилок. Якщо при спробі запису елементу масиву його індекс або значення задані невірно, значення елементу не привласнюється. Якщо при спробі читання елементу індекс не входить в допустимий діапазон, повертається 0. У обох випадках формується значення відкритого поля *error*, рівне *true*.

Взагалі кажучи, індексатор не обов'язково має бути пов'язаний з яким-небудь внутрішнім полем даних. У лістингу 7.4 приведений приклад класу *Pow2*, єдине призначення якого - формувати ступінь числа 2.

Лістинг 7.4. Індексатор без масиву

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

namespace examp43
{

   class Pow2
   {

      public ulong this[int i]
      {
         get
         {
            if (i >= 0)
            {
               ulong res = 1;
               for (int k = 0; k < i; k++)// Цикл отримання ступеня
                  unchecked { res *= 2; }// 1
               return res;
            }
            else return 0;
         }// get
      }//this

   }//Pow2




      class Class1
      {
       static void Main()
        {
           try
           {

              int n = 30;
              Pow2 pow2 = new Pow2();
              for (int i = 0; i < n; ++i)
                 Console.WriteLine("{0}\t{1}", i, pow2[i]);
           }
```

```
            catch (Exception e)
            {
               Console.WriteLine(e.Message);
               return;
            }
          }
      }//Class1
}// namespace
```

Оператор 1 виконується в контексті, що не перевіряє, для того, щоб виключення, пов'язане з переповнюванням, не генерувалося. В принципі, дана програма працює і без цього, але якщо помістити клас *Pow2* в контекст, що перевіряється, при значенні, що перевищує допустимий діапазон для типу *ulong*, виникне виключення.

Результат роботи програми:

```
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
```

Мова C# допускає використання *багатовимірних індексаторів*. Вони застосовуються для контролю за занесенням даних в багатовимірні масиви. Крім того, вони використовуються при вибірці даних з багатовимірних масивів, оформлених у вигляді класів. Наприклад:

```
int[,] a;
```

Якщо усередині класу оголошений такий двовимірний масив, то заголовок індексатора повинен мати вигляд

```
public  int this[int i, int j]
```

## 7.6. Операції класу

C# дозволяє перевизначити дію більшості операцій так, щоб при використанні з об'єктами конкретного класу вони виконували задані функції. Приклад:

```
MyObject a, b, c;
…
c = a + b;       // використовується операція складання для класу MyObject
```

Визначення власних операцій класу часто називають перевантаженням операцій. Перевантаження зазвичай застосовується для класів, що описують математичні або фізичні поняття, тобто таких класів, для яких семантика операцій робить програму зрозумілішою.

Операції класу описуються за допомогою методів спеціального вигляду (*функцій-операцій*). Перевантаження операцій схоже на перевантаження звичайних методів. Синтаксис операції:

[атрибути ] специфікатори об’явник_операції тіло

*Атрибути* розглядаються у розділі 12, як специфікатори одночасно використовуються ключові слова *public* і *static*. Крім того, операцію можна оголосити як зовнішню (*extern*).

*Об'явник операції* містить ключове слово *operator*, по якому і впізнається опис операції в класі. Тіло операції визначає дії, які виконуються при використанні операції у виразі. Тілом є блок, аналогічний тілу інших методів.

При описі операцій необхідно дотримуватись наступних правил:

– операція має бути описана як відкритий статичний метод класу (специфікатори *public static*);

– параметри в операцію повинні передаватися за значенням (тобто без ключових слів *ref* або *out*);

– сигнатури всіх операцій класу повинні розрізнятися.

У C# існують три види операцій класу: *унарні*, *бінарн*і і операції перетворення типу.

### 7.6.1. Унарні операції

Можна визначати в класі наступні унарні операції:

```
+     -     !     -     ++    --    true  false
```

Синтаксис об'явника унарної операції:

тип operator унарна_операція ( параметр )

Приклади заголовків унарних операцій:

```
public static int operator +(MyObject m )
```

public static MyObject operator -- ( MyObject m )
public static bool operator true(MyObject m )

Параметр, переданий в операцію, повинен мати тип класу, для якого вона визначається. Операція повинна повертати:

- для операцій +, - ! ~ величину будь-якого типу;
- для операцій ++ -- величину типу класу, для якого вона відзначається;
- для операцій *true* і *false* величину типу *bool*.

Операції не повинні змінювати значення переданого ним операнда. Операція, що повертає величину типу класу, для якого вона визначається, повинна створити новий об'єкт цього класу, виконати з ним необхідні дії і передати його як результат.

Префіксний і постфіксний інкременти не розрізняються.

Як приклад удосконалимо приведений в лістингу 7.3 клас Safearray для зручної і безпечної роботи з масивом. До класу внесені наступні зміни:

- доданий конструктор, що дозволяє ініціалізувати масив звичайним масивом або серією цілочисельних значень довільного розміру;
- додана операція інкремента;
- доданий допоміжний метод *Print* виведення масиву;
- змінена стратегія обробки помилок виходу за межі масиву;
- знята вимога, щоб елементи масиву приймали значення в заданому діапазоні.

Текст програми приведений в лістингу 7.5.

Лістинг 7.5. Визначення операції інкремента для класу

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

namespace examp44
{
   class SafeArray
   {
      public SafeArray(int size)  // конструктор
      {
         a = new int[size];
         length = size;
      }
      public SafeArray(params int[] arr)  //новий конструктор
      {
         length = arr.Length;
         a = new int[length];
         for (int i = 0; i < length; ++i) a[i] = arr[i];
      }
```

```
        public static SafeArray operator ++(SafeArray x)  // ++
        {
        SafeArray temp = new SafeArray(x.length );
        for ( int i = 0; i < x.length; ++i )
        temp[i] = ++x.a[i] ;
        return temp;
        }


      public int this[int i]  // індексатор
      {
        get
        {
        if ( i >= 0 && i < length ) return a[i];
        else throw new IndexOutOfRangeException(); // виключення
        }
        set
        {
          if (i >= 0 && i < length) a[i] = value;
          else throw new IndexOutOfRangeException(); // виключення
        }
      }
      public void Print(string name) // виведення на екран
      {
        Console.WriteLine(name + ": ");
        for (int i = 0; i < length; ++i)
          Console.Write("\t" + a[i]);
        Console.WriteLine();
      }
      int[] a; // закритий масив
      int length; // закрита розмірність
    }

    class Class1
    {
      static void Main()
      {
        try
        {
          int[] f = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
          SafeArray a1 = new SafeArray(f);
          a1.Print("Масив 1");
          a1++;
          a1.Print("Інкремент масиву 1");
        }
        catch (Exception e) // обробка виключення
        {
          Console.WriteLine(e.Message);
        }
      }
    }
}
```

### 7.6.2. Бінарні операції

Можна визначати в класі наступні бінарні операції:

+ - * / % & | ^ << >> == != > < >= <=

Синтаксис об'явника бінарної операції:

тип operator бінарна_операція (параметр1, параметр2)

Приклади заголовків бінарних операцій:

Cjblic static MyObject operator + (MyObject m1, MyObject m2 )
Cublic static bool operator = = (MyObject m1, MyObject m2 )

Хоч би один параметр, переданий в операцію, повинен мати тип класу, для якого вона визначається. Операція може повертати величину будь-якого типу.

Операції == і ! =, > і <, >= і <= визначаються тільки парами і зазвичай повертають логічне значення. Найчастіше в класі визначають операції порівняння на рівність і нерівність для того, щоб забезпечити порівняння об'єктів, а не їх посилань, як визначено за умовчанням для посилальних типів. Перевантаження операцій відношення вимагає знання інтерфейсів, тому вона розглядається пізніше.

Приклад визначення операції складання для класу Safearray, описаного в попередньому розділі, приведений в лістингу 7.6. Залежно від операндів операція або виконує поелементне складання двох масивів, або додає значення операнда до кожного елементу масиву.

Лістинг 7.6. Визначення операції складання для класу SafeArray

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

namespace examp47
{
    class SafeArray
   {
        public SafeArray( int size )
        {
         a = new int[size] ;
         length = size;
        }
```

```
public SafeArray( params int[ ] arr )
{
 length = arr.Length;
 a = new int[length] ;
 for ( int i = 0; i < length; ++i ) a[i] = arr[i];
}

public static SafeArray operator + ( SafeArray x, SafeArray y ) // +
{
 int len = x.length < y.length ? x.length : y.length;
 SafeArray temp = new SafeArray (len) ;
 for ( int i = 0; i < len; ++i ) temp[i] = x[i] + y[i];
 return temp;
 }

 public static SafeArray operator + ( SafeArray x, int y )       // +
 {
    SafeArray temp = new SafeArray(x.length);
    for (int i = 0; i < x.length; ++i) temp[i] = x[i] + y;
    return temp;
 }
 public static SafeArray operator +(int x, SafeArray y)          // +
 {
    SafeArray temp = new SafeArray(y.length);
    for (int i = 0; i < y.length; ++i) temp[i] = x + y[i];
    return temp;
 }
 public static SafeArray operator ++ ( SafeArray x )   // ++
 {
    SafeArray temp = new SafeArray(x.length);
    for ( int i = 0; i < x.length; ++i ) temp[i] = ++x.a[i] ;
    return temp;
 }
 public int this[int i]      // []
 {
    get
    {
       if (i >= 0 && i < length) return a[i];
       else throw new IndexOutOfRangeException();
    }
    set
    {
       if (i >= 0 && i < length) a[i] = value;
       else throw new IndexOutOfRangeException();
    }
 }
 public void Print( string name )
 {
```

```
            Console.WriteLine(name + ": " );
            for ( int i = 0; i < length; ++i ) Console.Write("\t" + a[i]);
            Console.WriteLine();
        }
        int[] a;   //закритий масив
        int length;         //закрита розмірність
    }

    class Classl
    {
      static void Main()
      {
        try
        {
        SafeArray a1 = new SafeArray( 5, 2, -1, 1, -2 );
        a1.Print ( "Масив 1" );
        SafeArray a2 = new SafeArray( 1, 0, 3 );
        a2.Print( "Масив 2" );
        SafeArray a3 = a1 + a2;
        a3.Print( "Сума масивів 1 и 2" );
        a1 = a1 + 100;       // 1
        a1.Print ( "Масив 1 + 100" );
        a1 = 100 + a1 ;       // 2
        a1.Print ("100 + масив 1" );
        a2 += ++a2+1;      // 3
        a2.Print( "++a2.a2 + a2 + 1" );
        }
        catch ( Exception e )
        {
        Console.WriteLine( e.Message ) ;
        }
      }
    }
}
```

Щоб забезпечити можливість складання з константою, операція складання перевантажена двічі для випадків, коли константа є першим і другим операндом (оператори 2 і 1).

Складну операцію привласнення += (оператор 3) визначати не потрібно, та це і неможливо. При її виконанні автоматично викликається спочатку операція складання, потім привласнення.

### 7.6.3. Операції перетворення типу

Операції перетворення типу забезпечують можливість явного і неявного перетворення між призначеними для користувача типами даних. Синтаксис об’явника операції перетворення типу:

```
implicit operator тип (параметр )        // неявне перетворення
explicit operator тип  (параметр )       // явне перетворення
```

Ці операції виконують перетворення з типу параметра в тип, вказаний в заголовку операції. Одним з цих типів має бути клас, для якого визначається операція. Таким чином, операції виконують перетворення або типу класу до іншого типу, або навпаки. Перетворювані типи не мають бути зв'язані стосунками спадкоємства. Приклади операцій перетворення типу для класу Monster:

```
public static implicit operator int( Monster m )
{
return m.health;
}

public static explicit operator Monster(int h )
{
return new Monster(h, 100, "FromInt");
}
```

Нижче приведені приклади використання цих перетворень в програмі. Не треба шукати в них сенс, вони просто ілюструють синтаксис:

```
Monster Masha = new Monster(200, 200, "Masha");
int i = Masha;          //неявне перетворення
Masha = (Monster) 500;    //явне перетворення
```

Неявне перетворення виконується автоматично:

- при привласненні об'єкту змінній цільового типу, як в прикладі;
- при використанні об'єкту у виразі, що містить змінні цільового типу;
- при передачі об'єкту в метод на місце параметра цільового типу;
- при явному приведенні типу.

*Явне перетворення* виконується при використанні операції приведення типу. Всі операції класу повинні мати різні сигнатури. Ключові слова implicit і explicit в сигнатуру не включаються, отже, для одного і того ж перетворення не можна визначити одночасно явну і неявну версії.

*Неявне перетворення* слід визначати так, щоб при його виконанні не виникала втрата точності і не генерувалися виключення. Якщо ці ситуації можливі, перетворення слід описати як явне.

## 7.7. Деструктор

У C# існує спеціальний вид методу, який називається деструктором. Він викликається складальником сміття безпосередньо перед видаленням об'єкту з пам'яті. У деструкторі описуються дії, що гарантують коректність подальшого видалення об'єкту, наприклад, перевіряється, чи всі ресурси, використовувані об'єктом, звільнені (файли закриті, видалене з'єднання розірване і т. п.).

Синтаксис деструктору:

[атрибути] [extern] ~ ім'я_класу( )
тіло

Як видно з визначення, деструктор не має параметрів, не повертає значення і не вимагає вказівки специфікаторів доступу. Його ім'я збігається з ім'ям класу і перед ним ставиться тильда (~), виконує зворотні по відношенню до конструктора дії. Тілом деструктору є блок або просто крапка з комою, якщо деструктор визначений як зовнішній (*extern*).

Складальник сміття видаляє об'єкти, на які немає посилань. Він працює відповідно до своєї внутрішньої стратегії в невідомі для програміста моменти часу. Оскільки деструктор викликається складальником сміття, неможливо гарантувати, що деструктор буде обов'язково викликаний в процесі роботи програми. Отже, його краще використовувати тільки для гарантії звільнення ресурсів, а “штатне” звільнення виконувати у іншому місці програми.

Застосування деструкторів уповільнює процес збірки сміття.

## 7.8. Вкладені типи

У класі можна визначати типи даних, внутрішні по відношенню до класу. Так визначаються допоміжні типи, які використовуються класом, що тільки містить їх. Механізм вкладених типів дозволяє приховати непотрібні деталі і більш повно реалізувати принцип інкапсуляції. Безпосередній доступ ззовні до такого класу неможливий (мається на увазі доступ по імені без уточнення). Для вкладених типів можна використовувати ті ж специфікатори, що і для полів класу.

Наприклад, введемо в наш клас Monster допоміжний клас Gun. Об'єкти цього класу без “господаря” даремні, тому його можна визначити як внутрішній:

```
using System;
namespace ConsoleApplication1
{
class Monster
{
```

```
class Gun
{
…
}
…
}
}
```

## 7.9. Рекомендації по програмуванню

Як правило, клас як тип, визначений користувачем, повинен містити приховані (private) поля і наступні функціональні елементи:

- конструктори, що визначають, як ініціалізувати об'єкти класу;
- набір методів і властивостей, що реалізовують характеристики класу;
- класи виключень, використовувані для повідомлень про помилки шляхом генерації виняткових ситуацій.

Класи, що моделюють математичні або фізичні поняття, зазвичай так само містять набір *операцій*, що дозволяють копіювати, привласнювати, порівнювати об'єкти і проводити з ними інші дії, потрібні по суті класу.

Перевантажені операції класу повинні мати інтуїтивно зрозумілий загальноприйнятий сенс (наприклад, не слід примушувати операцію + виконувати що-небудь, окрім складання або додавання). Якщо яка-небудь операція перевантажена, слід перенавантажувати і аналогічні операції, наприклад + і ++ (компілятор цього автоматично не зробить). При цьому операції повинні мати ту ж семантику, що і їх стандартні аналоги.

У переважній більшості класів для реалізації дій з об'єктами класу переважно використовувати не операції, а методи, оскільки їм можна дати осмислені імена.

Перевантажені методи, на відміну від операцій, застосовуються в класах завжди - як мінімум, використовується набір перевантажених конструкторів для створення об'єктів різними способами.

Методи із змінним числом параметрів реалізуються менш ефективно, чим звичайні, тому якщо, наприклад, потрібно передавати в метод два, три або чотири параметри, можливо, виявиться ефективнішим реалізувати не один метод з параметром *params*, а три перевантажені варіанти із звичайними параметрами.

# РОЗДІЛ 8. ІЄРАРХІЇ КЛАСІВ

Управляти великою кількістю розрізнених класів досить складно. З цією проблемою можна справитися шляхом об'єднання загальних для декількох класів властивостей в одному класі, і використовувати його як базовий.

Цю можливість надає механізм спадкоємства, який є щонайпотужнішим інструментом *ООП*. Він дозволяє будувати ієрархії, в яких класи-нащадки набувають властивостей класів-предків і можуть доповнювати їх або змінювати. Таким чином, спадкоємство забезпечує важливу можливість багатократного використання коду. Написавши і відлагодивши код базового класу, можна, не змінюючи його, за рахунок спадкоємства пристосувати клас для роботи в різних ситуаціях. Це економить час розробки і підвищує надійність програм.

Класи, розташовані ближче до початку ієрархії, об'єднують в собі загальні риси для всіх класів, що розташовані нижче. У міру просування вниз за ієрархією класів набувають все більше конкретних особливостей.

Отже, спадкоємство застосовується для наступних взаємозв'язаних цілей:

– виключення з програми фрагментів коду, що повторюються;

– спрощення модифікації програми;

– спрощення створення нових програм на основі тих, що існують.

Крім того, спадкоємство є єдиною можливістю використовувати об'єкти, початковий код яких недоступний, але в яких потрібно внести зміни.

## 8.1. Спадкоємство

Клас в C# може мати довільну кількість нащадків і лише одного предка. При описі класу ім'я його предка записується в заголовку класу після двокрапки. Якщо ім'я предка не вказане, за предка вважається базовий клас всієї ієрархії *System.Object*:

```
[атрибути]  [специфікатори] class ім'я_класу [: предки]
тіло_класу
```

Слово "предки" присутнє в описі класу в множині, хоча клас може мати тільки одного предка. Причина в тому, що клас разом з єдиним предком може успадковувати від інтерфейсів - спеціального виду класів, що не мають реалізації. Інтерфейси розглядаються в наступному розділі.

Розглянемо спадкоємство класів на прикладі. У 5 розділі був описаний клас *Monster,* моделюючий персонаж комп'ютерної гри. Допустимо, нам потрібно ввести в гру ще один тип персонажів, який повинен володіти властивостями об'єкту *Monster*. Буде логічне зробити новий об'єкт нащадком об'єкту *Monster* (лістинг 8.1).

Лістинг 8.1. Клас Daemon, предок класу Monster

```
using System;
namespace examp48
{
   class Monster
   {
      public Monster()
      {
         this.name = "Noname";
         this.health = 100;
         this.ammo = 100;
      }

      public Monster(string name): this()
      {
         this.name = name;
      }
      public Monster(int health, int ammo, string name)
      {
         this.name = name;
         this.health = health;
         this.ammo = ammo;
      }
      public string GetName()
      {
         return name;
      }
      public int GetHealth()
      {
         return health;
      }
      public int GetAmmo()
      {
         return ammo;
      }
      public void Passport()
      {
         Console.WriteLine("Monster {0} \t health = {1} ammo = {2}", name, health, ammo);

      }

      protected string name;       // закриті поля
      public int health, ammo;
   }

   class Daemon : Monster
   {
      public Daemon()
      {
         brain = 1;
      }
```

```
    public Daemon(string name, int brain)
      : base(name)     // 1
    {
      this.brain = brain;
    }

    public Daemon(int health, int ammo, string name, int brain)
      : base(health, ammo, name)     // 2
    {
      this.brain = brain;
    }

    new public void Passport()          // 3
    {
      Console.WriteLine(
      "Daemon {0} \t health = {1} ammo = {2} brain = {3}",
      name, health, ammo, brain);

    }

    public void Think()          // 4
    {
      Console.Write(name + " is ");
      for (int i = 0; i < brain; ++i) Console.Write(" thinking");
      Console.WriteLine("...");
    }
    int brain; // закрите поле
  }

  class Classl
  {
    static void Main()
     {
      Daemon Dima =  new Daemon("Dima", 3);   // 5
      Dima. Passport();         // 6
      Dima.Think();  // 7
      Dima.GetHealth();       // 8
      Dima.Passport();
     }
  }
}
```

У класі *Daemon* введені закрите поле *brain* і метод *Think*, визначені власні конструктори, а також перевизначений метод *Passport.* Всі поля і властивості класу *Monster* успадковуються в класі *Daemon*.

Результат роботи програми:

```
Daemon Dima        health = 100 ammo =  100 brain = 3
Dima  is  thinking  thinking  thinking...
Daemon Dima        health = 90 ammo =  100 brain = 3
```

Екземпляр класу *Daemon* використовує як власні (оператори 5-7), так і успадковані (оператор 8) елементи класу. Розглянемо загальні правила спадкоємства, використовуючи як приклад лістинг 8.1.

*Конструктори* не успадковуються, тому похідний клас повинен мати власні конструктори. Порядок виклику конструкторів визначається приведеними далі правилами:

- Якщо в конструкторі похідного класу явний виклик конструктора базового класу відсутній, то автоматично викликається конструктор базового класу без параметрів. Це правило використане в першому з конструкторів класу *Daemon*.
- Для ієрархії, що складається з декількох рівнів, конструктори базових класів викликаються, починаючи з самого верхнього рівня. Після цього виконуються конструктори тих елементів класу, які є об'єктами, в порядку їх оголошення в класі, а потім виконується конструктор класу. Таким чином, кожен конструктор ініціалізував свою частину об'єкту.
- Якщо конструктор базового класу вимагає вказівки параметрів, він має бути явним чином викликаний в конструкторі похідного класу в списку ініціалізації (оператори 1 і 2). Виклик виконується за допомогою ключового слова *base*. Викликається та версія конструктора, список параметрів якої відповідає списку аргументів, вказаних після слова *base*.

Поля, методи і властивості класу успадковуються, тому за бажання замінити елемент базового класу новим елементом слід явним чином вказати компілятору свій намір за допомогою ключового слова *new*. У лістингу 8.1 таким чином перевизначений метод виведення інформації про об'єкт *Passport.*

Метод *Passport* класу *Daemon* заміщає відповідний метод базового класу, проте можливість доступу до методу базового класу з методу похідного класу зберігається. Для цього перед викликом методу указується слово *base*, наприклад:

```
base.Passport( );
```

Виклик однойменного методу предка з методу нащадка завжди дозволяє зберегти функції предка і доповнити їх, не повторюючи фрагмент коду. Окрім зменшення об'єму програми це полегшує її модифікацію, оскільки зміни, внесені до методу предка, автоматично відбиваються у всіх його нащадках. У конструкторах метод предка викликається після списку параметрів і двокрапки, а в решті методів - в будь-якому місці за допомогою приведеного синтаксису.

Ось, наприклад, як виглядав би метод *Passport*, якби ми в класі *Daemon* хотіли не повністю перевизначити поведінку його предка, а доповнити його:

```
new public void Passport()
{
base.Passport();
Console.WriteLine(" brain = {1}", brain );
}
```

Елементи базового класу, визначені як *private*, в похідному класі недоступні. Тому в методі *Passport* для доступу до полів *name, health* і *ammo* довелося використовувати відповідні властивості базового класу. Інше рішення полягає в тому, щоб визначити ці поля зі специфікатором *protected,* в цьому випадку вони будуть доступні методам всіх класів, похідних від *Monster*. Обидва рішення мають свої переваги і недоліки.

Під час виконання програми об'єкти зберігаються в окремих змінних, масивах або інших колекціях. У багатьох випадках зручно оперувати об'єктами однієї ієрархії одноманітно, тобто використовувати один і той же програмний код для роботи з екземплярами різних класів. Бажано мати можливість описати:

- об'єкт, в який під час виконання програми заносяться посилання на об'єкти різних класів ієрархії;
- контейнер, в якому зберігаються об'єкти різних класів, які відносяться до однієї ієрархії;
- метод, в який можуть передаватися об'єкти різних класів ієрархії;
- метод, з якого залежно від типу об'єкту, який викликав його, викликаються відповідні методи.

Все це можливо завдяки тому, що об'єкту базового класу можна привласнити об'єкт похідного класу .

Давайте спробуємо описати масив об'єктів базового класу і занести туди об'єкти похідного класу. У лістингу 8.2 в масиві типу *Monster* зберігаються два об'єкти типу *Monster* і один - типу *Daemon*

Лістинг 8.2. Масив об’єктів різних типів

```
using System;
namespace examp48
{
   class Monster
   {
// Див. лістинг 8.1
   }

   class Daemon : Monster
   {
// Див. лістинг 8.1
   }

   class Classl
   {
      static void Main()
        {
         const int n = 3;
         Monster[] stado = new Monster[n];
         stado[0] = new Monster("Monia");
         stado[1] = new Monster("Monk");
         stado[2] = new Daemon ("Dimon", 3);
         foreach (Monster elem in stado) elem.Passport(); //1
         for (int i = 0; i < n; ++i) stado[i].ammo = 0;   //2
```

```
            Console.WriteLine();
            foreach (Monster elem in stado) elem.Passport();   //3
        }
    }
}
```

Результат роботи програми:

```
Monster Monia      health = 100 ammo = 100
Monster Monk       health = 100 ammo = 100
Monster Dimon      health = 100 ammo = 100

Monster Monia      health = 100 ammo = 0
Monster Monk       health = 100 ammo = 0
Monster Dimon      health = 100 ammo = 0
```

Об'єкт типу *Daemon* дійсно можна помістити в масив, що складається з елементів типу *Monster*, але для нього викликаються тільки методи і властивості, успадковані від предка. Це влаштовує нас в операторові 2, а в операторах 1 і 3 хотілося б, щоб викликався метод *Passport*, перевизначений в нащадку.

Отже, привласнювати об'єкту базового класу об'єкт похідного класу можна, але викликаються для нього тільки методи і властивості, визначені в базовому класі. Іншими словами, можливість доступу до елементів класу визначається типом посилання, а не типом об'єкту, на який вона вказує.

Це і зрозуміло: адже компілятор повинен ще до виконання програми вирішити, який метод викликати, і вставити в код фрагмент, передавальний управління на цей метод (цей процес називається *раннім зв'язуванням*). При цьому компілятор може керуватися тільки типом змінної, для якої викликається метод або властивість (наприклад, *stado[i].Ammo*). Те, що в цій змінній в різні моменти часу можуть знаходитися посилання на об'єкти різних типів, компілятор врахувати не може.

Отже, якщо ми хочемо, щоб методи, що викликаються, відповідали типу об'єкту, необхідно відкласти процес *зв'язування* до етапу виконання програми, а точніше - до моменту виклику методу, коли вже точно відомо, на об'єкт якого типу вказує посилання. Такий механізм в C# є - він називається *пізнім зв'язуванням* і реалізується за допомогою так званих віртуальних методів.

## 8.2. Віртуальні методи

Для позначення віртуальних функцій використовується ключове слово virtual. Воно записується в заголовку методу базового класу, наприклад:

```
virtual public void Passport( )
```

Слово *virtual* в перекладі з англійського означає "фактичний". Оголошення методу віртуальним означає, що всі посилання на цей метод вирішуватимуться

за фактом його виклику, тобто не на стадії компіляції, а під час виконання програми. Цей механізм називається пізнім *зв'язуванням.*

Для його реалізації необхідно, щоб адреси віртуальних методів зберігалися там, де ними можна буде у будь-який момент скористатися. Компілятор формує для цих методів таблицю віртуальних методів (*Virtual Method Table, VMT*) . У цю таблицю записуються адреси віртуальних методів (зокрема успадкованих) в порядку опису в класі. Для кожного класу створюється одна таблиця.

Кожен об'єкт під час виконання повинен мати доступ до *VMT*. Забезпечення цього зв'язку не можна доручити компілятору, оскільки він повинен встановлюватися під час виконання програми при створенні об'єкту. Тому зв'язок екземпляра об'єкту з *VMT* встановлюється за допомогою спеціального коду, що автоматично поміщається компілятором в конструктор об'єкту.

Якщо в похідному класі потрібно перевизначити віртуальний метод, використовується ключове слово override, наприклад:

```
override public void Passport( )
```

Перевизначений віртуальний метод повинен володіти таким же набором параметрів, як і однойменний метод базового класу. Це вимога цілком природна, якщо врахувати, що однойменні методи, що відносяться до різних класів, можуть викликатися з однієї і тієї ж точки програми.

Додамо в лістинг 8.2 два чарівні слова - virtual і override - в описи методів Passport, відповідно, базового і похідного класів (лістинг 8.3).

Лістинг 8.3. Віртуальні методи

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

namespace examp49
{
   class Monster
   {
...
      virtual public void Passport()
      {
            Console.WriteLine(
            "Monster {0} \t health = {1} ammo = {2}",
            name, health, ammo);
      }

      protected string name; // закриті поля
      public int health, ammo;
   }

 class Daemon : Monster
```

```
    {
      override public void Passport()
      {
        Console.WriteLine(
        "Daemon {0} \t health = {1} ammo = {2} brain = {3}", name, health, ammo, brain);
      }
    class Classl
    {
      static void Main()
      {
        const int n = 3;
        Monster[] stado = new Monster[n];
        stado[0] = new Monster("Monia");
        stado[1] = new Monster("Monk");
        stado[2] = new Daemon("Dimon", 3);
        foreach (Monster elem in stado) elem.Passport();
        for (int i = 0; i < n; ++i) stado[i].ammo = 0;
        Console.WriteLine();
        foreach (Monster elem in stado) elem.Passport();
      }
    }
}
```

Результат роботи програми:

```
Monster Monia          health = 100 ammo = 100
Monster Monk           health = 100 ammo = 100
Daemon Dimon           health =  100 ammo = 100 brain = 3

Monster Monia          health = 100 ammo = 0
Monster Monk           health = 100 ammo = 0
Daemon Dimon           health = 100 ammo = 0   brain = 3
```

Віртуальні методи базового класу визначають інтерфейс всієї ієрархії. Цей інтерфейс може розширюватися в нащадках за рахунок додавання нових віртуальних методів.

Виклик віртуального методу виконується так: з об'єкту береться адреса його таблиці *VMT*, з *VMT* вибирається адреса методу, а потім управління передається цьому методу. Таким чином, при використанні віртуальних методів зі всіх однойменних методів ієрархії завжди вибирається той, який відповідає фактичному типу об'єкту, що викликав його.

Виклик віртуального методу, на відміну від звичайного, виконується через додатковий етап отримання адреси методу з таблиці *VMT*, що декілька уповільнює виконання програми.

За допомогою віртуальних методів реалізується один з основних принципів об'єктно-орієнтованого програмування - поліморфізм. Це слово в перекладі з грецького означає "багато форм", що в даному випадку означає "один виклик - багато методів". Застосування віртуальних методів забезпечує гнучкість і можливість розширення функціональності класу.

Віртуальні методи незамінні і при передачі об'єктів в методи як параметри. У параметрах методу описується об'єкт базового типу, а при виклику в ньому передається об'єкт похідного класу. В цьому випадку віртуальні методи, що викликаються для об'єкту з методу, відповідатимуть типу аргументу, а не параметра.

При описі класів рекомендується визначати як віртуальних ті методи, які в похідних класах повинні реалізовуватися по-іншому. Якщо у всіх класах ієрархії метод виконуватиметься однаково, його краще визначити як звичайний метод.

## 8.3. Абстрактні класи

При створенні ієрархії об'єктів для виключення коду, що повторюється, часто буває логічно виділити їх загальні властивості в один батьківський клас. При цьому може опинитися, що створювати екземпляри такого класу не має сенсу, тому що ніякі реальні об'єкти їм не відповідають. Такі класи називають абстрактними.

Абстрактний клас служить тільки для породження нащадків. Як правило, в ньому задається набір методів, які кожен з нащадків реалізовуватиме по-своєму. Абстрактні класи призначені для представлення загальних понять, які передбачається конкретизувати в похідних класах.

Абстрактний клас задає інтерфейс для всієї ієрархії, при цьому методам класу може не відповідати ніяких конкретних дій. В цьому випадку методи мають порожнє тіло і оголошуються зі специфікатором *abstract*.

Якщо в класі є хоч би один абстрактний метод, весь клас також має бути описаний як абстрактний. Приклад приведений на лістингу 8.4.

Лістинг 8.4. Абстрактний клас

```
using System;

namespace examp50
{
   abstract class Spirit
   {
      public abstract void Passport();
   }

   class Monster : Spirit
   {
      public Monster()
      {
         this.name = "Noname";
         this.health = 100;
         this.ammo = 100;
      }

      public Monster(string name)
         : this()
```

```
    {
        this.name = name;
    }
    public Monster(int health, int ammo, string name)
    {
        this.name = name;
        this.health = health;
        this.ammo = ammo;
    }
    public string GetName()
    {
        return name;
    }

    public int GetHealth()
    {
        return health;
    }

    public int GetAmmo()
    {
        return ammo;
    }
    override public void Passport()
    {
        Console.WriteLine("Monster {0} \t health = {1} ammo = {2}", name, health, ammo);
    }

    public string name; // закриті поля
    public int health, ammo;
}

class Daemon : Monster
{
    public Daemon()
    {
        brain = 1;
    }

    public Daemon(string name, int brain)
        : base(name)  // 1
    {
        this.brain = brain;
    }

    public Daemon(int health, int ammo, string name, int brain)
        : base(health, ammo, name)  // 2
    {
        this.brain = brain;
    }
```

```
        override public void Passport()  // 3
        {
            Console.WriteLine(
            "Daemon {0} \t health = {1} ammo = {2} brain = {3}",
            name, health, ammo, brain);
        }

        int brain; // закрите поле
    }

    class Classl
    {
        static void Main()
        {
            Daemon Dim1 = new Daemon("Dima", 3);
            Dim1.Passport();

            Monster Dim2 = new Daemon("Dima", 8);
            Dim2.Passport();

            Monster Dim3 = new Monster();
            Dim3.Passport();

        }
    }
}
```

Результат роботи програми:

```
Daemon Dima        health = 100 ammo =  100 brain = 3
Daemon Dima        health = 100 ammo =  100 brain = 8
Monster Noname     health = 100 ammo =  100
```

Абстрактні класи використовуються при роботі із структурами даних, призначеними для зберігання об'єктів однієї ієрархії, і як параметри методів. Якщо клас, похідний від абстрактного, не перевизначає всі абстрактні методи, він також повинен описуватися як абстрактний.

Можна створити метод, параметром якого є абстрактний клас. На місце цього параметра при виконанні програми може передаватися об'єкт будь-якого похідного класу. Це дозволяє створювати поліморфні методи, що працюють з об'єктом будь-якого типу в межах однієї ієрархії. Поліморфізм в різних формах є могутнім і широко вживаним інструментом *ООП.*

## 8.4. Безплідні класи

У C# є ключове слово sealed, що дозволяє описати клас, від якого, в протилежність абстрактному, успадковувати забороняється:

```
sealed class Spirit
{
.
}
```

// class Monster : Spirit { ... }помилка!

Більшість вбудованих типів даних описана як *sealed.* Якщо необхідно використовувати функціональність безплідного класу, застосовується не спадкоємство, а вкладення (включення): у класі описується поле відповідного типу.

Вкладення класів це коли один клас включає поля, що є класами. Наприклад, якщо є об'єкт "двигун", а потрібно описати об'єкт "літак", логічно зробити двигун полем цього об'єкту, а не його предком.

Оскільки поля класу зазвичай закриті, виникає питання, як же користуватися методами включеного об'єкту. Загальноприйнятий спосіб полягає в тому, щоб описати метод охоплюючого класу, з якого викликати метод включеного класу. Такий спосіб взаємин класів відомий як модель включення-делегування. Приклад приведений в лістингу 8.4.

Лістинг 8.4. Модель включення

```
using System;

namespace examp51
{
   class Двигатель
   {
      public void Запуск()
      {
         Console.WriteLine("вжжжж!!");
      }
   }

   class Самолет
   {
      public Самолет()
      {
         левый = new Двигатель();
         правый = new Двигатель();
      }
      public void Запустить_двигатели()
      {
         левый.Запуск();
         правый.Запуск();
      }
      Двигатель левый, правый;
   }

   class Class1
   {
      static void Main()
      {
      Самолет АН24_1 = new Самолет();
      АН24_1.Запустить_двигатели();
      }
   }
}
```

Результат роботи програми:
вжжжж!!
вжжжж!!

У методі “Запустити двигуни” запит на запуск двигунів передається, або, як прийнято говорити, делегується вкладеному класу.

## 8.5. Клас object

Кореневий клас *System.Object* всій ієрархії об'єктів *.NET*, названий в C# *object*, забезпечує всіх спадкоємців декількома важливими методами. Похідні класи можуть використовувати ці методи безпосередньо або перевизначати їх.

Клас object часто використовується і безпосередньо при описі типу параметрів методів для додання ним спільності, а також для зберігання посилань на об'єкти різного типу - таким чином реалізується поліморфізм.

Відкриті методи класу System.Object перераховані нижче.

1. Метод *Equals* з одним параметром повертає значення true, якщо параметр об'єкту, що визивається, посилаються на одну і ту ж область пам'яті. Синтаксис:

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

2. Метод *Equals* з двома параметрами повертає значення true, якщо обидва параметри посилаються на одну і ту ж область пам'яті. Синтаксис:

```
public static bool Equals( object ob1, object ob2);
```

3. Метод *GetHashCode* формує хеш-кодування-код об'єкту і повертає число, що однозначно ідентифікує об'єкт. Це число використовується в різних структурах і алгоритмах бібліотеки. Якщо перевизначається метод *Equals*, необхідно перенавантажувати і метод *GetHashCode*. Синтаксис:

```
public virtual int GetHashCode();
```

4. Метод *GetType* повертає поточний поліморфний тип об'єкту, тобто не тип посилання, а тип об'єкту, на який він в даний момент указує. Повертане значення має тип *Type*. Це абстрактний базовий клас ієрархії, що використовується для отримання інформації про типи під час виконання. Синтаксис:

```
public Type GetType();
```

5. Метод *ReferenceEquals* повертає значення *true*, якщо обидва параметри посилаються на одну і ту ж область пам'яті. Синтаксис:

public static bool ReferenceEquals(object obi, object ob2);

6. Метод *ToString* за умовчанням повертає для посилальних типів повне ім'я класу у вигляді рядка, а для значущих - значення величини, перетворене в рядок. Цей метод перевизначають для того, щоб можна було виводити інформацію про стан об'єкту. Синтаксис:

public virtual string ToString();

У похідних об'єктах ці методи часто перевизначають. Наприклад, можна перевизначити метод *Equals* для того, щоб задати власні критерії порівняння об'єктів, тому що часто буває зручнішим використовувати для порівняння не посилальну семантику (рівність посилань), а значущу (рівність значень).

Приклад застосування і перевизначення методів класу *object* для класу *Monster* приведений в лістингу 8.5.

Лістинг 8.5. Перевантаження методів класу *object*

```
using System;

namespace ConsoleApplicationl
{
   class Monster
   {
         public Monster(int health, int ammo, string name)
         {
            this.health = health;
            this.ammo = ammo;
            this.name = name;
         }
         public override bool Equals(object obj)
         {
           if ( obj == null || GetType() != obj.GetType() ) return false;

            Monster temp = (Monster) obj;
            return health == temp.health &&
            ammo == temp.ammo          &&
            name == temp.name;
         }
```

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

        public override string ToString()
        {
           return string.Format("Monster {0} \t health = {1} ammo = {2}",
           name, health, ammo);
        }
      string name;
      int health, ammo;
   }

   class Class1
   {
        static void Main()
        {
           Monster X = new Monster( 80, 80, "Вася" );
           Monster Y = new Monster( 80, 80, "Вася" );
           Monster Z = X;
           if ( X == Y ) Console.WriteLine("X == Y ");
           else Console.WriteLine("X != Y ") ;
           if ( X == Z ) Console.WriteLine(" X == Z ") ;
           else Console.WriteLine("X != Z ") ;
           if ( X.Equals(Y) ) Console.WriteLine( " X Equals Y " );
           else Console.WriteLine( " X not Equals Y " );
           Console.WriteLine(X.GetType());
        }
   }
}
```

Результат роботи програми:

X != Y
X = = Z
X Equals Y
ConsoleApplicationl.Monster

У методі *Equals* спочатку перевіряється переданий в нього аргумент. Якщо він рівний *null* або його тип не відповідає типу об'єкту, що викликав метод, повертається значення *false*. Значення *true* формується у разі попарної рівності всіх полів об'єктів.

Метод *GetHashCode* просто делегує свої функції відповідному методу одного з полів. Метод *ToString* формує рядок, що містить значення полів об'єкту.

Аналізуючи результат роботи програми, можна побачити, що в операції порівняння на рівність порівнюються посилання, а в перевантаженому методі *Equals* - значення.

## 8.6. Рекомендації по програмуванню

Спадкоємство класів надає програмістові великі можливості організації коду і його багатократного використання. Спадкоємство класу Y від класу X означає, що Y є різновидом класу X, тобто конкретнішу, приватну концепцію. Базовий клас є більш загальним поняттям, ніж Y. Скрізь, де можна використовувати X, можна використовувати і Y, але не навпаки (пригадайте, що на місце базового класу можна передавати будь-який з похідних). Необхідно пам'ятати, що під час виконання програми не існує ієрархії класів і передачі повідомлень об'єктам базового класу з похідних - є тільки конкретні об'єкти класів, поля яких формуються на основі ієрархії на етапі компіляції.

Головна перевага спадкоємства полягає в тому, що на рівні базового класу можна написати універсальний код, за допомогою якого працювати також з об'єктами похідного класу, що реалізується за допомогою віртуальних методів.

Як віртуальні мають бути описані методи, які виконують у всіх класах ієрархії одну і ту ж функцію, але, можливо, різними способами. Нехай, наприклад, всі об'єкти ієрархії повинні уміти виводити інформацію про себе. Оскільки ця інформація зберігається в різних полях похідних класів, функцію виводу не можна реалізувати в базовому класі. Природно назвати її у всіх класах однаково і оголосити як віртуальну з тим, щоб її можна було викликати залежно від фактичного типу об'єкту, з яким працюють через базовий клас.

Для представлення загальних понять, які передбачається конкретизувати в похідних класах, використовують абстрактні класи. Як правило, в абстрактному класі задається набір методів, тобто інтерфейс, який кожен з нащадків реалізовуватиме по-своєму.

Альтернативним спадкоємству механізмом використання одним класом іншого є вкладення, коли один клас є полем іншого. Вкладення представляє стосунки класів "Y містить X" або "Y реалізується за допомогою X".

Для вибору між спадкоємством і вкладенням служить відповідь на питання про те, чи може у Y бути декілька об'єктів класу X ("Y містить X"). Крім того, вкладення використовується замість спадкоємства тоді, коли про класи X і Y не можна сказати, що Y є різновидом X, але при цьому Y використовує частину функціональності X ("Y реалізується за допомогою X").

# РОЗДІЛ 9. ІНТЕРФЕЙСИ І СТРУКТУРНІ ТИПИ

У цьому розділі розглядаються спеціальні види класів: інтерфейси, структури і перелічення.

## 9.1. Синтаксис інтерфейсу

*Інтерфейс* є “крайнім випадком” абстрактного класу. У ньому задається набір абстрактних методів, властивостей і індексаторів, які мають бути реалізовані в похідних класах. Основна ідея використання інтерфейсу полягає в тому, щоб до об'єктів таких класів можна було звертатися однаковим чином.

Кожен клас може визначати елементи інтерфейсу по-своєму. Так досягається поліморфізм: об'єкти різних класів по-різному реагують на виклики одного і того ж методу.

Синтаксис інтерфейсу аналогічний синтаксису класу:

**[атрибути] [специфікатори] interface ім'я_інтерфейсу [:предки]**
**тіло_інтерфейсу [ ; ]**

Для інтерфейсу можуть бути вказані специфікатори, *new, public, protected, internal* і *private*. Специфікатор *new* застосовується для вкладених інтерфейсів і має такий же сенс, як і відповідний модифікатор методу класу. Решта специфікаторів управляє видимістю інтерфейсу. У різних контекстах визначення інтерфейсу допускаються різні специфікатори. За умовчанням інтерфейс доступний тільки із збірки, в якій він описаний (internal).

Інтерфейс може успадковувати властивості декількох інтерфейсів, в цьому випадку предки перераховуються через кому. Тіло інтерфейсу складають абстрактні методи, шаблони властивостей і індексаторів, а також події.

Інтерфейс не може містити константи, поля, операції, конструктори, деструктори, типи і будь-які статичні елементи.

Як приклад розглянемо інтерфейс IAction, що визначає базову поведінку персонажів комп'ютерної гри, що зустрічалися в попередніх розділах. Допустимо, що будь-який персонаж повинен уміти виводити себе на екран, атакувати і красиво вмирати:

```
interface IAction
{
void Draw();
int Attack(int a);
void Die();
int Power { get; set}
}
```

У інтерфейсі *IAction* задані заголовки трьох методів і шаблон властивості *Power*.

Відмінності інтерфейсу від абстрактного класу:

- елементи інтерфейсу за умовчанням мають специфікатор доступу *public* і не можуть мати специфікаторів, заданих явним чином;
- інтерфейс не може містити полів і звичайних методів - всі елементи інтерфейсу мають бути абстрактними;
- клас, в списку предків якого задається інтерфейс, повинен визначати всі його елементи, тоді як нащадок абстрактного класу може не перевизначати частину абстрактних методів предка (в цьому випадку похідний клас також буде абстрактним);
- клас може мати в списку предків декілька інтерфейсів, при цьому він повинен визначати всі їх методи.

У бібліотеці *.NET* визначена велика кількість стандартних інтерфейсів, які описують поведінку об'єктів різних класів. Наприклад, якщо потрібно порівнювати об'єкти за принципом більше або менше, відповідні класи повинні реалізувати інтерфейс *IComparable*. Ми розглянемо найбільш споживані стандартні інтерфейси в подальших розділах посібника.

## 9.2. Реалізація інтерфейсу

У списку предків класу спочатку вказується його базовий клас, якщо він є, а потім через кому - інтерфейси, які реалізує цей клас. Таким чином, в C# підтримується одиночне спадкоємство для класів і множинне - для інтерфейсів. Це дозволяє додати похідному класу властивості декількох базових інтерфейсів, реалізовуючи їх на свій розсуд.

Наприклад, реалізація інтерфейсу *IAction* в класі *Monster* показана на лістингу 9.1:

Лістинг 9.1. Реалізація інтерфейсу

```
using System;

namespace Iterface
{
interface IAction
{
void Draw();
int Attack(int a);
void Die();
void Passport();
int Power { get; set;}
}
   class Monster : IAction
   {
      public Monster (int health, int ammo, string name)
      {
         this.health = health;
         this.ammo = ammo;
         this.name = name;
      }
```

```
    public void Draw()
    {
      Console.WriteLine( "Тут був " + name );
    }
  public int Attack(int ammo_)
  {
     ammo -= ammo_;
     if (ammo > 0) Console.WriteLine("Ба-бах!");
     else ammo = 0;
     return ammo;
  }
    public void Die()
  {
     Console.WriteLine("Monster " + name + " RIP");
     health = 0;
  }
   public int Power
    {
     get
       {
        return ammo * health;
       }
     set
       {
          if (value > 0)
          ammo = health = value;
       }
   }
      public void Passport()
       {
        Console.WriteLine("Monster {0} \t health = {1} ammo = {2}",
        name, health, ammo);
       }
    string name;
    int health, ammo;
}
class Class1
     {
    public static void Act(IAction A)
    {
       A.Passport();
    }
    static void Main()
       {
          Monster X = new Monster( 80, 80, "Вася" ); // 1 Об'єкт класу
          X.Draw();
          IAction Y = new Monster( 80, 80, "Миша" ); // 2 Об'єкт типу інтерфейсу
          Y.Draw();
          Console.WriteLine("Power - " + Y.Power);
          Y.Power = 100;
          Y.Attack(-500);
          Y.Draw();
```

```
            Y.Passport();
            Act(Y);
          }
        }
}
```

Природно, що сигнатури методів в інтерфейсі і реалізації повинні повністю збігатися. Для елементів інтерфейсу, що реалізовуються, в класі слід указувати специфікатор *public*. До цих елементів можна звертатися двома способами: через об'єкт класу і через об'єкт типу відповідного інтерфейсу. Це показано відповідно в операторах 1 і 2.

Результат роботи програми:
Тут був Вася
Тут був Міша
Power - 6400
Ба – бах!
Тут був Міша
Monster Міша health = 100 ammo = 600
Monster Міша health = 100 ammo = 600

Існує другий спосіб реалізації інтерфейсу в класі: *явна вказівка імені інтерфейсу* перед елементом, що реалізовується. Специфікатори доступу при цьому не указуються. До таких елементів можна звертатися в програмі тільки через об'єкт типу інтерфейсу, наприклад:

```
class Monster : IAction
{
int IAction.Power
{
get
{
return ammo * health;
  }
  }

void IAction.Draw()
{
Console.WriteLine( " Тут був " + name );
}
…
}
…
IAction Actor = new Monster(10,10,"Маша");
Actor.Draw();                  // звернення через об'єкт типу інтерфейсу
```

```
// Monster Vasia = new Monster(50, 50, "Вася" );
// Vasia.Draw();      помилка!
```

Таким чином, при явному завданні імені інтерфейсу, що реалізовується, відповідний метод не входить в інтерфейс класу. Це дозволяє спростити його в тому випадку, якщо якісь елементи інтерфейсу не потрібні кінцевому користувачеві класу.

Крім того, явне завдання імені інтерфейсу, що реалізовується, перед ім'ям методу дозволяє уникнути конфліктів при множинному спадкоємстві, якщо елементи з однаковими іменами або сигнатурою зустрічаються більш ніж в одному інтерфейсі. Хай, наприклад, клас *Monster* підтримує два інтерфейси: один для управління об'єктами, а інший для тестування:

```
interface ITest
{
void Draw();
}

interface IAction
{
void Draw();
int Attack( int a );
void Die();
int Power { get; }
}

class Monster : IAction, ITest
{
void ITest.Draw()
{
Console.Writel_ine("Testing " + name );
}
void IAction.Draw()
{
Console.WriteLine( " Тут був " + name );
}
…
}
```

Обидва інтерфейси містять метод *Draw* з однією і тією ж сигнатурою. Розрізняти їх допомагає явна вказівка імені інтерфейсу. Звертатися до цих методів можна, використовуючи операцію приведення типу, наприклад:

```
Monster Vasia = new MONSTER( 50,   50,   "Вася" );
((Itest)Vasia).Draw();       // результат:  Тут був Вася
((Iaction)Vasia).Draw();   // результат:  Testing Вася
```

Втім, якщо від таких методів не потрібна різна поведінка, можна реалізувати метод першим способом (із специфікатором public), компілятор не заперечує

```
:
class Monster: IAction. ITest
{
public void Draw()
{
Console.WriteLine( " Тут був " + name ):
}
…
}
```

До методу *Draw*, описаного таким чином, можна звертатися будь-яким способом: через об'єкт класу *Monster*, через інтерфейс *IAction або ITest.*

Конфлікт виникає в тому випадку, якщо компілятор не може визначити з контексту звернення до елементу, елемент якого саме з інтерфейсів, що реалізовуються, потрібно викликати. При цьому завжди допомагає явне завдання імені інтерфейсу.

### 9.3. Робота з об'єктами через інтерфейси. Операції is і as

При роботі з об'єктом через об'єкт типу інтерфейсу буває необхідно переконатися, що об'єкт підтримує даний інтерфейс. Перевірка виконується за допомогою бінарної операції *is*. Ця операція визначає, чи сумісний поточний тип об'єкту, що знаходиться зліва від ключового слова *is*, з типом, заданим праворуч.

Результат операції рівний true, якщо об'єкт можна перетворити до заданого типу, і false – в іншому випадку. Операція зазвичай використовується в наступному контексті:

```
if ( об’єкт is тип )
{
//  виконати  перетворення  "об'єкту" до "типу"
//  виконати дії з перетвореним об'єктом
}
```

Допустимо, ми оформили якісь дії з об'єктами у вигляді методу з параметром типу *object*. Перш ніж використовувати цей параметр усередині методу для звернення до методів, описаних в похідних класах, потрібно виконати перетворення до похідного класу. Для безпечного перетворення слід перевірити, чи можливо воно, наприклад так:

```
static vod Act(object A)
{
if ( A is  IAction )
{
```

```
    IAction Actor = (IAction) A;
    Actor.Draw();
}
}
```

У метод *Act* можна передавати будь-які об'єкти, але на екран будуть виведені тільки ті, які підтримують інтерфейс *IAction*.

Недоліком використання операції *is* є те, що перетворення фактично виконується двічі: при перевірці і при власне перетворенні. Ефективнішою є інша операція - *as*. Вона виконує перетворення до заданого типу, а якщо це неможливо, формує результат null, наприклад:

```
static void Act(object A)
{
IAction Actor = A as IAction;
if (Actor != null )Actor.Draw();
}
```

Обидві розглянуті операції застосовуються як до інтерфейсів, так і до класів.

## 9.4. Інтерфейси і спадкоємство

Інтерфейс може не мати або мати скільки завгодно інтерфейсів-предків, в останньому випадку він успадковує всі елементи всіх своїх базових інтерфейсів, починаючи з самого верхнього рівня. Базові інтерфейси мають бути доступні як і їх нащадки. Як і в звичайній ієрархії класів, базові інтерфейси визначають загальну поведінку, а їх нащадки конкретизують і доповнюють його. У інтерфейсі-нащадку можна також вказати елементи, які змінюють успадковані елементи з такою ж сигнатурою. В цьому випадку перед елементом указується ключове слово new, як і в аналогічній ситуації в класах. За допомогою цього слова відповідний елемент базового інтерфейсу приховується. Приклад:

```
interface IBase
{
void F(int i );
}
interface  Ileft : IBase
{
new void F( int i );  // перевизначення методу F
{

interface Iright : IBase
{
void G();
}
interface IDerived : ILeft, IRight {}
```

```
class A
{
void Test( IDerived d )     {
d.F(1); //       Викликається      ILeft.F
((IBase)d). F(1):     //       Викликається      IBase.F
((ILeft )d). F(1);     //'       Викликається      ILeft. F
((IRight)d).F(1);     //       Викликається      IBase.F
}
}
```

Метод F з інтерфейсу IBase прихований інтерфейсом ILeft, не дивлячись на те що в ланцюжку IDerived - IRight - IBase він не перевизначався.

Клас, що реалізовує інтерфейс, повинен визначати всі його елементи, зокрема успадковані. Якщо при цьому явно указується ім'я інтерфейсу, воно повинне посилатися на той інтерфейс, в якому був описаний відповідний елемент, наприклад:

```
class A : IRight
{
IRight.G()   { ... }
IBase.F( int i )   { ... }       // IRight.F( int i )  - не можна
}
```

Інтерфейс, на власні або успадковані елементи якого є явне посилання, має бути вказаний в списку предків класу, наприклад:

```
class B : A
{
  //       IRight.G() { ... }    не можна!
}

class C : A, IRight
{
IRight.G() { ... }      // можна
IBase.F( int i ) { ... }        // можна
}
```

Клас успадковує всі методи свого предка, зокрема ті, які реалізовували інтерфейси. Він може перевизначити ці методи за допомогою специфікатора *new*, але звертатися до них можна буде тільки через об'єкт класу. Якщо використовувати для звернення посилання на інтерфейс, викликається не перевизначена версія:

```
interface IBase
{
 void A();
}
```

```
class Base : IBase
{
public void A() { ... }
}
class Derived: Base
{
new public void A() { ... }
}
...
Derived d = new Derived ();
d.A();  // викликається Derived.A();
IBase id = d;
id.A(); // викликається Base.A();
```

Проте якщо інтерфейс реалізується за допомогою віртуального методу класу, після його перевизначення в нащадку будь-який варіант звернення (через клас або через інтерфейс) приведе до одного і того ж результату:

```
interface IBase
{
 void A();
}
class Base : IBase
{
public virtual void A() { ... }
}
class Derived: Base
{
  Public override void A() { ... }
}
...
Derived d = new Derived ();
d.A();  // викликається Derived.A();
IBase id = d;
id.A(); // викликається Base.A();
```

Метод інтерфейсу, реалізований явною вказівкою імені, оголошувати віртуальним забороняється. При необхідності перевизначити в нащадках його поведінку користуються наступним прийомом: з цього методу викликається інший, захищений метод, який оголошується віртуальним. У приведеному далі прикладі метод *A* інтерфейсу *IBase* реалізується за допомогою захищеного віртуального методу *A,* який можна перевизначати в нащадках класу *Base*:

```
interface IBase
{
void A();
}

class Base : IBase
{
void IBase.A() { A_(); }
protected virtual void A_() { ... }
}

class Derived: Base
{
protected override void A() { ... }
}
```

Існує можливість *повторно реалізувати інтерфейс*, вказавши його ім'я в списку предків класу разом з класом-предком, що вже реалізував цей інтерфейс. При цьому реалізація перевизначених методів базового класу до уваги не береться:

```
interface IBase
{
void A();
}

class Base : IBase
{
  void  IBase.A() { ... }      // не використовується в Derived
}

class Derived : Base, IBase
{
public void A() { ... }
}
```

Якщо клас успадковує від класу і інтерфейсу, які містять методи з однаковими сигнатурами, успадкований метод класу сприймається як реалізація інтерфейсу, наприклад:

```
interface Interfacel
{
void F( ):
}
```

```
class Class1
{
public void    F() { ... }
public void    G{} { ... }
}

class Class2 : Class1, Interface1
{
new public void G( ) { ... }
}
```

Тут клас *Class2* успадковує від класу *Class1* метод *F*. Інтерфейс *Interface1* також містить метод *F*. Компілятор не видає помилку, тому що клас *Class2* містить метод, відповідний для реалізації інтерфейсу.

Взагалі при реалізації інтерфейсу враховується наявність "відповідних" методів в класі незалежно від їх походження. Це можуть бути методи (описані в поточному або базовому класі) які реалізують інтерфейс явним або неявним чином.

## 9.5. Стандартні інтерфейси .NET

У бібліотеці класів *.NET* визначена множина стандартних інтерфейсів, задаючих бажану поведінку об'єктів. Наприклад, інтерфейс *IComparable* задає метод порівняння об'єктів за принципом більше або менше, що дозволяє виконувати їх сортування. Реалізація інтерфейсів *IEnumerable* і *IEnumerator* дає можливість проглядати вміст об'єкту за допомогою конструкції *foreach*, а реалізація інтерфейсу *ICloneable* - клонувати об'єкти.

Стандартні інтерфейси підтримуються багатьма стандартними класами бібліотеки. Наприклад, робота з масивами за допомогою циклу *foreach* можлива саме тому, що тип *Array* реалізує інтерфейси *IEnumerable* і *IEnumerator*. Можна створювати і власні класи, що підтримують стандартні інтерфейси, що дозволить використовувати об'єкти цих класів стандартними способами.

### 9.5.1. Порівняння об'єктів (інтерфейс IComparable)

Інтерфейс *IComparable* визначений в просторі імен System. Він містить всього один метод CompareTo, що повертає результат порівняння двох об'єктів, - поточного і переданого йому як параметр:

```
interface IComparable
{
int CompareTo( object obj )
}
```

Метод повинен повертати:

- 0, якщо поточний об'єкт і параметр рівні;
- від’ємне число, якщо поточний об'єкт менше параметра;
- додатне число, якщо поточний об'єкт більше параметра.

Реалізуємо інтерфейс *IComparable* в знайомому нам класі *Monster.* Як критерій порівняння об'єктів виберемо поле *health*. У лістингу 9.2 приведена програма, що сортує масив монстрів за збільшенням величини, що характеризує їх здоров'я (елементи класу, що не використовуються в даній програмі, не приводяться).

Лістинг 9.2. Приклад реалізації інтерфейсу *IComparable*

```
using System;

// Стандартні інтерфейси .NET

namespace examp55
{
   class Monster : IComparable
   {
      public Monster(int health, int ammo, string name)
      {
         this.health = health ;
         this.ammo = ammo;
         this.name = name;
      }

      virtual public void Passport()
      {
         Console.WriteLine("Monster {0} \t healt h = {1} ammo = {2}",
         name, health, ammo);
      }
      public int CompareTo(object obj)          // реализація інтерфейсу
      {
         Monster temp = (Monster)obj;
         if (this.health > temp.health) return 1;
         if (this.health < temp.health) return -1;
         return 0;
      }
      string name;
      int health, ammo;
   }

   class Class1
   {
      static void Main()
      {
         const int n = 3;
         Monster[] stado = new Monster[n];
         stado[0] = new Monster(50, 50, "V");
         stado[1] = new Monster(80, 80, "P");
```

```
            stado[2] = new Monster(40, 10, "M");
            Array.Sort(stado);   // сортування стало можливим
            foreach (Monster elem in stado) elem.Passport();
        }
    }
}
```

Результат роботи програми:

```
Monster Маша      health = 40 ammo = 10
Monster Вася      health = 50 ammo = 50
Monster Петя      health = 80 ammo = 80
```

Якщо декілька об'єктів мають однакове значення критерію сортування, їх відносний порядок проходження після сортування не зміниться.

У багатьох алгоритмах потрібно виконувати сортування об'єктів по різних критеріях. У C# для цього використовується інтерфейс IComparer.

### 9.5.2 Сортування по різних критеріях (інтерфейс IComparer)

Інтерфейс *IComparer* визначений в просторі імен *System.Collections*. Він містить один метод *CompareTo*, що повертає результат порівняння двох об'єктів, переданих йому як параметри:

```
interface IComparer
{
int Compare ( object ob1, object ob2 )
}
```

Принцип застосування цього інтерфейсу полягає в тому, що для кожного критерію сортування об'єктів описується невеликий допоміжний клас, що реалізовує цей інтерфейс. Об'єкт цього класу передається в стандартний метод сортування масиву як другий аргумент (існує декілька перевантажених версій цього методу).

Приклад сортування масиву об'єктів з попереднього лістингу по іменах (властивість *Name*, клас *Sortbyname*) і кількості озброєнь (властивість *Ammo*, клас *Sortbyammo*) приведений в лістингу 9.3. Класи параметрів сортування оголошені вкладеними, оскільки вони потрібні тільки об'єктам класу *Monster*.

Лістинг 9.3. Сортування по двом критеріям

```
using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace examp56
{
   class Monster
```

```
{
   string name;
   int health;
   int ammo;

   public Monster(int health, int ammo, string name)
   {
      this.health = health;
      this.ammo = ammo;
      this.name = name;
   }

   public int Ammo
   {
      get { return ammo; }
      set
      {
         if (value > 0) ammo = value; else ammo = 0;
      }
   }
   public string Name
   {
      get { return name; }
   }
   virtual public void Passport()
   {
      Console.WriteLine("Monster {0} \t health - {1} ammo - {2}",
         name, health, ammo);
   }

   public class SortByName : IComparer
   {
      int IComparer.Compare(object ob1, object ob2)
      {
         Monster m1 = (Monster)ob1;
         Monster m2 = (Monster)ob2;
         return String.Compare(m1.Name, m2.Name);
      }
   }
      public class SortByAmmo : IComparer
      {
         int IComparer.Compare(object ob1, object ob2)
         {
            Monster m1 = (Monster)ob1;
            Monster m2 = (Monster)ob2;
            if (m1.Ammo > m2.Ammo) return 1;
            if (m1.Ammo < m2.Ammo) return -1;
            return 0;
         }
      }
   }
```

```
class Program
   {
      static void Main(string[] args)
      {
         const int n = 3;
         Monster[] stado = new Monster[n];
         stado[0] = new Monster(50, 50, "Вася");
         stado[1] = new Monster(80, 80, "Петя");
         stado[2] = new Monster(40, 10, "Маша");
         Console.WriteLine("Сортировка по имени:");
         Array.Sort(stado, new Monster.SortByName());
         foreach (Monster elem in stado) elem.Passport();
         Console.WriteLine("Сортировка по вооружению:");
         Array.Sort(stado, new Monster.SortByAmmo());
         foreach (Monster elem in stado) elem.Passport();
      }
   }
}
```

Результат роботи програми:
Сортування по імені:
Monster Вася health = 50 ammo 50
Monster Маша health = 40 ammo 10
Monster Петя health = 80 ammo 30

Сортування по озброєнню:
Monster Маша health = 40 ammo10
Monster Вася health = 50 ammo50
Monster Петя health = 80 ammo80

### 9.5.3 Перевантаження операцій відношення

Якщо клас реалізує інтерфейс IComparable, його екземпляри можна порівнювати між собою за принципом більше або менше. Логічно дозволити використовувати для цього операції відношення, які перенавантажують їх. Операції повинні перевантажуватися парами: < i >, <= i >= == i ! =. Перевантаження операцій зазвичай виконується шляхом делегування, тобто звернення до перевизначених методів *Compare* і *Equals*.

Якщо клас реалізує інтерфейс *IComparable*, потрібно перевизначити метод *Equals* і пов'язаний з ним метод *GetHashcode*. Обидва методи успадковано від базового класу *object*.

У лістингу 9.4 операцій відношення перевантажені для класу *Monster*. Як критерій порівняння об'єктів за принципом більше або менше виступає поле *health*, а при порівнянні на рівність реалізується значуща семантика, тобто попарно порівнюються всі поля об'єктів

Лістинг 9.4. Перевантаження операцій відношення

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

namespace examp57
{
class Monster : IComparable
 {
    string name;
    int health;
    int ammo;

    public Monster( int health, int ammo, string name )
       {
       this. health = health;
       this.ammo = ammo;
       this.name = name;
       }
   public override bool Equals( object obj )
   {
       if ( obj == null || GetType() != obj.GetType() ) return false;
       Monster temp = (Monster) obj;
       return health == temp.health
           && ammo == temp.ammo
           && name == temp.name;
   }
    public override int GetHashCode()
    {
       return name.GetHashCode();
    }
       public static bool operator ==( Monster a, Monster b )
       {
       return a.Equals( b );
       }
// варіант оператора:
//public static bool operator == ( Monster a. Monster b )
// {
//return ( a.CompareTo( b ) == 0 ):
// }


    public static bool operator != ( Monster a, Monster b )
    {
       return !a.Equals(b);
    }
// варіант:
//public static bool operator != ( Monster a, Monster b )
// {
//return ( a.CompareTo( b ) != 0 );
// }
```

```
public static bool operator < ( Monster a, Monster b )
{return ( a.CompareTo( b ) < 0);}

public static bool operator > ( Monster a, Monster b )
{return ( a.CompareTo( b ) > 0);}

public static bool operator <= (Monster a, Monster b)
{return ( a.CompareTo(b) <= 0 );}

public static bool operator >= ( Monster a, Monster b )
{return ( a.CompareTo( b ) >= 0);}

public int CompareTo( object obj )
{
   Monster temp = (Monster) obj; if ( this.health > temp.health ) return 1;
   if ( this.health < temp.health ) return -1;
   return 0;
}

}

   class Program
   {
      static void Main(string[] args)
      {
         Monster Вася = new Monster( 70, 80, "Вася" );
         Monster Петя = new Monster( 80, 80, "Петя" );
         if ( Вася > Петя ) Console.WriteLine ("Вася більше Петі" );
         else if ( Вася == Петя ) Console.WriteLine("Вася = Петя" );
         else       Console.WriteLine("Вася менше Петі");

      }
   }
}
```

Результат роботи програми:
Вася менше Петі

### 9.5.4. Клонування об'єктів (інтерфейс ICloneable)

*Клонування* - це створення копії об'єкту. Копія об'єкту називається клоном. Як вам відомо, при привласненні одного об'єкту посилального типу іншому копіюється посилання, а не сам об'єкт (рис. 9.1, *а*). Якщо необхідно скопіювати в іншу область пам'яті поля об'єкту, можна скористатися методом *MemberwiseClone*, який будь-який об'єкт успадковує від класу *object*. При цьому об'єкти, на які указують поля об'єкту, що у свою чергу є посиланнями, не копіюються (рис. 9.1, *б*). Це називається *поверхневим клонуванням*.

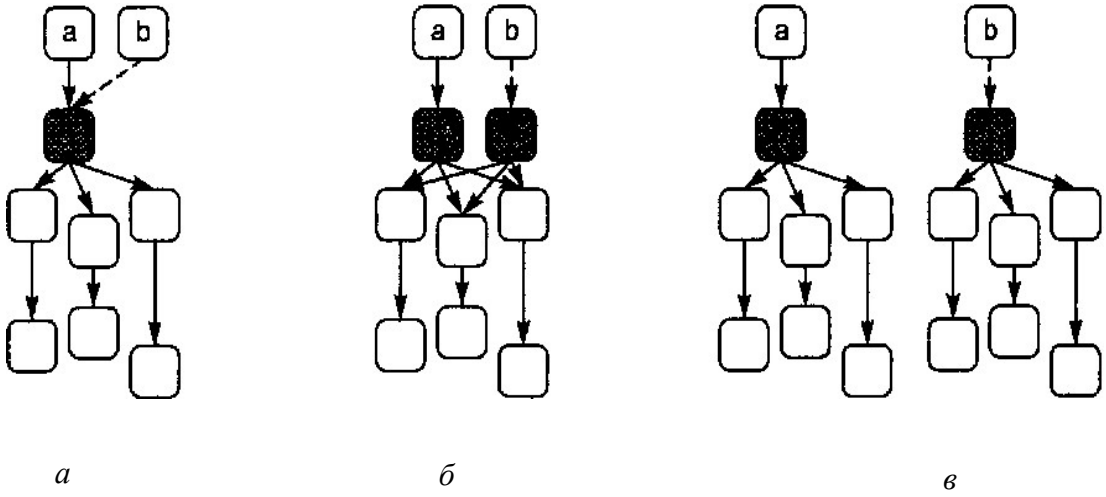



Рис. 9.1. Клонування об'єктів

Для створення повністю незалежних об'єктів необхідне *глибоке клонування*, коли в пам'яті створюється дублікат всього дерева об'єктів, тобто об'єктів, на які посилаються поля об'єкту, поля полів і так далі (рис. 9.1, в*)*. Алгоритм глибокого клонування дуже складний, оскільки вимагає рекурсивного обходу всіх посилань об'єкту і відстежування циклічних залежностей.

Об'єкт, що має власні алгоритми клонування, повинен оголошуватися як спадкоємець інтерфейсу *ICloneable* і перевизначати його єдиний метод *Clone.* У лістингу 9.4 приведений приклад створення поверхневої копії об'єкту класу *Monster* за допомогою методу *MemberwiseClone*, а також реалізований інтерфейс *ICloneable.* У демонстраційних цілях в ім'я клона об'єкту додано слово "Клон".

Метод *MemberwiseClone* можна викликати тільки з методів класу. Він не може бути викликаний безпосередньо, оскільки оголошений в класі *object* як захищений (*protected*).

Лістинг 9.5. Клонування об'єктів

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace examp58
{
   class Monster : ICloneable
   {
      string name;
      int health;
      int ammo;

      public Monster(int health, int ammo, string name)
      {
         this.health = health;
         this.ammo = ammo;
```

```
            this.name = name;
        }
        public Monster ShallowClone()   // поверхнева копія
        { return (Monster)this.MemberwiseClone(); }
        public object Clone()  // призначена копія для користувача
        {
            return new Monster(this.health, this.ammo, "Клон " + this.name);
        }
        virtual public void Passport()
        {
            Console.WriteLine("Monster {0} \t health = {1} ammo = {2}",
            name, health, ammo);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
        Monster Вася = new Monster(70, 80, "Вася" );
        Monster X = Вася;
        X.Passport();
        Monster Y = Вася.ShallowClone();
        Y.Passport();
        Monster Z = (Monster)Вася.Clone();
        Z.Passport();
        }
    }
}
```

Об'єкт *X* посилається на ту ж область пам'яті, що і об'єкт *Вася*. Отже, якщо ми внесемо зміни до одного з цих об'єктів, це відіб'ється на іншому. Об'єкти *Y* і *Z*, створені шляхом клонування, володіють власними копіями значень полів і незалежні від початкового об'єкту.

### 9.5.5. Перебір об'єктів (інтерфейс IEnumerable) і ітератори

Оператор *foreach* є зручним засобом перебору елементів об'єкту. Масиви і всі стандартні колекції бібліотеки *.NET* дозволяють виконувати такий перебір завдяки тому, що в них реалізовані інтерфейси *IEnumerable* і *IEnumerator*. Для застосування оператора *foreach* до призначеного для користувача типу даних потрібно реалізувати в нім ці інтерфейси. Давайте подивимося, як це робиться.

Інтерфейс *IEnumerable* (що перераховує) визначає всього один метод - *GetEnuraerator*, що повертає об'єкт типу *IEnumerator* (нумератор), який можна використовувати для проглядання елементів об'єкту.

Інтерфейс *IEnumerator* задає три елементи:

- властивість *Current*, що повертає поточний елемент об'єкту;
- метод *Movenext*, що просуває нумератор на наступний елемент об'єкту;
- метод *Reset*, що встановлює нумератор в початок перегляду.

Цикл *foreach* використовує ці методи для перебору елементів, з яких складається об'єкт.

Таким чином, якщо потрібно, щоб для перебору елементів класу міг застосовуватися цикл *foreach*, необхідно реалізувати чотири методи: *GetEnumerator, Current, MoveNext* і *Reset*. Наприклад, якщо внутрішні елементи класу організовані в масив, потрібно буде описати закрите поле класу, що зберігає поточний індекс в масиві, в методі *Movenext* задати зміну цього індексу на 1 з перевіркою виходу за межу масиву, в методі *Current* - повернення елементу масиву по поточному індексу і так далі.

Це не цікава робота, а виконувати її доводиться часто, тому і були введені засоби, що полегшують виконання перебору в об'єкті, - *ітератори*. *Ітератором* є блок коду, задаючий послідовність перебору елементів об'єкту. На кожному проході циклу *foreach* виконується один крок ітератора, що закінчується видачею чергового значення. Видача значення виконується за допомогою ключового слова *yield*.

Розглянемо створення ітератора на прикладі лістингу 9.6. Нехай потрібно створити об'єкт, що містить бойову групу екземплярів типу *Monster.* Для простоти обмежимо максимальну кількість бійців в групі десятьма.

Лістинг 9.6. Клас з ітератором

```
using System;
using System.Collections;
namespace examp59
{
   class Monster
   {
       string name;
       int health;
       int ammo;

       public Monster( int health, int ammo, string name )
          {
          this. health = health;
          this.ammo = ammo;
          this.name = name;
          }

       public Monster()
       {
          name = "ЛЮДА";
          ammo = 100;
          health = 100;
       }

       public Monster(string name)
       {
          this.name = name;
          ammo = 200;
          health = 200;
```

```
    }

    public void Passport()
    {
       Console.WriteLine(" {0}  {1}  {2}  ", name, ammo, health);

    }
}

class Daemon : Monster
{
public int brain;
public Daemon(): base()
    {
       brain = 1;
    }
}

class Stado : IEnumerable    // 1
{
   private Monster[] mas;
   private int n;

   public Stado()
   {
       mas = new Monster[10]; n = 0;
   }
   public IEnumerator GetEnumerator()
   {
       for (int i = 0; i < n; ++i)
          yield return mas[i];            // 2
   }
   public IEnumerable Backwards()// у зворотному порядку
   {
         for (int i = n - 1; i >= 0; --i ) yield return mas[i];
   }

   public IEnumerable MonstersOnly()       // тільки монстри
   {
       for ( int i = 0; i < n; ++i )
       if ( mas [i].GetType().Name == "Monster" ) yield return mas[i];
   }

   public void Add( Monster m )
   {
      if ( n >= 10 ) return;
      mas[n] = m;
      ++n;
   }
}
class Program
{
```

```
        static void Main(string[] args)
        {

        Stado s = new Stado();
        s.Add( new Monster());
        s.Add( new Monster("Вася"));
        s.Add( new Monster("Петя"));
        foreach ( Monster i in s )  i.Passport();
        foreach ( Monster i in s.Backwards())   i.Passport();
        foreach ( Monster i in s.MonstersOnly()) i.Passport();
        }
    }
}
```

Все, що потрібно зробити для підтримки перебору, - вказати, що клас реалізує інтерфейс *IEnumerable* (оператор 1), і описати *imepamop* (оператор 2). Доступ до нього може бути здійснений через методи *MoveNext* і *Current* інтерфейсу *IEnumerator*.

За кодом, приведеним в лістингу 9.6, стоїть велика внутрішня робота компілятора. На кожному кроці циклу *foreach* для ітератора створюється “оболонка” - службовий об'єкт, який запам'ятовує поточний стан ітератора і виконує все необхідне для доступу до елементів об'єкту, що проглядаються. Іншими словами, код, що становить ітератор, не виконується так, як він виглядає - у вигляді безперервної послідовності, а розбитий на окремі ітерації, між якими стан ітератора зберігається.

У лістингу 9.7 приведений приклад ітератора, що перебирає чотири задані рядки.

Лістинг 9.7. Простий ітератор

```
using System;
using System.Collections;
namespace examp60
{
    class Num : IEnumerable
    {
        public IEnumerator GetEnumerator()
        {
            yield return "one";
            yield return "two";
            yield return "three";
            yield return "oops";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
        foreach ( string s in new Num() ) Console.WriteLine( s );
```

```
        }
    }
}
```

Результат роботи програми:
one
two
three
oops

Наступний приклад демонструє перебір значень в заданому діапазоні (від 1 до 5):

```
using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace examp61
{
    class Program
    {

        public static IEnumerable Count( int from, int to )
        {
            from = 1;
            while ( from <= to ) yield return from++;
        }

        static void Main(string[] args)
        {
            foreach (int i in Count(1, 5)) Console.WriteLine(i);
        }
    }
}
```

Перевага використання ітераторів полягає в тому, що для одного і того ж класу можна задати різний порядок перебору елементів. У лістингу 9.8 описано дві додаткові стратегії перебору елементів класу *Stado*, введеного в лістингу 9.6, - перебір в зворотному порядку і вибірка тільки тих об'єктів, які є екземплярами класу *Monster* (для цього використаний метод отримання типу об'єкту *GetType*, успадкований від базового класу *object)*.

Лістинг 9.8. Реалізація декількох стратегій перебору

```
using System;
using System.Collections;

namespace examp62
{
    class Monster
```

```
{
   string name;
   int health;
   int ammo;
   public Monster(int health, int ammo, string name)
   {
      this.health = health;
      this.ammo = ammo;
      this.name = name;
   }
   public Monster()
   {
      name = "ЛЮДА";
      ammo = 100;
      health = 100;
   }
   public Monster(string name)
   {
      this.name = name;
      ammo = 200;
      health = 200;
   }
   public void Passport()
   {
      Console.WriteLine(" {0}  {1}  {2}  \n", name, ammo, health);

   }
}
class Daemon : Monster
{
   public int brain;
   public Daemon()
      : base()
   {
      brain = 1;
   }
}
class Stado : IEnumerable    // 1
{
   private Monster[] mas;
   private int n;
   public Stado()
   {
      mas = new Monster[10]; n = 0;
   }
   public IEnumerator GetEnumerator()
   {
      for (int i = 0; i < n; ++i)
         yield return mas[i];   // 2
   }
```

```
    public IEnumerable Backwards()// у зворотному порядку
    {
       for (int i = n - 1; i >= 0; --i) yield return mas[i];
    }

    public IEnumerable MonstersOnly()       // тільки монстри
    {
       for (int i = 0; i < n; ++i)
          if (mas[i].GetType().Name == "Monster") yield return mas[i];
    }
    public void Add(Monster m)
    {
       if (n >= 10) return;
       mas[n] = m;
       ++n;
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
       Stado s = new Stado();
       s.Add( new Monster());
       s.Add( new Monster("Вася"));
       s.Add( new Monster("Петя"));
       s.Add(new Daemon());
        foreach ( Monster i in s )                  i.Passport();
        foreach ( Monster i in s.Backwards())   i.Passport();
        foreach ( Monster i in s.MonstersOnly()) i.Passport();
    }
  }
}
```

Блок ітератора синтаксично є звичайним блоком і може зустрічатися в тілі методу, операції або частині *get* властивості, якщо відповідне повертане значення має тип *IEnumerable* або *IEnumerator*.

У тілі блоку ітератора можуть зустрічатися дві конструкції:

- yield return формує значення, що видається на черговій ітерації;
- yield break сигналізує про завершення ітерації.

Ключове слово yield має спеціальне значення для компілятора тільки в цих конструкціях.

Код блоку ітератора виконується не так, як звичайні блоки. Компілятор формує службовий об'єкт-нумератор, при виклику методу *MoveNext* якого виконується код блоку ітератора, що видає чергове значення за допомогою ключового слова *yield*. Наступний виклик методу *MoveNext* об'єкту-нумератора відновлює виконання блоку ітератора з моменту, на якому він був припинений в попередній раз.

## 9.6. Структури

Структура - тип даних, аналогічний класу, але ряд важливих відмінностей, що має, від нього:

- структура є значущою, а не посилальним типом даних, тобто екземпляр структури зберігає значення своїх елементів, а не посилання на них, і розташовується в стеку, а не в хіпові;
- структура не може брати участь в ієрархіях спадкоємства, вона може реалізовувати тільки інтерфейси;
- у структурі заборонено визначати конструктор за умовчанням, оскільки він визначений неявно і привласнює всім її елементам значення за умовчанням (нулі відповідного типу);
- у структурі заборонено визначати деструктори.

**Строго кажучи, будь-який значущий тип C# є структурним.**

Відмінності від класів обумовлюють *сферу застосування структур*: типи даних, що мають невелику кількість полів, з якими зручніше працювати як із значеннями, а не як з посиланнями. Накладні витрати на динамічне виділення пам'яті для невеликих об'єктів можуть значно понизити швидкодію програми, тому їх ефективніше описувати як структури, а не як класи.

З іншого боку, передача структури в метод за значенням вимагає і додаткового часу, і додаткової пам'яті.

Синтаксис структури:

**[атрибути ] [специфікатори] struct ім'я_ структури [ : інтерфейси ] тіло _структури**

*Специфікатори* структури мають такий же сенс, як і для класу. У специфікаторів доступу допускаються тільки *public, internal* і *private* (останній - тільки для вкладених структур).

*Інтерфейси*, що реалізовуються структурою, перераховуються через кому. Тіло структури може складатися з констант, полів, методів, властивостей, подій, індексаторів, операцій, конструкторів і вкладених типів. Правила їх опису і використання аналогічні відповідним елементам класів, за виключенням деяких відмінностей, витікаючих із згаданих раніше:

- оскільки структури не можуть брати участь в ієрархіях, для їх елементів не можуть використовуватися специфікатори *protected* і *protected internal*;
- структури не можуть бути абстрактними (*abstract*), до того ж за умовчанням вони безплідні (*sealed*);
- методи структур не можуть бути абстрактними і віртуальними;

- перевизначатися (тобто описуватися із специфікатором override) можуть тільки методи, успадковані від базового класу *object*;
- параметр *this* інтерпретується як значення, тому його можна використовувати для посилань, але не для привласнення;
- при описі структури не можна задавати значення полів за умовчанням - це буде зроблено в конструкторі за умовчанням, створюваному автоматично (конструктор привласнює значущим полям структури нулі, а посилальним - значення *null*).

У лістингу 9.9 приведений приклад опису структури, що представляє комплексне число. Для економії місця зі всіх операцій приведений тільки опис складання. Зверніть увагу на перевантажений метод *ToString*: він дозволяє виводити екземпляри структури на консоль, оскільки неявно викликається в методі *Console.WriteLine*. Використані в методі специфікатори формату описані в додатку.

Лістинг 9.9. Приклад структури

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

namespace examp63
{
    struct Complex
    {
      public double re, im;
      public Complex( double re_, double im_ )
      {
          re = re_; im = im_;      // можна використовувати this.re, this.im
      }
      public static Complex operator + ( Complex a, Complex b )
      {
return new Complex( a.re + b.re, a.im + b.im );
}

      public override string ToString()
      {
return (string.Format( "({0,2:0.##};{1,2:0.##})" , re, im ));
}
   }

    class Classl
   {
static void Main()
{
Complex a = new Complex( 1.2345 , 5.6 );
Console.WriteLine( "a = " + a );
```

```
Complex b;
b.re = 10; b.im = 1;
Console.WriteLine( "b = " + b );
Complex c = new Complex();
Console.WriteLine( "c = " + c );
c = a + b;
Console.WriteLine( "c = " + c );
}
}
}
```

Результат роботи програми:

```
a = (1.23; 5.6)
b = (10;  1)
c = (  0;  0)
c = ( 11.23;6.6)
```

При виведенні екземпляра структури на консоль виконується *упаковка*, тобто неявне перетворення в посилальний тип. При зворотному перетворенні - з посилального типу в структурний - виконується розпаковування.

*Привласнення структур* має, що природно, значущу семантику, тобто при привласненні створюється копія значень полів. Те ж саме відбувається і при передачі структур як параметрів за значенням. Для економії ресурсів ніщо не заважає передавати структури в методи по посиланню за допомогою ключових слів *ref* або *out*.

Особливо значний виграш в ефективності можна отримати, використовуючи масиви структур замість масивів класів. Наприклад, для масиву з 100 екземплярів класу створюється 101 об'єкт, а для масиву структур - один об'єкт. Приклад роботи з масивом структур, описаних в попередньому лістингу:

```
Complex [] mas = new Complex[4];
for ( int i = 0;  i < 4;  ++i )
mas[i].re = i;
mas[i].im = 2*i;
foreach ( Complex elem in mas ) Console.WriteLine( elem );
```

Якщо помістити цей фрагмент замість тіла методу *Main* в лістингу 9.9, отримаємо наступний результат:

```
( 0 0)
( 1 2)
( 2 4)
( 3 6)
```

## 9.7. Перелічення

При написанні програм часто виникає потреба визначити декілька зв'язаних між собою іменованих констант, при цьому їх конкретні значення можуть бути не важливі. Для цього зручно скористатися переліченим типом даних, всі можливі значення якого задаються списком цілочисельних констант, наприклад:

enum Menu { Read, Write, Append, Exit }

enum Веселка { Червоний, Оранжевий, Жовтий, Зелений, Синій, Фіолетовий }

Для кожної константи задається її символічне ім'я. За умовчанням константам привласнюються послідовні значення типу int, починаючи з 0, але можна задати і власні значення, наприклад:

enum Nums { two = 2, three, four, ten = 10, eleven, fifty = ten + 40 };

Константам three і four привласнюються значення 3 і 4, константі eleven - 11. Імена констант усередині кожного перелічення мають бути унікальними, а значення можуть бути різними.

Перевага перелічення перед описом іменованих констант полягає в тому, що зв'язані константи наочніше; крім того, компілятор виконує перевірку типів, а інтегроване середовище розробки підказує можливі значення констант, виводячи їх список.

Синтаксис перелічення:

**[атрибути ] [специфікатори ] enum ім'я _ перелічення [ : базовий_тип ] тіло _ перелічення [ ; ]**

*Специфікатори* перелічення мають такий же сенс, як і для класу, причому допускаються тільки специфікатори new, public, protected, internal і *private*.

*Базовий тип* - це тип елементів, з яких побудовано перелічення. За умовчанням використовується тип int, але можна задати тип і явним чином, вибравши його серед цілочисельних типів (окрім char), а саме: *byte, sbyte, short, ushort, int, uint, long* і *ulong*. Необхідність в цьому виникає, коли значення констант неможливо або незручно представляти за допомогою типу *int*.

Тіло перелічення складається з імен констант, кожною з яких може бути привласнене значення. Якщо значення не вказане, воно обчислюється збільшенням одиниці до значення попередньої константи. Константи за умовчанням мають специфікатор доступу *public*.

Перелічення часто використовуються як вкладені типи, ідентифікуючи значення з якого-небудь обмеженого набору. Приклад такого перелічення приведений в лістингу 9.10.

Лістинг 9.10. Приклад перелічення

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

namespace examp64
{
   struct Боец
   {
      public enum Воинское_Звание
      { Рядовой, Сержант, Лейтенант, Майор, Полковник, Генерал }

      public string Фамилия;
      public Воинское_Звание Звание;
   }

   class Program
   {
      static void Main(string[] args)
      {
         Боец x;
         x.Фамилия = " Иванов ";
         x.Звание = Боец.Воинское_Звание.Сержант;
         Console.WriteLine(x.Звание + " " + x.Фамилия);
      }
   }
}
```

Результат роботи програми: Сержант Іванов

Перелічення зручно використовувати для представлення бітових прапорів, наприклад:

```
enum Flags : byte
{
b0, b1. b2, b3 = 0x04, b4 = 0x08, b5 = 0x10, b6 = 0x20, b7 = 0x40
}
```

### 9.7.1 Операції з переліченнями

Із змінними переліченого типу можна виконувати арифметичні операції ( +, - ++, --), логічні порозрядні операції (*, &, |, ~), порівнювати їх за допомогою операцій відношення (<, < =, >, >= = =, ! = ) і отримувати розмір в байтах (*sizeof*).

При використанні змінних переліченого типу в цілочисельних виразах і операціях привласнення потрібне явне *перетворення типу*. Змінній перелічено-

го типу можна привласнити будь-яке значення, уявне за допомогою базового типу, тобто не тільки одне із значень, що входять в тіло перелічення. Привласнюване значення стає новим елементом перелічення.

Приклад:

```
Flags a = Flags.b2 | Flags.b4;
Console.WriteLine( "a = {0}      {0,2:X}", a );
++a;
Console.WriteLine( "a =  {0}      {0.2:X}", a );
int x = (int) a;
Console.WriteLine( "x = {0}      {0.2:X}", x );
Flags b = (Flags) 65;
Console.WriteLine(  "b = {0}      {0.2:X}", b );
```

Результат роботи цього фрагмента програми ( {0,2:x} позначає шістнадцятирічний формат виведення):

```
a = 10  0A
a = 11  0B
x = 11  B
b = 65  41
```

Інший приклад використання операцій з переліченнями приведений в лістингу 9.11.

Лістинг 9.11. Операції з переліченнями

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

namespace examp64
{
   struct Боец
   {
      public enum Воинское_Звание
      { Рядовой, Сержант, Лейтенант, Майор, Полковник, Генерал }
      public string Фамилия;
      public Воинское_Звание Звание;
   }

   class Program
   {
      static void Main(string[] args)
      {
         Боец x;
         x.Фамилия = " Иванов ";
```

```
            x.Звание = Боец.Воинское_Звание.Сержант;
            for (int i = 1976; i < 2006; i += 5)
            {
               if (x.Звание < Боец.Воинское_Звание.Генерал) ++x.Звание;
               Console.WriteLine("Рік: {0} {1} {2}", i, x.Звание, x.Фамилия);
            }
         }
      }
   }
```

Результат роботи програми:

```
Рік      1976  Лейтенант Иванов
Рік      1981  Майор Иванов
Рік      1986  Полковник Иванов
Рік      1991  Генерал Иванов
Рік      1996  Генерал Иванов
Рік      2001  Генерал Иванов
```

### 9.7.2. Базовий клас System.Enum

Всі перелічення в C# є нащадками базового класу *System.Enum*, який забезпечує їх деякими корисними методами.

Статичний метод *GetName* дозволяє отримати символічне ім'я константи по її номеру, наприклад:

```
Console.WriteLine( Enum.GetName(typeof( Flags ),8));  // b4
Console.WriteLine(Enum.GetName(typeof(Боец.Воинское_Звание          ),1));
//Сержант
```

Статичні методи *GetNames* і *GetValues* формують, відповідно, масиви імен і значень констант, складових перелічення, наприклад:

```
Array names = Enum.GetNames(typeof(Flags));
Console.WriteLine("Кількість елементів в переліченні: " + names.Length );
foreach (string elem in names) Console.Write("    " + elem);
Array values = Enum.GetValues(typeof(Flags));
foreach (Flags elem in values ) Console.Write("    " + (byte) elem);
```

Статичний метод *IsDefined* повертає значення *true*, якщо константа із заданим символічним ім'ям описана у вказаному переліченні, і *false* в іншому випадку, наприклад:

```
if (Enum.IsDefined(typeof(Flags),   "b5"))
Console.WriteLine("Константа з ім'ям b5 існує " );
else Console.WriteLine("Константа з ім'ям b5 не існує " );
```

Статичний метод *GetUnderlyingType* повертає ім'я базового типу, на якому побудовано перелічення. Наприклад, для перелічення Flags буде отримано *System.Byte*:

```
Console.WriteLine(Enum.GetUnderlyingType(typeof(Flags) ) );
```

## 9.8. Рекомендації по програмуванню

Інтерфейси найчастіше використовуються для завдання загальних властивостей об'єктів різних ієрархій. Основна ідея інтерфейсу полягає в тому, що до об'єктів класів, що реалізовують інтерфейс, можна звертатися однаковим чином. При цьому кожен клас може визначати елементи інтерфейсу по-своєму.

Якщо якийсь набір дій має сенс тільки для якоїсь конкретної ієрархії класів, що реалізовують ці дії різними способами, доречніше задати цей набір у вигляді віртуальних методів абстрактного базового класу ієрархії.

У C# підтримується одиночне спадкоємство для класів і множинне - для інтерфейсів. Це дозволяє додати похідному класу властивості декількох базових інтерфейсів. Клас повинен визначати всі методи всіх інтерфейсів, які є в списку його предків.

У бібліотеці *.NET* визначена велика кількість стандартних інтерфейсів. Реалізація стандартних інтерфейсів у власних класах дозволяє використовувати для об'єктів цих класів стандартні засоби мови і бібліотеки.

Наприклад, для забезпечення можливості сортування об'єктів стандартними методами слід реалізувати у відповідному класі інтерфейси *IComparable* або *IComparer*. Реалізація інтерфейсів *IEnumerable* і *IEnumerator* дає можливість проглядати вміст об'єкту за допомогою конструкції *foreach*, а реалізація інтерфейсу *ICloneable* - клонувати об'єкти.

Використання *ітераторів* спрощує організацію перебору елементів і дозволяє задати для одного і того ж класу різні стратегії перебору.

Сфера застосування *структур* - типи даних, що мають невелику кількість полів, з якими зручніше працювати як із значеннями, а не як з посиланнями. Накладні витрати на динамічне виділення пам'яті для екземплярів невеликих класів можуть значно понизити швидкодію програми, тому їх ефективніше описувати як структури.

Перевага використання перелічення для опису зв'язаних між собою значень полягає в тому, що це наочніше і інкапсульовано, чим велика кількість іменованих констант. Крім того, компілятор виконує перевірку типів, а інтегроване середовище розробки підказує можливі значення констант, виводячи їх список.

# РОЗДІЛ 10. ДЕЛЕГАТИ, ПОДІЇ І ПОТОКИ ВИКОНАННЯ

У цьому розділі розглядаються делегати і події - два взаємозв'язані засоби мови C#, що дозволяють організувати ефективну взаємодію об'єктів. Крім того приводяться початкові відомості про розробку багатопотокових застосувань.

## 10.1. Делегати

Делегат - це вид класу, призначений для зберігання посилань на методи. Делегат можна передати як параметр, а потім викликати інкапсульований в ньому метод. Делегати використовуються для підтримки подій, а також як самостійна конструкція мови. Розглянемо спочатку другий випадок.

### 10.1.1. Опис делегатів

Опис делегата задає сигнатуру методів, які можуть бути викликані з його допомогою:

**[ атрибути ] [ специфікатори ] delegate тип ім'я_делегата ([ параметри ])**

*Специфікатори* делегата мають той же сенс, що і для класу, причому допускаються тільки специфікатори *new, public, protected, internal* і *private*.
*Тип* описує повертаєме значення методів, що викликаються за допомогою делегата, а необов'язковими параметрами делегата є параметри цих методів. Делегат може зберігати посилання на декілька методів і викликати їх по черзі. При цьому сигнатури всіх методів повинні збігатися.

Приклад опису делегата: public delegate void D  (int i );

Тут описаний тип делегата, який може зберігати посилання на методи, повертаючі void і що приймають один параметр цілого типу.
Делегат, як і всякий клас, є типом даних. Його базовим класом є клас System.Delegate, що забезпечує свого “нащадка” деякими корисними елементами, які ми розглянемо пізніше. Успадковувати від делегата не можна.

Оголошення делегата можна розміщувати безпосередньо в просторі імен або усередині класу.

### 10.1.2. Використання делегатів

Для того, щоб скористатися делегатом, необхідно створити його екземпляр і задати імена методів, на які він посилатиметься. При виклику екземпляра делегата викликаються всі задані в ньому методи.

- Делегати застосовуються в основному для наступних цілей:

- отримання можливості визначати метод, що викликається, не за допомогою компіляції, а динамічно під час виконання програми;
- забезпечення зв'язку між об'єктами за типом “джерело - спостерігач”;
- створення універсальних методів, в які можна передавати інші методи;
- підтримка механізму зворотних викликів.

Розглянемо спочатку приклад реалізації першої з цих цілей. У лістингу 10.1 оголошується делегат, за допомогою якого один і той же оператор використовується для виклику двох різних методів (Coo1 і Hack).

Лістинг 10.1. Просте використання делегата

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

namespace examp65
{

delegate void Del ( ref string s );                    // оголошення делегата

   class Classl
  {
   public static void C001 ( ref string s )            // метод 1
   {
      Console.WriteLine( "викликаний метод Cool" );
        s = s.Replace('l', '1').Replace('O', '0').Replace('o', '0');
   }

  public static void Hack ( ref string s )             // метод 2

   {
      char[] mas = s.ToCharArray();
      s = "";
      for (int i = 0; i < mas.Length; ++i)
           s+= ((i & 1) == 0)? char.ToUpper(mas[i]):mas[i];
   }

   static void Main()
   {
      string s = "cOol hackers";
      Del d;                                           // екземпляр делегата
 /*
      d = new Del(C001);                               // ініціалізація методом 1
      d += new Del(Hack);
 */

      d = null;
      for (  int i = 0; i < 2; i++)
      {
```

```
            if (i == 0) d = new Del(C001);
            if (i == 1)d = new Del(Hack);                    // ініціалізація методом 2
            d( ref s );                                 // використання делегата для виклику методів
            Console.WriteLine(s);
        }
      }
    }
}
```

Результат роботи програми:

```
cOOl  hackers
COOl hAcKeRs
```

Використання делегата має той же синтаксис, що і виклик методу. Якщо делегат зберігає посилання на декілька методів, вони викликаються послідовно в тому порядку, в якому були додані в делегат.

Додавання методу в список виконується або за допомогою методу *Combine*, успадкованого від класу *System.Delegate*, або, що зручніше, за допомогою перевантаженої операції складання. От як виглядає змінений метод *Main* з попереднього лістингу, в якому одним викликом делегата виконується перетворення початкового рядка відразу двома методами:

```
static void Main()
{
    string s = "cool hackers";
    Del d = new Del ( C00l ) ;
    d += new Del( Hack );// додавання методу в делегат
    d( ref s ) ;
    Console.WriteLine( s );        // результат: C001 hAcKeRs
}
```

При виклику послідовності методів за допомогою делегата необхідно враховувати наступне:

- сигнатура методів повинна в точності відповідати делегатові;
- методи можуть бути як статичними, так і звичайними методами класу;
- кожному методу в списку передається один і той же набір параметрів;
- якщо параметр передається по посиланню, зміни параметра в одному методі відіб'ються на його значенні при виклику наступного методу;
- якщо параметр передається з ключовим словом *out* або метод повертає значення, результатом виконання делегата є значення, сформоване останнім з методів списку (у зв'язку з цим рекомендується формувати списки тільки з делегатів, що мають повертаєме значення типу *void*);
- якщо в процесі роботи методу виникло виключення, не оброблене в тому ж методі, подальші методи в списку не виконуються, а відбувається пошук обробників в охоплюючих делегат блоках;
- спроба викликати делегат, в списку якого немає жодного методу, викликає генерацію виключення *System.NullReferenceException*.

### 10.1.3. Патерн “спостерігач”

Розглянемо застосування делегатів для забезпечення зв'язку між об'єктами за типом “джерело - спостерігач”. В результаті розбиття системи на множину спільно працюючих класів з'являється необхідність підтримувати узгоджений стан взаємозв'язаних об'єктів. При цьому бажано уникнути жорсткої зв'язаності класів, оскільки це часто негативно позначається на можливості багатократного використання коду.

Для забезпечення гнучкого, динамічного зв'язку між об'єктами під час виконання програми застосовується наступна стратегія. Об'єкт, званий джерелом, при зміні свого стану, який може представляти інтерес для інших об'єктів, посилає їм повідомлення. Ці об'єкти називаються спостерігачами. Отримавши повідомлення, спостерігач опитує джерело, щоб синхронізувати з ним свій стан. Прикладом такої стратегії може служити зв'язок об'єкту з різними його уявленнями, наприклад, зв'язок електронної таблиці із створеними на її основі діаграмами.

Програмісти часто використовують одну і ту ж схему організації і взаємодії об'єктів в різних контекстах. За такими схемами закріпилася назва патерни, або шаблони проектування. Описана стратегія відома під назвою патерн “спостерігач”.

Спостерігач (*observer*) визначає між об'єктами залежність типу «один до багатьох», так що при зміні стані одного об'єкту всі залежні від нього об'єкти отримують сповіщення і автоматично оновлюються. Розглянемо приклад (лістинг 10.2), в якому демонструється схема сповіщення джерелом трьох спостерігачів. Гіпотетична зміна стану об'єкту моделюється повідомленням «OOPS!». Один з методів в демонстраційних цілях зроблений статичним.

Лістинг 10.2. Сповіщення спостерігачів за допомогою делегата

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace examp66
{
   public delegate void Del(object o);                  // оголошення делегата

   class Subj                                           // клас-джерело
   {
      Del dels;                                         // оголошення екземпляра делегата

      public void Register (Del d )                     // реєстрація делегата
      {
          dels += d;
      }

      public void OOPS()                                // щось відбулося
       {
      Console.WriteLine("OOPS!");
```

```
            if ( dels != null ) dels(null);                    // сповіщення спостерігачів

            }
         }

        class ObsA      // клас-спостпрігач
        {
            public void Do( object o )                         // реакція на подію джерела
            {
            Console.WriteLine("Бачу, що OOPS!" );
            }
        }

        class ObsB                                             // клас-спостерігач
        {
            public static void See( object o )                 // реакція на подію джерела {
            {
            Console.WriteLine("Я теж бачу, що OOPS!");
            }
         }

          class Program
          {
            static void Main(string[] args)
            {
                Subj s = new Subj();   // об'єкт класу-джерела
               ObsA o1 = new ObsA();                           //          об'єкти
               ObsA o2 = new ObsA();                           // класу-спостерігача
               s.Register( new Del( o1.Do ) );                 // реєстрація методів
               s.Register( new Del( o2.Do ) );                 // спостерігачів в джерелі
               s.Register( new Del( ObsB.See ) );              // (екземпляри делегата )
               s.OOPS();                                       // ініціалізація події
            }
          }
}
```

У джерелі оголошується екземпляр делегата, в цей екземпляр заносяться методи тих об'єктів, які хочуть отримувати повідомлення про зміну стану джерела. Цей процес називається *реєстрацією делегатів*. При реєстрації ім'я методу додається до списку. Зверніть увагу: для статичного методу називається ім'я класу, а для звичайного методу - ім'я об'єкту. При настанні “години Х” всі зареєстровані методи по черзі викликаються через делегат.

Результат роботи програми:

OOPS!

Бачу, що OOPS!

Бачу, що OOPS!

Я теж бачу, що OOPS!

Для забезпечення зворотного зв'язку між спостерігачем і джерелом делегат оголошений з параметром типу *object*, через який в метод, що викликається, пе-

редається посилання на об'єкт. Отже, в методі, що викликається, можна отримувати інформацію про стан об'єкту, що викликається, і посилати йому повідомлення (тобто викликати методи цього об'єкту).

Зв'язок “джерело – спостерігач” встановлюється під час виконання програми для кожного об'єкту окремо. Якщо спостерігач більше не хоче отримувати повідомлення від джерела, можна видалити відповідний метод з писку делегата за допомогою методу *Remove* або перевантаженій операції віднімання, наприклад:

```
public void UnRegister( Del  d )                    // видалення делегата
{
    dels -= d;
}
```

### 10.1.4. Операції

Делегати можна порівнювати на *рівність* і *нерівність*. Два делегати рівні, якщо вони обидва не містять посилань на методи або якщо вони містять посилання на одні і ті ж методи в одному і тому ж порядку. Порівнювати можна навіть делегати різних типів за умови, що вони мають один і той же тип повертаємого значення і однакові списки параметрів.

Делегати, які розрізняються тільки іменами, вважаються за тих, які мають різні типи.

З делегатами одного типу можна виконувати операції простого і складного привласнення, наприклад:

```
Del d1 = new Del (o1.Do):                   // o1.Do
Del d2 = new Del (o2.Do);                   // o2.Do
Del d3 = d1 + d2;                           //  ol.Do і o2.Do
d3 += d1;                                   //  ol.Do. o2.Do і ol.Do
d3 -=  d2;                                  //  o1.Do і ol.Do
```

Ці операції можуть знадобитися, наприклад, в тому випадку, якщо в різних обставинах потрібно викликати різні набори і комбінації наборів методів.

Делегат, як і рядок *string*, є незмінним типом даних, тому при будь-якій зміні створюється новий екземпляр, а старий згодом видаляється складальником сміття.

### 10.1.5. Передача делегатів в методи

Оскільки делегат є класом, його можна передавати в методи як параметр. Цим забезпечується *функціональна параметризація*: у метод можна передавати не тільки різні дані, але і різні функції їх обробки. Функціональна параметризація застосовується для створення універсальних методів і забезпечення можливості зворотного виклику.

Як *універсальний метод* можна привести метод виведення таблиці значень функції. У метод передається діапазон значень аргументу, крок його зміни і вид обчислюваної функції. Цей приклад наводиться далі.

*Зворотним викликом* (*callback*) є виклик функції, яка передається в іншу функцію як параметр. Розглянемо рис. 10.1. Допустимо, в бібліотеці описана функція *A,* параметром якої є ім'я іншої функції. У коді описується функція з необхідною сигнатурою (B) і передається у функцію *A. Виконання* функції *A* приводить до виклику *B*, тобто управління передається з бібліотечної функції назад в код.

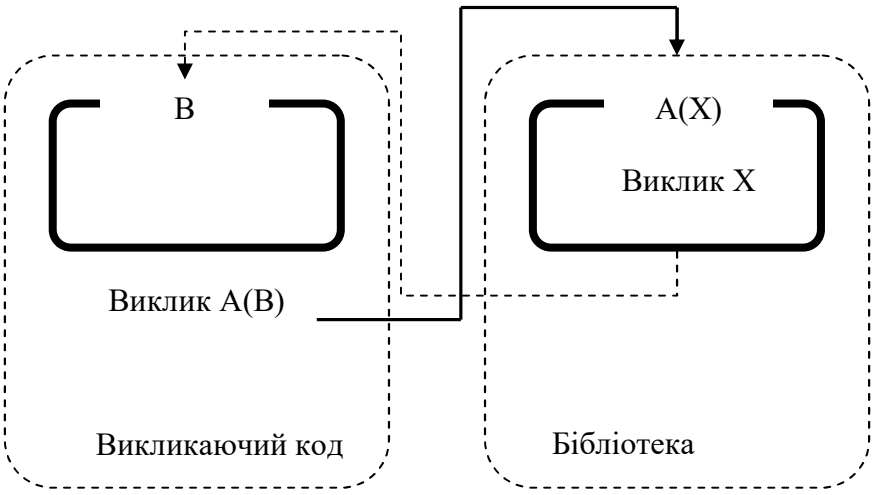



Рис. 10.1. Механізм зворотного виклику

Механізм зворотного виклику широко використовується в програмуванні. Наприклад, він реалізується в багатьох стандартних функціях *Windows*.

Приклад передачі делегата як параметр приведений в лістингу 10.3. Програма виводить таблицю значень функції на заданому інтервалі з кроком, рівним одиниці.

Лістинг 10.3. Передача делегата через список параметрів

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

namespace examp67
{
public delegate double Fun( double x );          // оголошення делегата

   class Classl
   {
     public static void Table (Fun F, double x, double b )
     {
        Console.WriteLine( "     X        Y     " );
        while (x <= b)
        {
```

```
                Console.WriteLine("| {0,8:0.000} | {1,8:0.000}", x, F(x));
                x += 1;
            }
            Console.WriteLine( " ....................." );
        }
        public static double Simple(double x)
        {
            return 1;
        }

        static void Main(string[] args)
        {
            Console.WriteLine("Таблиця функції Sin");
            Table(new Fun( Math.Sin), -2, 2 );
            Console.WriteLine(" Таблиця функції Simple " );
            Table( new Fun(Simple), 0, 3 );
        }
}
    }
```

Результат роботи програми:

```
Таблиця функції Sin
 -------X ------- Y --------
I   -2.000   I  -0,909    I
I   -1.000   I  -0,841    I
I    0.000   I   0,000    I
I    1,000   I   0.841    I
I    2,000   I   0,909    I
-------------------------------


Таблиця функції Simple
 -------X ------- Y -------
I   0.000   I  1,000    I
I   1.000   I  1,000    I
I   2.000   I  1,000    I
I   3.000   I  1,000    I
-------------------------------
```

Має місце спрощений синтаксис для делегатів. Перше спрощення полягає в тому, що в більшості випадків явним чином створювати екземпляр делегата не потрібно, оскільки він створюється автоматично по контексту. Друге спрощення полягає в можливості створення так званих анонімних методів - фрагментів коду, що описуються безпосередньо в тому місці, де використовується делегат. У лістингу 10.4 використано обидва спрощення для реалізації тих же дій, що і лістингу 10.3.

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

namespace examp68
{
   public delegate double Fun(double x);        // об'ява делегата
   class Classl
   {
      public static void Table(Fun F, double x, double b)
      {
         Console.WriteLine("     X        Y     ");
         while (x <= b)
         {
            Console.WriteLine("| {0,8:0.000} | {1,8:0.000}", x, F(x));
            x += 1;
         }
         Console.WriteLine(" .....................");
      }

      public static double Simple(double x)
      {
         return 1;
      }

      static void Main(string[] args)
      {
         Console.WriteLine(" Таблиця функції Sin ");
         Table(Math.Sin, -2, 2);
         Console.WriteLine(" Таблиця функції Simple ");
         Table(delegate(double x){return 1;}, 0, 3);
      }
   }
}
```

У першому випадку екземпляр делегата, відповідного функції *Sin*, створюється автоматично. Щоб це могло відбутися, список параметрів і тип повертаємого значення функції мають бути сумісні з делегатом. У другому випадку не потрібно оформляти фрагмент коду у вигляді окремої функції *Simple,* як це було зроблено в попередньому лістингу, - код функції оформляється як анонімний метод і вбудовується прямо в місце передачі.

Альтернативою використанню делегатів як параметрів є віртуальні методи. Метод виведення значень функції у вигляді таблиці можна реалізувати за допомогою абстрактного базового класу. Він містить два методи: метод виведення таблиці і абстрактний метод, для обчислення функції. Для виведення таблиці конкретної функції необхідно створити похідний клас, який перевизначає цей абстрактний метод. Реалізація методу виведення таблиці за допомогою спадкоємства і віртуальних методів приведена в лістингу 10.5.

Листинг 10.5. Альтернатива параметрам-делегатам

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

namespace examp69
{
   abstract class TableFun
   {
   public abstract double F(double x );

      public void Table(double x, double b )
      {
      Console.WriteLine( "      X         Y      " );
         while ( x <= b )
         {
            Console.WriteLine(" {0,8:0.000} | {1,8:0.000} |", x, F(x));
            x+=1;
         }
         Console.WriteLine( " ....................." );
      }
   }

   class SimpleFun : TableFun
   {
      public override double F( double x ) {
      return 1;
   }

}

   class SinFun : TableFun
   {
      public override double F( double x )
      {
       return Math.Sin(x);
      }
   }

      class Classl
      {
         static void Main()
         {
           TableFun a = new SinFun();
           Console.WriteLine(" Таблица функции Sin " );
            a.Table(-2, 2 );
            a = new SimpleFun();
            Console.WriteLine(" Таблица функции Simple " );
            a.Table(0,3);
      }
   }
}
```

Результат роботи цієї програми такий же, як і попередньої, але в даному випадку застосування делегатів переважно.

### 10.1.6. Обробка виключень при виклику делегатів

Раніше наголошувалося, що якщо в одному з методів списку делегата генерується виключення, наступні методи не викликаються. Це можна уникнути, якщо забезпечити явний перебір всіх методів в блоці, що перевіряється, і обробляти виникаючі виключення. Всі методи, задані в екземплярі делегата, можна отримати за допомогою успадкованого методу *GetInvocationList*. Цей прийом ілюструє лістинг 10.6, що є зміненим варіантом лістингу 10.1.

Лістинг 10.6. Перехоплення виключень при виклику делегата

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

namespace examp70
{

delegate void Del ( ref string s );

   class Classl
  {

   public static void Cool ( ref string s )
    {
      Console.WriteLine( "викликаний метод Cool" );
        s = s.Replace('l', '1').Replace('O', '0').Replace('o', '0');
    }

   public static void Hack ( ref string s )
   {
      Console.WriteLine( "викликаний метод Hack" );
      char[] mas = s.ToCharArray();
      s = "";
      for (int i = 0; i < mas.Length; ++i)
           s+= ((i & 1) == 0)? char.ToUpper(mas[i]):mas[i];
   }

   public static void BadHack ( ref string s )
   {
      Console.WriteLine( "викликаний метод Badhack" );
      throw new Exception();                              // імітація помилки
   }

   static void Main()
   {
```

```
      string s = "cool hackers";
      Del d = new Del(Cool);                    // створення екземпляра делегата
      d += new Del(BadHack);                    // доповнення списку методів
      d += new Del(Hack);                       // доповнення списку методів

    foreach ( Del fun in d.GetInvocationList() )
    {

      try
      {
      fun( ref s );                             // виклик кожного методу із списку
      }

      catch (Exception er)
      {
         Console.WriteLine(er.Message);
         Console.WriteLine( "Exception in method " + fun.Method.Name);
      }
    }

   Console.WriteLine( "результат - " + s );

    }
  }
}
```

Результат роботи програми:
викликаний метод *Cool*
викликаний метод *BadHack*
Exception of type *System.Exception was thrown*.
Exception in method *BadHack*
викликаний метод *Hack*
результат – *C001 hAcKeRs*

У цій програмі окрім методу базового класу *GetInvocationList* використана властивість *Method*. Цю властивість повертає результат типу *MethodInfo*. Клас *MethodInfo* містить множину властивостей і методів, що дозволяють отримати повну інформацію про метод, наприклад його специфікатори доступу, ім'я і тип повертаємого значення.

## 10.2. Події

Подія - це елемент класу, що дозволяє йому посилати іншим об'єктам повідомлення про зміну свого стану. При цьому для об'єктів, що є спостерігачами події, активізуються методи-обробники цієї події. Обробники мають бути зареєстровані в об'єкті-джерелі події. Таким чином, механізм подій формалізує на мовному рівні патерн “спостерігач”, який розглядався в попередньому розділі.

Механізм подій можна також описати за допомогою моделі “публікація - підписка”: один клас, що є відправником (*sender*) повідомлення, публікує події, які він може ініціювати, а інші класи, повідомлення, що є одержувачами (*receivers*), підписуються на отримання цих подій.

Події побудовані на основі делегатів: за допомогою делегатів викликаються методи-обробники подій. Тому створення події в класі складається з наступних частин:

опис делегата, задаючого сигнатуру обробників подій;

опис події;

опис методу (методів), що ініціюють подію.

Синтаксис події схожий на синтаксис делегата:

[ атрибути ]  [ специфікатори ] event тип ім'я_події

Для подій застосовуються специфікатори *new, public, protected, internal, private, static, virtual, sealed, override, abstract* і *extern*, які вивчалися при розгляді методів класів. Подія може бути статичною (*static*)  або звичайною. При цьому вона відповідно пов'язана з класом або з екземпляром класу.

Тип події - це тип делегата, на якому заснована подія.

Приклад опису делегата і відповідної йому події:

```
public delegate void Del( object o );     // оголошення делегата
class A
{
public event Del Oops;      // оголошення події
    …
}
```

Обробка подій виконується в класах-одержувачах повідомлення. Для цього в них описуються методи-обробники подій, сигнатура яких відповідає типу делегата. Кожен об'єкт (не клас!), охочий отримувати повідомлення, повинен зареєструвати в об'єкті-відправнику цей метод.

Як бачите, це в точності той же самий механізм, який розглядався в попередньому розділі. Єдина відмінність полягає в тому, що при використанні подій не потрібно описувати метод, реєструючий обробники, оскільки події підтримують операції += і -=, що додають обробник в список і що видаляють його із списку.

Подія - це зручна абстракція для програміста. Насправді вона складається із закритого статичного класу, в якому створюється екземпляр делегата, і двох методів, призначених для додавання і видалення обробника із списку цього делегата.

У лістингу 10.7 приведений код з лістингу 10.2, перероблений з використанням подій.

Лістинг 10.7. Сповіщення спостерігачів за допомогою подій

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

namespace examp71
{
   public delegate void Del();                    // оголошення делегата

   class Subj                                     // клас-джерело
   {
      public event Del Oops;                      // оголошення події
      public void CryOops()                       // метод, що ініціює подію
      {
         Console.WriteLine("OOPS!");
         if (Oops != null) Oops();
      }
   }

   class ObsA                                     // клас-спостерігач
   {
      public void Do()                            // реакція на подію джерела
      {
         Console.WriteLine("Бачу, що OOPS!");
      }
   }

   class ObsB                                     // клас-спостерігач
   {
      public static void See()                    // реакція на подію джерела
      {
         Console.WriteLine("Я теж бачу, що OOPS!");
      }
   }

   class Classl
   {
      static void Main()
      {
      Subj s = new Subj();                        // об'єкт класу-джерела
      ObsA ol = new ObsA();                       //        об'єкти
      ObsA o2 = new ObsA();                       //        класу-спостерігача
      s.Oops += new Del( ol.Do );                 //        додавання
      s.Oops += new Del( o2.Do );                 //        обробників
      s.Oops += new Del ( ObsB.See );             //        до події
      s.CryOops();                                // ініціація події
      }
   }
}
```

Зовнішній код може працювати з подіями тільки таким чином: додавати обробники в список або видаляти їх, оскільки поза класом можуть використовуватися тільки операції += і -=. Тип результату цих операцій - *void*, на відміну від операцій складного привласнення для арифметичних типів. Іншого способу доступу до списку обробників немає.

Усередині класу, в якому описана подія, з ним можна звертатися, як із звичайним полем, що має тип делегата: використовувати операції відношення, привласнення і так далі. Значення події за умовчанням - *null*. Наприклад, в методі Cryoops виконується перевірка на *null* для того, щоб уникнути генерації виключення *System.NullReferenceException*.

У бібліотеці *.NET* описана величезна кількість стандартних делегатів, призначених для реалізації механізму обробки подій. Більшість цих класів оформлена по одних і тих же правилах:

– ім'я делегата закінчується суфіксом *EventHandler*;

– делегат отримує два параметри:

– перший параметр задає джерело події і має тип *object*;

– другий параметр задає аргументи події і має тип *EventArgs* або похідний від нього.

Якщо обробникам події потрібна специфічна інформація про подію, то для цього створюють клас, похідний від стандартного класу *EventArgs*, і додають в нього необхідну інформацію. Якщо делегат не використовує таку інформацію, можна не описувати делегат і власний тип аргументів, а обійтися стандартним класом делегата *System.EventHandler*.

Ім'я обробника події прийнято складати з префікса *On* і імені події. У лістингу 10.8 приведений приклад з лістингу 10.7, оформлений відповідно до стандартних угод *.NET.* Знайдіть вісім відмінностей!

Лістинг 10.8. Використання стандартного делегата *EventHandler*

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

namespace examp72
{
   class Subj
   {
      public event EventHandler Oops;
      public void CryOops()
      {
         Console.WriteLine("OOPS!");
         if ( Oops != null ) Oops (this, null );
      }
   }
```

```
class ObsA
{
   public void OnOops(object sender, EventArgs e )
   {
      Console.WriteLine("Бачу, що OOPS!");
   }

}

   class ObsB
   {

      public static void OnOops( object sender, EventArgs e )
      {
         Console.WriteLine("Я теж бачу, що OOPS!");
      }

   }

class Classl
{
   static void Main()
   {
      Subj s = new Subj();

      ObsA ol = new ObsA();
      ObsA o2 = new ObsA();

      s.Oops += new EventHandler(ol.OnOops);
      s.Oops += new EventHandler(o2.OnOops);
      s.Oops += new EventHandler(ObsB.OnOops);
      s.CryOops ();
   }
  }
}
```

Можна спростити цю програму, використовуючи можливість неявного створення делегатів при реєстрації обробників подій. Відповідний варіант приведений в лістингу 10.9.

Лістинг 10.9. Використання делегатів і анонімних методів

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

namespace examp73
{
    class Subj
    {
     public event EventHandler Oops;
```

```
    public void CryOops()
        {
            Console.WriteLine("OOPS!" );
            if ( Oops != null ) Oops( this, null );
        }
    }
    class ObsA
    {
        public void OnOops(object sender, EventArgs e )
        {
            Console.WriteLine("Бачу, що OOPS!" );
        }
    }

    class ObsB
    {
        public static void OnOops(object sender, EventArgs e )
        {
        Console.WriteLine("Я теж бачу, що OOPS!" );
        }
    }

class Classl
{
    static void Main()
    {
    Subj s = new Subj();
    ObsA o1 = new ObsA();
    ObsA o2 = new ObsA();

    s.Oops += o1.OnOops;
    s.Oops += o2.OnOops;
    s.Oops += ObsB.OnOops;
    s.Oops += delegate(object sender, EventArgs e)
    {
       Console.WriteLine("Я з вами!");
    };
    s.CryOops ();

    }
}
}
```

Події включені в багато стандартних класів .NET, наприклад, в класи простору імен Windows.Forms, використовувані для розробки Windows-додатків.

## 10.3. Багатопотокові додатки

Додаток *.NET* складається з одного або декількох *процесів*. Процесу належать виділена для нього область оперативної пам'яті і ресурси. Кожен процес може складатися з декількох *доменів* (частин) додатків, ресурси яких ізольовані один від одного. В рамках домена може бути запущене декілька потоків вико-

нання. Потоком (*thread*) є частина виконуваного коду програми. У кожному процесі є первинний потік, виконуючий роль точки входу в додаток. Для консольних застосувань це метод *Main*.

Багатопотокові застосування створюють як для багатопроцесорних, так і для однопроцесорних систем. Основною метою при цьому є підвищення загальної продуктивності і скорочення часу реакції додатку. Управління потоками здійснює операційна система. Кожен потік отримує деяку кількість квантів часу, після закінчення якого управління передається іншому потоку. Це створює у користувача однопроцесорної машини враження одночасної роботи декількох потоків і дозволяє, наприклад, виконувати введення тексту одночасно з тривалою операцією по передачі даних.

Недоліки багатопоточності:

- велика кількість потоків веде до збільшення накладних витрат, пов'язаних з їх перемиканням, яке знижує загальну продуктивність системи;
- у багатопотокових застосуваннях виникають проблеми синхронізації даних, що пов'язані з доступом до одних і тих же даних з боку декількох потоків.

### 10.3.1. Клас Thread

Підтримка багатопоточності здійснюється в .NET в основному за допомогою простору імен System.Threading. Деякі типи цього простору описані в таблиці 10.1

Таблиця 10.1.

Деякі типи простору імен System.Threading

| Тип | Опис |
|---|---|
| *Interlocked* | Клас, що забезпечує синхронізований доступ до змінних, які використовуються в різних потоках |
| *Monitor* | Клас, що забезпечує синхронізацію доступу до об'єктів |
| *Mutex* | Клас-примітив синхронізації, який використовується також для синхронізації між процесами |
| *ReaderWriterLock* | Клас, що визначає блокування, що підтримує один доступ на запис і декілька, - на читання |
| *Thread* | Клас, який створює потік, встановлює його пріоритет, отримує інформацію про стан |
| *ThreadPool* | Клас, використовуваний для управління набором взаємозв'язаних потоків, - пулом потоків |
| *Timer* | Клас, що визначає механізм виклику заданого методу в задані інтервали часу для пулу потоків |

Продовження таблиці 10.1

| Тип | Опис |
|---|---|
| *WaitHandle* | Клас, що інкапсулює об'єкти синхронізації, які чекають доступу до ресурсів, що розділяються |
| *IOCompletionCallback* | Клас, одержуючий зведення про операцію введення-виведення, що завершилася |
| *ThreadStart* | Делегат, що представляє метод, який має бути виконаний при запуску потоку |
| *TimerCallback* | Делегат, що представляє метод, оброблювальний виклики від класу *Timer* |
| *WaitCanback* | Делегат, що представляє метод для елементів класу *Threadpool* |
| *ThreadPriority* | Перелічення, що описує пріоритет потоку |
| *ThreadState* | Перелічення, що описує стан потоку |

Первинний потік створюється автоматично. Для запуску вторинних потоків використовується клас *Thread*. При створенні об'єкту-потоку йому передається делегат, що визначає метод, виконання якого виділяється в окремий потік:

**Thread t = new Thread ( new ThreadStart( ім'я_методу ) ) ;**

Після створення потоку заданий метод починає в ньому свою роботу, а первинний потік продовжує виконуватися. У лістингу 10.10 приведений приклад одночасної роботи двох потоків.

Лістинг 10.10. Створення вторинного потоку

```
using System;
using System.Threading;
namespace ConsoleApplication1
{
   class Program
   {
      static public void Hedgehog()         // метод для вторинного потоку
      {
         for (int i = 0; i < 6; ++i)
         {
            Console.WriteLine(i);
            Thread.Sleep(1000);
         }
      }
      static void Main()
      {
         Console.WriteLine( "Первинний потік " +
         Thread.CurrentThread.GetHashCode());
         Thread ta = new Thread(new ThreadStart(Hedgehog) );
         Console.WriteLine("Вторинний потік " + ta.GetHashCode() );
```

```
            ta.Start();
            for ( int i = 0; i > -6;  --i )
            {
            Console.Write( " " + i );
            Thread.Sleep( 400 );
            }
        }
    }
}
```

Результат роботи програми:

```
Первинний потік 1
Вторинний потік 3
  00
  -1   -21
  -3   -42
  -53
 4
5
```

У лістингу використовується метод *Sleep*, що зупиняє функціонування потоку на задану кількість мілісекунд. Як бачите, обидва потоки працюють одночасно. Якби вони працювали з одним і тим же файлом, він був би зіпсований так само, як і приведене виведення на консоль, тому такий спосіб розпаралелювання обчислень має сенс тільки для роботи з різними ресурсами.

У таблиці 10.2 перераховані основні елементи класу *Thread*.

Таблиця 10.2.

Основні елементи класу Thread

| Елемент | Вигляд | Опис |
|---|---|---|
| CurrentThread | Статична властивість | Повертає посилання на потік, що виконується (тільки для читання) |
| IsAlive | Властивість | Повертає *true* або *false* залежно від того, запущений потік чи ні |
| IsBackground | Властивість | Повертає або встановлює значення, яке показує, чи є цей потік фоновим |
| Name | Властивість | Установка текстового імені потоку |
| Priority | Властивість | Отримати/встановити пріоритет потоку (використовуються значення переліченя *ThreadPrority*) |
| ThreadState | Властивість | Повертає стан потоку (використовуються значення переліченя *ThreadState*) |
| Abort | Метод | Генерує виключення *ThreadAbortException*. Виклик цього методу зазвичай завершує роботу потоку |
| GetData, SetData | Статичні методи | Повертає (встановлює) значення для вказаного слота в поточному потоці |

Продовження таблиці 10.2

| Елемент | Вигляд | Опис |
|---|---|---|
| GetDomain, GetDomainID | Статичні методи | Повертає посилання на домен додатку (ідентифікатор домена додатку), в рамках якого працює потік |
| GetHashCode | Метод | Повертає хеш-код для потоку |
| Sleep | Статичний метод | Припиняє виконання поточного потоку на задану кількість мілісекунд |
| Interrupt | Метод | Переривает роботу поточного потоку |
| Join | Метод | Блокує викликаючий потік до завершення іншого потоку або вказаного проміжку часу і завершує потік |
| Resume | Метод | Відновлює роботу після припинення потоку |
| Start | Метод | Починає виконання потоку, визначеного делегатом *ThreadStart* |
| Suspend | Метод | Припиняє виконання потоку. Якщо виконання потоку вже припинене, то ігнорується |

Можна створити декілька потоків, які спільно використовуватимуть один і той же код. Приклад приведений в лістингу 10.11.

Лістинг 10.11. Потоки, що використовують один об'єкт

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
   class Classl
   {
      public void Do()
      {
         for ( int i = 0; i < 4; ++i )
         {
            Console.Write(  " " + i ); Thread.Sleep( 3 );
         }
      }
   }
   class Program
   {
      static void Main()
      {
         Classl a = new Classl();
         Thread tl = new Thread( new ThreadStart(a.Do));
         tl.Name = "Second";
         Console.WriteLine("Потік " + tl.Name );
```

```
            tl.Start();
            Thread t2 = new Thread( new ThreadStart( a.Do));
            t2.Name = "Third";
            Console.WriteLine( "Потік " + t2.Name);
            t2.Start();
        }
    }
}
```

Результат роботи програми:
Потік Second
Потік Third
0 0 1 1 2 2 3 3

Варіанти виведення можуть декілька розрізнятися, оскільки один потік перериває виконання іншого в невідомі моменти часу.

Для того, щоб блок коду міг використовуватися в кожен момент тільки одним потоком, застосовується оператор *lock*. Формат оператора:

**lock ( вираз ) блок_операторів**

*Вираз* визначає об'єкт, який потрібно заблокувати. Для звичайних методів як вираз використовується ключове слово *this*. Для статичних - *typeof* (клас). Блок операторів задає критичну секцію коду, яку потрібно заблокувати.

Наприклад, блокування операторів в приведеному раніше методі *Do* виглядає таким чином:

```
public void Do()
{
      Lock( this )
      {
          for ( int i = 0; i < 4; ++i )
            {
            Console.Write( " " + i );
            Thread.Sleep( 30 );
            }
      }
  }
```

Результат роботи програми:
Потік Second
Потік Third
0 1 2 3 0 1 2 3

### 10.3.2. Асинхронні делегати

Делегат можна викликати на виконання або синхронно, як у всіх приведених раніше прикладах, або асинхронний за допомогою методів *BeginInvoke* і *EndInvoke*. При виклику делегата за допомогою методу *BeginInvoke* середовище виконання створює для виконання методу окремий потік і повертає управління операторові, наступному за викликом. При цьому в початковому потоці можна продовжувати обчислення

Якщо при виклику *BeginInvoke* був вказаний метод зворотного виклику, цей метод викликається після завершення потоку. Метод зворотного виклику також задається за допомогою делегата, при цьому використовується стандартний делегат *AsyncCallback*. У методі, зворотного виклику для набуття повертаємого значення і вихідних параметрів застосовується метод *EndInvoke.*

Якщо метод зворотного виклику не був вказаний в параметрах методу *BeginInvoke*, метод *EndInvoke* можна використовувати в потоці, що ініціював запит. У лістингу 10.11 наводяться два приклади асинхронного виклику методу, що виконує розкладання числа на множники.

Клас *Factorizer* містить метод *Factorize*, що виконує розкладання на множники. Цей метод асинхронно викликається двома способами: у методі *Num1* метод зворотного виклику задається в *BeginInvoke*, в методі *Num2* мають місце очікування завершення потоку і безпосередній виклик *EndInvoke*.

Лістинг 10.11. Асинхронні делегати

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;

namespace AsynchroneDelegat
{
                                        // асинхронний делегат
    public delegate bool AsyncDelegate(int Num, out int m1, out int m2);

    // клас, що виконує розкладання числа на множники
    public class Factorizer
    {
       public bool Factorize(int Num, out int m1, out int m2)
       {
          m1 = 1; m2 = Num;
          for (int i = 2; i < Num; i++)
             if (0 == (Num % 1)) { m1 = i; m2 = Num / i; break; }
          if (1 == m1) return false;
          else return true;
       }
    }
                                        // клас, одержуючий делегати і результати
    public class PNum
    {
       private int Number;
```

```
    public PNum(int number)
    {
        Number = number;
    }

    [OneWayAttribute()]
                                    // метод, одержуючий результати
    public void Res(IAsyncResult ar)
    {
        int m1, m2;
        // отримання делегата з IAsyncResult
        AsyncDelegate ad = (AsyncDelegate)((AsyncResult)ar).AsyncDelegate;
        // отримання результатів виконання методу Factorize
        ad.EndInvoke(out m1, out m2, ar);
        // виведення результатів
        Console.WriteLine("Перший спосіб: множетелі {0} : {1} {2}", Number, m1, m2);
    }

}
                                    // демонстраційний клас
public class Simple
{
    // спосіб 1: використовується функція зворотного виклику
    public void Num1()
    {
        Factorizer f = new Factorizer();
        AsyncDelegate ad = new AsyncDelegate(f.Factorize);
        int Num = 1000589023, tmp;
                                // створення екземпляра класу, який буде викликаний
                                // після завершення роботи методу Factorize
        PNum n = new PNum(Num);
        // завдання делегата методу зворотного виклику
        AsyncCallback callback = new AsyncCallback(n.Res);
        // асинхронний виклик методу Factorize
        IAsyncResult ar = ad.BeginInvoke(Num, out tmp, out tmp, callback, null);
                                // тут виконання якихось подальших дій
    }
                            // спосіб 2: використовується очікування закінчення виконання
    public void Num2()
    {
        Factorizer f = new Factorizer();
        AsyncDelegate ad = new AsyncDelegate(f.Factorize);

        int Num = 1000589023, tmp;

                                    // створення екземпляра класу, який буде викликаний
                                    // після завершення роботи методу Factorize
        PNum n = new PNum(Num);
        // завдання делегата методу зворотного виклику
        AsyncCallback callback = new AsyncCallback(n.Res);
```

```
                                        // асинхронний виклик методу Factorize
            IAsyncResult ar = ad.BeginInvoke(Num, out tmp, out tmp, null, null);
                                        // очікування завершення
            ar.AsyncWaitHandle.WaitOne(10000, false);
            if (ar.IsCompleted)
            {
               int m1, m2;
                                    // отримання результатів виконання методу Fractorize
               ad.EndInvoke(out m1, out m2, ar);
                                        // виведення результатів
               Console.WriteLine("Другий спосіб : множетелі {0} : {1} {2}", Num, m1, m2);
            }
         }
      }
      class Program
      {
         static void Main()
         {
            Simple s = new Simple();
            s.Num1();
            s.Num2();
         }

      }
   }
```

Результат роботи програми:
Перший спосіб : множники 1000589023 : 7 142941289
Другий спосіб : множники 1000589023 : 7 142941289

Атрибут [*OneWayAttribute()*] позначає метод якщо не має повертаємого значення і вихідних параметрів.

## 10.4. Рекомендації по програмуванню

*Делегати* широко застосовуються в бібліотеці .NET як самостійно, так і для підтримки механізму подій.

*Делегат* є особливим видом класу, що нагадує собою інтерфейс, але, на відміну від нього, задає тільки одну сигнатуру методу. У мові C ++ аналогом делегата є вказівка на функцію, але вона не володіє безпекою і зручністю використання делегата. Завдяки делегатам стає можливою гнучка організація взаємодії, що дозволяє підтримувати узгоджений стан взаємозв'язаних об'єктів.

Основною метою створення багатопотокових застосувань є підвищення загальної продуктивності програми. Проте розробка багатопотокових застосувань складніша, оскільки при цьому виникають проблеми синхронізації даних, пов'язаних з можливістю доступу до одних і тих же даних з боку декількох потоків.

# РОЗДІЛ 11. РОБОТА З ФАЙЛАМИ

*Файл* - іменована інформація на зовнішньому носієві, наприклад на жорсткому або гнучкому магнітному диску. Логічно файл можна представити як кінцева кількість послідовних байтів, тому такі пристрої, як дисплей, клавіатура і принтер, також можна розглядати як файли. Передача даних із зовнішнього пристрою в оперативну пам'ять називається *читанням*, або *введенням*, зворотний процес - *записом*, або *виведенням*.

Введення-виведення в C# виконується за допомогою підсистеми введення-виведення і класів бібліотеки *.NET*. У цьому розділі розглядається обмін даними з файлами і консоллю. Обмін даними реалізується за допомогою потоків.

*Потік* (*stream*) - це абстрактне поняття, що відноситься до будь-якого перенесення даних від джерела до приймача. Потоки забезпечують надійну роботу як із стандартними, так і з визначеними користувачем типами даних. Потік визначається як послідовність байтів і не залежить від конкретного пристрою, з яким проводиться обмін (оперативна пам'ять, файл на диску, клавіатура або принтер).

Обмін з потоком для підвищення швидкості передачі даних проводиться, як правило, через спеціальну область оперативної пам'яті - *буфер*. Буфер виділяється для кожного відкритого файлу. При записі у файл вся інформація спочатку прямує в буфер і там накопичується до тих пір, поки весь буфер не заповниться. Тільки після цього або після спеціальної команди скидання відбувається передача даних на зовнішній пристрій. При читанні з файлу дані спочатку прочитуються в буфер, причому не стільки, скільки запрошується, а скільки поміщається в буфер.

Механізм буферізації дозволяє швидше і ефективно обмінюватися інформацією із зовнішніми пристроями.

Для підтримки потоків бібліотека *.NET* містить ієрархію класів, основна частина якої представлена на рис. 11.1. Ці класи визначені в просторі імен *System.IO*. Окрім класів там описана велика кількість перелічень для завдання різних властивостей і режимів.

Класи бібліотеки дозволяють працювати в різних режимах з файлами, каталогами і областями оперативної пам'яті. Короткий опис класів приведений в таблиці 11.1.

Як можна бачити з таблиці, виконувати обмін із зовнішніми пристроями можна на рівні:

- двійкового представлення даних (*BinaryReader*, *BinaryWriter*);
- байтів (FileStream);
- тексту, тобто символів (*StreamWriter*, *StreamReader*).

У *.NET* використовується кодування *Unicode*, в якому кожен символ кодується двома байтами. Класи, що працюють з текстом, є оболонками класів, що використовують байти, і автоматично виконують те, що кодується з байтів в символи і назад.

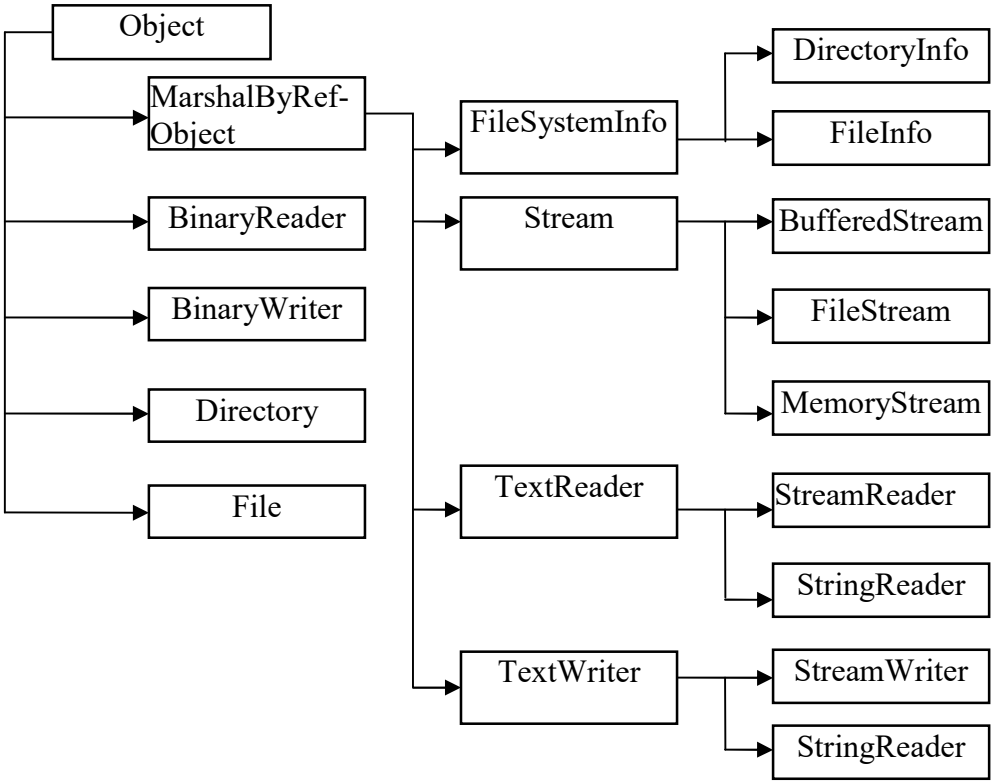



Рис. 11.1. Класи бібліотеки *.NET* для роботи з потоками

Двійкові і байтові потоки зберігають дані в тому ж вигляді, в якому вони представлені в оперативній пам'яті, тобто при обміні з файлом відбувається побітове копіювання інформації. Двійкові файли застосовуються не для перегляду їх людиною, а для використання в програмах.

*Доступ* до файлів може бути *послідовним*, коли черговий елемент можна прочитати (записати) тільки після аналогічної операції з попереднім елементом, і *довільним*, або *прямим*, при якому виконується читання (запис) довільного елементу за заданою адресою. Текстові файли дозволяють виконувати тільки послідовний доступ, в двійкових і байтових потоках можна використовувати обидва методи.

Прямий доступ у поєднанні з відсутністю перетворень забезпечує високу швидкість отримання потрібної інформації.

Методи *форматованого введення*, за допомогою яких можна виконувати введення з клавіатури або з текстового файлу значень арифметичних типів, в C# не підтримуються. Для перетворення з символьного в числове представлення використовуються методи класу *Convert* або метод *Parse*, розглянуті в попередніх розділах.

Таблиця 11.1

Основні класи простору імен *System.IO*

| Клас | Опис |
|---|---|
| BinaryReader, BinaryWriter | Читання і запис значень простих вбудованих типів (цілочисельних, логічних, рядкових і т. п.) у внутрішній формі уявлення |
| BufferedStream | Тимчасове зберігання потоку байтів (наприклад, для подальшого перенесення в постійне сховище) |
| Directory, Directorylnfo, File, Filelnfo | Робота з каталогами або фізичними файлами: створення, видалення, придбання властивостей. Можливості класів *File* і *Directory* реалізовані в основному у вигляді статичних методів. Аналогічні класи *DirectoryInfo* і *FileInfo* використовують звичайні методи |
| FileStream | Довільний (прямий) доступ до файлу, представленого як потік байтів |
| MemoryStream | Довільний доступ до потоку байтів в оперативній пам'яті |
| StreamWriter, StreamReader | Читання з файлу і запис у файл текстової інформації (довільний доступ не підтримується) |
| StringWriter, StringReader | Робота з текстовою інформацією в оперативній пам'яті |

Форматоване виведення, тобто перетворення з внутрішньої форми представлення числа в символьну, зрозумілу людині, виконується за допомогою перевантажених методів *ToString*.

Окрім перерахованих класів в бібліотеці .NET є класи *XmlTextReader* і *XmlTextWriter*, призначені для формування і читання коду у форматі *XML*.

Розглянемо прості способи роботи з файловими потоками. Використання класів файлових потоків в програмі припускає наступні операції:

1. Створення потоку і зв’язування його з фізичним файлом.
2. Обмін (уведення-виведення).
3. Закриття файлу.

Кожен клас файлових потоків містить декілька варіантів конструкторів, за допомогою яких можна створювати об'єкти цих класів різними способами і в різних режимах.

Наприклад, файли можна відкривати тільки для читання, тільки для запису або для читання і запису. Ці режими доступу до файлу містяться в переліченні *FileAccess*, визначеному в просторі імен *System. IO*. Константи перелічення приведені в таблиці 11.2.

Таблиця 11.2

Значення *FileAccess*

| Значення | Опис |
|---|---|
| Read | Відкрити файл тільки для читання |
| ReadWrite | Відкрити файл для читання і запису |
| Write | Відкрити файл тільки для запису |

Можливі режими відкриття файлу визначені в переліченні *FileMode* (таблиця. 11.3).

Таблиця 11.3

Значення *FileMode*

| **Значення** | **Опис** |
|---|---|
| Append | Відкрити файл, якщо він існує, і встановити поточний покажчик в кінець файлу. Якщо файл не існує, створити новий файл |
| Create | Створити новий файл. Якщо в каталозі вже існує файл з таким же ім'ям, він буде стертий |
| CreateNew | Створити новий файл. Якщо в каталозі вже існує файл з таким же ім'ям, виникає виключення IOException |
| Open | Відкрити існуючий файл |
| OpenOrCreate | Відкрити файл, якщо він існує. Якщо немає, створити файл з таким ім'ям |
| Truncate | Відкрити існуючий файл. Після відкриття вміст файлу видаляється |

Режим *FileMode.Append* можна використовувати тільки спільно з доступом типу *FileAccess.Write*, тобто для файлів, що відкриваються для запису.

Режими сумісного *використання* файлу різними користувачами визначає перелічення *FileShare* (таблиця 11.4).

Таблиця 11.4

Значення *FileShare*

| Значення | Опис |
|---|---|
| None | Сумісне використання відкритого файлу заборонене. Запит на відкриття даного файлу завершується повідомленням про помилку |
| Read | Дозволяє відкривати файл для читання одночасно декільком користувачам. Якщо цей прапор не встановлений, запити на відкриття файлу для читання завершуються повідомленням про помилку |
| ReadWrite | Дозволяє відкривати файл для читання і запису одночасно декільком користувачам |
| Write | Дозволяє відкривати файл для запису одночасно декільком користувачам |

## 11.1. Потоки байтів

Уведення-виведення у файл на рівні байтів виконується за допомогою класу *FileStream*, який є спадкоємцем абстрактного класу *Stream*, що визначає набір стандартних операцій з потоками. Елементи класу *Stream* описані в таблиці 11.5.

Таблиця 11.5

Елементи класу Stream

| Елемент | Опис |
|---|---|
| BeginRead, BeginWrite | Почати асинхронне введення або виведення |
| CanRead, CanSeek, CanWrite | Властивості, що визначають, які операції підтримує потік: читання, прямий доступ і/або запис |
| Close | Закрити поточний потік і звільнити пов'язані з ним ресурси (сокети, покажчики на файли і т. п.) |
| EndRead, EndWrite | Чекати завершення асинхронного введення; закінчити асинхронне виведення |
| Flush | Записати дані з буфера в пов'язане з потоком джерело даних і очистити буфер. Якщо для даного потоку буфер не використовується, то цей метод нічого не робить |
| Length | Повернути довжину потоку в байтах |
| Position | Повернути поточну позицію в потоці |
| Read, ReadByte | Підрахувати послідовність байтів (або один байт) з поточного потоку і перемістити покажчик в потоці на кількість лічених байтів |
| Seek | Встановити поточний покажчик потоку на задану позицію |
| SetLength | Встановити довжину поточного потоку |
| Write, WriteByte | Записати послідовність байтів (або один байт) в поточний потік і перемістити покажчик в потоці на кількість записаних байтів |

Клас *FileStream* реалізує ці елементи для роботи з дисковими файлами. Для визначення режимів роботи з файлом використовуються стандартні перелічення *FileMode*, *FileAccess* і *FileShare*. Значення цих перелічень приведені в таблицях 11.2-11.4. У лістингу 11.1 представлений приклад роботи з файлом. У прикладі демонструються читання і запис одного байта і масиву байтів, а також позиціонування в потоці.

Лістинг 11.1. Приклад використання потоку байтів

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

namespace examp74
{
    class Class1
    {
        static void Main()
        {
        FileStream f = new FileStream( "test.txt",
        FileMode.Create, FileAccess.ReadWrite );
        f.WriteByte( 100 );                          // у початок файлу записується число 100
        byte[] x = new byte[10];
        for ( byte i = 0; i < 10; ++i )
        {
            x[i] = (byte)(10 - i);
            f.WriteByte(i); // записується 10 чисел від 0 до 9
        }
        f.Write(x, 0, 5);                        // записується 5 елементів масиву
        byte[] y = new byte[20];
        f.Seek( 0, SeekOrigin.Begin ); // поточний покажчик - на початок
        f.Read( y, 0, 20 ); // читання з файлу в масив
        foreach ( byte elem in y ) Console.Write( " " + elem );
        Console.WriteLine();
        f.Seek(5, SeekOrigin.Begin); // поточний покажчик - на 5-й елемент
        int a = f.ReadByte();       // читання 5-го елементу
        Console.WriteLine(a);
        a = f.ReadByte();           // читання 6-го елементу
        Console.WriteLine(a);
        Console.WriteLine( " Поточна позиція в потоці " + f.Position );
        f.Close();
        }
    }
}
```

Результат роботи програми:

```
100   01234   56789   10 98760000
4
5
Поточна позиція в потоці 7
```

Поточна позиція в потоці спочатку встановлюється на початок файлу (для будь-якого режиму відкриття, окрім *Append*) і зрушується на одну позицію при записі кожного байта.

Для установки бажаної позиції читання використовується метод *Seek*, що має два параметри: перший задає зсув в байтах щодо точки відліку, що задаєть-

ся другим. Точки відліку задаються константами перелічення *SeekOrigin*: початок файлу - *Begin*, поточна позиція - *Current* і кінець файлу - *End*.

У даному прикладі файл створювався в поточному каталозі. Можна вказати і повний шлях до файлу, наприклад:

```
FileStream f = new FileStream( @"D:\C#\test.txt",
FileMode.Create, FileAccess.ReadWrite );
```

Операції по відкриттю файлів можуть завершитися невдало, наприклад, при помилці в імені існуючого файлу або за відсутності вільного місця на диску, тому рекомендується завжди контролювати результати цих операцій.

У разі непередбачених ситуацій середовище виконання генерує різні виключення, обробку яких слід передбачити в програмі, наприклад:

- *FileNotFoundException*, якщо файлу у вказаному каталозі не існує;
- *DirectoryNotFoundException,* якщо не існує вказаний каталог;
- *ArgumentException*, якщо невірно заданий режим відкриття файлу;
- *IOException*, якщо файл не відкривається із-за помилок введення-виведення.

Можливі і інші виняткові ситуації.

Зручно обробляти найбільш вірогідні помилки роздільно, щоб надати користувачеві програми в повідомленні, що виводиться, найбільш точну інформацію. У приведеному далі прикладі окремо перехоплюється помилка в імені файлу, а потім обробляється решта всіх можливих помилок:

```
try
{
FileStream f = new FileStream(@"d:\C#\test.tx", FileMode.Open, FileAccess.Read );
…                                                    // дії з файлом
f.Close();
}
catch( FileNotFoundException e)
{
        Console.WriteLine( e.Message ) ;
        Console.WriteLine( " Перевірте правильність імені файлу!" ); return;
}
catch( Exception e )
{
        Console.WriteLine( "Error:  " + e.Message ); return;
}
```

При закритті файлу звільняються всі пов'язані з ним ресурси, наприклад, для файлу, відкритого для запису, у файл вивантажується вміст буфера. Тому рекомендується завжди закривати файли після закінчення роботи, особливо файли, відкриті для запису. Якщо буфер потрібно вивантажити, не закриваючи файл, використовується метод *Flush*.

## 11.2. Асинхронне уведення-виведення

Клас Stream (і, відповідно, *FileStream*) підтримує два способи виконання операцій уведення-виведення: синхронний і асинхронний. За умовчанням файли відкриваються в синхронному режимі, тобто подальші оператори виконуються тільки після завершення операцій введення-виведення. Для тривалих файлових операцій ефективніше виконувати уведення-виведення асинхронно, в окремому потоці виконання. При цьому в первинному потоці можна виконувати інші операції.

Для асинхронного уведення-виведення необхідно відкрити файл в асинхронному режимі, для цього використовується відповідний варіант перевантаженого конструктора. Асинхронна операція введення ініціюється за допомогою методу *BeginRead*. Окрім характеристик буфера, в який виконується введення, в цей метод передається делегат, який задає метод, що виконується після завершення введення.

Цей метод може ініціювати обробку отриманої інформації, відновити операцію читання або виконати будь-які інші дії, наприклад, перевірити успішність введення і повідомити про його завершення. Зазвичай в цьому методі викликається метод *EndRead*, який завершує асинхронну операцію.

Аналогічно виконується і асинхронне виведення. У лістингу 11.2 приведений приклад асинхронного читання з файлу великого об'єму і паралельного виконання діалогу з користувачем.

Лістинг 11.2. Асинхронне введення

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

namespace examp75
{
   class Demo
   {
      FileStream f;
      byte[] buf = new byte[66666666];
      AsyncCallback callback;

      public void UserInput() // діалог з користувачем
      {
         string s;
         do
         {
            Console.WriteLine("Введіть рядок. Enter для завершення ");
            s = Console.ReadLine();
         } while (s.Length != 0);
      }

      public void OnCompletedRead(IAsyncResult ar)              // 1
```

```
        {
            int bytes = f.EndRead(ar);
            Console.WriteLine("Налічено " + bytes);
        }
        public void AsyncRead()
        {
            try
            {
                f = new FileStream("1.txt", FileMode.Open,
                FileAccess.Read, FileShare.Read, buf.Length, true);     // 2
                callback = new AsyncCallback( OnCompletedRead );  //3
                f.BeginRead( buf, 0, buf.Length, callback, null );          // 4
            }
            catch (Exception e)
            {
                Console.WriteLine("Error " + e.Message);
            }
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Demo d = new Demo();
            d.AsyncRead();
            d.UserInput();
        }
    }
}
```

Для зручності сприйняття операції читання з файлу і діалогу з користувачем оформлені в окремий клас *Demo*.

Метод *OnCompletedRead* (оператор 1) повинен отримувати один параметр стандартного типу *IAsyncResult*, що містить відомості про завершення операції, які передаються в метод *EndRead*.

Файл відкривається в асинхронному режимі, про це говорить значення true останнього параметра конструктора (оператор 2). У операторові 3 створюється екземпляр стандартного делегата *AsyncCallback*, який ініціалізувався методом *OnCompletedRead*.

За допомогою цього делегата метод *OnCompletedRead* передається в метод *BeginRead* (оператор 4), який створює окремий потік, починає асинхронне введення і повертає управління в потік, що викликає. Зворотний виклик методу *OnCompletedRead* відбувається при завершенні операції введення. При достатньо довгому файлі *verybigfile* можна переконатися, що запрошення до введення в методі *UserInput* видається раніше, ніж повідомлення про завершення операції введення з методу *OnCompletedRead*.

Приклад, приведений в лістингу 11.2, максимально спрощений для демонстрації методів *BeginRead* і *EndRead*, тому в ньому немає необхідних перевірок наявності файлу, успішності читання і так далі.

## 11.3. Потоки символів

Символьні потоки *StreamWriter* і *StreamReader* працюють з *Unicode-символами.* Ці потоки є спадкоємцями класів *TextWriter* і *TextReader*. У таблицях 11.6 і 11.7 приведені найбільш важливі елементи цих класів. Як бачите, довільний доступ для текстових файлів не підтримується.

Таблиця 11.6

Найбільш важливі елементи базового класу *TextWriter*

| Елемент | Опис |
|---|---|
| Close | Закрити файл і звільнити пов'язані з ним ресурси. Якщо в процесі запису використовується буфер, він буде автоматично очищений |
| Flush | Очистити всі буфери для поточного файлу і записати накопичені в них дані в місце їх постійного зберігання. Сам файл при цьому не закривається |
| NewLine | Використовується для завдання послідовності символів, що означають початок нового рядка. За умовчанням використовується послідовність “повернення каретки” - “переведення рядка” ( \r \n) |
| Write | Записати фрагмент тексту в потік |
| WriteLine | Записати рядок в потік і перейти на інший рядок |

Таблиця 11.7

Найбільш важливі елементи класу *TextReader*

| Елемент | Опис |
|---|---|
| Peek | Повернути наступний символ, не змінюючи позицію покажчика у файлі |
| Read | Рахувати дані з вхідного потоку |
| ReadBlock | Рахувати з вхідного потоку вказану користувачем кількість символів і записати їх в буфер, починаючи із заданої позиції |
| ReadLine | Рахувати рядок з поточного потоку і повернути його як значення типу *string*. Порожній рядок (null) означає кінець файлу (EOF) |
| ReadToEnd | Рахувати всі символи до кінця потоку, починаючи з поточної позиції, і повернути дані як один рядок типу *string* |

Ви вже знайомі з деякими методами, приведеними в цих таблицях: впродовж всього посібника постійно використовувалися методи читання з текстових потоків і запису в текстові потоки, але не для дискових файлів, а для консолі, яка є їх окремим випадком.

У лістингу 11.3 створюється текстовий файл, в який записуються два рядки. Другий рядок формується з перетворених чисельних значень змінних і пояснюючого тексту. Вміст файлу можна подивитися в будь-якому текстовому редакторові. Файл створюється в тому ж каталозі, куди середовище записує виконуваний файл. За умовчанням це каталог ...\ConsoleApplication1\bin\Debug.

Лістинг 11.3. Виведення в текстовий файл

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

namespace examp76
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
              StreamWriter f = new StreamWriter( "text.txt" );
              f.WriteLine( "ABCDEF:" );
              double a = 12.234;
              int b = 29;
              f.WriteLine( " a = {0,6:C} b = {1,2:X}", a, b );
              Console.WriteLine(" a = {0,6:C} b = {1,2:X}", a, b);
              f.Close();
            }
            catch( Exception e )
            {
              Console.WriteLine( "Error: " + e.Message );
              return;
            }
        }
    }
}
```

У лістингу 11.4 файл, створений в попередньому прикладі, виводиться на екран.

Лістинг 11.4. Читання текстового файлу

```
using System;
using System.IO;
namespace exam77
{
    class Class1
        {
            static void Main()
            {
                try
                {
                    StreamReader f = new StreamReader( "1.txt" );
                    string s = f.ReadToEnd();
                    Console.WriteLine(s);
                    f.Close();
                }
                catch( FileNotFoundException e )
                {
                    Console.WriteLine( e.Message );
                    Console.WriteLine( " Проверьте правильность имени файла!" );
                    return;
                }
                catch( Exception e )
                {
                    Console.WriteLine("Error:" + e.Message);
                    return;
                }
        }
    }
}
```

У цій програмі весь файл прочитується за один прийом за допомогою методу *ReadToEnd*. Частіше виникає необхідність прочитувати файл порядково, такий приклад приведений в лістингу 11.5.

Лістинг 11.5. Порядкове читання текстового файлу

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

namespace examp78
{
    class Class1
    {
        static void Main()
        {
            try
            {
```

```
            StreamReader f = new StreamReader( "1.txt" );
            string s;
            long i = 0;
            while ( ( s = f.ReadLine() ) != null )
            Console.WriteLine( " {0} : {1} ", ++i, s );
            f.Close();
          }
          catch( FileNotFoundException e )
            {
            Console.WriteLine( e.Message );
            Console.WriteLine( " Перевірте правильність імені файлу!" );
            return;
            }
          catch ( Exception e )
          {
            Console.WriteLine( "Error: " + e.Message );
            return;
          }
        }
    }
}
```

Приклад перетворення чисел, що містяться в текстовому файлі, в їх внутрішню форму представлення приведений в лістингу 11.6. У програмі обчислюється сума чисел в кожному рядку.

На вміст файлу накладаються строгі обмеження: числа мають бути розділені рівно одним пропуском, після останнього числа в рядку пропуску бути не повинно, файл не повинен закінчуватися символом переведення рядків. Методи розбиття рядка і перетворення в цілочисельне представлення розглядалися раніше.

Лістинг 11.6. Перетворення рядків в числа

```
using System;
using System.IO;
namespace examp80
{ class Class1
  { static void Main()
          {
                try
                {
                        StreamReader f = new StreamReader("1.txt" );
                        string s;
                        const int n = 20;
                        int[] a = new int[n];
                        string[] buf;
                        while (( s = f.ReadLine() ) != null )
                        {
                                buf = s.Split(' ') ;
                                long sum = 0;
                                for ( int i = 0; i < buf.Length; ++i )
                                {
```

```
                                a [i] = Convert.ToInt32( buf[i] );
                                sum += a [i] ;
                        }
                        Console.WriteLine( "{0} сумма: {1}", s, sum );
                }
                f.Close();
        }
        catch( FileNotFoundException e )
        {
                Console.WriteLine( e.Message );
                Console.WriteLine( " Перевірте правильність імені файлу!" );
                return;
        }
        catch ( Exception e )
        {
                Console.WriteLine( "Error; " + e.Message );
                return;
        }
    }
  }
}
```

Результат роботи програми:
1 2 4 сума: 7
3 44 -3 6 сума: 50
8 1 1 сума: 10

## 11.4. Двійкові потоки

Двійкові файли зберігають дані в тому ж вигляді, в якому вони представлені в оперативній пам'яті, тобто у внутрішній формі представлення. Двійкові файли застосовуються не для перегляду їх людиною, а для використання в програмах. Вихідний потік *BinaryWriter* підтримує довільний доступ, тобто є можливість виконувати запис в довільну позицію двійкового файлу.

Двійковий файл відкривається на основі базового потоку, який найчастіше використовує потік *FileStream*.

Основні методи двійкових потоків приведені в таблицях 11.8 і 11.9.

Таблиця 11.8

Найбільш важливі елементи класу BinaryWriter

| Елемент | Опис |
|---|---|
| BaseStream | Базовий потік, з яким працює об'єкт BinaryWriter |
| Close | Закрити потік |
| Flush | Очистити буфер |
| Seek | Встановити позицію в потоці |
| Write | Записати значення в потік |

Таблиця 11.9

Найбільш важливі елементи класу BinaryReader

| Елемент | Опис |
|---|---|
| BaseStream | Базовий потік, з яким працює об'єкт *BinaryReader* |
| Close | Закрити потік |
| PeekChar | Повернути наступний символ без переміщення внутрішнього покажчика в потоці |
| Read | Рахувати потік байтів або символів і зберегти в масиві |
| ReadXXXX | Рахувати з потоку дані певного типу (наприклад, *Readboolean, Readbyte, ReadInt32* і т. д.) |

У лістингу 11.7 приведений приклад формування двійкового файлу. У файл записується послідовність дійсних чисел, а потім для демонстрації довільного доступу третє число замінюється числом 8888.

Лістинг 11.7. Формування двійкового файлу

```
using System;
using System.IO;
namespace examp81
{
    class Class1
    {
        static void Main()
        {
            try
            {
                BinaryWriter fout = new BinaryWriter(
                new FileStream(@"D:\1.txt", FileMode.Create));
                double d = 0;
                while ( d < 4 )
                {
                  fout.Write( d );
                  d += 0.33;
                };
                fout.Seek( 16, SeekOrigin.Begin ); // другий елемент файлу
                fout.Write( 8888d );
                fout.Close();
            }
            catch( Exception e )
            {
                Console.WriteLine( "Error: " + e.Message );
                return;
            }
        }
    }
}
```

При створенні двійкового потоку в нього передається об'єкт базового потоку. При установці покажчика поточної позиції у файлі враховується довжина кожного значення типу double - 8 байт.

У лістингу 11.8 приводиться програма, яка за допомогою екземпляра *Binaryreader* прочитує вміст файлу в масив дійсних чисел, а потім виводить цей масив на екран.

При читанні береться до уваги, що метод *ReadDouble* при виявленні кінця файлу генерує виключення *EndOfStreamException*. Оскільки в даному випадку це не помилка, тіло обробника виключень порожнє.

Лістинг 11.8. Читання двійкового файлу

```
using System;
using System.IO;
namespace examp82
{
    class Class1
    {
        static void Main()
        {
            try
            {
            FileStream f =
            new FileStream( @"D:\1.txt", FileMode.Open );
            BinaryReader fin = new BinaryReader( f );
            long n = f.Length / 8; // кількість чисел у файлі
            double[] x = new double[n];
            long i = 0;
            try
            {
              while (true) x[i++] = fin.ReadDouble(); // читання
            }
            catch ( EndOfStreamException e ) {}
            foreach (double d in x) Console.Write(" " + d); // виведення
            fin.Close();
            f.Close();
            }
            catch ( FileNotFoundException e )
            {
            Console.WriteLine( e.Message );
            Console.WriteLine( " Перевірте правильність імені файлу!" );
            return;
            }
            catch ( Exception e )
            {
            Console.WriteLine( "Error: " + e.Message );
            return;
            }
        }
    }
}
```

Результат роботи програми:

0 0,33 8888 0,99 1,32 1,65 1,98 2,31 2,64 2,97 3,3 3,63 3,96

## 11.5. Консольне уведення-виведення

Консольні застосування мають обмежену область застосування, найпоширенішим з яких є навчання мові програмування. Для організації уведення і виведення використовується відомий вам клас *Console*, визначений в просторі імен *System*. У цьому класі визначено три стандартні потоки: вхідний потік *Console.In* класу *TextReader* і вихідні потоки *Console.Out* і *Console.Error* класу *TextWriter*.

За умовчанням вхідний потік пов'язаний з клавіатурою, а вихідні - з екраном. Проте можна перенаправити ці потоки на інші пристрої за допомогою методів *SetIn* і *SetOut* або засобами операційної системи (перенаправлення за допомогою операцій <, > і >>.

При обміні з консоллю можна застосовувати методи вказаних потоків, але частіше використовуються методи класу *Console: Read, ReadLine, Write* і *WriteLine*. Ці методи просто передають управління методам, що розташовуються нижче*: In, Out і Error*.

Використання двох вихідних потоків корисно, якщо треба розділити на нормальне виведення програми і її повідомлення про помилки. Наприклад, нормальне виведення програми можна направити у файл, а повідомлення про помилки - на консоль або у файл журналу.

## 11.6. Робота з каталогами і файлами

У просторі імен *System.IO* є чотири класи, призначені для роботи з фізичними файлами і структурою каталогів на диску: *Directory, File, DirectoryInfo* і *FileInfo*. З їх допомогою можна виконувати створення, видалення, переміщення файлів і каталогів, а також набуття їх властивостей.

Класи *Directory* і *File* реалізують свої функції через статичні методи. *DirectoryInf*o і *FileInfo* володіють схожими можливостями, але вони реалізуються шляхом створення об'єктів відповідних класів. Класи DirectoryInfo і FileInfo походять від абстрактного класу *FilesystemInfo*, який забезпечує їх базовими властивостями, описаними в таблиці 11.10.

Таблиця 11.10

Властивості класу FileSystemInfo

| Властивість | Опис |
|---|---|
| Attributes | Отримати або встановити атрибути для даного об'єкту файлової системи. Для цієї властивості використовуються значення перелічення *FileAttributes* |
| CreationTime | Отримати або встановити час створення об’єкту файлової системи |
| Exists | Визначити, чи існує даний об'єкт файлової системи |
| Extension | Отримати розширення файлу |
| FullName | Повернути ім'я файлу або каталога з вказівкою повного шляху |
| LastAccessTime | Отримати або встановити час останнього звернення до об'єкту файлової системи |
| LastWriteTime | Отримати або встановити час останнього внесення змін в об'єкт файлової системи |
| Name | Повернути ім'я файлу. Ця властивість доступна тільки для читання. Для каталогів повертає ім'я останнього каталога в ієрархії. |

Клас *DirectoryInfo* містить елементи, що дозволяють виконувати необхідні дії з каталогами файлової системи. Ці елементи перераховані в таблиці 11.11

Таблиця 11.11

Елементи класу *DirectoryInfo*

| Елемент | Опис |
|---|---|
| Create, CreateSubDirectory | Створити каталог або підкаталог по вказаному шляху у файловій системі |
| Delete | Видалити каталог зі всім його вмістом |
| GetDirectories | Повернути масив рядків, що представляють всі підкаталоги |
| GetFiles | Отримати файли в поточному каталозі у вигляді масиву об'єктів класу FileInfo |
| MoveTo | Перемістити каталог і весь його вміст на нову адресу у файловій системі |
| Parent | Повернути батьківський каталог |

У лістингу 11.9 приведений приклад, в якому створюються два каталоги, виводиться інформація про них і робиться спроба видалення каталога.

Лістинг 11.9. Використання класу *DirectorInfo*

```
using System;
using System.IO;
namespace examp83
{ class Class1
        { static void DirInfo( DirectoryInfo di)
                {
                        // Виведення інформації про каталог
                        Console.WriteLine("===== Directory Info =====" );
                        Console.WriteLine("Full Name: " + di.FullName );
                        Console.WriteLine("Name: "      + di.Name );
                        Console.WriteLine("Parent: "    + di.Parent );
                        Console.WriteLine("Creation: "  + di.CreationTime );
                        Console.WriteLine("Attributes: "+ di.Attributes );
                        Console.WriteLine("Root: "      + di.Root );
                        Console.WriteLine("============================");
                }

                static void Main()
                {
                        DirectoryInfo di1 = new DirectoryInfo( @"c:\MyDir" );
                        DirectoryInfo di2 = new DirectoryInfo( @"c:\MyDir\temp" );

                        try
                        {
                                // Створити каталоги
                                di1.Create();
                                di2.Create();
                                // Вивести інформацію про каталоги
                                DirInfo(di1);
                                DirInfo(di2);
                                // Спробувати видалити каталог
                                Console.WriteLine( " Спроба видалити {0}.", di1.Name );
                                di1.Delete(true);
                        }
                        catch ( Exception )
                        {
                                Console.WriteLine( " Спроба не вдалася " );
                        }
                }
        }
}
```

Результат роботи програми:

```
===== Directory Info =====
Full Name: c:\MyDir
Name: MyDir Parent:
Creation: 30.04.2022 17:14:44
```

Attributes: Directory
Root: c:\
==========
===== Directory Info =====
Full Name: *c:\MyDir\temp*
Name: *temp Parent*: *MyDir*
Creation: *30.04.2022 17:14:44*
Attributes: *Directory*
Root: *c:*
==========
Спроба видалити *MyDir*.
Спроба не вдалася

Каталог не порожній, тому спроба його видалення не вдалася. Втім, якщо використовувати перевантажений варіант методу Delete з одним параметром, задаючим режим видалення, можна видалити і непорожній каталог:

```
dil.Delete( true );          // видаляє непорожній каталог
```

Зверніть увагу на властивість *Attributes*. Деякі її можливі значення, задані в переліченні *FileAttributes*, приведені в таблиці 11.12.

Таблиця 11.12

Деякі значення перелічення *FileAttributes*

| Значення | Опис |
|---|---|
| Archive | Використовується додатками при виконанні резервного копіювання, а в деяких випадках - при видаленні старих файлів |
| Compressed | Файл є стислим |
| Directory | Об'єкт файлової системи є каталогом |
| Encrypted | Файл є зашифрованим |
| Hidden | Файл є прихованим |
| Normal | Файл знаходиться в звичайному стані, і для нього встановлені будь-які інші атрибути. Цей атрибут не може використовуватися з іншими атрибутами |
| Offline | Файл, розташований на сервері, кешированій в сховищі на клієнтському комп'ютері. Можливо, що дані цього файлу вже застаріли |
| Readonly | Файл доступний тільки для читання |
| System | Файл є системним |

Лістинг 11.10 демонструє використання класу *FileInfo* для копіювання всіх файлів з розширенням jpg з каталога *d:\foto* в каталог *d:\temp*. Метод *Exists* дозволяє перевірити, чи існує початковий каталог.

```
using System;
using System.IO;
namespace examp84
{
    class Class1
    {
        static void Main()
        {
            try
            {
                string DestName = @"d:\fot";
                DirectoryInfo dest = new DirectoryInfo(DestName);
                dest.Create(); // створення цільового каталога
                DirectoryInfo dir = new DirectoryInfo( @"d:\temp" );
                if ( !dir.Exists ) // перевірка існування каталога
                {
                    Console.WriteLine("Каталог " +
                    dir.Name + " не існує" );
                    return;
                }
                FileInfo[] files = dir.GetFiles( "*.jpg" ); // список файлів
                foreach (FileInfo f in files)
                f.CopyTo( dest +  "1.txt" ); // копіювання файлів
                Console.WriteLine( "Скопійовано " +
                files.Length + " jpg-файлов" );
                Console.ReadLine();
            }
            catch (Exception e )
            {
                Console.WriteLine( "Error: " + e.Message );
                Console.ReadLine();
            }
        }
    }
}
```

Використання класів Fi1e і Directory аналогічно, за винятком того, що їх методи є статичними і, отже, не вимагають створення об'єктів.

## 11.7. Збереження об'єктів (серіалізація)

У C# є можливість зберігати на зовнішніх носіях не тільки дані примітивних типів, але і об'єкти. Збереження об'єктів називається серіалізацією, а відновлення збережених об'єктів - десеріалізацією. При серіалізації об'єкт перетвориться в лінійну послідовність байтів. Це складний процес, оскільки об'єкт може включати множину успадкованих полів і посилання на вкладені об'єкти, які, у свою чергу, теж можуть складатися з об'єктів складної структури.

На щастя, серіалізація виконується автоматично, досить просто помітити клас, який сериалізується за допомогою атрибуту [*Serializable*] . Атрибути розглядаються в розділі 12, поки ж достатньо знати, що атрибути - це додаткові відомості про клас, які зберігаються в його метаданих. Ті поля, які зберігати не потрібно, позначаються атрибутом [*NonSerialized*], наприклад:

```
[Serializable]
class Demo
{
        public int a = 1;
        [NonSerialized]
        public double y;
        public Monster X. Y;
}
```

Об'єкти можна зберігати в одному з двох форматів: двійковому або SOAP (у вигляді XML-файла). У першому випадку слід підключити до програми простір імен *System.Runtime.Serialization.Formatters.Binary*, у другому - простір *System.Runtime.Serialization.Formatters.Soap*.

Розглянемо *збереження об'єктів у двійковому форматі*. Для цього використовується клас *BinaryFormatter*, в якому визначено два методи:

Serialize(потік, об'єкт);

Deserialize(потік);

Метод Serialize зберігає заданий об'єкт в заданому потоці, метод *Deserialize* відновлює об'єкт із заданого потоку.

У лістингу 11.11 об'єкт, приведеного раніше класу *Demo* зберігається у файлі на диску з ім'ям Demo.bin. Цей файл можна проглянути, відкривши його, наприклад, в *Visual Studio.NET*.

Лістинг 11.11. Сериалізація об'єкту

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
namespace Examp85
{
    [Serializable]
    abstract class Spirit
    {
        public abstract void Passport();
    }
    [Serializable]
    class Monster : Spirit
    {
        string name;
        int health, ammo;
        public Monster( int health, int ammo, string name )
        {
            this.health = health;
            this.ammo = ammo;
            this.name = name;
        }
        override public void Passport()
        {
            Console.WriteLine("Monster {0} \t health = {1} ammo = {2} " ,
            name, health, ammo );
        }}
```

```
    [Serializable]
    class Demo
    {
        public int a = 1;

        [NonSerialized]
        public double b;
        public Monster X, Y;
    }

    class Class1
    {
        static void Main()
        {
            // Запис об'єкту на диск
            Demo d = new Demo();
            d.X = new Monster(100, 80, "Вася");
            d.Y = new Monster(120, 50, "Петя");
            d.a = 2;
            d.b = 5;
            d.X.Passport();
            d.Y.Passport();
            Console.WriteLine(d.a);
            Console.WriteLine(d.b);
            FileStream f = new FileStream("c:\\Demo.bin",
            FileMode.Create);
            BinaryFormatter bf = new BinaryFormatter();
            bf.Serialize(f, d); // збереження об'єкту d в потоці f
            f.Close();

            // Зчитування з диска

            f = new FileStream("c:\\Demo.bin",
            FileMode.Open);
            bf = new BinaryFormatter();
            d = (Demo)bf.Deserialize(f); // відновлення об'єкту
            d.X.Passport();
            d.Y.Passport();
            Console.WriteLine(d.a);
            Console.WriteLine(d.b);
            f.Close();
        }
    }
}
```

Результат роботи програми:

```
Monster Вася      health = 100 ammo = 80
Monster Петя      health = 120 ammo = 50
2
2
Monster Вася      health = 100 ammo = 80
Monster Петя      health = 120 ammo = 50
```

2
0
Отже, *для збереження об'єкту в двійковому форматі необхідно*:

1. Підключити простір імен *System.Runtime.Serialization. Formatters.Binary.*
2. Помітити клас, що зберігається, і пов'язані з ним класи атрибутом [*Serializable*].
3. Створити потік і пов'язати його з файлом на диску або з областю оперативної пам'яті.
4. Створити об'єкт класу *BinaryFormatter*.
5. Зберегти об'єкти в потоці.
6. Закрити файл.

Як видно з лістингу 11.11 після збереження об'єкту на диску він зчитується з файлу.

При серіалізації зберігається все дерево об'єктів. При цьому значення поля *y* не було збережено, оскільки воно було помічене як таке, що не треба зберігати.

## 11.8. Рекомендації по програмуванню

Більшість програм тим або іншим чином працюють із зовнішніми пристроями, наприклад, як консоль, файл на диску або мережене з'єднання. Взаємодія із зовнішніми пристроями організовується за допомогою потоків, які підтримуються множиною класів бібліотеки .NET.

Потік визначається як послідовність байтів і не залежить від конкретного пристрою, з яким проводиться обмін. Класи бібліотеки дозволяють працювати з потоками в різних режимах і на різних рівнях: на рівні двійкового представлення даних, байтів і тексту. Двійкові і байтові потоки зберігають дані у внутрішньому представленні, текстові - в кодуванні *Unicode*.

Потік можна відкрити в синхронному або асинхронному режимі для читання, запису або додавання. Доступ до файлів може бути послідовним і довільним. Текстові файли дозволяють виконувати тільки послідовний доступ, в двійкових і байтових потоках можна використовувати обидва методи. Прямий доступ у поєднанні з відсутністю перетворень забезпечує високу швидкість обміну.

Методи форматованого введення для значень арифметичних типів в C# не підтримуються. Для перетворення з символьного в числове представлення використовуються методи класу *Convert* або метод *Parse*. Форматоване виведення виконується за допомогою перевантаженого методу *ToString*, результат виконання якого передається в методи текстових файлів.

Рекомендується завжди перевіряти успішність відкриття існуючого файлу, перехоплювати виключення, що виникають при перетворенні значень арифметичних типів, і явним чином закривати файл, в який виконувався запис.

Тривалі операції з файлами ефективніше виконувати в асинхронному режимі.

Для збереження об'єктів (серіалізація) використовується атрибут [*Serializable*]. Об'єкти можна зберігати в одному з двох форматів: двійковому або SOAP (у вигляді XML-файла).

# РОЗДІЛ 12. ЗБІРКИ, БІБЛІОТЕКИ, АТРИБУТИ, ДИРЕКТИВИ

У цьому розділі розглядаються питання створення і використання бібліотек, дається інформація про атрибути, простір імен і директиви препроцесора.

## 12.1. Збірки

В результаті компіляції в середовищі .NET створюється збірка - файл з розширенням exe або dll, який містить код на проміжній мові, метадані типів, маніфест і ресурси (рис. 12.1)

| Маніфест |
|---|
| Метадані |
| Код на мові IL |
| Ресурси (не обов'язково) |

Рис. 12.1. Збірка, що складається з одного файлу

Проміжна мова (Intermediate Language, IL) не містить інструкцій, залежних від операційної системи і типу комп'ютера, що забезпечує дві основні можливості:

- виконання додатку на будь-якому типі комп'ютера, для якого існує середовище виконання CLR;
- повторне використання коду, написаного на будь-якій .NET- мові.

IL-код можна проглянути за допомогою дизасемблера ILDasm.exe, який знаходиться в папці ...\SDK\bin\ каталога розміщення Visual Studio.NET. Після запуску ILDasm можна відкрити будь-який файл середовища .NET з розширенням exe або dll; за допомогою команди File > Open. У вікні програми відкриється список всіх елементів збірки, відомості про кожне можна отримати подвійним клацанням. При цьому відкривається вікно, в якому для методів виводиться доступний для сприйняття дізасембльований код.

*Метадані типів* - це відомості про типи, використовувані в збірці. Компілятор створює метадані автоматично. У них міститься інформація про кожен тип, наявний в програмі, і про кожен його елемент. Наприклад, для кожного класу описуються всі його поля, методи, властивості, події, базові класи і інтерфейси.

Середовище виконання використовує метадані для пошуку визначень типів і їх елементів в збірці, для створення екземплярів об'єктів, перевірки виклику методів і так далі. Компілятор, редактор коду і засоби відладки також широко використовують метадані, наприклад, для виведення підказок і діагностичних повідомлень.

*Маніфест* - це набір метаданих про саму збірку, включаючи інформацію про всі файли, що входять до складу збірки, версії збірки, а також зведення про

всі зовнішні збірки, на які вона посилається. Маніфест створюється компілятором автоматично, програміст може доповнювати його власними атрибутами.

Найчастіше збірка складається з єдиного файлу, проте вона може включати і декілька фізичних файлів (модулів). В цьому випадку маніфест або включається до складу одного з файлів, або міститься в окремому файлі. Багатофайлові збірки використовуються для прискорення завантаження додатку - це має сенс для збірок великого об'єму, робота з якими проводиться видалено.

На логічному рівні збірка є сукупністю взаємозв'язаних типів - класів, інтерфейсів, структур, перелічень, делегатів і ресурсів. Бібліотека .NET є сукупністю збірок, яку використовують додатки. Так само можна створювати і власні збірки, які можна буде використовувати або в рамках одного застосування (*приватні збірки*), або спільно різними застосуваннями (*відкриті збірки*). За умовчанням всі збірки є приватними.

*Маніфест збірки містить:*

- ідентифікатор версії;
- список всіх внутрішніх модулів збірки;
- список зовнішніх збірок, необхідних для нормального виконання збірки;
- інформацію про природну мову, використовувану в збірці (наприклад, російському);
- “сильне” ім'я (*strong name*) - спеціальний варіант імені збірки, використовуваний для відкритих збірок;
- необов'язкову інформацію, пов'язану з безпекою;
- необов'язкову інформацію, пов'язану із зберіганням ресурсів усередині збірки.

Ідентифікатор версії відноситься до всіх елементів збірки. Він дозволяє уникати конфліктів імен і підтримувати одночасне існування і використання різних версій одних і тих же збірок. Ідентифікатор версії складається з двох частин: інформаційної версії у вигляді текстового рядка і версії сумісності у вигляді чотирьох чисел, розділених крапками:

- основний номер версії (*major version*);
- додатковий номер версії (*minor version*);
- номер збірки (build number);
- номер ревізії (revision number).

Середовище виконання застосовує ідентифікатор версій для визначення того, які з відкритих збірок сумісні з вимогами клієнта. Наприклад, якщо клієнт запрошує збірку 3.1.0.0, а присутня тільки версія 3.4.0.0, збірка не буде пізнана як відповідна, оскільки вважається, що в додаткових версіях можуть відбутися зміни в типах і їх елементах. Різні номери ревізії допускають, але не гарантують сумісність. Номер збірки на сумісність не впливає, оскільки найчастіше він змінюється при установці патча (*patch*).

Ідентифікатор версії формується автоматично, але за бажання можна задати його вручну за допомогою атрибуту [*AssemblyVersion*], який розглядається далі.

*Інформація про безпеку* дозволяє визначити, чи надати клієнтові доступ до запрошуваних елементів збірки. У маніфесті збірки визначені обмеження системи безпеки.

*Ресурси* є, наприклад, файлами зображень, що поміщаються на форму, текстові рядки, значки додатку і так далі. Зберігання ресурсів усередині збірки забезпечує їх захист і спрощує розгортання додатку. Середовище *Visual Studio.NET* надає можливості автоматичного впровадження ресурсів в збірку.

Відкриті і приватні збірки розрізняються по способах розміщення на комп'ютері користувача, іменуванні і політиці версій. Приватні збірки повинні знаходитися в каталозі додатку, що використовує збірку, або в його підкаталогах.

Відкриті збірки розміщуються в спеціальному каталозі, який називається глобальним кешем збірок (*Global Assembly Cache, GAC*). Для ідентифікації відкритої збірки використовується вже згадуване сильне ім'я (*strong name*), яке має бути унікальним.

## 12.2. Створення бібліотеки

Для створення бібліотеки треба при розробці проекту в середовищі Visual Studio.NET вибрати шаблон *Class Library* (бібліотека класів). В розділі 8 була створена проста ієрархія класів персонажів комп'ютерної гри. У цьому розділі ми оформимо її у вигляді бібліотеки, тобто збірки з розширенням dll. Для збірки задано ім'я MonsterLib (рис. 12.2).

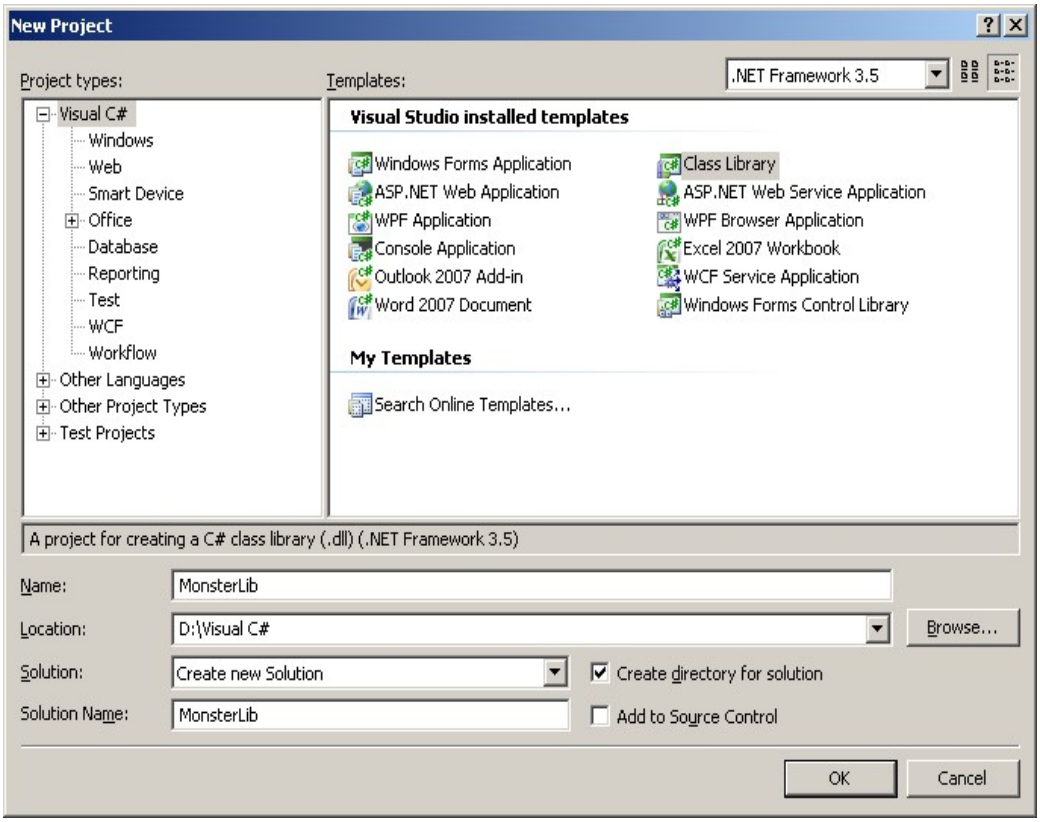



Рис. 12.2. Створення бібліотеки

Текст модуля приведений в лістингу 12.1. В порівнянні з модулем з розділу 8 в нього додані специфікатори доступу *public* для всіх трьох класів, що входять в бібліотеку.

Лістинг 12.1. Бібліотека монстрів

```
namespace MonsterLib
{
   using System;
   public abstract class Spirit
   {
      public abstract void Passport();
   }

   public class Monster : Spirit
   {
      public Monster()
      {
         this.name = "Noname";
         this.health = 100;
         this.ammo = 100;
      }

      public Monster(string name)
         : this()
      {
         this.name = name;
      }
      public Monster(int health, int ammo, string name)
      {
         this.name = name;
         this.health = health;
         this.ammo = ammo;
      }

      public int Ammo
      {
         get { return ammo; }
         set
         {
            if (value > 0) ammo = value; else ammo = 0;
         }
      }
      public int Health
      {
         get { return health; }
         set
         {
            if (value > 0) health = value; else health = 0;
         }
      }
```

```
        override public void Passport()
        {
            Console.WriteLine("Monster {0} \t health = {1} ammo = {2}", name, health, ammo);
        }

        public string name; // закриті поля
        public int health, ammo;
    }

    public class Daemon : Monster
    {
        public Daemon()
        {
            brain = 1;
        }

        public Daemon(string name, int brain)
            : base(name)  // 1
        {
            this.brain = brain;
        }

        public Daemon(int health, int ammo, string name, int brain)
            : base(health, ammo, name)
        {
            this.brain = brain;
        }

        override public void Passport()
        {
            Console.WriteLine(
            "Daemon {0} \t health = {1} ammo = {2} brain = {3}",
            name, health, ammo, brain);
        }
        public void Think()
        {
            Console.Write( name + " is");
            for (int i = 0; i < brain; ++i);
            Console.Write( " thinking");
            Console.WriteLine("...");
        }

        int brain; // закрите поле
    }
}
```

Скомпілювавши бібліотеку, ви виявите файл *Monsterlib.dll* у каталогах *...\bin\ Debug і ...\obj\Debug*. Відкривши файл *Monsterlib.dll* за допомогою програми *ILdasm.exe*, можна отримати повну інформацію про створену бібліотеку (рис. 12.3).

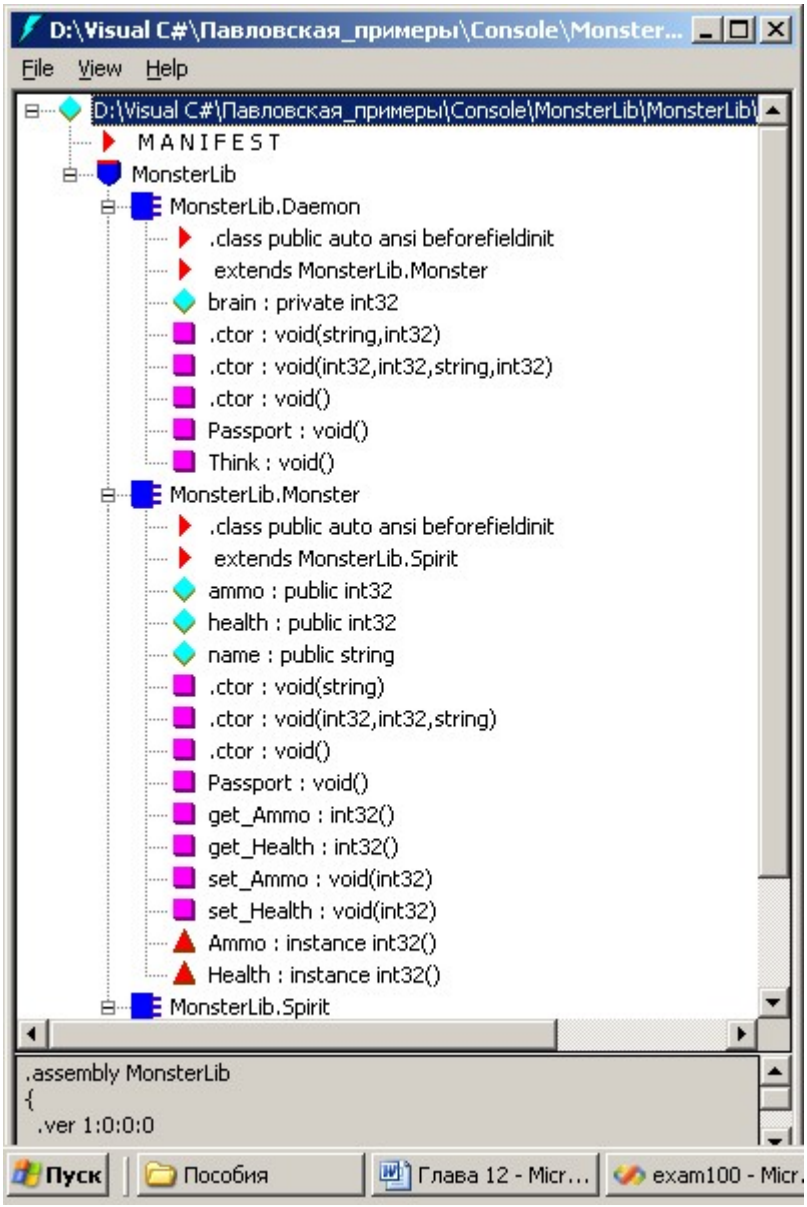



Рис. 12.3. Перегляд бібліотеки за допомогою дизасемблера Lidasm.exe

Будь-яка бібліотека - це сервер, що надає свої ресурси клієнтам. Створимо клієнтське застосування, що виконує ті ж функції, що і додаток з розділу “Віртуальні методи” (див. розділ 8, лістінг 8.3), але з використанням бібліотеки *Monsterlib.dll*. Для того, щоб компілятор міг її виявити, необхідно після створення проекту (як завжди, це - консольний додаток) підключити посилання на бібліотеку за допомогою команди Project > Add Reference (Додати посилання). Для пошуку каталогу, що містить бібліотеку, слід використовувати кнопку *Browse.*

Після підключення бібліотеки можна користуватися її відкритими елементами таким же чином, неначебто вони були описані в тому ж модулі. Текст додатку приведений в лістингу 12.2.

Лістинг 12.2. Клієнтське застосування

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

namespace exam100
{
   using MonsterLib;

   class Program
   {
      static void Main(string[] args)
      {
         const int n = 3;
         Monster[] stado = new Monster[n];
         stado[0] = new Monster( "Monia" );
         stado[1] = new Monster( "Monk" );
         stado[2] = new Daemon ( "Dimon", 3 );
         foreach ( Monster elem in stado ) elem. Passport();
         for ( int i = 0; i < n; ++i ) stado[i].Ammo = 0;
         Console.WriteLine();
         foreach ( Monster elem in stado ) elem.Passport();
      }
   }
}
```

Результат роботи програми:

```
Monster Monia      health = 100 ammo = 100
Monster Monk       health = 100 ammo = 100
Monster Dimon      health = 100 ammo = 100  brain = 3;

Monster Monia      health = 100 ammo = 0
Monster Monk       health = 100 ammo = 0
Monster Dimon      health = 100 ammo = 0  brain = 3;
```

Перевага .NET полягає в тому, що завдяки стандартним угодам можна використовувати бібліотеки незалежно від мови, на якій вони були написані. Таким чином, можна було б написати клієнтське застосування, наприклад, на мові VB.NET.

## 12.3. Рефлексія

*Рефлексія* - це отримання інформації про типи під час виконання програми. Наприклад, можна отримати список всіх класів і інтерфейсів збірки, список елементів кожного з класів, список параметрів кожного методу і так далі. Вся інформація береться з метаданих збірки. Для використання рефлексії необхідні клас *System.Type і* типи простору імен *System.Reflection*.

У класі *Type* описані методи, які дозволяють отримати інформацію про типи. У просторі імен *System.Reflecion* описані типи, що підтримують *Type*, а також класи, які служать для організації пізнього зв’язування і динамічного завантаження збірок.

Властивості і методи класу *Type* приведені в таблиці 12.1

Таблиця 12.1

Елементи класу *Type*

| Елемент | Опис |
|---|---|
| IsAbstract, IsArray, IsNestedPublic, IsClass, IsNestedPrivate, IsCOMObject, IsEnum, Islnterface, IsPrimitive, IsSealed, IsValueType | Властивості, що дозволяють отримати відповідні характеристики конкретного типу в програмі (наприклад, чи є він абстрактним, чи є він масивом, класом і т. п.). Приведені не всі властивості |
| GetConstructors, GetEvents, GetFields, GetInterfaces, GetMethods, GetMembers, GetNestedTypes, GetProperties | Методи, що повертають масив з набором відповідних елементів (конструкторів, подій, полів і т. п.). Повертаєме значення відповідає імені методу, наприклад, GetFields повертає масив типу FieldInfo, GetMethods - масив типу MethodInfo. Для кожного з методів є парний йому (без символу S в кінці імені), який призначений для роботи з одним заданим в параметрі елементом (наприклад, GetMethod і GetMethods) |
| FindMembers | Метод повертає масив типу MemberInfo на основі заданих критеріїв пошуку |
| GetType | Метод повертає об'єкт типу Type за іменем, заданому у вигляді рядка |
| InvokeMember | Метод використовується для пізнього зв’язування заданого елементу |

Скористатися цими методами можна після створення екземпляра класу Type. Оскільки це абстрактний клас, звичайний спосіб створення об'єктів за допомогою операції *new* непридатний, зате існують три інших способи:

1. У базовому класі *object* описаний метод *GetType*, яким можна скористатися для будь-якого об'єкту, оскільки він успадковується. Метод повертає об'єкт типу *Type*, наприклад:

```
Monster X = new Monster();
Type t = X.GetType();
```

2. У класі *Type* описаний статичний метод *GetType* з одним параметром рядкового типу, на місце якого потрібно передати ім'я класу (типу), наприклад:

```
Type t = Type.GetType("Monster");
```

3. Операція *typeof* повертає об'єкт класу *Type* для типу, заданого як параметр, наприклад:

Type t = typeof (Monster);

При використанні другого і третього способів створювати екземпляр досліджуваного класу немає необхідності.

Як видно з таблиці 12.1, багато методів класу *Type* повертають екземпляри стандартних класів (наприклад, *MemberInfo*). Ці класи описані в просторі імен *System.Reflection*. Найбільш важливі з цих класів перераховані в таблиці 12.2.

Таблиця 12.2

Деякі класи простору імен *System.Reflection*

| Тип | Опис |
|---|---|
| Assembly | Містить методи для отримання інформації про збірку, а також для завантаження збірки і виконання з нею різних операцій |
| AssemblyName | Дозволяє отримувати інформацію про збірку (ім'я, версія, сумісність, природна мова і т. п.) |
| EventInfo | Зберігає інформацію про подію |
| FieldInfo | Зберігає інформацію про поле |
| MemberInfo | Абстрактний базовий клас, що визначає загальні елементи для класів EventInfo, FieldInfo, MethodInfo і PropertyInfo |
| MethodInfo | Зберігає інформацію про метод |
| Module | Дозволяє звернутися до модуля в багатофайловій збірці |
| ParameterInfo | Зберігає інформацію про параметр |
| PropertyInfo | Зберігає інформацію про властивість |

У лістингу 12.3 приведені приклади використання методів і класів.

Лістинг 12.3. Отримання інформації про класи

```
using System;
using System.Reflection;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace exam100
{
   using MonsterLib;
   class Program
   {
      static void Info(Type t)
      {
         Console.WriteLine("======= Класс " + t.FullName);
         if (t.IsAbstract) Console.WriteLine("абстрактний ");
         if (t.IsClass)    Console.WriteLine("звичайний ");
```

```
            if (t.IsEnum)     Console.WriteLine("перечислений");
            Console.WriteLine();
            MethodInfo[] met = t.GetMethods();
            foreach (MethodInfo m in met) Console.WriteLine(m);
            Console.WriteLine();
            PropertyInfo[] prs = t.GetProperties();
            Console.WriteLine();
        }
        static void Main(string[] args)
        {
         Type t = typeof(Spirit);
         Info(t);
         t = typeof(Monster);
         Info(t);
         t = typeof(Daemon);
         Info(t);
        }
    }
}
```

Результат роботи програми приведений на рис. 12.4.

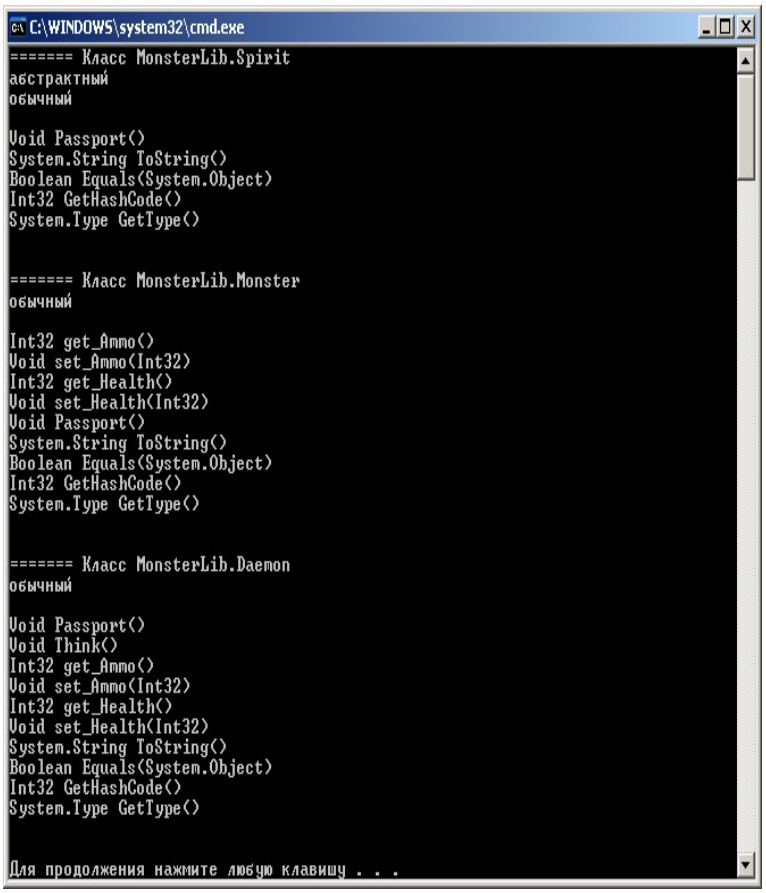



Рис. 12.4. Результат роботи програми

## 12.4. Атрибути

Атрибути - це додаткові відомості про елементи програми (класів, методів, параметрів і т. д.). За допомогою атрибутів можна додавати інформацію в метаданні збірки і потім витягувати її під час виконання програми. Атрибут є спеціальним видом класу і походить від базового класу *System. Attribute.*

Атрибути діляться на стандартні і призначених для користувача. У бібліотеці .NET передбачена множина стандартних атрибутів, які можна використовувати в програмах. Якщо всієї різноманітності стандартних атрибутів не вистачить, щоб задовольнити вимоги програміста, він може описати власні класи атрибутів, після чого застосовувати їх точно так, як і стандартні.

При використанні (специфікації) атрибутів вони задаються в секції атрибутів, що розташовується безпосередньо перед елементом, для опису якого вони призначені. Секція береться в квадратні дужки і може містити декілька атрибутів, що перераховуються через кому. Порядок проходження атрибутів довільний.

Для кожного атрибуту задаються ім'я, а також необов'язкові параметри і тип елементу збірки, до якого відноситься атрибут. Простий приклад атрибуту:

```
[Serializable]
class Monster{
[NonSerialized]
string name;
int health, ammo;}
```

Атрибут [*Serializable*] означає, що об'єкти цього класу можна зберігати в зовнішній пам'яті, відноситься до всього класу *Monster*. При цьому поле *name* помічене атрибутом [*Nonserialized*] говорить про те, що це поле зберігатися не повинно.

Зазвичай з контексту зрозуміло, до якого елементу збірки відноситься атрибут, проте в деяких випадках можуть виникнути неоднозначності. Для їх усунення перед ім'ям атрибуту записується тип елементу збірки - уточнююче ключове слово, відокремлюване від атрибуту двокрапкою. Ключові слова і відповідні елементи збірки, до яких можуть відноситися атрибути наведені в таблиці 12.3.

Таблиця 12.3

Типи елементу збірки, що задаються для атрибутів

| Ключове слово | Опис |
|---|---|
| assembly | Атрибут відноситься до всієї збірки |
| field | Атрибут відноситься до поля |
| event | Атрибут відноситься до події |
| method | Атрибут відноситься до методу |
| param | Атрибут відноситься до параметрів методу |
| property | Атрибут відноситься до властивості |
| return | Атрибут відноситься до повертає мого значення |
| type | Атрибут відноситься до класу або структури |

Нехай, наприклад, перед методом описаний гіпотетичний атрибут ABC:

```
[ABC]
public void Do()   { ... }
```

За умовчанням він відноситься до методу. Щоб вказати, що атрибут відноситься не до методу, а до його поверємого значення, слід написати:

```
[return:ABC]
public void Do()   { ... }
```

Атрибут може мати параметри. Вони записуються в круглих дужках через кому після імені атрибуту і бувають *позиційними і іменованими*. Іменований параметр указується у формі *ім'я = значення*, для позиційного просто задається значення. Наприклад, для використаного в наступному фрагменті коди атрибуту *ClsCompliant* заданий позиційний параметр *true*. Атрибути, що відносяться до збірки, повинні розташовуватися безпосередньо після директив *using*, наприклад:

```
using System;
[assembly:CLSCompliant(true)]
namespace ConsoleApplicationl
{
...

}
```

Атрибут [*CLSCompliant*] визначає, задовольняє програмний код угодам CLS (Common Language Specification) чи ні. Стандартні атрибути, як і інші типи класів, мають набір конструкторів, які визначають, яким чином використовувати (специфікувати) атрибут. Фактично, при використанні атрибуту указується найбільш відповідний конструктор, а величини, невказані в конструкторі, задаються через іменовані параметри в кінці списку параметрів.

Стандартний атрибут [*Stathread*], старанно видалений зі всіх лістингів в цій книзі, відноситься до методу, перед яким він записаний. Він має значення тільки для додатків, що використовують модель *COM*, і задає модель потоків в рамках моделі *COM*. Приклад застосування ще одного стандартного атрибуту, [*Conditional*], приведений далі в розділі “Директиви препроцесора”.

Атрибути рівня збірки зберігаються у файлі *AssemblyInfo.cs*, автоматично створюваному середовищем для будь-якого проекту. Для явного завдання номера версії збірки можна записати атрибут [*AssemblyVersion*], наприклад:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

## 12.5. Простір імен

Простір імен - це контейнер для типів, що визначає зону їх видимості. Просторам імен запобігають конфлікти імен і використовуються для двох взаємозв'язаних цілей:

– логічного групування елементів програми, розташованих в різних фізичних файлах;

– групування імен, що надаються збіркою в користування іншим модулям.

У всіх програмах, створених раніше, використовувався простір імен, що створюється за умовчанням. Реальні програми найчастіше розробляються групою програмістів, кожен з яких працює зі своїм набором фізичних файлів (одиниць компіляції), що зберігають елементи створюваного застосування. Якщо в різних файлах описати простори імен з одним і тим же ім'ям, то при побудові додатку, що складається з цих файлів, буде зкомпоновано єдиний простір імен.

Простори імен можуть бути вкладеними, наприклад:

```
namespace State
{
  namespace City
  {
    ...
  }
}
```

Таке оголошення аналогічно наступному:

```
namespace State.City
{
…
}
```

Вкладені простори імен, як ви напевно встигли відмітити, широко застосовуються в бібліотеці .NET.

Існує *три способи використання типу*, визначеного в якому-небудь просторі імен:

1. Використовувати повністю кваліфіковане ім'я. Наприклад, в просторі імен *System.Runtime.Serialization.Formatters.Binary* описаний клас BinaryFormatter.

Створення об'єкту цього класу за допомогою кваліфікованого імені виглядає так:

```
System.Runtime.Serialization.Formatters.Binary.BinaryFormatter bf =
new System.Runtime.Serialization.Formatters.Binary.BinaryFormatter();
```

2. Використовувати директиву *using*, за допомогою якої імпортуються всі імена із заданого простору. В цьому випадку попередній приклад прийме вигляд

```
using System.Runtime.Serialization.Formatters.Binary;
…
```

```
BinaryFormatter bf = new BinaryFormatter();
```

Директива *using* повинна розташовуватися зовні або усередині простору імен, проте до любих описів типів.

3. Використовувати псевдонім типу. Це робиться за допомогою другої форми директиви*using*:

```
using BinF =
   System.Runtime.Serialization.Formatters.Binary.BinaryFormatter;
…
BinF bf = new BinF();
```

Перший спосіб застосовується при одноразовому використанні імені типу “неглибоко” вкладених просторів імен, другий - в більшості решти випадків, що ми і робили у всіх прикладах, а третій можна рекомендувати при багаторазовому використанні довгого імені типу.

Має місце можливість застосовувати псевдонім простору імен за допомогою операції ::, наприклад:

```
using SI0 = System.I0                  // псевдонім простору імен
using MIO = MyLibrary.IO;              // псевдонім простору імен
class Program
{
      static void Main()
      {
        // використання псевдонімів
        SI0 :: Stream s = new MIO :: Empty Stream();
      …
      }
}
```

Використання псевдоніма для простору імен гарантує, що подальші підключення інших просторів імен до цієї збірки не вплинуть на існуючі визначення. Зліва від операції **::** можна вказати ідентифікатор *global*. Він гарантує, що пошук ідентифікатора, розташованого праворуч операції, виконуватиметься тільки в глобальному просторі імен. Мета користування цього ідентифікатора та ж: не допустити змін існуючих визначень при розробці наступних версій програми, в яких в неї бути додані нові простори імен, що містять елементи з такими іменами.

Таким чином, збірки забезпечують фізичне групування типів, а простори імен - логічне. В світі мережевого програмування, коли програмістові доступні десятки тисяч класів, простори імен абсолютно необхідні як для класифікації і пошуку, так і для запобігання конфліктам імен типів.

## 12.6. Директиви препроцесора

*Препроцесором* в мові С ++ називається попередній етап компіляції, що формує остаточний варіант тексту програми. У мові C#, нащадку C++, препроцесор практично відсутній, але деякі директиви збереглися. Призначення директив - виключати з процесу компіляції фрагменти коду при виконанні певних умов, виводити повідомлення про помилки і попередження, а також структурувати код програми. Кожна директива розташовується на окремому рядку і не закінчується крапкою з комою, на відміну від операторів мови. У одному рядку з директивою може розташовуватися тільки коментар виду //. Перелік і короткий опис директив приведені в таблиці 12.4.

Розглянемо детальніше застосування директив умовної компіляції. Вони використовуються для того, щоб виключити компіляцію окремих частин програми. Це буває корисно при відладці або, наприклад, за підтримки декількох версій програми для різних платформ.

Таблиця 12.4

Директиви препроцесора

| Найменування | Опис |
|---|---|
| #define, #undef | Визначення (наприклад, #define DEBUG) і відміна визначення (#undef DEBUG) символьної константи, яка використовується директивами умовної компіляції. Директиви розміщуються до першої лексеми одиниці компіляції. Допускається повторне визначення однієї і тієї ж константи |
| #if,#elif, #else, #endif | Директиви умовної компіляції. Код, що знаходиться в області їх дії, компілюється або пропускається залежно від того, чи була раніше визначена символьна константа |
| #line | Завдання номера рядка і імені файлу, про який видаються повідомлення, що виникають при компіляції (наприклад, #line 200 "ku_ku.txt"). При цьому в діагностичних повідомленнях компілятора ім'я компільованого файлу замінюється вказаним, а рядки нумеруються, починаючи із заданого першим параметром номера |
| # error, #warning | Виведення *при компіляції* повідомлення, вказаного в рядку директиви. Після виконання директиви компіляція припиняється (наприклад, #error Далі компілювати не можна). Після виконання директиви #warning компіляція продовжується |
| #region, #endregion | Визначення фрагмента коду, який можна буде скрутити або розвернути засобами редактора коду. Фрагмент розташовується між цими директивами |
| #pragma | Дозволяє відключити (#pragma warning disable) або відновити (#pragma warning restore ) видачу всіх або перерахованих в директиві попереджень компілятора |

Формат директив:

**# константний_вираз**
**…**
**[ #elif константний_вираз**
**...]**
**[ #elif константний_вираз**
**...]**
**[ #else**
**... ]**
**#endif**

Кількість директив #elif довільна. Блоки коду, що виключаються, можуть містити як описи, так і виконуваних операторів. Константний вираз може містити одну або декілька символьних констант, об'єднаних знаками операцій = =, !=, !, && і ||. Також допускаються круглі дужки. Константа вважається рівною *true*, якщо вона була раніше визначена за допомогою директиви *#define*. Приклад застосування директив приведений на лістингу 12.4.

Лістинг 12.4. Застосування директив умовної компіляції

```
// #define VAR1
// #define VAR2

using System;

namespace ConsoleApplication1
{
        class Classl
        {
#if     VAR1

                static void F(){ Console.WriteLine( "Вариант 1" ): }
#elif   VAR2
                static void F(){ Console.WriteLine( "Вариант 2" ); }
#else
                static void F(){ Console.WriteLine( "Основной вариант" ); }
#endif
                        static void Main()
                        {
                                                F();
                        }
        }
}
```

Залежно від того, визначення якої символьної константи розкоментувати, в компіляції братиме участь один з трьох методів *F*.

Директива *#define* застосовується не тільки у поєднанні з директивами умовної компіляції. Можна застосовувати її разом із стандартним атрибутом *Conditional* для умовного управління виконанням методів. Метод виконуватиметься, якщо константа визначена. Приклад приведений в лістингу 12.5. Зверніть увагу на те, що для застосування атрибуту необхідно підключити простір імен *System.Diagnostics*.

Лістинг 12.5. Використання атрибуту Conditional

```
// #define VAR1
#define VAR2
using System;
using System.Diagnostics;

namespace ConsoleApplication1
{
        class Classl
        {
                [Conditional ("VAR1")]
                static void A(){ Console.WriteLine( " Виконується метод A" ); }
                [Conditional ("VAR2")]
                static void B(){ Console.WriteLine( " Виконується метод B" ); }

                static void Main()
                {
                A(); B();
                }
        }
}
```

У методі *Main* записані виклики обох методів, проте в даному випадку буде виконаний тільки метод *B*, оскільки символьна константа *Var1* не визначена.

# РОЗДІЛ 13. СТРУКТУРИ ДАНИХ, КОЛЕКЦІЇ І КЛАСИ-ПРОТОТИПИ

У даному розділі приводиться огляд основних структур даних і їх реалізація в бібліотеці *.NET* у вигляді колекцій. Крім того, описуються класи-прототипи (*generics*), часткові типи (*partial types*) і типи, що обнуляються (*nullable types*).

## 13.1. Абстрактні структури даних

Найчастіше в програмах використовуються наступні структури даних: масив, список, стек, черга, бінарне дерево, хеш-таблиця, граф і множина. Далі надані короткі характеристики кожній їз цих структур.

*Масив* - це кінцева сукупність однотипних величин. масив займає безперервну область пам'яті і надає доступ до своїх елементів по індексу. Пам'ять під масив виділяється до початку роботи з ним і згодом не змінюється.

*У списку* кожен елемент пов'язаний з наступним і, можливо, з попереднім. У першому випадку список називається однозвязковим, в другому - двозвязковим. Також застосовуються терміни однонапрямковим і двонапрямковим. Якщо останній елемент зв'язати покажчиком з першим, виходить кільцевий список. Кількість елементів в списку може змінюватися в процесі роботи програми.

Кожен елемент списку містить ключ, що ідентифікує цей елемент. Ключ зазвичай буває або цілим числом, або рядком і є частиною даних, що зберігаються в кожному елементі списку. Як ключ в процесі роботи із списком можуть виступати різні частини даних. Наприклад, якщо створюється список із записів, що містять прізвище, рік народження, стаж роботи і стать, будь-яка частина запису може виступати як ключ: при впорядковуванні списку по зростанню ключа буде прізвище, а при пошуку, наприклад, ветеранів праці ключем можна зробити стаж. Ключі різних елементів списку можуть збігатися.

Над списками можна виконувати наступні операції:

- додавання елементу в кінець списку;
- читання елементу із заданим ключем;
- вставка елементу в задане місце списку;
- видалення елементу із заданим ключем;
- впорядковування списку по ключу.

Список не забезпечує довільний доступ до елементу, тому при виконанні операцій читання, вставки і видалення виконується послідовний перебір елементів, поки не буде знайдений елемент із заданим ключем. Для списків великого об'єму перебір елементів може займати значний час, оскільки середній час пошуку елементу пропорційний кількості елементів в списку.

*Стек* - це окремий випадок однонапрямкового списку, додавання елементів в який і вибірка з якого виконуються з одного кінця, званого вершиною стека. Інші операції із стеком не визначені. При вибірці елемент виключається із стека. Говорять, що стек реалізує принцип обслуговування LIFO (Last In - First

Out, останнім прийшов - першим пішов). Стек найпростіше уявити собі як закриту з одного кінця вузьку трубу, в яку кидають м'ячики. Дістати перший кинутий м'ячик можна тільки після того, як вийняті всі останні. Стеки широко застосовуються в системному програмному забезпеченні, компіляторах, в різних рекурсивних алгоритмах.

*Черга* - це окремий випадок однонапрямкового списку, додавання елементів в який виконується в один кінець, а вибірка - з іншого кінця. Інші операції з чергою не визначені. При вибірці елемент виключається з черги. Говорять, що чергу реалізує принцип обслуговування FIFO (First In - First Out, першим прийшов - першим пішов). Чергу найпростіше уявити собі, постоявши в ній час-другий. У програмуванні черги застосовуються, наприклад, в моделюванні, диспетчеризації завдань операційною системою.

*Бінарне дерево* - це динамічна структура даних, що складається з вузлів, кожен з яких містить окрім даних не більше двох посилань на різні бінарні піддерева. На кожен вузол є рівно одне посилання. Початковий вузол називається коренем дерева.

Приклад бінарного дерева приведений на рис. 13.1 (корінь зазвичай зображається зверху). Вузол, що не має піддерев, називається листом. Витікаючи вузли називаються предками, що входять - нащадками. Висота дерева визначається кількістю рівнів, на яких розташовуються його вузли.

Якщо дерево організоване таким чином, що для кожного вузла всі ключі його лівого піддерева менші ключа цього вузла, а всі ключі його правого піддерева - більші, воно називається деревом пошуку. Однакові ключі не допускаються.

У дереві пошуку можна знайти елемент по ключу, рухаючись від кореня і переходячи на ліве або праве піддерево залежно від значення ключа в кожному вузлі. Такий пошук набагато ефективніший пошуку за списком, оскільки час пошуку визначається висотою дерева, а вона пропорційна двійковому логарифму кількості вузлів.

Втім, швидкість пошуку в значній мірі залежить від порядку формування дерева: якщо на вхід подається впорядкована або майже впорядкована послідовність ключів, дерево вироджується в список. Для прискорення пошуку застосовується процедура балансування дерева, що формує дерево, піддерева якого розрізняються не більше ніж на один елемент. Бінарні дерева застосовуються для ефективного пошуку і сортування даних.

Хеш-таблиця, асоціативний масив, або словник - це масив, доступ до елементів якого здійснюється не по номеру, а по деякому ключу. Можна сказати, що це таблиця, що складається з пар “ключ-значення” (таблиця 13.1). Хеш-таблиця ефективно реалізує операцію пошуку значення по ключу. При цьому ключ перетвориться в число (хеш - код), яке використовується для швидкого знаходження потрібного значення в хеш-таблиці.

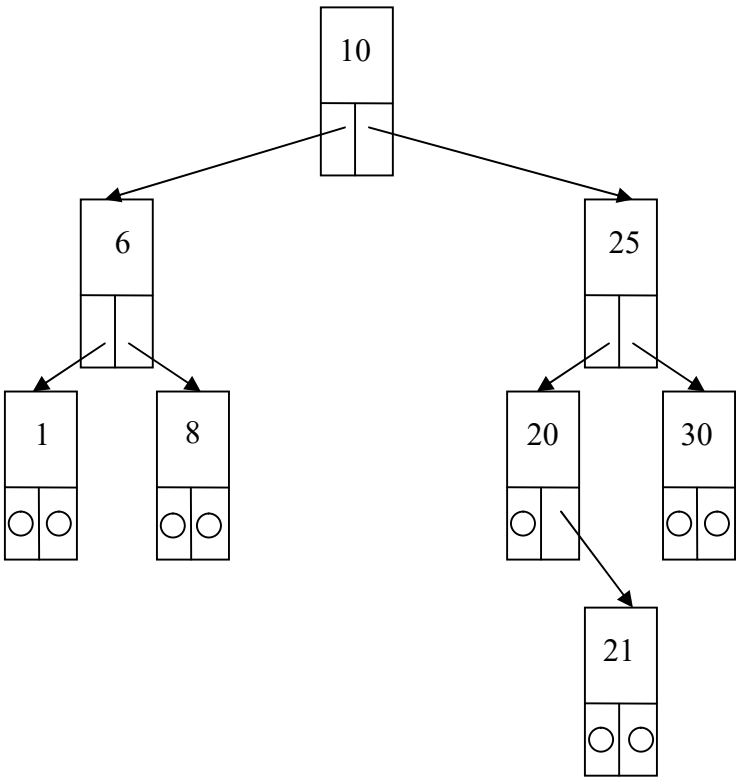



Рис. 13.1. Приклад бінарного дерева пошуку

Таблиця 13.1

Приклад хеш-таблиці

| Ключ | Значення |
|---|---|
| boy | хлопчик |
| girl | дівчинка |
| dog | собачка |

Перетворення виконується за допомогою *хеш-функції*, або *функції розстановки*. Ця функція проводить які-небудь перетворення внутрішнього представлення ключа. Наприклад, обчислює середнє арифметичне кодів символів ключа. Якщо хеш-функція розподіляє сукупність можливих ключів рівномірно по множині індексів масиву, то доступ до елементу по ключу виконується майже так само швидко, як в масиві. Якщо хеш-функція генерує для різних ключів однакові хеш-коди, час пошуку зростає і стає порівнянним з часом пошуку в списку.

*Сенс* хеш-функції полягає в тому, щоб відобразити широку множину ключів у вужчу множину індексів. При цьому неминуче виникають так звані колізії, коли хеш-функція формує для двох різних елементів одні і ті ж хеш-коди.

*Граф* - це сукупність вузлів і ребер, що сполучають різні вузли. Наприклад, можна уявити собі карту автомобільних доріг як граф з містами, як вузли і шосе між містами, як ребра. Множина реальних практичних завдань можна описати в термінах графів, що робить їх структурою даних, часто використовуваною при написанні програм.

*Множина* - це неврегульована сукупність елементів. Для множин визначені операції перевірки приналежності елементу множині, включення і виключення елементу, а також об'єднання, перетину і віднімання множин.

Описані структури даних називаються абстрактними, оскільки в них не задається реалізація допустимих операцій.

У бібліотеках більшості сучасних об'єктно-орієнтованих мов програмування представлені стандартні класи, що реалізовують основні абстрактні структури даних. Такі класи називаються колекціями, або контейнерами. Для кожного типу колекції визначені методи роботи з її елементами, не залежні від конкретного типу даних, які зберігаються в колекції, тому один і той же вид колекції можна використовувати для зберігання даних різних типів. Використання колекцій дозволяє скоротити терміни розробки програм і підвищити їх надійність.

Кожен вид колекції підтримує свій набір операцій над даними, і швидкодія цих операцій може бути різною. Вибір виду колекції залежить від того, що потрібно робити з даними в програмі і які вимоги пред'являються до її швидкодії. Наприклад, при необхідності частої вставки і видалення елементів з середини послідовності слід використовувати *список*, а не масив, а якщо включення елементів виконується головним чином в кінець або початок послідовності - *чергу*. Тому вивчення можливостей стандартних колекцій і їх грамотне застосування є необхідними умовами створення ефективних і професійних програм.

У бібліотеці .NET визначена множина стандартних класів, що реалізовують більшість перерахованих раніше абстрактних структур даних.

Основні простори імен, в яких описані ці класи:

- *System.Collections*;
- *System.Col1ections.Specialized*;
- *System.Col1ections.Generic*.

У наступних розділах коротко описані основні елементи цих просторів імен.

## 13.2. Простір імен *System.Collections*

У просторі імен *System.Collections* визначені набори стандартних колекцій і інтерфейсів, які реалізовані в цих колекціях. У таблиці 13.2 приведені найбільш важливі інтерфейси.

Таблиця 13.2

Інтерфейси простору імен System.Collections

| Інтерфейс | Призначення |
|---|---|
| ICollection | Визначає загальні характеристики (наприклад, розмір) для набору елементів |
| IComparer | Дозволяє порівнювати два об'єкти |
| IDictionary | Дозволяє представляти вміст об'єкту у вигляді пар “ім'я-значення” |
| IDictionaryEnumerator | Використовується для нумерації вмісту об'єкту, що підтримує інтерфейс *IDictionary* |

Продовження таблиці 13.2

| Інтерфейс | Призначення |
|---|---|
| IEnumerable | Повертає інтерфейс *IEnumerator* для вказаного об'єкту |
| IEnumerator | Зазвичай використовується для підтримки оператора *foreach* відносно об'єктів |
| IHashCodeProvider | Повертає хеш-код для реалізації типу із застосуванням вибраного користувачем алгоритму хешування |
| IList | Підтримує методи додавання, видалення і індексування елементів в списку об'єктів |

У таблиці 13.3 перераховані основні колекції, визначені в просторі *System. Collections.*

Таблиця 13.3

Колекції простору імен *System.Collections*

| Клас | Призначення | Найважливіші з реалізованих інтерфейсів |
|---|---|---|
| ArrayList | Масив, що динамічно змінює свій розмір | IList, ICollection, IEnumerable, ICloneabl |
| BitArray | Компактний масив для зберігання бітових значень | ICollection, IEnumerable, ICloneable |
| Hashtable | Хеш-таблиця | IDictionary, ICollection, IEnumerable, ICloneable |
| Queue | Черга | ICollection, ICloneable, IEnumerable |
| SortedList | Колекція, відсортована по ключах. Доступ до елементів - по ключу або по індексу | IDictionary, ICollection, IEnumerable, ICloneable |
| Stack | Стек | ICollection, IEnumerable |

Простір імен System.Collections.Specialized включає спеціалізовані колекції, наприклад колекцію рядків StringCollection і хеш- таблицю із рядковими ключами StringDictionary.

Як приклад стандартної колекції розглянемо клас *ArrayList.*

## 13.3. Клас *ArrayList*

Основним недоліком звичайних масивів є те, що об'єм пам'яті, який потрібен для зберігання їх елементів, має бути виділений до початку роботи з маси-

вом. Клас *ArrayList* дозволяє програмістові не піклуватися про виділення пам'яті і зберігати в одному і тому ж масиві елементи різних типів.

За умовчанням при створенні об'єкту типу *ArrayList* будується масив з 16 елементів типу *object*. Можна задати бажану кількість елементів в масиві, передавши його в конструктор або встановивши як значення властивості *Capacity*, наприклад:

```
ArrayList arrl = new ArrayList();          // створюється масив з 16 елементів
ArrayList arr2 = new ArrayList(1000);  // створюється масив з 1000 елементів
ArrayList arr3 = new ArrayList();
arr3.Capacity = 1000;                          // кількість елементів задається
```

Основні методи і властивості класу *Arraylist* перераховані в таблиці 13.4.

Таблиця 13.4

Основні елементи класу Arraylist

| Елемент | Вигляд | Опис |
|---|---|---|
| Capacity | Властивість | Ємність масиву (кількість елементів, які можуть зберігатися в масиві) |
| Count | Властивість | Фактична кількість елементів масиву |
| Item | Властивість | Отримати або встановити значення елементу по заданому індексу |
| Add | Метод | Додавання елементу в кінець масиву |
| AddRange | Метод | Додавання серії елементів в кінець масиву |
| BinarySearch | Метод | Двійковий пошук у відсортованому масиві або його частині |
| Clear | Метод | Видалення всіх елементів з масиву |
| Clone | Метод | Поверхневе копіювання елементів одного масиву в інший масив |
| CopyTo | Метод | Копіювання всіх або частини елементів масиву в одновимірний масив |
| GetRange | Метод | Набуття значень підмножини елементів масиву у вигляді об'єкту типу *Arraylist* |
| IndexOf | Метод | Пошук входження елементу (повертає індекс елементу або -1, якщо елемент не знайдений) |
| Insert | Метод | Вставка елементу в задану позицію (по заданому індексу) |
| InsertRange | Метод | Вставка групи елементів, починаючи із заданої позиції |
| LastIndexOf | Метод | Пошук останнього входження елементу в одновимірний масив |
| Remove | Метод | Видалення першого входження заданого елементу |
| RemoveAt | Метод | Видалення елементу з масиву по індексу |

Продовження таблиці 13.4

| Елемент | Вигляд | Опис |
|---|---|---|
| RemoveRange | Метод | Видалення групи елементів з масиву |
| Reverse | Метод | Зміна порядку проходження елементів на зворотний |
| SetRange | Метод | Установка значень елементів масиву в заданому діапазоні |
| Sort | Метод | Впорядковування елементів масиву або його частини |
| TrimToSize | Метод | Установка ємності масиву рівною фактичній кількості елементів |

Клас *ArrayList* реалізований через клас *Array*, тобто містить закрите поле цього класу. Оскільки всі типи в C# є нащадками класу *object*, масив може містити елементи довільного типу. Навіть якщо в масиві зберігаються звичайні цілі числа, тобто елементи значущого типу, внутрішній клас є масивом посилань на екземпляри типу *object*, які є упакованим типом-значенням. Відповідно, при занесенні в масив виконується упаковка, а при витяганні - розпаковування елементу. Це не може не позначитися на швидкодії алгоритмів, які використовують *ArrayList*.

Якщо при додаванні елементу в масив виявляється, що фактичну кількість елементів масиву перевищує його ємність, вона автоматично подвоюється, тобто відбувається повторне виділення пам'яті і переписування туди всіх існуючих елементів.

Приклад занесення елементів в екземпляр класу *ArrayList*:

```
arrl.Add(123);
arrl.Add( -2 );
arrl.Add("Вася");
```

Доступ до елементу виконується по індексу, проте при цьому необхідно явним чином привести отримане посилання до цільового типу, наприклад:

```
int a = (int) arrl [0];
int b = (int) arrl [l];
string s = (string) arrl [2];
```

Спроба приведення до невідповідного типу зберігається в елементі, викликає генерацію виключення *InvalidCastException.*

Для підвищення надійності програм застосовується наступний прийом: екземпляр класу *ArrayList* оголошується закритим полем класу, в якому необхідно зберігати колекцію значень певного типу, а потім описуються методи роботи з цією колекцією, які делегують свої функції методам *ArrayList.* Цей спосіб ілюструється в лістингу 13.1, де створюється клас для зберігання об'єктів типу *Monster* і похідних від нього.

Лістинг 13.1. Колекція об'єктів

```
using System;
using System.Collections;
namespace ConsoleApplicationl
{
    class Monster { ... }
    class Daemon : Monster { ... }

    class Stado : IEnumerable
    {
        private ArrayList list;
        public Stado()                  { list = new ArrayList(); }
        public void Add( Monster m )    { list.Add( m ) ; }
        public void RemoveAt( int i )   {list.RemoveAt( i ); }
        public void Clear()             { list.Clear(); }
        public IEnumerator GetEnumerator()
        { return list.GetEnumerator(); }
    }
    class Classl
    {
        static void Main()
        {
            Stado stado = new Stado();
            stado.Add( new Monster( "Monia" ) );
            stado.Add( new Monster(  "Monk" ) );
            stado.Add( new Daemon ( "Dimon", 3 ) );
            stado.RemoveAt( 1 );
            foreach ( Monster x in stado ) x. Passport();
        }
    }
 }
```

Результат роботи програми:

```
Monster Monia      health = 100 ammo = 100
Daemon Dimon       health = 100 ammo = 100 brain = 3
```

Недоліком цього рішення є те, що для кожного методу стандартної колекції доводиться описувати метод-оболонку, що викликає стандартний метод. У C# з'явилися класи-прототипи (*generics*), що дозволяють вирішити цю проблему. Ми розглянемо їх в наступному розділі.

## 13.4. Класи-прототипи

Багато алгоритмів не залежать від типів даних, з якими вони працюють. Простими прикладами можуть служити сортування і пошук. Можливість відокремити алгоритми від типів даних надають класи-прототипи (*generics*) - класи, що мають як параметри типи даних. Найчастіше ці класи застосовуються для зберігання даних, тобто як контейнерні класи, або колекції.

У другу версію бібліотеки .NET додані колекції, що параметризуються, для представлення основних структур даних - стека, черги, списку, словника і так

далі. Ці колекції, розташовані в просторі імен *System.Collections.Generic*, дублюють аналогічні колекції простору імен *System.Collections*, розглянуті в розділі "Простір імен *System.Collections*". У таблиці 13.5 приводиться відповідність між звичайними та парамеризованими колекціями бібліотеки .NET (параметри, що визначають типи даних, що зберігаються в колекції, вказані в кутових дужках).

Таблиця 13.5

Параметризовані колекції бібліотеки .NET

| Клас-прототип | Звичайний клас |
|---|---|
| Comparer<T> | Comparer |
| Dictionary<K,T> | HashTable |
| LinkedList<T> | -- |
| List<T> | ArrayList |
| Queue<T> | Queue |
| SortedDictionary<K,T> | SortedList |
| Stack<T> | Stack |

У колекцій, описаних в бібліотеці .NET версій 1.0 і 1.1, є два основні недоліки, обумовлених тим, що в них зберігаються посилання на тип *object*:

- у одній і тій же колекції можна зберігати елементи будь-якого типу, отже, помилки при переміщенні в колекцію неможливо проконтролювати на етапі компіляції, а при витяганні елементу потрібне його явне перетворення;
- при зберіганні в колекції елементів значущих типів виконується великий об'єм дій з упаковки і розпаковування елементів, що в значній мірі знижує ефективність роботи.

Параметром класу-прототипу є тип даних, з яким він працює. Це позбавляє від перерахованих недоліків. Як приклад розглянемо застосування універсального "двійника" класу *Arraylist* - класу *List<t>* - для зберігання колекції об'єктів класів *Monster* і *Daemon*, а також для зберігання цілих чисел.

Лістинг 13.2. Використання універсальної колекції List<T>

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

namespace ConsoleApplication1
{
    using MonsterLib;
    class Program
    {
        static void Main()
        {
            List<Monster> stado = new List<Monster>();
            stado.Add( new Monster(  "Monia" ) );
            stado.Add( new Monster(  "Monk" ) );
            stado.Add( new Daemon ( "Dimon", 3 ) );
```

```
            foreach ( Monster x in stado ) x.Passport();
            List<int> lint = new List<int>();
            lint.Add( 5 ); lint.Add( 1 ); lint.Add( 3 );
            lint.Sort();
            int a = lint[2];
            Console.WriteLine( a );
            foreach ( int x in lint ) Console.Write( x + " " );
        }
    }
}
```

Результат роботи програми:

```
Monster Monia     health = 100 ammo = 100
Monster Monk      health = 100 ammo = 100
Daemon Dimon      health = 100 ammo = 100 brain = 3
5
1 3 5
```

У лістингу 13.2 дві колекції. Перша (*stado*) містить елементи, що призначені для користувача класів, які знаходяться в бібліотеці *Monsterlib.dll*, створеною в попередньому розділі. У колекції, для якої оголошений тип елементів *Monster*, завдяки поліморфізму можна зберігати елементи будь-якого похідного класу, але не елементи інших типів.

Здавалося б, в порівнянні із звичайними колекціями це обмеження, а не універсальність, проте на практиці колекції, в яких дійсно потрібно зберігати значення різних, не зв'язаних між собою типів, майже не використовуються. Гідністю ж такого обмеження є те, що компілятор може виконати контроль типів під час компіляції, а не виконання програми, що підвищує її надійність і спрощує пошук помилок.

Колекція int складається з цілих чисел, причому для роботи з ними не потрібні ні операції упаковки і розпаковування, ні явні перетворення типу при отриманні елементу з колекції, як це було в звичайних колекціях (див. лістинг 13.1).

Класи-прототипи називають також родовими або шаблонними, оскільки вони є зразками, по яких під час виконання програми будуються конкретні класи. При цьому відомості про класи, які є параметрами класів-прототипів, витягуються з метаданих.

Використання стандартних параметризованих колекцій для зберігання і обробки даних є хорошим стилем програмування, оскільки дозволяє скоротити терміни розробки програм і підвищити їх надійність.

У лістингу 13.3 приведений ще один приклад застосування параметризованих колекцій. Програма зчитує вміст текстового файлу, розбиває його на слова і підраховує кількість повторень кожного слова в тексті. Для зберігання слів і числа їх повторень використовується словник Dictionary<T,K>. У цього класу два параметри: тип ключів і тип значень, що зберігаються у словнику. Як ключі використовуються слова, зчитані з файлу, а значеннями є лічильники цілого типу, які збільшуються на одиницю, коли слово зустрічається в черговий раз.

Лістинг 13.3. Формування частотного словника

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
namespace ConsoleApplicationl
{
    class Program
    {
        static void Main()
        {
            StreamReader f = new StreamReader( @"d:\C#\text.txt");          //1
            string s = f.ReadToEnd();                                       // 2
            char[] separators = {'.', ' ', ',' , '!' };                     //3
            List<string> words = new List<string> ( s.Split(separators) );  //4
            Dictionary<string, int> map = new Dictionary<string, int>();    //5

            foreach ( string w in words )                                   //6
            {
                if ( map.ContainsKey( w ) ) map[w]++;
                else                        map[w] = 1;
            }
            foreach ( string w in map.Keys )                                //7
            Console.WriteLine(  "{0}\t{1}", w, map[w] );
        }
    }
}
```

Нехай початковий файл text. txt містить рядки:
*Ехал Грека через реку. Видит Грека, в реке рак.*
*Сунул Грека в реку руку, рак за руку Греку цап!*

Тоді результат роботи програми виглядатиме так:

| | |
|---|---|
| Ехал | 1 |
| Грека | 3 |
| через | 1 |
| реку | 2 |
| | 4 |
| Видит | 1 |
| в | 2 |
| реке | 1 |
| рак | 2 |
| | |
| Сунул | 1 |
| руку | 2 |
| за | 1 |
| Греку | 1 |
| цап | 1 |

Декілька пояснень до програми. У операторові 1 відкривається текстовий файл, довжина якого не повинна перевищувати 32767 символів, тому що в операторові 2 весь його вміст зчитується в окремий рядок.

Звичайно, для реальної роботи такий спосіб не рекомендується. Крім того, для файлів, що відкриваються для читання, програма обов'язково повинна обробляти виключення *FileNotFoundException.*

У операторові 3 задається масив роздільників, переданий як параметр методу *Split*, що формує масив рядків, кожен з яких містить окреме слово початкового файлу. Цей масив використовується для ініціалізації екземпляра *words* класу List<string>. Застосування стандартного класу дозволяє не піклуватися про виділення місця під масив слів.

Оператор 5 описує словник, а в циклі 6 виконується його заповнення шляхом перегляду списку слів *words*. Якщо слово зустрічається вперше, в значення, відповідне слову як ключу, заноситься одиниця. Якщо слово вже зустрічалося, значення збільшується на одиницю.

У циклі 7 виконується виведення словника шляхом перегляду всіх його ключів (для цього використовується властивість словника *Keys*, що повертає колекцію ключів) і вибірки відповідних значень.

Метод *Split* не дуже інтелектуальний: він розглядає пропуск, розташований після розділового знаку, як окреме слово. Для точнішого розбиття на слова використовуються регулярні вирази, які розглядаються в наступному розділі.

Зверніть увагу на те, наскільки використання стандартних колекцій скорочує початковий текст програми. Звичайно, на ретельне вивчення їх можливостей потрібно багато часу, проте це окупається багато разів.

Для повноти картини слід додати, що разом з параметризованими класами в просторі імен *System.Collections.Generic* описані параметризовані інтерфейси, які перераховані в таблиці 13.6.

Таблиця 13.6

Параметризовані інтерфейси бібліотеки .NET

| Парамеризований інтерфейс | Звичайний інтерфейс |
|---|---|
| ICollection<T> | ICollection |
| IComparable<T> | IComparable |
| IDictionary<K,T | IDictionary |
| IEnumerable<T> | IEnumerable |
| IEnumerator<T> | IEnumerator |
| IList<T> | IList |

## 13.5. Створення класу-прототипу

Мова C# дозволяє створювати власні класи-прототипи і їх різновиди - інтерфейси, структури, делегати і події, а також узагальнені (generic) методи звичайних класів.

Розглянемо створення класу-прототипу на прикладі стека, приведеному в специфікації C#. Параметр типу даних, які зберігаються в стеку, указується в кутових дужках після імені класу, а потім використовується таким же чином, як і звичайні типи:

```
public class Stack<T>
{
  T[] items:
  int count;
  public void Push(T item ){...}  // переміщення в стек
  public T      Pop(){  ...  }        // витягання із стека
}
```

При використанні цього класу на місце параметра T підставляється реальний тип, наприклад int:

```
Stack<int> stack = new Stack<int>();
stack.Push( 3 ) ;
int x = stack.Pop();
```

Тип Stack<int> називається *сконструйованим* типом (constructed type). Цей тип створюється під час виконання програми при його першій згадці в програмі. Якщо в програмі зустрінеться клас *Stack* з іншим значущим типом, наприклад *double*, середовище виконання створить іншу копію коду для цього типу. Навпаки, для всіх посилальних типів буде використана одна і та ж копія коду, оскільки робота з покажчиками на різні типи виконується однаковим чином. Клас-прототип може містити довільну кількість параметрів типу. Для кожного з них можуть бути задані обмеження (constraints), вказуючи, яким вимогам повинен задовольняти аргумент, відповідний цьому параметру, наприклад, може бути вказано, що це має бути значущий тип або тип, який реалізує деякий інтерфейс.

Синтаксично обмеження задаються після ключового слова where, наприклад:

```
public class Stack<T>
where T : struct
{ ... }
```

Тут за допомогою слова struct записано обмеження, що елементи стека мають бути значущого типу. Для посилального типу вживається ключове слово class. Для кожного типу, класу, що є параметром, може бути задано один рядок обмежень, який може включати один клас, а за ним - довільна кількість інтерфейсів, що перераховуються через кому.

Вказівка як обмеження імені класу означає, що відповідний аргумент може бути ім'ям або цього класу, або його нащадка. Вказівка імені інтерфейсу означає, що тип-аргумент повинен реалізовувати даний інтерфейс - це дозволяє використовувати усередині класу-прототипу, наприклад, операції по перерахуванню елементів, їх порівнянню і тому подібне.

Окрім класу і інтерфейсів в обмеженнях можна задати вимогу, щоб тип-аргумент мав конструктор за умовчанням без параметрів, що дозволяє створювати об'єкти цього типу в тілі методів класу-прототипу. Ця вимога записується у вигляді виразу new(), наприклад:

```
public class EntityTable<K, E>
where K: IComparable<K>, IPersistable
where E: Entity. new()
{
      public void Add( K key. E entity )
      {  …
      if ( key.CompareTo( x ) < 0 )   { ... }
      …
      }
}
```

В даному прикладі на перший аргумент класу Entitytable накладаються два обмеження по інтерфейсах, а для другого аргументу задано, що він може бути тільки класом Entity або його нащадком і мати конструктор без параметрів.

Завдання обмежень дозволяє компілятору виконувати більш строгий контроль типів і, таким чином, уникнути багатьох помилок, які інакше виявилися б тільки під час виконання програми.

Для завдання типізованим елементам класу-прототипу значень за умовчанням використовується ключове слово *default*. При цьому елементам посилального типу привласнюється *null*, а елементам значущого типу - 0.

## 13.6. Узагальнені методи

Іноді зручно мати окремий метод, який параметризується яким-небудь типом даних. Розглянемо цей випадок на прикладі методу сортування.

Відомо, що “самого кращого” алгоритму сортування не існує. Стандартні методи сортування реалізують алгоритми, які хороші для більшості застосувань, але не для всіх, тому може виникнути необхідність реалізувати власний метод.

У лістингу 13.4 приведений приклад сортування методом вибору. Алгоритм полягає в тому, що спочатку вибирається найменший елемент масиву і міняється місцями з першим елементом, потім є видимими елементи, починаючи з другого, і найменший з них міняється місцями з другим елементом, і т. д., всього *n - 1 раз.* На останньому проході циклу при необхідності міняються місцями попередній і наступний елементи масиву.

Лістинг 13.4. Сортування вибором

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

```
namespace ConsoleApplicationl
{
    class Program
    {
        static void Sort<T> ( ref T[] a )                                   // 1
            where T : IComparable<T>                                        // 2
        {
            T buf;
            int n = a.Length;
            for ( int i = 0; i < n - 1; ++i )
            {
                int im = i;
                for ( int j = i + 1; j < n; ++j )
                if (  a[j].CompareTo(a[im])  < 0 )  im = j;                 //3
                buf = a[i]; a[i] = a[im]; a[im] = buf;
            }
        }
        static void Main()
        {
        int[] a = {1, 6, 4, 2, 7, 5, 3 };
        Sort<int>(ref a);                                                   // 4
        foreach ( int elem in a ) Console.WriteLine( elem );
        double[] b = {  1.1, 5.2, 5.21, 2, 7, 6, 3 };
        Sort( ref b );                                                      // 5
        foreach ( double elem in b ) Console.WriteLine( elem );
        string[] s = { "qwe",   "qwer",  "df",  "asd" };
        Sort( ref s );                                                      // 6
        foreach ( string elem in s ) Console.WriteLine( elem );
        }
    }
}
```

Параметризовані типи і  методи дозволяють:

- описувати способи зберігання і алгоритми обробки даних незалежно від типів даних;
- виконувати контроль типів під час компіляції, а не виконання програми;
- збільшити швидкість обробки даних за рахунок виключення операцій упаковки, розпаковування і перетворення типу.

Як уже згадувалося, окрім класів-прототипів і узагальнених методів можна описати параметризовані інтерфейси, структури і делегати.

За допомогою параметризованих інтерфейсів можна визначити список функцій, які можуть бути реалізовані різним чином для різних класів, що реалізовують ці інтерфейси. Параметризовані інтерфейси можна реалізовувати в класі-прототипі, використовуючи як аргументи інтерфейсу параметри типу, що реалізовує інтерфейс, або в звичайному класі, підставляючи як параметри інтерфейса конкретні типи.

Параметризуючі делегати, дозволяють створювати узагальнені алгоритми, логіку яких можна змінювати передаючими параметри.

## 13.7. Часткові типи

Надається можливість розбивати опис типу на частини і зберігати їх в різних фізичних файлах, створюючи так звані часткові типи (*partial types*). Це необхідно для класів великого об'єму або, що актуальніше, для відділення частини коду з програми, яка написана вручну. Крім того, така можливість полегшує відладку програми, дозволяючи відокремити відлагоджені частини класу від нових.

Для опису окремої частини типу використовується модифікатор *partial*. Він може застосовуватися до класів, структур і інтерфейсів, наприклад:

```
public partial class A
{
  …
}
public partial class A
{
  …
}
```

Після сумісної компіляції цих двох частин виходить такий же код, неначебто клас був описаний звичайним способом. Всі частини одного і того ж часткового типу повинні компілюватися одночасно.

Модифікатор *partial* не є ключовим словом і повинен стояти безпосередньо перед одним з ключових слів *class, struct* або *interface* в кожній з частин. Всі частини визначення одного класу мають бути описані в одному і тому ж просторі імен.

Якщо модифікатор *partial* указується для типу, опис якого складається тільки з однієї частини, це не є помилкою.

Модифікатори доступу для всіх частин типу мають бути узгодженими. Якщо хоч би одна з частин містить модифікатор *abstract* або *sealed*, клас вважається відповідно абстрактним або безплідним.

Клас-прототип також може оголошуватися по частинах, в цьому випадку у всіх частинах мають бути присутніми одні і ті ж параметри типу з одними і тими ж обмеженнями.

Якщо частковий тип є спадкоємцем декількох інтерфейсів, в кожній частині не потрібно перераховувати всі інтерфейси: зазвичай в одній частині оголошується один інтерфейс і описується його реалізація, в іншій частині - інший інтерфейс і так далі. Набором базових інтерфейсів для типу, оголошеного в декількох частинах, є об'єднання базових інтерфейсів, визначених в кожній частині.

## 13.8. Типи, що обнуляються

У програмуванні існує проблема, яким чином задати змінній значення , якщо вона неініціалізована. Ця проблема вирішується різними способами. Один із способів полягає в тому, щоб привласнити змінній яке-небудь значення, що не входить в діапазон допустимих для неї. Наприклад, якщо величина може набувати тільки додатних значень, їй привласнюється -1. Ясно, що для багатьох випадків цей підхід непридатний. Інший спосіб - зберігання логічної ознаки, по якій можна визначити, чи привласнено змінною значення. Цей спосіб не може використовуватися, наприклад, для значень, повертаємих з методу.

Ця проблема вирішується введенням типів спеціального вигляду (nullable). Тип, що обнуляється, є структурою, що зберігає разом із значенням величини (властивість Value) логічну ознаку, по якій можна визначити, чи було привласнено значення цій величині (властивість HasValue).

Якщо значення величини було привласнене, властивість *HasValue* має значення *true*. Якщо значення величини рівне *null*, властивість *HasValue* має значення *false*, а спроба набути значення через властивість *Value* викликає генерацію виключення.

Тип, що обнуляється, будується на основі базового типу, за яким слідує символ ?, наприклад:

```
int? x = 123;
int? y = null;
if ( x.HasValue ) Console.WriteLine( x ); // замість x можна записати x.Value
if ( y.HasValue ) Console.WriteLine( y );
```

Існують явні і неявні перетворення з типів, що обнуляються, в звичайних і назад, при цьому виконується контроль можливості набуття значення, наприклад:

```
int      i = 123;
int?     x = i;          //int     --> int?
double? y = x;           // int?   -- > double?
int?     z = (int?) y;   // double? --> int?
int      j = (int) z;    // int?    --> int
```

Для величин типів, що обнуляються, визначені операції відношення. Операції == і != повертають значення *true*, якщо обидві величини мають значення *null*. Природно, що значення *null* вважається за не рівне будь-якому ненульовому значенню. Операції <, >, < = і >= дають в результаті *false*, якщо хоч би один з операндів має значення *null*.

Арифметичні операції з величинами типів, що обнуляються, дають в результаті *null*, якщо хоч би один з операндів рівний *null*, наприклад:

```
int? x = null;
int? y = x + 1;          // y = null
```

Для величин типів, що обнуляються, введена ще одна операція - об'єднання **??** (*null coalescing operator*). Це бінарна операція, результат якої дорівнює першому операнду, якщо він не рівний *null*, і другому інакше. Іншими словами, ця операція надає значення, що заміщається, для *null*, наприклад:

```
int? x = null;
int   y = x ?? 0;                // y = 0
x = 1;
y = x ?? 0;                      // y = 1
```

Типи, що обнуляються, зручно використовувати при роботі з базами даних і XML.

## 13.9. Рекомендації по програмуванню

Алгоритм роботи програми багато в чому залежить від способу організації її даних, тому дуже важливо до початку розробки алгоритму вибрати оптимальні структури даних, ґрунтуючись на вимогах до функціональності і швидкодії програми.

Для різних завдань необхідні різні способи зберігання і обробки даних, тому необхідно добре уявляти собі як характеристики і сфери застосування абстрактних структур даних, так і їх конкретну реалізацію у вигляді колекцій бібліотеки. Вивчення можливостей стандартних колекцій і їх грамотне застосування є необхідною умовою створення ефективних і професійних програм, дозволяє скоротити терміни розробки програм і підвищити їх надійність.

Недоліками колекцій перших версій бібліотеки .NET є відсутність контролю типів на етапі компіляції і неефективність при зберіганні елементів значущих типів. Колекції, що параметризуються, з'явилися у версії 2.0 бібліотек, позбавлені від цих недоліків, тому в програмах рекомендується використовувати саме колекції, вибираючи найбільш відповідні класи залежно від вирішуваного завдання.

Для реалізації алгоритмів, незалежних від типів даних, слід використовувати класи-прототипи і узагальнені методи. Вони не знижують ефективність програми в порівнянні із звичайними класами і методами, оскільки код для конкретного типу генерується середовищем CLR під час виконання програми. Окрім класів-прототипів і узагальнених методів можна описати інтерфейси, що параметризуються, структури і делегати.

Часткові типи зручно використовувати при розробці об'ємних класів групою програмістів і для спрощення відладки програм. Типи, що обнуляються, застосовують для роботи з даними, для яких необхідно уміти визначати, чи було їм привласнено значення.

# РОЗДІЛ 14. ДОДАТКОВІ ЗАСОБИ C#

У цьому розділі описані додаткові засоби мови C# і середовища *Visual Studio*: вказівки, регулярні вирази і документація у форматі XML. В кінці розділу дається коротке введення в основні сфери професійного застосування C#: ASP.NET (веб - форми і веб - служби) і ADO.NET (бази даних).

Вказівки, без яких не мислять своє життя програмісти, що використовують C і C+ +, у мові C# рекомендується застосовувати тільки у разі потреби, оскільки вони зводять нанівець багато переваг цієї мови. Документування коди у форматі XML і регулярні вирази застосовуються ширше, але відносяться до додаткових можливостей мови, тому не були розглянуті раніше.

Веб - форми, веб - служби і робота з базами даних є одними з основних сфер застосування C#, але не розглядаються через те, що подібні теми не входять в базовий курс програмування.

## 14.1. Небезпечний код

Однією з основних переваг мови C# є його схема роботи з пам'яттю: автоматичне виділення пам'яті під об'єкти і автоматичне прибирання сміття. При цьому неможливо звернутися за неіснуючою адресою пам'яті або вийти за межі масиву, що робить програми надійнішими і безпечнішими і унеможливлює появи цілого класу помилок.

Проте в деяких випадках виникає необхідність працювати з адресами пам'яті безпосередньо, наприклад, при взаємодії з операційною системою, написанні драйверів або програм, час виконання яких критично. Таку можливість надає так званий небезпечний (*unsafe*) код.

Небезпечним називається код, виконання якого середовище CLR не контролює. Він працює безпосередньо з адресами областей пам'яті за допомогою вказівок і має бути явним чином помічений за допомогою ключового слова *unsafe*, яке визначає так званий небезпечний контекст виконання.

Ключове слово *unsafe* може використовуватись або як специфікатор, або як оператор. У першому випадку його указують разом з іншими специфікаторами при описі класу, делегата, структури, методу, поля і так далі - скрізь, де допустимі інші специфікатори. Наприклад:

```
public unsafe struct Node
 {
        public int    Value; .
        public Node* Left;
        public Node* Right;
}
```

Вся структура Node позначається як небезпечна, що робить можливим використання в ній вказівок Left і Right. Можна застосувати і інший варіант опису, в якому небезпечними оголошуються тільки відповідні поля структури:

```
public struct Node
{
        public int Value;
        public unsafe Node* Left;
        public unsafe Node* Right;
}
```

Оператор *unsafe* має наступний синтаксис:

***unsafe* блок**

Всі оператори, що входять в блок, виконуються в небезпечному контексті.

Компіляція кода, що містить небезпечні фрагменти, повинна проводитися з ключем */unsafe*. Цей режим можна встановити шляхом налаштування середовища *Visual Studio* (Project ► Properties ► Configuration Properties ► Build ► Allow Unsafe Code).

### 14.1.1. Синтаксис вказівок

Вказівки призначені для зберігання адреси областей пам'яті. Синтаксис оголошення вказівок:

**тип* змінна;**

Тут *тип* - це тип величини, на яку указує змінна, тобто величини, що зберігається за записаною в змінній адресою. Тип не може бути класом, але може бути структурою, переліченням, вказівкою, а також одним із стандартних типів*: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool* і *void*. Останнє означає, що вказівка посилається на змінну невідомого типу. Вказівка на тип *void* застосовується в тих випадках, коли конкретний тип об'єкту, адресу якого потрібно зберігати, не визначений (наприклад, якщо в одній і тій же змінній в різні моменти часу потрібно зберігати адреси об'єктів різних типів). Вказівці на тип *void* можна привласнити значення вказівки будь-якого типу, а також порівнювати його з будь-якими вказівками, але перед виконанням яких-небудь дій з областю пам'яті, на яку вона посилається, потрібно перетворити його до конкретного типу явним чином. Приклади оголошення вказівок:

```
int*     a;       // вказівка на int
Node*   pNode;      // вказівка на описану раніше структуру Node
void*    p;      // вказівка на невизначений тип
int*[] m;        // одновимірний масив вказівок на int
int**    d;      // вказівка на вказівку на int
```

У одному операторові можна описати декілька вказівок одного і того ж типу, наприклад:

```
int* a,  b,  c;  // три вказівки на int
```

Величини типу вказівки можуть бути локальними змінними, полями, параметрами і повертаємим значенням функції. Ці величини підпорядковуються загальним правилам визначення зони дії і часу життя.

### 14.1.2. Перетворення та ініціалізація вказівок

Для вказівок підтримуються неявні перетворення з будь-якого типу вказівки до типу *void**. Будь-якій вказівці можна привласнити константу *null*. Крім того, допускаються явні перетворення:

- між вказівками будь-якого типу;
- між вказівками будь-якого типу і цілими типами.

Коректність перетворень лежить на совісті програміста. Перетворення ніяк не впливають на величини, на які посилаються вказівки, але при спробі набуття значення по вказівкам невідповідного типу поведінка програми не визначена.

Нижче перераховані способи привласнення значень вказівкам:

1. Привласнення вказівки адреси існуючого об'єкту:

1.1. За допомогою операції отримання адреси:

```
int   a = 5;
int* p = &a;
```

1.2. За допомогою значення іншої вказівки:

```
int* r = p;
```

1.3. За допомогою імені масиву, яке трактується як адреса:

```
int[] b = new int[] {10, 20, 30, 50};                    //масив
fixed ( int* t = b ) {   ...   }; // привласнення адреси початку масиву
fixed ( int* t = &b[0] ) {   ...   };              //те ж саме
```

Оператор *fixed* розглядається пізніше.

2. Привласнення вказівки адреси області пам'яті в явному вигляді:

```
char* v = (char *) 0x12F69e;
```

Тут 0x12F69e - шістнадцятирічна константа, (char *) - операція приведення типу: константа перетвориться до типу вказівки на char. Використовувати цей спосіб можна тільки в тому випадку, якщо адреса точно відома, інакше може виникнути виключення.

3. Надання нульового значення:

```
int* xx = nul1;
```

4. Виділення області пам'яті в стеку і привласнення її адреси вказівці:

```
int* s = stackalloc int [10];
```

Тут операція *stackalloc* виконує виділення пам'яті під 10 величин типу *int* (масив з 10 елементів) і записує адресу початку цієї області пам'яті в змінну *s*, яка може трактуватися як ім'я масиву.

### 14.1.3. Операції з вказівками

Всі операції з вказівками виконуються в небезпечному контексті. Вони перераховані в таблиці 14.1.

Таблиця 14.1

Операції з вказівками

| Операція | Опис |
|---|---|
| * | Набуття значення, яке знаходиться за адресою, що зберігається у вказівці |
| -> | Доступ до елементу структури через вказівку |
| [ ] | Доступ до елементу масиву через вказівку |
| & | Отримання адреси змінною |
| ++, -- | Збільшення і зменшення значення вказівки на один елемент, що адресується |
| +, - | Складання з цілою величиною і віднімання вказівки |
| ==, != , <>, <=, >= | Порівняння адрес, що зберігаються у вказівках. Виконується як порівняння беззнакових цілих величин |
| stackalloc | Виділення пам'яті в стеку під змінну, на яку посилається вказівка |

Розглянемо приклади застосування операцій. Якщо у вказівку занесена адреса об'єкту, дістати доступ до цього об'єкту можна за допомогою операцій розадресації і доступу до елементу.

Операція розадресації або розіменування призначена для доступу до величини, адреса якої зберігається у вказівці. Цю операцію можна використовувати як для отримання, так і для зміни значення величини, наприклад:

```
int a = 5;                                  // ціла змінна
int* p = &a;                    // ініціалізація вказівки адресою a
Console.WriteLine(*p ):                     // результат: 5
Console.WriteLine( ++ (*p) ) ;              // результат: 6
int[] b = new int[] {10, 20, 30, 50};       // масив
                  fixed ( int* t = b )      // ініціалізація вказівки
                                             // адресою початку масиву
      int* z = t;                           // ініціалізація вказівки
                                          // значенням іншої вказівки
```

```
for (int i = 0; i < b.Length; ++i )
      {
        t[i] += 5;
        *z += 5;
        ++z;
      }
Console.WriteLine( &t[5] - t ); // операція віднімання вказівок
```

Оператор *fixed* фіксує об'єкт, адреса якого заноситься до вказівок, для того, щоб його не переміщав складальник сміття і, таким чином, вказівки залишалися коректними. Фіксація відбувається на час виконання блоку, який записаний після круглих дужок.

У приведеному прикладі доступ до елементів масиву виконується двома способами: шляхом індексації вказівки *t* і шляхом розадресації вказівки *z*.

Конструкцію **змінна* можна використовувати в лівій частині оператора привласнення, оскільки вона визначає адресу області пам'яті. Для простоти цю конструкцію можна вважати за ім'я змінної, на яку посилається вказівка. З нею допустимі всі дії, визначені для величин відповідного типу.

*Арифметичні операції* з вказівками (складання з цілим, віднімання, інкремент і декремент) автоматично враховують розмір типу величин, що адресуються вказівками. Ці операції застосовні тільки до вказівок одного типу і мають сенс в основному при роботі із структурами даних, елементи яких розміщені в пам'яті послідовно, наприклад, з масивами.

*Інкремент* переміщає вказівка до наступного елементу масиву, *декремент* - до попереднього. Фактично значення вказівки змінюється на величину *sizeof* (тип), де *sizeof* - операція отримання розміру величини вказаного типу (у байтах). Ця операція застосовується тільки в небезпечному контексті, з її допомогою можна отримувати розміри не тільки стандартних, але і призначених для користувача типів даних. Для структури результат може бути більше суми довжин складових її полів із-за вирівнювання елементів.

Якщо вказівка на певний тип збільшується або зменшується на константу, його значення змінюється на величину цієї константи, помножену на розмір об'єкту даного типу, наприклад:

```
short* p; ...
p++;                    // значення p збільшується на 2
long* q;
q++;                    // значення q збільшується на 4
```

*Різниця* двох вказівок - це різниця їх значень, що ділиться на розмір типу в байтах. Так, результат виконання останньої операції виведення в приведеному прикладі дорівнює 5. Підсумовування двох вказівок не допускається.

При записі виразів з вказівками слід звертати увагу на пріоритети операцій. Як приклад розглянемо послідовність дій, задану в операторові

```
*p++ =10;
```

Оскільки інкремент постфіксний, він виконується після виконання операції привласнення. Таким чином, спочатку за адресою, записаною в вказівку *p*, буде записано значення 10, а потім вказівка збільшиться на кількість байтів, відповідну його типу. Те ж саме можна записати докладніше:

```
*p = 10; p++ ;
```

Вираз (*p)++, навпаки, інкрементує значення, на яке посилається вказівка.

У наступному прикладі кожен байт беззнакового цілого числа *x* виводиться на консоль за допомогою вказівки *t*:

```
uint x = 0xAB10234F;
byte* t = (byte*)&x;
for ( int i = 0; i < 4: ++i )
Console.Write("{0: X}  ", *t++ );        // результат: 4F 23 10 AB
```

Як бачите, спочатку вказівка t був встановлений на молодший байт змінної x. Лістинг 14.1 ілюструє доступ до поля класу і елементу структури:

Лістинг 14.1. Доступ до поля класу і елементу структури за допомогою вказівок.

```
using System;
namespace ConsoleApplicationl
{
      class A
      {
            public int value = 20;
      }
      struct B
      {
            public int a;
      }
      class Program
      {
            unsafe static void Main()
              {
                  A n = new A( ) ;
                  fixed (int* pn = &n.value ) ++ (*pn);
                  Console.WriteLine("n = " + n.value );        // результат: 21
                  B b;
                  B* pb = &b;
                  pb->a = 100;
                  Console.WriteLine( b.a );                    // результат: 100
              }
      }
}
```

Операція stackalloc дозволяє виділити пам'ять в стеку під задану кількість величин заданого типу:

**stackalloc тип [ кількість ]**

Кількість задається цілочисельним виразом. Якщо пам'яті недостатньо, генерується виключення System.StackOverflowException. Виділена пам'ять нічим не ініціюється і автоматично звільняється при завершенні блоку, що містить цю операцію. Приклад виділення пам'яті під п'ять елементів типу int і їх заповнення числами від 0 до 4:

```
int* p = stackalloc int [5];
for ( int i = 0; i < 5: ++i )
{
  p[1]-1;
  Console.Write( p[i] + "  "   );   // результат:  0 1 2 3 4 .
}
```

У лістингу 14.2 приведений приклад роботи з вказівками, узятий із специфікації C#. Метод IntToString перетворить передане йому ціле значення в рядок символів, заповнюючи його шляхом доступу через вказівку.

Лістинг 14.2. Приклад роботи з вказівками: переведення числа в рядок

```
using System;
class Test
{
    static string IntToString (int value)
    {
        int n = value >= 0 ? value : -value;
        unsafe {
                    char* buffer = stackalloc char[16];
                    char* p = buffer + 16;
                    do {
                    *--p = (char)(n % 10 +  '0' );
                    n /= 10;
                    } while ( n != 0 );
                    if ( value < 0 ) *-- p = '-';
                    return new string(p, 0, (int)(buffer + 16 - p ) );
                    }
    }
      static void Main( )
      {
          Console.WriteLine(IntToString( 12345 ) );
          Console.WriteLine(IntToString( -999 ) );
      }
}
```

## 14.2. Регулярні вирази

Регулярні вирази призначені для обробки текстової інформації і забезпечують:

- ефективний пошук в тексті за заданим шаблоном;
- редагування, заміну і видалення підрядків;
- формування підсумкових звітів за наслідками роботи з текстом.

За допомогою регулярних виразів зручно обробляти файли у форматі HTML, файли журналів або довгі текстові файли. Для підтримки регулярних виразів в бібліотеку .NET включені класи, об'єднані в простір імен System.Text.RegularExpressions.

### 14.2.1. Метасимволи

Регулярний вираз - це шаблон (зразок), по якому виконується пошук відповідного йому фрагмента тексту. Мова опису регулярних виразів складається з символів двох видів: звичайних і метасимволів. Звичайний символ представляє у виразі сам себе, а метасимвол - деякий клас символів, наприклад, будь-яку цифру або букву.

Наприклад, регулярний вираз для пошуку в тексті фрагмента “Вася” записується за допомогою чотирьох звичайних символів Вася, а вираз для пошуку двох цифр, що йдуть підряд, складається з двох метасимволів \d\d.

За допомогою комбінацій метасимволів можна описувати складні шаблони для пошуку. Наприклад, можна описати шаблон для IP-адреси, адреси електронної пошти, різних форматів дати, заголовків певного вигляду і так далі.

У таблиці 14.2 описані найбільш споживані метасимволи, що є класами символів.

Метасимволи, перераховані в таблиці 14.3, уточнюють позицію в рядку, в якому слід шукати збіг з регулярним виразом, наприклад, тільки на початку або в кінці рядка. Ці метасимволи є уявними, тобто в тексті їм не відповідає ніякий реальний символ.

Наприклад, вираз ^cat відповідає символам cat, що зустрічається на початку рядка, вираз cat$ - символам cat, що зустрічається в кінці рядка (тобто за ними йде символ перекладу рядка), а вираз ^$ представляє порожній рядок, тобто початок рядка, за яким відразу ж слідує його кінець.

Таблиця 14.2

Класи символів

| Клас символів | Опис | Приклад |
|---|---|---|
| . | Будь-який символ, окрім \n | Вираз c.t відповідає фрагментам cat, cut, clt. |
| [ ] | Будь-який одиночний символ з послідовності, записаної усередині дужок. Допускається використання діапазонів символів | Вираз c[aul]t відповідає фрагментам cat, cut і clt, а вираз c[a-z]t - фрагментам cat, cbt, cct, cdt,…, czt |
| [^] | Будь-який одиночний символ, що не входить в послідовність, записану усередині дужок. Допускається використання діапазонів символів | Вираз c[^aul]t відповідає фрагментам cbt, c2t, cXt и т. д., а выражение c[^a-zA-Z]t - фрагментам c1t, cCt і т.д. |
| \w | Будь-який алфавітно-цифровий символ, тобто символ з множини прописних і рядкових букв і десяткових цифр | Вираз c\wt відповідає фрагментам cat, cut, clt, /0t і т.д., але не відповідає фрагментам c{t, c;t і т.д. |
| \W | Будь-який не алфавітно-цифровий символ, тобто символ, що не входить в множину прописних і рядкових букв і десяткових цифр | Вираз c\Wt відповідає фрагментам c{t, c;t, c t і т. д., але не відповідає фрагментам cat, cut, clt і т.д. |
| \s | Будь-який пробільний символ, наприклад символ пробілу, табуляції (\t., \v), перекладу рядка (\n \r), нової сторінки (\f) | Вираз \s\w\w\w\s відповідає будь-якому слову з трьох букв, оточеному пробільними символами |
| \S | Будь-який не пробільний символ, тобто символ, що не входить в множину пробільних | Вираз \s\S\S\s відповідає будь-яким двом непробільним символам, оточеним пробільними |
| \d | Будь-яка десяткова цифра | Вираз c\dt відповідає фрагментам c1t, c2t,c9t |
| \D | Будь-який символ, що не є десятковою цифрою | Вираз C\Dt не відповідає фрагментам c1t, c2t, …, c9t |

Таблиця 14.3

Уточнюючі метасимволи

| Метасимвол | Опис |
|---|---|
| ^ | Фрагмент, співпадаючий з регулярним виразом, слід шукати тільки на початку рядка |
| $ | Фрагмент, співпадаючий з регулярним виразом, слід шукати тільки в кінці рядка |
| \A | Фрагмент, співпадаючий з регулярним виразом, слід шукати тільки на початку багаторядкового рядка |
| \Z | Фрагмент, співпадаючий з регулярним виразом, слід шукати тільки в кінці багаторядкового рядка |
| \b | Фрагмент, співпадаючий з регулярним виразом, починається або закінчується на границі слова (тобто між символами, відповідними метасимволам \w і \W ) |
| \B | Фрагмент, співпадаючий з регулярним виразом, не повинен зустрічатися на границі слова |

У регулярних виразах часто використовують повторення. *Повторення* - це метасимволи, які розташовуються безпосередньо після звичайного символу або класу символів і задають кількість його повторень у виразі. Наприклад, якщо потрібно записати вираз для пошуку в тексті п'яти що йдуть підряд цифр, замість метасимволів \d\d\d\d\d можна записати \d{5}. Такому виразу відповідатимуть фрагменти 11111, 12345, 53332 і так далі

Найбільш вживані повторення перераховані в таблиці 14.4.

Таблиця 14.4

Повторення

| Метасимвол | Опис | Приклад |
|---|---|---|
| * | Нуль або більш за повторення попереднього елементу | Вираз ca*t відповідає фрагментам ct, cat, caat, caaaaaaaaaat і так далі |
| + | Одне або більш за повторення попереднього елементу | Вираз ca+t відповідає фрагментам cat, caat, caaaaaaaaaat і так далі |
| ? | Жодного або одне повторення попереднього елементу | Вираз ca?t відповідає фрагментам ct і cat |
| {n} | Рівно n повторень попереднього елементу | Вираз ca{3}t відповідає фрагменту caaat, а вираз (cat){2} - фрагменту catcat |
| {n.} | Принаймні n повторень попереднього елементу | Вираз ca {3}t відповідає фрагментам caaat, caaaat, caaaaaaaaaaat і так далі |
| {n, m} | Від n до m повторень попереднього елементу | Вираз ca{2,4}t відповідає фрагментам caat, caaat і caaaat |

Окрім розглянутих елементів регулярних виразів можна використовувати конструкцію вибору з декількох елементів. Варіанти вибору перераховуються через вертикальну межу. Наприклад, якщо потрібно визначити, чи присутній в тексті хоч би один елемент із списку “cat”, “dog” і “horse”, можна використовувати вираз

**cat|dog|horse**

При пошуку використовується так званий «ледачий» алгоритм, по якому пошук припиняється при знаходженні найкоротшого з можливих фрагментів, співпадаючих з регулярним виразом.

Приклади простих регулярних виразів:

ціле число (можливо, із знаком):
[-+]?\d+
дійсне число (може мати знак і дробову частину, відокремлену крапкою):
[-+]?\d+\.?\d*
російський номер автомобіля (спрощено):
[A-Z]\d{3}[A-Z]{2}\d\dRUS

Якщо потрібно описати у виразі звичайний символ, співпадаючий з яким-небудь метасимволом, перед ним стоїть символ “\”. Так, для пошуку в тексті символу крапки слід записати \., а для пошуку косої межі - \\. Наприклад, для пошуку в тексті імені файлу cat.doc слід використовувати регулярний вираз cat\.doc.

Для групування елементів виразу використовуються круглі дужки. *Групування* застосовується у багатьох випадках, наприклад, якщо потрібно задати повторення не для окремого символу, а для послідовності. Крім того, групування служить для запам'ятовування в деякій змінній фрагмента тексту, що збігся з виразом, взятим в дужки. Ім'я змінної задається в кутових дужках або апострофах:

**(?<ім'я_змінної> фрагмент_виразу**

Фрагмент тексту, що збігся при пошуку з фрагментом регулярного виразу, заноситься в змінну із заданим ім'ям. Нехай, наприклад, потрібно виділити з тексту номери телефонів, записаних у вигляді nnn-nn-nn. Регулярний вираз для пошуку номера можна записати так:

**(?<num>\d\d\d-\d\d-\d\d)**

При аналізі тексту в змінну з ім'ям num послідовно записуватимуться знайдені номери телефонів.

Розглянемо ще один варіант застосування групування - для формування зворотних посилань. Всі конструкції, взяті в круглі дужки, автоматично нуме-

руються, починаючи з 1. Ці номери можна використовувати для посилань на відповідну конструкцію. Наприклад, вираз (\w)\1 використовується для пошуку здвоєних символів в словах (*wall, mass, cooperate*).

Круглі дужки можуть бути вкладеними, при цьому номер конструкції визначається порядком відкриваючої дужки у виразі. Приклади:

(Вася)\s+( дасть)\s+(?<sum>\d+)\sкрб\.\s+ Ну що ж ти. \1

У цьому виразі три під вирази взяті в дужки. Посилання на перше з них виконується в кінці виразу. З цим виразом співпадуть, наприклад, фрагменти

Вася дасть 5 крб.  Ну що ж ти, Вася

Вася дасть 54356 крб.  Ну що ж ти, Вася

Вираз, що задає IP-адресу:

((\d{1.3}\.){3}\d{1.3})

Адреса складається з чотирьох груп цифр, розділених крапками. Кожна група може включати від однієї до трьох цифр. Приклади IP-адрес: 212.46.197.69, 212.194.5.106, 209.122.173.160. Перша група, взята в дужки задає всю адресу. Їй привласнюється номер 1. У неї вкладені другі дужки, що визначають межі для повторення {3}.

Змінну, ім'я якої задається усередині виразу в кутових дужках, також можна використовувати для зворотних посилань в подальшій частині виразу. Наприклад, пошук подвійних символів в словах можна виконати за допомогою виразу (?<s>\w)\k<s>, де s - ім'я змінної, в якій запам'ятовується символ \k - елемент синтаксису.

У регулярний вираз можна поміщати коментарі. Оскільки вирази зазвичай простіше писати, чим читати, це - дуже корисна можливість. Коментар або поміщається всередину конструкції (?# ), або розташовується, починаючи від символу # до кінця рядка.

### 14.2.2. Класи бібліотеки .NET для роботи з регулярними виразами

Класи бібліотеки .NET для роботи з регулярними виразами об'єднані в простір імен *System.Text.RegularExpressions*.

Почнемо з класу *Regex,* що представляє власне регулярний вираз. Клас є незмінним, тобто після створення екземпляра, його коректування не допускається. Для опису регулярного виразу в класі визначено декілька перевантажених *конструкторів:*

Regex () - створює порожній вираз;

Regex(String) - створює заданий вираз;

Regex(String, Regexoptions) - створює заданий вираз і задає параметри для його обробки за допомогою елементів перелічення RegexOptions (наприклад, розрізняти або не розрізняти прописні і рядкові букви).

Приклад конструктора, задаючого вираз для пошуку в тексті слів, що повторюються, розташованих підряд і розділених довільною кількістю пропусків, незалежно від регістра:

```
Regex гх = new Regex( @"\b(?<word>\w+)\s+(\k<word>)\b",
RegexOptions.IgnoreCase );
```

Пошук фрагментів рядка, відповідних заданому виразу, виконується за допомогою методів *IsMatch, Match* і *Matches*.

Метод *IsMatch* повертає *true*, якщо фрагмент, відповідний виразу, в заданому рядку знайдений, і *false* в іншому випадку. У лістингу 14.3 приведений приклад пошуку слів, що повторюються, в двох тестових рядках. У регулярний вираз, приведений раніше, доданий фрагмент, що дозволяє розпізнавати розділові знаки.

Лістинг 14.3. Пошук в рядку дубльованих слів (методом IsMatch)

```
using System;
using System.Text.RegularExpressions;

namespace exam102
{
 public class Test
   {
      static void Main(string[] args)
      {
      Regex r = new Regex( @"\b(?<word>\w+)[.,:;!? ]\s*(\k<word>)\b",
         RegexOptions.IgnoreCase);
      string tstl = "Oh. oh! Give me more!";
      if ( r.IsMatch( tstl ) ) Console.WriteLine(" tstl yes" );
      elseConsole.WriteLine( " tstl no" );
      string tst2 = "Oh give me. give me more!";
      if ( r.IsMatch( tst2 ) ) Console.WriteLine(" tst2 yes" );
      elseConsole.WriteLine( " tst2 no"   );
      }
   }
}
```

Результат роботи програми:

tst1 yes

tst2 no

Слова, що повторюються, в рядку tst2 розташовуються не підряд, тому вони не відповідають регулярному виразу. Для пошуку слів, що повторюються, розташованих в довільних позиціях рядка, в регулярному виразі потрібно замінити пропуск (\s) “будь-яким символом” (.) :

```
Regex r = new Regex( @"\b(?<word>\w+)[..:;!? ].*(\k<word>)\b", RegexOp-
tions.IgnoreCase ) ;
```

Метод *Match* класу Regex, на відміну від методу Ismatch, не просто визначає, чи відбувся збіг, а повертає об'єкт класу *Match* - черговий фрагмент, що збігся із зразком. Розглянемо лістинг 14.4, в якому використовується цей метод.

Лістинг 14.4. Виділення з рядка слів і чисел (методом *Match*)

```
using System;
using System.Text.RegularExpressions;

   public class Test
   {
      static void Main(string[] args)
      {
         string text = "Салат - $4, борщ - $3,одеколон - $10.";
         string pattern = @"(\w+) - \$(\d)[.,]";
         Regex   r       = new Regex(pattern);
         Match   m       = r.Match(text);
         int     total   = 0;

         while ( m.Success)
         {
            Console.WriteLine( m );
            total += int.Parse( m.Groups[2].ToString() );
            m = m.NextMatch();
         }

         Console.WriteLine( "Разом: $" + total );
      }
   }
```

Результат роботи програми:

```
Салат - $4,
борщ - $3,
одеколон - $10.
Разом: $17
```

При першому зверненні до методу *Match* повертається перший фрагмент рядка, що збігся з регулярним виразом *pattern*. У класі *Match* визначена властивість *Groups*, що повертає колекцію фрагментів, що збіглися з підвиразами в круглих дужках. Нульовий елемент колекції містить весь фрагмент, перший елемент - фрагмент, що збігся з підвиразом в перших дужках, другий елемент - фрагмент, що збігся з підвиразом в других дужках, і так далі. Якщо при визначенні виразу задати фрагментам імена, як це було зроблено в попередньому лістингу, можна буде звернутися до них по цих іменах, наприклад:

```
string pattern - @"(?'name'\w+) - \$(?'price'\d + )[.,]";
…
total += int.Parse( m.Groups["price"].ToString() );
```

Метод *NextMatch* класу *Match* продовжує пошук в рядку з того місця, на якому закінчився попередній пошук.

Метод *Matches* класу *Regex* повертає об'єкт класу *Matchcollection* - колекцію всіх фрагментів заданого рядка, що збіглися із зразком.

Розглянемо тепер приклад застосування методу *Split* класу *Regex*. Цей метод розбиває заданий рядок на фрагменти відповідно до роздільників, заданих за допомогою регулярного виразу, і повертає ці фрагменти в масиві рядків. У лістингу 14.5 рядок з лістингу 14.4 розбивається на окремі слова.

Лістинг 14.5. Розбиття рядка на слова (методом *Split*)

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

public class Test
   {
      static void Main(string[] args)
      {
         string text = "Салат - $4, борщ - $3, одеколон - $10.";
         string pattern = "[- ,.]+";
         Regex   r           = new Regex(pattern);
         List<string> words = new List<string>(r.Split(text));
         foreach (string word in words) Console.WriteLine(word);

      }
   }
```

Результат роботи програми:

```
Салат
$4
борщ
$3
одеколон
$10
```

Метод Replace класу Regex дозволяє виконувати заміну фрагментів тексту. Визначено декілька перевантажених версій цього методу. От як виглядає приклад простого застосування методу в його статичному варіанті, замінюючого всі входження символу $ символами у. о. :

```
string text  = "Салат -   $4, борщ -$3, одеколон - $10.";
string textl = Regex.Replace( text, @"\$", "у.о." );
```

Інші версії методу дозволяють задавати будь-які дії із заміни за допомогою делегата *MatchEvaluator*, який викликається для кожного входження фрагмента, що збігся із заданим регулярним виразом.

Окрім класів Regex і Match в просторі імен System.Text.RegularExpres-sions визначені допоміжні класи, наприклад, клас Capture - фрагмент, що збігся з підвиразом в круглих дужках; клас CaptureCollection - колекція фрагментів, що збіглися зі всіма підвиразами в поточній групі; клас Group містить колекцію Capture для поточного збігу з регулярним виразом і так далі.

Як реальний приклад застосування регулярних виразів розглянемо програму аналізу файлу журналу веб-сервера. Це текстовий файл, кожен рядок якого містить інформацію про одне з'єднання з сервером. Чотири рядки файлу приведено нижче:

```
ppp-48.pool-113.spbnit.ru - - [31/May/2022:02:08:32 +0400] "GET / HTTP/1.1" 200
2434 "http://www.price.ru/bin/price/firminfo_f?fid=10922&where=01&base=2"
"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
81.24.130.7 - - [31/May/2022:08:13:17 +0400]  "GET /swf/menu.swf HTTP/1.1” 200
4682 "-" "Mozilla/4.0 (compatible; MSIE 5.01; Windows 98)"
81.24.130.7 - - [31/May/2022:08:13:17 +0400] "GET /swf/header.swf HTTP/1.1" 200
21244 "-" "Mozilla/4.0 (compatible; MSIE 5.01; Windows 98)"
gate.solvo.ru - - [31/May/2022:10:43:03 +0400] "GET / HTTP/1.0" 200 2422
"http://www.price.ru/bin/price/firminfo_f?fid=10922&where=01&base=l"
"Mozilla/4.0 (compatible; MSIE 5.0; Windows NT; DigExt)"
```

Подібні файли можуть мати дуже значний об'єм, тому складання підсумкового звіту в зручному форматі має важливе значення. Якщо розглядати кожен рядок файлу як сукупність полів, розділених пропусками, то поле номер 0 містить адресу, з якої виконувалося з'єднання з сервером, поле номер 5 - операцію (GET при завантаженні інформації), поле 8 - ознаку успішності виконання операції (200 -  успішно) і, нарешті, поле 9- кількість переданих байтів.

Приведена в лістингу 14.6 програма формує у форматі HTML підсумковий звіт, що містить таблицю адрес, з яких виконувалося звернення до сервера, і сумарну кількість переданих байтів для кожної адреси.

Лістинг 14.6. Аналіз журналу веб-сервера

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

```
public class Testm
{
   public static void Main(string[] args)
   {
      StreamReader f = new StreamReader("access.log" );
      StreamWriter w = new StreamWriter("report.htm" );
      Regex get   =   new Regex("GET" );
      Regex r  =   new Regex(" ");
      string s,entry;
      int value;
      string[] items   = new string[40];
      Dictionary<string,int> table = new Dictionary<string, int>();
      while ((s = f.ReadLine()) != null)
      {
         items = r.Split(s);
         if (get.IsMatch(items[5]) && items[8] == "200")
         {
            entry = items[0];
            value = int.Parse(items[9]);
            if (table.ContainsKey(entry)) table[entry] += value;
            else table[entry] = value;
         }
      }
            f.Close();
            w.Write("<html><head><title> Report </title></head><body>" +
            "<table border =1 <tr><td> Computer <td> Bytes </tr>");
            foreach ( string item in table.Keys )
            w.Write( "<tr><td>{0}<td>{1}</tr>", item, table[item] );
            w.Write( "</table></body>" );
            w.Close();
   }
}
```

Фрагмент результату роботи програми показаний у вигляді таблиці 14.5

Таблиця 14.5

Фрагмент журналу веб-сервера

| Computer | Bytes |
|---|---|
| ppp-48 pool-113.spbnit.ru | 107039 |
| Test223.sovam.com | 2422 |
| 210.82.124,83 | 200052 |
| ipblock209-209.octetgroup.net | 162781 |
| 81.24.130.7 | 74756 |
| gate.solvo.ru | 113692 |
| 212.113.108.164 | 261199 |

Файл access.log прочитується по рядкам, кожен рядок розбивається на поля, які заносяться в масив items. Потім, якщо завантаження пройшло успішно, про що свідчать значення GET і 200 полів 5 і 8, кількість переданих байтів (поле 9) перетвориться в ціле і заноситься в хеш-таблицю по ключу, яким служить адреса, що зберігається в полі 0.

Для формування HTML-файла *report.htm* використовуються відповідні теги. Файл можна проглянути, наприклад, за допомогою *Internet Explorer*. Програма вийшла дуже компактною за рахунок використання стандартних класів бібліотеки *.NET*.

## 14.3. Документування у форматі XML

XML (extensible Markup Language) - це мова розмітки тексту. Розміткою є все, що не відноситься до змісту: структура документа, формат, вигляд і так далі. Розмітка здійснюється за допомогою тегів - елементів, які виділені кутовими дужками. Теги в XML завжди парні: відкриваючий тег записується перед фрагментом, що розмічається, а закриваючий - після нього. Закриваючий тег виглядає так само, як відкриваючий, але перед ним ставиться символ “\”, наприклад:

**<summary> Клас для роботи з регулярними виразами </summary>**

У тегах можуть бути присутніми *атрибути* - елементи вигляду *ім'я = значення*, що уточнюють опис елементу.

Мова XML широко поширена в Інтернеті завдяки її універсальності. Корпоративні застосування використовують *XML* як основний формат для обміну даними. Строго кажучи, *XML* є не мовою, а системою правил для опису мов. Багато технологій, що становлять *.NET* нерозривно пов'язані з *XML*, тому в просторі імен *System. XML* бібліотеки *.NET* описано множину класів для роботи з *XML*. Тут даються тільки загальні поняття мови *XML.*

Будь-який програмний продукт потрібно документувати. Відповідність версій документації і програми - серйозна проблема, яка вирішується в .NET вбудовуванням документації прямо в код програми за допомогою коментарів і *XML*-тегів.

Коментарі, що використовуються для побудови файлів документації, починаються з символів /// і розміщуються перед відповідним елементом програми. Усередині коментарів записуються теги, що відносяться до елементу, що коментується, - класу, методу, параметру методу і тому подібне. Теги перераховані в таблиці 14.6.

Таблиця 14.6

Теги документування

| Тег | Опис |
|---|---|
| <c> | Форматування фрагменту коду |
| <code> | Багаторядковий код (використовується в секції <example>) |
| <example> | Приклад використання класу або методу |
| <exception> | Виключення, що генерується класом |
| <include file=’файл’ path= ’путь[@name= ”ид”]’ ’ /> | Посилання на коментарі в іншому файлі, в якому знаходяться описи елементів початкового коду |
| <list> | Перелічення у вигляді списку |
| <param> | Опис параметра методу |
| <paramref> | Посилання на параметр |
| <permission> | Права доступу до елементу |
| <remarks> | Докладний опис елементу (класу, методу і т. п.) |
| <returns> | Повертанєме значення методу |
| <see cref=”элемент”> | Посилання на елемент класу |
| <seealso ref=”элемент”> | Посилання на документацію виду “див. також” |
| <summary> | Короткий опис елементу (класу, методу і т.п.) |
| <value> | Опис значення властивості |

Приклади тегів:

```
///<summary> Функція обчислення синуса </summary>
///<param name="i"> Аргумент функції </paгam>
///<seealso cref="System.Double"> double </seealso>
///<returns> Повертає величину синуса </returns>
///<remarks> Для обчислень використовується розкладання
/// у ряд Тейлора </remarks>
public double Sin( double i )   { ... }
```

# ЛАБОРАТОРНІ РОБОТИ

## Лабораторна робота 1. Лінійні програми

Теоретичний матеріал: розділи 1-3.
Напишіть програму для розрахунку по двох формулах.

1. $z_1 = \cos\alpha + \sin\alpha + \cos 3\alpha + \sin 3\alpha$; $z_2 = \frac{1}{4} - \frac{1}{4}\sin\left(\frac{5}{2}\pi - 8\alpha\right)$.

2. $z_1 = \cos\alpha + \sin\alpha + \cos 3\alpha + \sin 3\alpha$; $z_2 = 2\sqrt{2}\cos\alpha \cdot \sin\left(\frac{\pi}{4} + 2\alpha\right)$.

3. $z_1 = \frac{\sin 2\alpha + \sin 5\alpha - \sin 3\alpha}{\cos\alpha + 1 - 2\sin^2 2\alpha}$; $z_2 = 2\sin\alpha$.

4. $z_1 = \frac{\sin 2\alpha + \sin 5\alpha - \sin 3\alpha}{\cos\alpha - \cos 3\alpha + \cos 5\alpha}$; $z_2 = tg3\alpha$.

5. $z_1 = 1 - \frac{1}{4}\sin^2 2\alpha + \cos 2\alpha$; $z_2 = \cos^2\alpha + \cos^4\alpha$.

6. $z_1 = \cos\alpha + \cos 2\alpha + \cos 6\alpha + \cos 7\alpha$; $z_2 = 4\cos\frac{\alpha}{2} \cdot \cos\frac{5}{2}\alpha \cdot \cos 4\alpha$.

7. $z_1 = \cos^2\left(\frac{3}{8}\pi - \frac{\alpha}{4}\right) - \cos^2\left(\frac{11}{8}\pi + \frac{\alpha}{4}\right)$; $z_2 = \frac{\sqrt{2}}{2}\sin\frac{\alpha}{2}$.

8. $z_1 = \cos^4 x + \sin^2 y + \frac{1}{4}\sin^2 2x - 1$; $z_2 = \sin(y+x) \cdot \sin(y-x)$.

9. $z_1 = (\cos\alpha - \cos\beta)^2 - (\sin\alpha - \sin\beta)^2$; $z_2 = -4\sin^2\frac{\alpha - \beta}{2} \cdot \cos(\alpha + \beta)$.

10. $z_1 = \frac{\sin\left(\frac{\pi}{2} + 3\alpha\right)}{1 - \sin(3\alpha - \pi)}$; $z_2 = ctg\left(\frac{5}{4}\pi + \frac{3}{2}\alpha\right)$.

11. $z_1 = \frac{1 - 2\sin^2\alpha}{1 + \sin 2\alpha}$; $z_2 = \frac{1 - tg\alpha}{1 + tg\alpha}$.

12. $z_1 = \frac{\sin 4\alpha}{1 + \cos 4\alpha} - \frac{\cos 2\alpha}{1 + \cos 2\alpha}$; $z_2 = ctg\left(\frac{3}{2}\pi - \alpha\right)$.

13. $z_1 = \frac{\sin\alpha + \cos(2\beta - \alpha)}{\cos\alpha - \sin(2\beta - \alpha)}$; $z_2 = \frac{1 + \sin 2\beta}{\cos 2\beta}$.

14. $z_1 = \frac{\cos\alpha + \sin\alpha}{\cos\alpha - \sin\alpha}$; $z_2 = tg2\alpha + \sec 2\alpha$.

15. $z_1 = \frac{\sqrt{2b + 2\sqrt{b^2 - 4}}}{\sqrt{b^2 - 4b + 2}}$; $z_2 = \frac{1}{\sqrt{b+2}}$.

16. $z_1 = \frac{x^2 + 2x - 3 + (x+1)\sqrt{x^2 - 9}}{x^2 - 2x - 3 + (x-1)\sqrt{x^2 - 9}}$; $z_2 = \sqrt{\frac{x+3}{x-3}}$.

17. $z_1 = \frac{\sqrt{(3m+2)^2 - 24m}}{3\sqrt{m} - \frac{2}{\sqrt{m}}}$; $z_2 = -\sqrt{m}$.

18. $z_1 = \left(\dfrac{a+2}{\sqrt{2a}} - \dfrac{a}{\sqrt{2a}+2} + \dfrac{2}{a-\sqrt{2a}}\right)\dfrac{\sqrt{a}-\sqrt{2}}{a+2}$; $\quad z_2 = \dfrac{1}{\sqrt{a}+\sqrt{2}}$.

19. $z_1 = \left(\dfrac{1+a+a^2}{2a+a^2} + 2 - \dfrac{1-a+a^2}{2a-a^2}\right)^{-1}\left(5-2a^2\right)$; $\quad z_2 = \dfrac{4-a^2}{2}$.

20. $z_1 = \dfrac{(m-1)\sqrt{m}-(n-1)\sqrt{n}}{\sqrt{m^3 n}+nm+m^2-m}$; $\quad z_2 = \dfrac{\sqrt{m}-\sqrt{n}}{m}$.

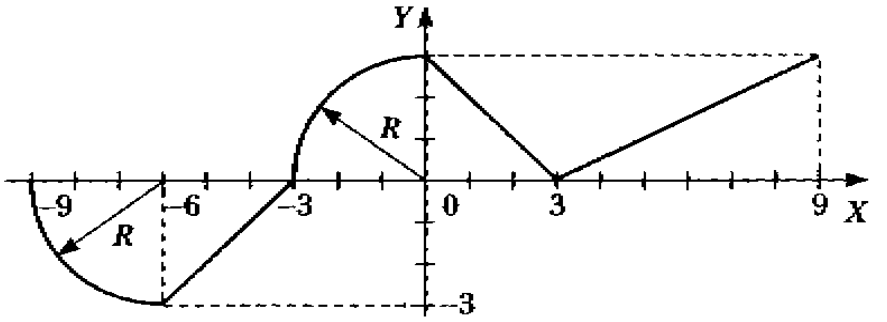
## Лабораторна робота 2. Розгалужені обчислювальні процеси

Теоретичний матеріал: розділ 4.

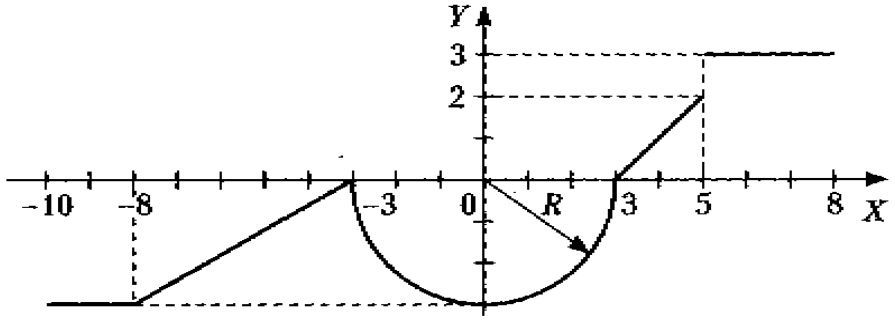
### Завдання 1. Обчислення значення функції

Написати програму, яка по введеному значенню аргументу обчислює значення функції, заданої у вигляді графіка. Параметр R вводиться з клавіатури.

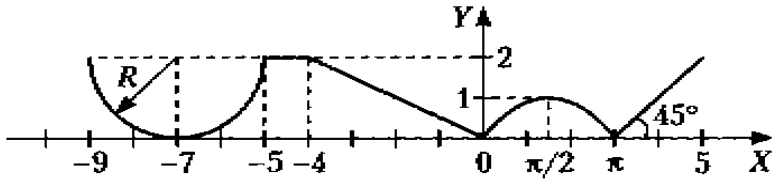
1



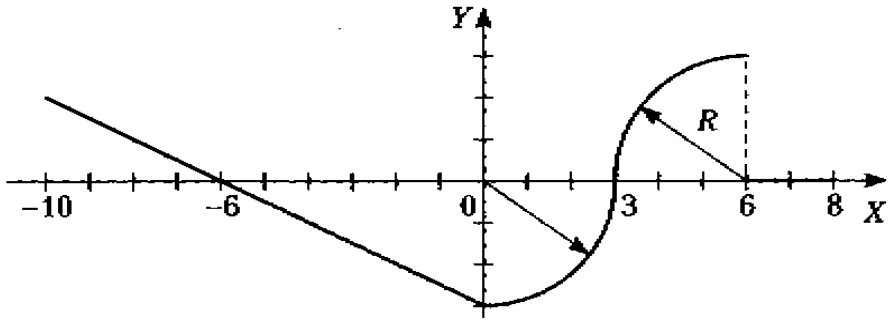

2



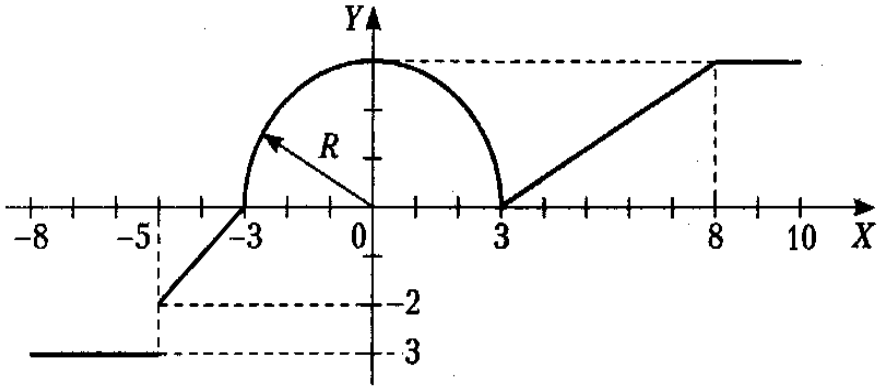

3



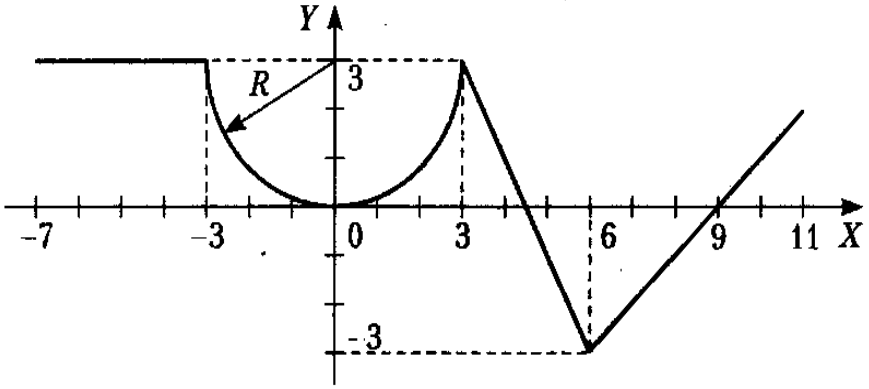

4



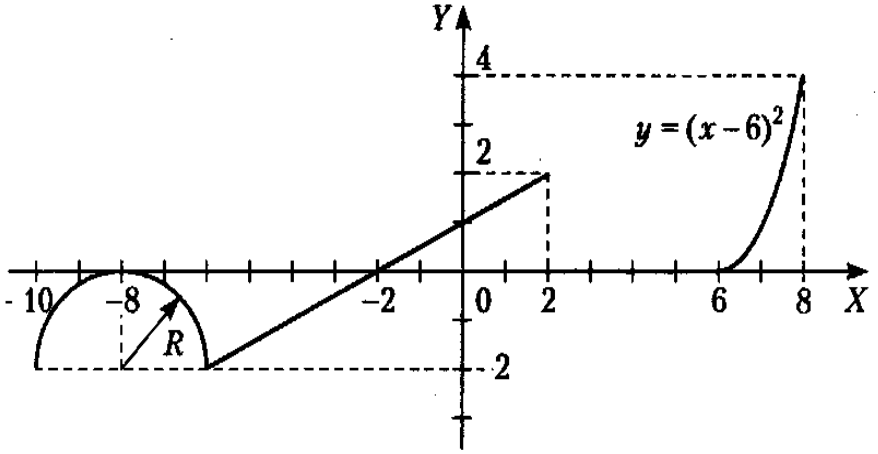
6


Y
R
-8
-5
-3
0
3
8
10
X
-2
-3

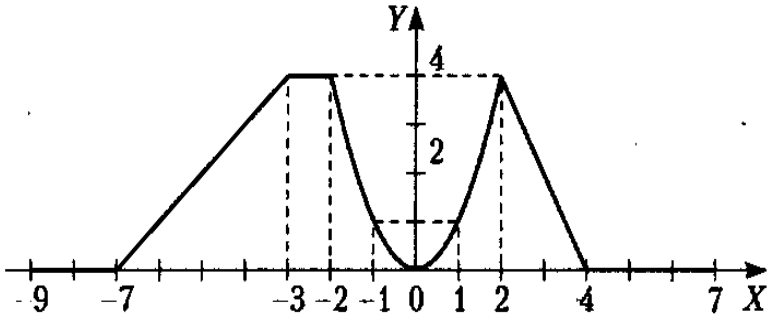
7


Y
R
3
-7
-3
0
3
6
9
11
X
-3

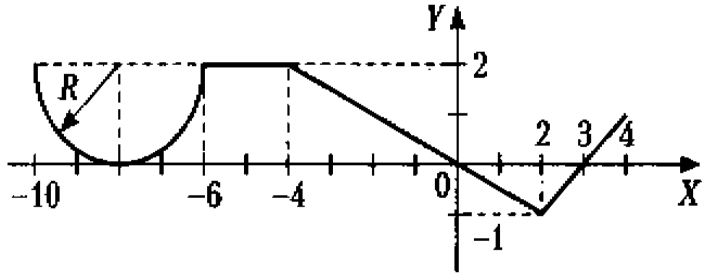
8


Y
4
2
$y = (x-6)^2$
-10
-8
R
-2
0
2
6
8
X
-2

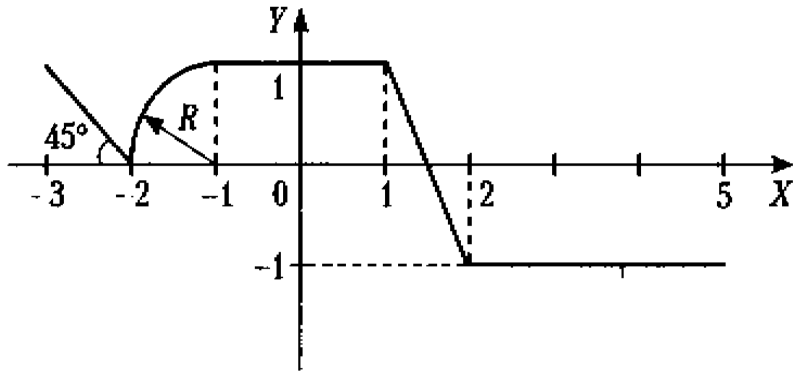
9


Y
4
2
-9
-7
-3
-2
-1
0
1
2
4
7
X

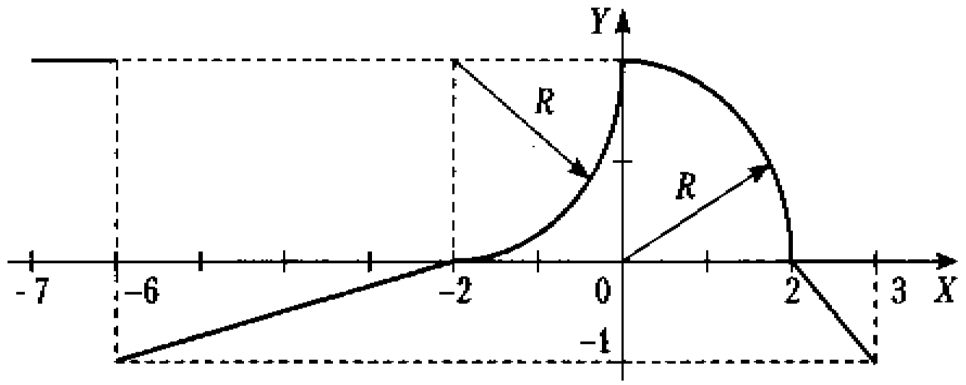
10


Y
2
R
2 3 4
-10
-6
-4
0
X
-1

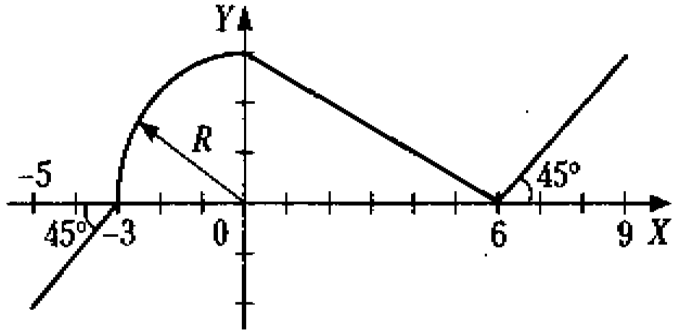
11


Y
1
R
45°
-3
-2
-1
0
1
2
5
X
-1

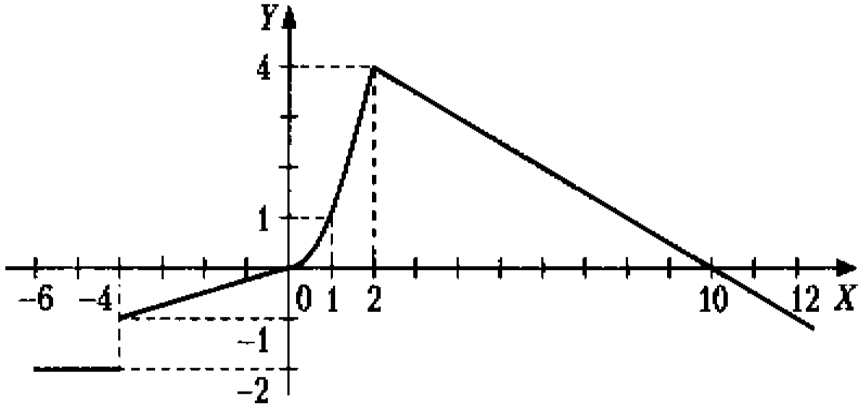
12


Y
R
R
-7
-6
-2
0
2
3
X
-1

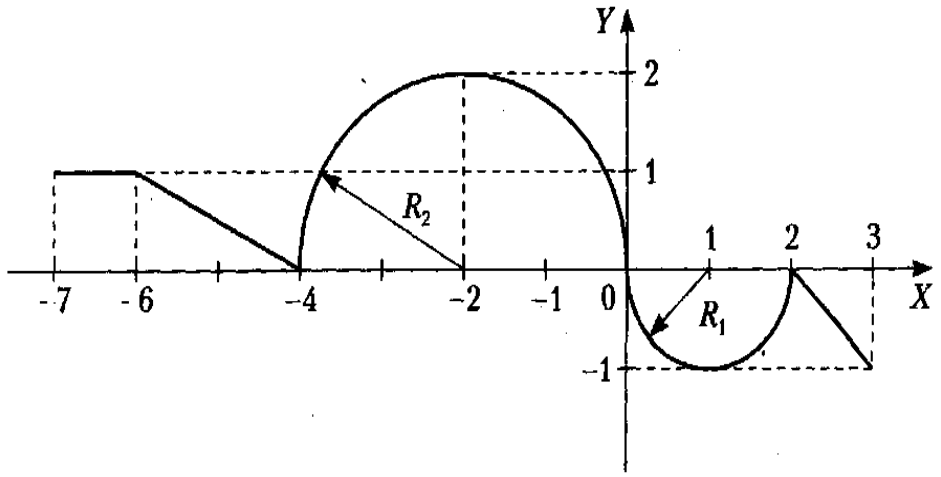
13


Y
R
-5
45°
45°
-3
0
6
9
X

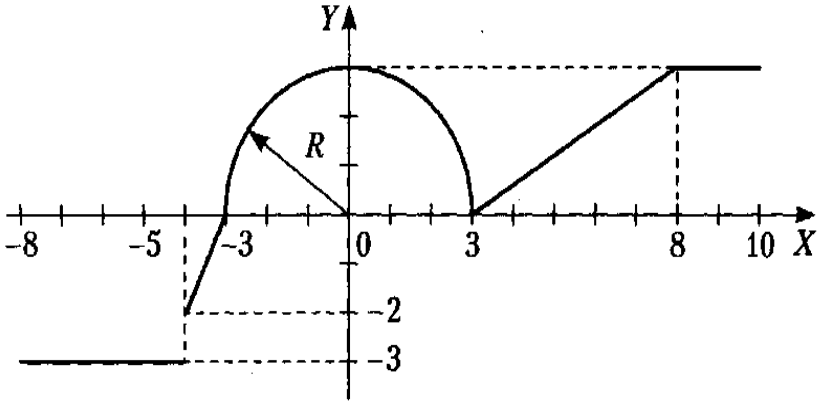
14


Y
4
1
-6
-4
0
1
2
10
12
X
-1
-2

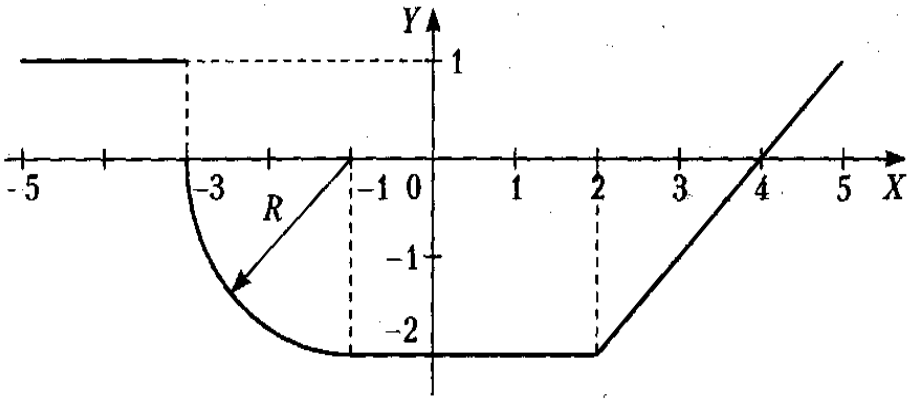
15


Y
2
1
$R_2$
1
2
3
-7
-6
-4
-2
-1
0
$R_1$
X
-1

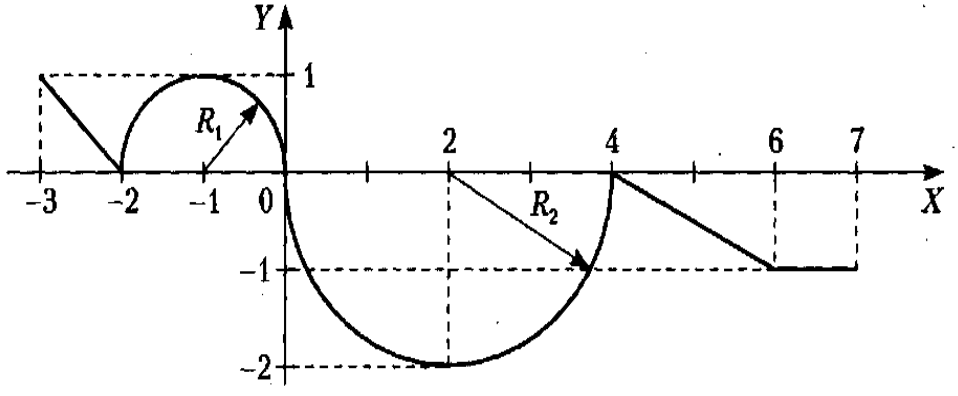
16


Y
R
-8
-5
-3
0
3
8
10
X
-2
-3

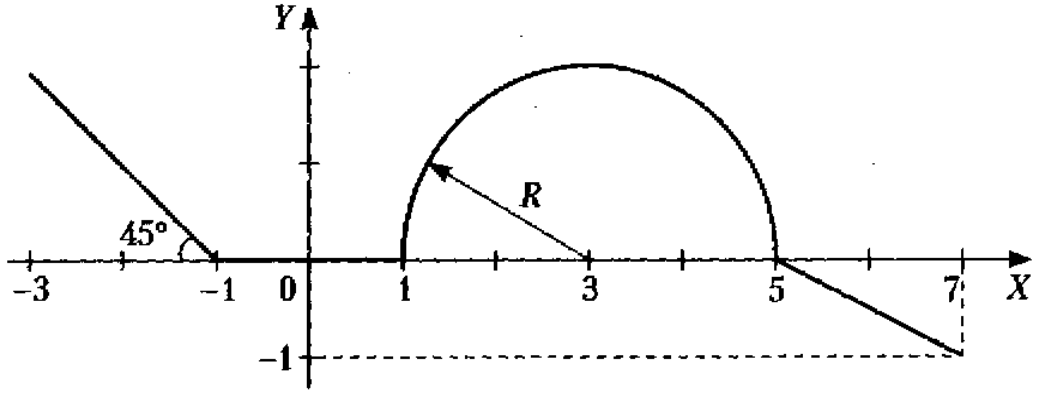
17


Y
1
-5
-3
R
-1
0
1
2
3
4
5
X
-1
-2

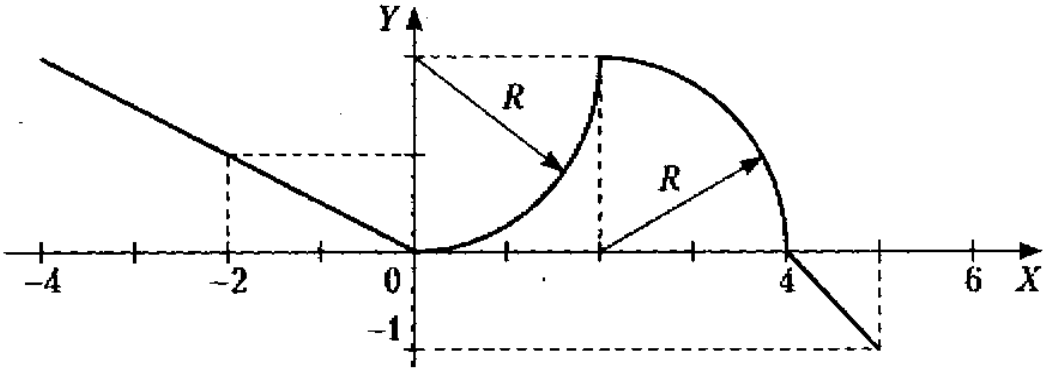
18


Y
1
$R_1$
2
4
6
7
-3
-2
-1
0
$R_2$
X
-1
-2

19




20

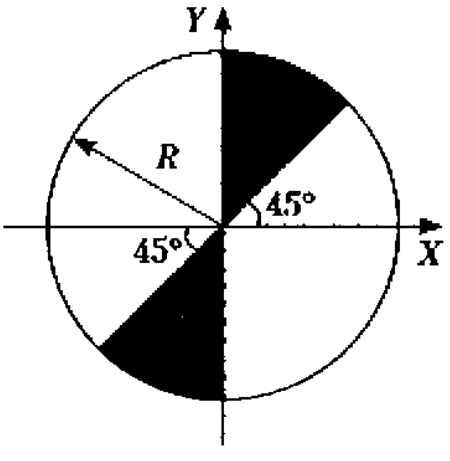



**Завдання 2. Потрапляння точки в заштриховану область**

Написати програму, яка визначає, чи потрапляє точка із заданими координатами в область, яка зафарбована на рисунку сірим кольором. Результат роботи програми вивести у вигляді текстового повідомлення.

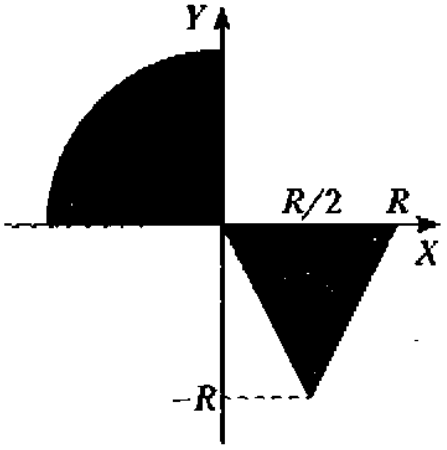
№ Область

1

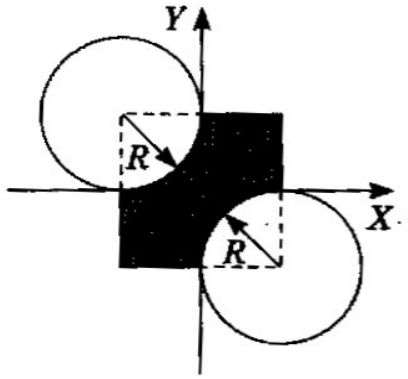



№ Область

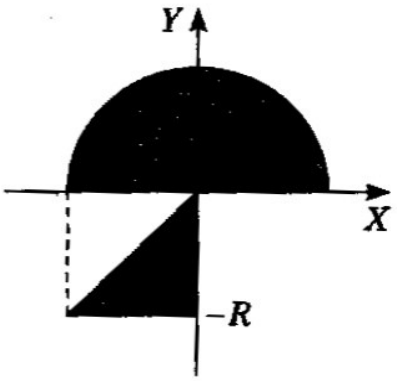
2



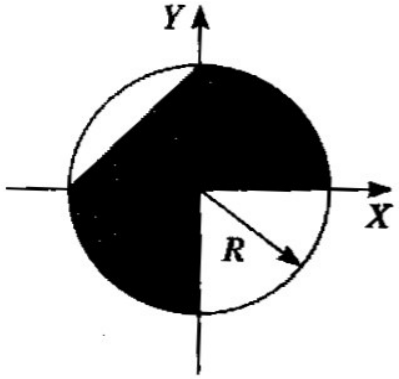
3



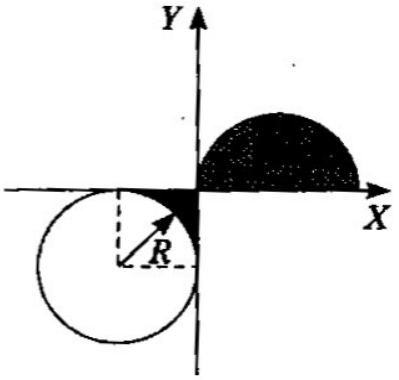

4



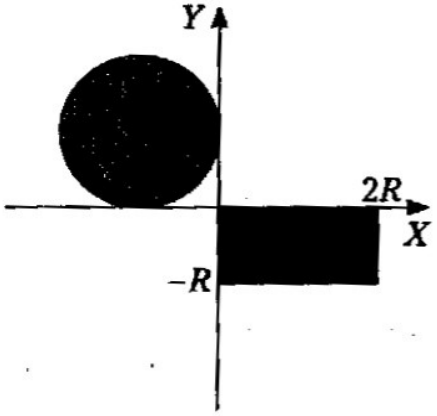

5



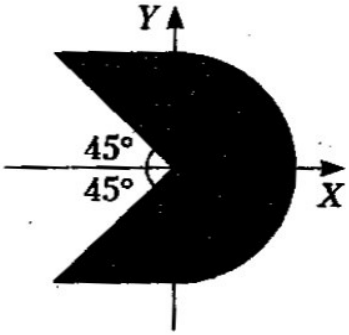

6



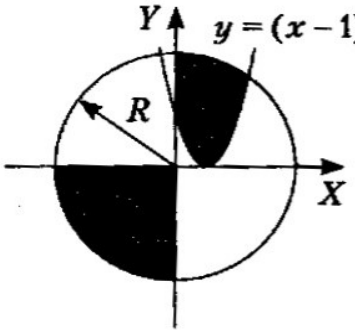

7



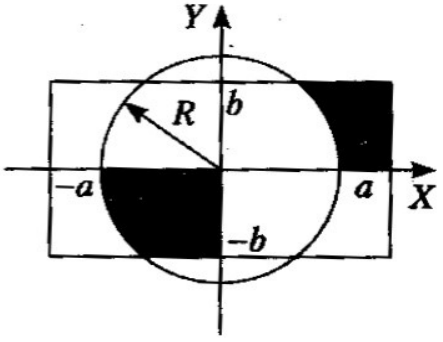

8



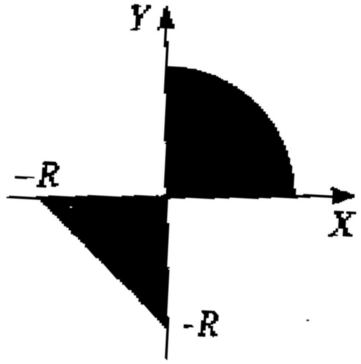

9



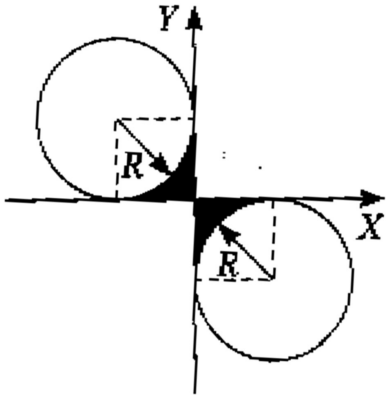

10



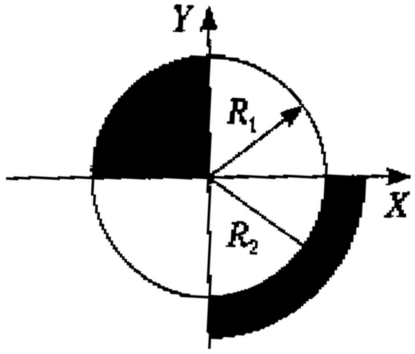
11



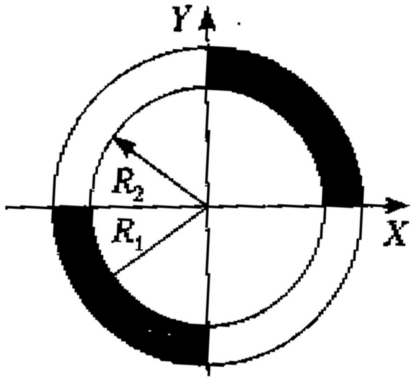

12



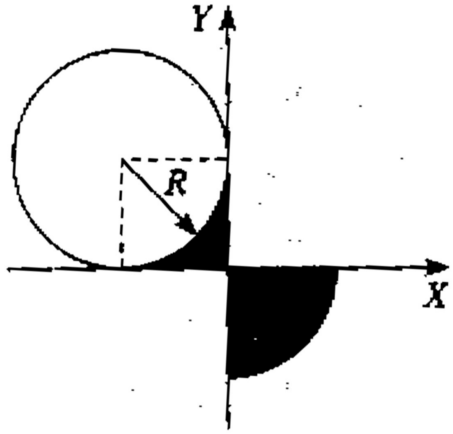

13



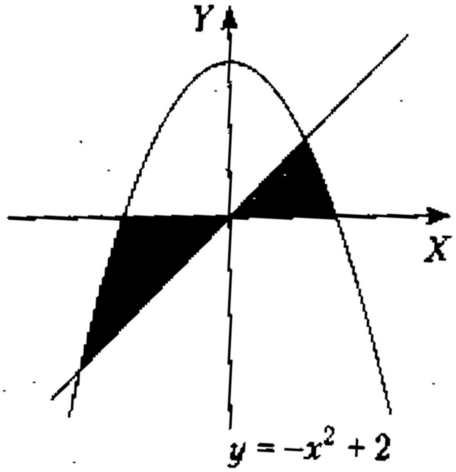

14



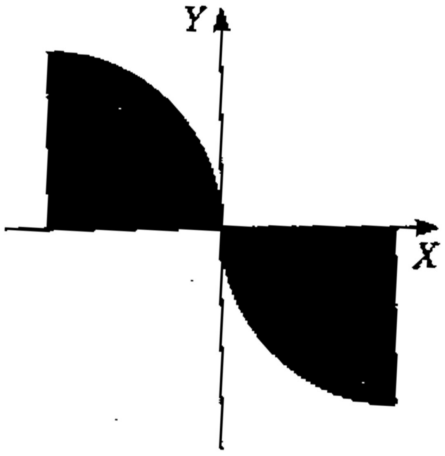

15



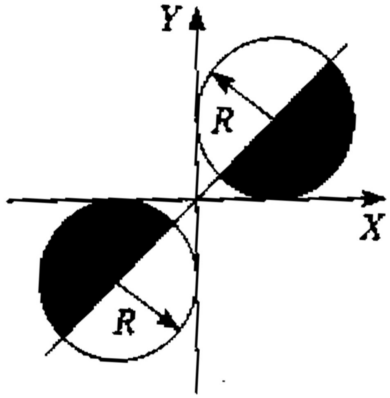

16



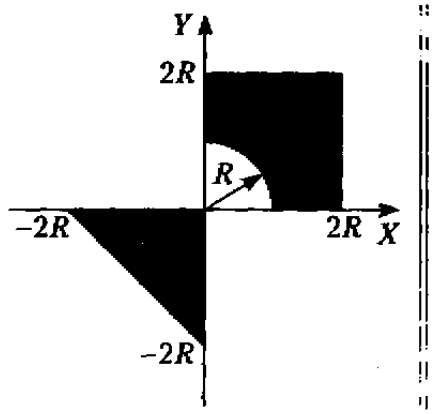

17



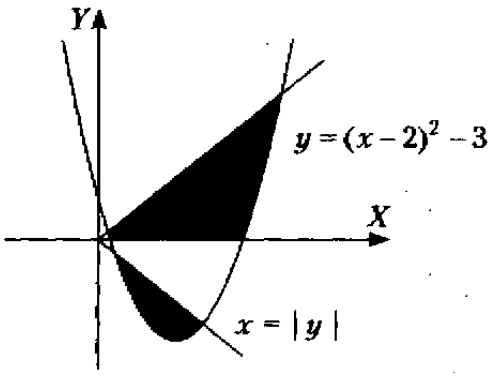

18



19




20




## Лабораторна робота 3. Організація циклів

Теоретичний матеріал: розділ 4.

### Завдання 1. Таблиця значень функції

Обчислити і вивести на екран у вигляді таблиці значення функції, заданої графічно (див. завдання 1 лабораторної роботи 2), на інтервалі від $x_{нач}$ до $x_{кон}$ з кроком *dx*. Інтервал і крок задати так, щоб перевірити всі гілки програми. Таблицю забезпечити заголовком і шапкою.

### Завдання 2. Серія пострілів по мішені

Для десяти пострілів, координати яких задаються з клавіатури, вивести текстові повідомлення про попадання в мішень із завдання 2 лабораторної роботи 2.

### Завдання 3. Ряди Тейлора

Обчислити і вивести на екран у вигляді таблиці значення функції, заданої за допомогою ряду Тейлора, на інтервалі від $x_{нач}$ до $x_{кон}$ з кроком *dx* з точністю *e*. Таблицю забезпечити заголовком і шапкою. Кожен рядок таблиці повинен містити значення аргументу, значення функції і кількість підсумованих членів ряду.

1. $\ln\dfrac{x+1}{x-1} = 2\sum\limits_{n=0}^{\infty}\dfrac{1}{(2n+1)x^{2n+1}} = 2\left(\dfrac{1}{x}+\dfrac{1}{3x^3}+\dfrac{1}{5x^5}+\ldots\right), |x|>1.$
2. $e^{-x} = \sum\limits_{n=0}^{\infty}\dfrac{(-1)^n x^n}{n!} = 1-x+\dfrac{x^2}{2!}-\dfrac{x^3}{3!}+\dfrac{x^4}{4!}-\ldots, |x|<\infty.$
3. $e^{x} = \sum\limits_{n=0}^{\infty}\dfrac{x^n}{n!} = 1+x+\dfrac{x^2}{2!}+\dfrac{x^3}{3!}+\dfrac{x^4}{4!}+\ldots, |x|<\infty.$
4. $\ln(x+1) = \sum\limits_{n=0}^{\infty}\dfrac{(-1)^n x^{n+1}}{n+1} = x-\dfrac{x^2}{2}+\dfrac{x^3}{3}-\dfrac{x^4}{4}+\ldots, -1<x\le 1.$

5. $\ln\frac{1+x}{1-x}=2\sum_{n=0}^{\infty}\frac{x^{2n+1}}{2n+1}=2\left(x+\frac{x^3}{3}+\frac{x^5}{5}+...\right),\ |x|<1.$

6. $\ln(1-x)=-\sum_{n=1}^{\infty}\frac{x^n}{n}=-\left(x+\frac{x^2}{2}+\frac{x^4}{4}+...\right),\ -1\le x<1.$

7. $arcctgx=\frac{\pi}{2}+\sum_{n=0}^{\infty}\frac{(-1)^{n+1}x^{2n+1}}{2n+1}=\frac{\pi}{2}-x+\frac{x^3}{3}-\frac{x^5}{5}+...,\ |x|\le 1.$

8. $arctgx=\frac{\pi}{2}+\sum_{n=0}^{\infty}\frac{(-1)^{n+1}}{(2n+1)x^{2n+1}}=\frac{\pi}{2}-\frac{1}{x}+\frac{1}{3x^3}-\frac{1}{5x^5}+...,\ x>1.$

9. $arctgx=\sum_{n=0}^{\infty}\frac{(-1)^n x^{2n+1}}{(2n+1)}=x-\frac{x^3}{3}+\frac{x^5}{5}-\frac{x^7}{7}+...,\ |x|\le 1.$

10. $arthx=\sum_{n=0}^{\infty}\frac{x^{2n+1}}{2n+1}=x+\frac{x^3}{3}+\frac{x^5}{5}+\frac{x^7}{7}+...,\ |x|<1.$

11. $arthx=\sum_{n=0}^{\infty}\frac{1}{(2n+1)x^{2n+1}}=\frac{1}{x}+\frac{1}{3x^3}+\frac{1}{5x^5}+...,\ |x|>1.$

12. $arctgx=-\frac{\pi}{2}+\sum_{n=0}^{\infty}\frac{(-1)^{n+1}}{(2n+1)x^{2n+1}}=-\frac{\pi}{2}-\frac{1}{x}+\frac{1}{3x^3}-\frac{1}{5x^5}+...,\ x\le 1.$

13. $e^{-x^2}=\sum_{n=0}^{\infty}\frac{(-1)^n x^{2n}}{n!}=1-x^2+\frac{x^4}{2!}-\frac{x^6}{3!}+\frac{x^8}{4!}-...,\ |x|<\infty.$

14. $\cos x=\sum_{n=0}^{\infty}\frac{(-1)^n x^{2n}}{(2n)!}=1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}+...,\ |x|<\infty.$

15. $\frac{\sin x}{x}=\sum_{n=0}^{\infty}\frac{(-1)^n x^{2n}}{(2n+1)!}=1-\frac{x^2}{3!}+\frac{x^4}{5!}-\frac{x^6}{7!}+...,\ |x|<\infty.$

16. $\ln x=2\sum_{n=0}^{\infty}\frac{(x-1)^{2n+1}}{(2n+1)(x+1)^{2n+1}}=2\left(\frac{x-1}{x+1}+\frac{(x-1)^3}{3(x+1)^3}+\frac{(x-1)^5}{5(x+1)^5}+...\right),\ x>0.$

17. $\ln x=\sum_{n=0}^{\infty}\frac{(-1)^n(x-1)^{n+1}}{(n+1)}=(x-1)-\frac{(x-1)^2}{2}+\frac{(x-1)^3}{3}+...,\ 0<x\le 2.$

18. $e^x=\sum_{n=0}^{\infty}\frac{x^n}{n!}=1+x+\frac{x^2}{2!}+\frac{x^3}{3!}+\frac{x^4}{4!}+...,\ |x|<\infty.$

19. $\arcsin x=x+\sum_{n=1}^{\infty}\frac{1\cdot 3...(2n-1)\cdot x^{2n+1}}{2\cdot 4...2n\cdot(2n+1)}=x+\frac{x^3}{2\cdot 3}+\frac{1\cdot 3\cdot x^5}{2\cdot 4\cdot 5}+\frac{1\cdot 3\cdot 5\cdot x^7}{2\cdot 4\cdot 6\cdot 7}+\frac{1\cdot 3\cdot 5\cdot 7\cdot x^9}{2\cdot 4\cdot 6\cdot 8\cdot 9}...,\ |x|<1.$

20. $\arccos x=\frac{\pi}{2}-\left(x+\sum_{n=1}^{\infty}\frac{1\cdot 3...(2n-1)\cdot x^{2n+1}}{2\cdot 4...2n\cdot(2n+1)}\right)=\frac{\pi}{2}-\left(x+\frac{x^3}{2\cdot 3}+\frac{1\cdot 3\cdot x^5}{2\cdot 4\cdot 5}+\frac{1\cdot 3\cdot 5\cdot x^7}{2\cdot 4\cdot 6\cdot 7}...\right),\ |x|<1.$

## Лабораторна робота 4. Прості класи

Теоретичний матеріал: розділ 5
Кожен клас, що розробляється, повинен, як правило, містити наступні елементи: приховані поля, конструктори з параметрами і без параметрів, методи, властивості. Методи і властивості повинні забезпечувати несуперечливий, повний, мінімальний і зручний інтерфейс класу. При виникненні помилок повинні викидатися виключення.
У програмі повинна виконуватися перевірка всіх розроблених елементів класу.

**Варіант 1**
Описати клас, що реалізовує десятковий лічильник, який може збільшувати або зменшувати своє значення на одиницю в заданому діапазоні. Передбачити ініціалізацію лічильника значеннями за умовчанням і довільними значеннями. Лічильник має два методи: збільшення і зменшення, - і властивість, що дозволяє отримати його поточний стан. При виході за межі діапазону викидаються виключення.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 2**
Описати клас, що реалізовує шістнадцятирічний лічильник, який може збільшувати або зменшувати своє значення на одиницю в заданому діапазоні. Передбачити ініціалізацію лічильника значеннями за умовчанням і довільними значеннями. Лічильник має два методи: збільшення і зменшення, - і властивість, що дозволяє отримати його поточне багатство. При виході за межі діапазону викидаються виключення.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 3**
Описати клас, що представляє трикутник. Передбачити методи для створення об'єктів, переміщення на плоскість, зміни розмірів і обертання на заданий кут. Описати властивості для отримання стану об'єкту. При неможливості побудови трикутника викидається виключення.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 4**
Побудувати опис класу, що містить інформацію про поштову адресу організації. Передбачити можливість роздільної зміни складових частин адреси і перевірки допустимості значень, що вводяться. У разі неприпустимих значень полів викидаються виключення.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 5**
Скласти опис класу для представлення комплексних чисел. Забезпечити виконання операцій складання, віднімання і множення комплексних чисел.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 6**
Скласти опис класу для вектора, заданого координатами його кінців в тривимірному просторі. Забезпечити операції складання і віднімання векторів з отриманням нового вектора (суми або різниці), обчислення скалярного твору двох векторів, довжини вектора, косинуса кута між векторами.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 7**
Скласти опис класу прямокутників із сторонами, паралельними осям координат. Передбачити можливість переміщення прямокутників на плоскість, зміну розмірів, побудову найменшого прямокутника, що містить два задані прямокутники, і прямокутники, що є загальною частиною (перетином) двох прямокутників.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 8**
Скласти опис класу для представлення дати. Передбачити можливості установки дати і зміни її окремих полів (рік, місяць, день) з перевіркою допустимості значень, що вводяться. У разі неприпустимих значень полів викидаються виключення. Створити методи зміни дати на задану кількість днів, місяців і років.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 9**
Скласти опис класу для представлення часу. Передбачити можливості установки часу і зміни його окремих полів (година, хвилина, секунда) з перевіркою допустимості значень, що вводяться. У разі неприпустимих значень полів викидаються виключення. Створити методи зміни часу на задану кількість годинника, хвилин і секунд.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 10**
Скласти опис класу многочленна виду $ax^2 + bx + c$. Передбачити методи, що реалізовують:
обчислення значення многочленна для заданого аргументу;
операцію складання, віднімання і множення многочленів з отриманням нового об'єкту-многочленна;
виведення на екран опису многочленна.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 11**
Описати клас, що представляє трикутник. Передбачити методи для створення об'єктів, обчислення площі, периметра і точки перетину медіан. Описати властивості для отримання стану об'єкту. При неможливості побудови трикутника викидається виключення.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 12**
Описати клас, що представляє круг. Передбачити методи для створення об'єктів, обчислення площі круга, довжини кола і перевірки попадання заданої точки всередину круга. Описати властивості для отримання стану об'єкту.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 13**
Описати клас для роботи з рядком, що дозволяє зберігати тільки двійкове число і виконувати з ним арифметичні операції. Передбачити ініціалізацію з перевіркою допустимості значень. У разі неприпустимих значень викидаються виключення.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 14**
Описати клас дробів - раціональних чисел, що є відношенням двох цілих чисел. Передбачити методи складання, віднімання, множення і ділення дробів.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 15**
Описати клас “файл”, що містить відомості про ім'я, дату створення і довжину файлу. Передбачити ініціалізацію з перевіркою допустимості значень полів. У разі неприпустимих значень полів викидаються виключення. Описати метод додавання інформації в кінець файлу і властивості для отримання стану файлу.
Написати програму, що демонструє всі розроблені елементи класу.

**Вариант 16**
Описати клас “кімната”, що містить відомості про метраж, висоту стель і кількість вікон. Передбачити ініціалізацію з перевіркою допустимості значень полів. У разі неприпустимих значень полів викидаються виключення. Описати методи обчислення площі і об'єму кімнати і властивості для отримання стану об'єкту.
Написати програму, що демонструє всі розроблені елементи класу.

**Вариант 17**
Описати клас, що представляє нелінійне рівняння ax - cos(x)= 0. Описати метод, що обчислює вирішення цього рівняння на заданому інтервалі методом ділення навпіл і що викидає виключення у разі відсутності кореня. Описати властивості для отримання стану об'єкту.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 18**
Описати клас, що представляє квадратне рівняння вигляду ax + bx + з = 0. Описати метод, що обчислює вирішення цього рівняння і що викидає виключення у разі відсутності кореня. Описати властивості для отримання стану об'єкту.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 19**
Описати клас “процесор”, що містить відомості про марку, тактову частоту, об'єм кеша і вартості. Передбачити ініціалізацію з перевіркою допустимості значень полів. У разі неприпустимих значень полів викидаються виключення. Описати властивості для отримання стану об'єкту. Описати клас “материнська плата”, що включає клас “процесор” і об'єм встановленої оперативної пам'яті. Передбачити ініціалізацію з перевіркою допустимості значень поля об'єму пам'яті. У разі неприпустимих значень поля викидається виключення. Описати властивості для отримання стану об'єкту.
Написати програму, що демонструє всі розроблені елементи класів.

**Варіант 20**
Описати клас “кольорова крапка”. Для крапки задаються координати і колір. Колір описується за допомогою трьох складових (червоний, зелений, синій). Передбачити різні методи ініціалізації об'єкту з перевіркою допустимості значень. Допустимим діапазоном для кожної складової є [0, 255]. У разі неприпустимих значень полів викидаються виключення. Описати властивості для отримання стану об'єкту і метод зміни кольору.
Написати програму, що демонструє всі розроблені елементи класу.

## Лабораторна робота 5. Одновимірні масиви

Теоретичний матеріал: Розділ 6.

**Варіант 1**
У одновимірному масиві, що складається з *n* речових елементів, обчислити:
- суму від’ємних елементів масиву;
- множення елементів масиву, розташованих між максимальним і мінімальним елементами.

Упорядкувати елементи масиву за збільшенням.

**Варіант 2**
У одновимірному масиві з *n* дійсних елементів обчислити:
- суму додатних елементів масиву;
- множення елементів масиву, розташованих між максимальним по модулю і мінімальним по модулю елементами.

Упорядкувати елементи масиву по убуванню.

**Варіант 3**
У одновимірному масиві з *n* цілих елементів обчислити:
- множення елементів масиву з парними номерами;
- суму елементів масиву, розташованих між першим і останнім нульовими елементами.

Перетворити масив так, щоб спочатку розташовувалися всі додатні елементи, а потім - всі від’ємні (елементи, рівні нулю, вважати додатними).

**Варіант 4**

У одновимірному масиві з *n* дійсних елементів обчислити:

- суму елементів масиву з непарними номерами;
- суму елементів масиву, розташованих між першим і останнім від'ємними елементами.

Стиснути масив, видаливши з нього всі елементи, модуль яких не перевищує одиницю. Елементи, що звільнилися в кінці масиву, заповнити нулями.

**Вариант 5**

У одновимірному масиві з *n* дійсних елементів обчислити:

- максимальний елемент масиву;
- суму елементів масиву, розташованих до останнього додатного елементу.

Стиснути масив, видаливши з нього всі елементи, модуль яких знаходиться в інтервалі [a, b]. Елементи, що звільнилися в кінці масиву, заповнити нулями.

**Варіант 6**

У одновимірному масиві з *n* дійсних елементів обчислити:

- мінімальний елемент масиву;
- суму елементів масиву, розташованих між першим і останнім додатними елементами.

Перетворити масив так, щоб спочатку розташовувалися всі елементи, рівні нулю, а потім - всі останні.

**Варіант 7**

У одновимірному масиві з *n* цілих елементів обчислити:

- номер максимального елементу масиву;
- множення елементів масиву, розташованих між першим і другим нульовими елементами.

Перетворити масив так, щоб в першій його половині розташовувалися елементи, що стояли в непарних позиціях, а в другій половині - елементи, що стояли в парних позиціях.

**Варіант 8**

У одновимірному масиві з *n* дійсних елементів обчислити:

- номер мінімального елементу масиву;
- суму елементів масиву, розташованих між першим і другим від'ємними елементами.

Перетворити масив так, щоб спочатку розташовувалися всі елементи, модуль яких не перевищує одиницю, а потім - всі останні.

**Варіант 9**

У одновимірному масиві з *n* дійсних елементів обчислити:

- максимальний по модулю елемент масиву;
- суму елементів масиву, розташованих між першим і другим додатними елементами.

Перетворити масив так, щоб елементи, рівні нулю, розташовувалися після всіх останніх.

**Варіант 10**

У одновимірному масиві з *n* цілих елементів обчислити:

- мінімальний по модулю елемент масиву;
- суму модулів елементів масиву, розташованих після першого елементу, рівного нулю.

Перетворити масив так, щоб в першій його половині розташовувалися елементи, що стояли в парних позиціях, а в другій половині - елементи, що стояли в непарних позиціях.

**Варіант 11**

У одновимірному масиві з *n* дійсних елементів обчислити:

- номер мінімального по модулю елементу масиву;
- суму модулів елементів масиву, розташованих після першого від'ємного елементу.

Стиснути масив, видаливши з нього всі елементи, величина яких знаходиться в інтервалі [a, b]. Елементи, що звільнилися в кінці масиву, заповнити нулями.

**Варіант 12**

У одновимірному масиві з *n* дійсних елементів обчислити:

- номер максимального по модулю елементу масиву;
- суму елементів масиву, розташованих після першого додатного елементу.

Перетворити масив так, щоб спочатку розташовувалися всі елементи, ціла частина яких лежить в інтервалі [a, d], а потім - всі останні.

**Варіант 13**

У одновимірному масиві з *n* дійсних елементів обчислити:

- кількість елементів масиву,що знаходяться в діапазоні від A до B;
- суму елементів масиву, розташованих після максимального елементу. Упорядкувати елементи масиву по убуванню модулів.

**Варіант 14**

У одновимірному масиві з *n* дійсних елементів обчислити:

- кількість елементів масиву, рівних нулю;
- суму елементів масиву, розташованих після мінімального елементу.

Упорядкувати елементи масиву за збільшенням модулів.

**Варіант 15**

У одновимірному масиві з *n* дійсних елементів обчислити:

- кількість елементів масиву, більших 3;
- множення елементів масиву, розташованих після максимального по модулю елементу.

Перетворити масив так, щоб спочатку розташовувалися всі від'ємні, а потім - всі додатні (елементи, рівні нулю, вважати додатними).

**Варіант 16**

У одновимірному масиві з *n* дійсних елементів обчислити:

- кількість від'ємних елементів масиву;
- суму модулів елементів масиву, розташованих після мінімального по модулю елементу.

Замінити всі від'ємні елементи масиву їх квадратами і упорядкувати елементи масиву за збільшенням.

**Варіант 17**

У одновимірному масиві з *n* цілих елементів обчислити:

- кількість додатних елементів масиву;
- суму елементів масиву, розташованих після останнього елементу, рівного нулю.

Перетворити масив так, щоб спочатку розташовувалися всі елементи, ціла частина яких не перевищує одиницю, а потім - всі останні.

**Варіант 18**

У одновимірному масиві з *n* дійсних елементів обчислити:

- кількість елементів масиву, менших *C*;
- суму цілих частин елементів масиву, розташованих після останнього від'ємного елементу.

Перетворити масив так, щоб спочатку розташовувалися всі елементи, що відрізняються від максимального не більше ніж на 20%, а потім - всі останні.

**Варіант 19**

У одновимірному масиві з *n* дійсних елементів обчислити:

- множення від'ємних елементів масиву;
- суму додатних елементів масиву, розташованих до максимального елементу.

Змінити порядок проходження елементів в масиві на зворотний.

**Варіант 20**

У одновимірному масиві з *n* дійсних елементів обчислити:

- множення додатних елементів масиву;
- суму елементів масиву, розташованих до мінімального елементу.

Упорядкувати за збільшенням окремо елементи, що стоять на парних місцях, і елементи, що стоять на непарних місцях.

## Лабораторна робота 6. Двовимірні масиви

Теоретичний матеріал: розділ 6.

**Варіант 1**
Дана цілочисельна прямокутна матриця. Визначити:

- кількість рядків, що не містять жодного нульового елементу;
- максимальне з чисел, що зустрічаються в заданій матриці більше одного разу.

**Варіант 2**
Дана цілочисельна прямокутна матриця. Визначити кількість стовпців, що не містять жодного нульового елементу.
Характеристикою рядка цілочисельної матриці назвемо суму її додатних парних елементів. Переставляючи рядки заданої матриці, розташувати їх відповідно до зростання характеристик.

**Варіант 3**
Дана цілочисельна прямокутна матриця. Визначити:

- кількість стовпців, які містять хоч би один нульовий елемент;
- номер рядка, в якому знаходиться щонайдовша серія однакових елементів.

**Варіант 4**
Дана цілочисельна квадратна матриця. Визначити:

- множину елементів в тих рядках, які не містять від'ємних елементів;
- максимум серед сум елементів діагоналей, паралельних головній діагоналі матриці.

**Варіант 5**
Дана цілочисельна квадратна матриця. Визначити:

- суму елементів в тих стовпцях, які не містять від'ємних елементів;
- мінімум серед сум модулів елементів діагоналей, паралельних побічній діагоналі матриці.

**Варіант 6**
Дана цілочисельна прямокутна матриця. Визначити:
суму елементів в тих рядках, які містять хоч би один від'ємний елемент; номери рядків і стовпців всіх седлових точок матриці.

**ПРИМІТКА**
**Матриця А має седловую точку Аij, якщо Аij є мінімальним елементом в *i* -у рядку і максимальним - в *j*-м стовпці.**

**Варіант 7**

Для заданої матриці розміром *8x8* знайти такі *k* при яких *k*-й рядок збігається з *k*-м стовпцем.

Знайти суму елементів в тих рядках, які містять хоч би один від'ємний елемент.

**Варіант 8**

Характеристикою стовпця цілочисельної матриці назвемо суму модулів його від'ємних непарних елементів. Переставляючи стовпці заданої матриці, розташувати їх відповідно до зростання характеристик.

Знайти суму елементів в тих стовпцях, які містять хоч би один від'ємний елемент.

**Варіант 9**

Сусідами елементу $A_{ij}$ в матриці назвемо елементи $A_{kl}$, де i - 1< *до* < i + 1, j - 1 < l < j + 1, (до, j) не рівно (i, j). Операція згладжування матриці дає нову матрицю того ж розміру, кожен елемент якої виходить як середнє арифметичне наявних сусідів відповідного елементу початкової матриці. Побудувати результат згладжування заданої речової матриці розміром 10 x 10.

У згладженій матриці знайти суму модулів елементів, розташованих нижче за головну діагональ.

**Варіант 10**

Елемент матриці називається локальним мінімумом, якщо він строго менше всіх його сусідів (визначення сусідніх елементів див. у варіанті 9). Підрахувати кількість локальних мінімумів заданої матриці розміром 10 x 10.

Знайти суму модулів елементів, розташованих вище за головну діагональ.

**Варіант 11**

Коефіцієнти системи лінійних рівнянь задані у вигляді прямокутної матриці. За допомогою допустимих перетворень привести систему до трикутного вигляду.

Знайти кількість рядків, середнє арифметичне елементів яких менше заданої величини.

**Варіант 12**

Ущільнити задану матрицю, видаляючи з неї рядки і стовпці, заповнені нулями.

Знайти номер першою з рядків, що містять хоч би один додатний елемент.

**Варіант 13**

Здійснити циклічне зрушення елементів прямокутної матриці *n* елементів управо або вниз (залежно від введеного режиму), *n* може бути більше кількості елементів в рядку або стовпці.

**Варіант 14**
Дана цілочисельна прямокутна матриця. Визначити номер першого із стовпців, що містять хоч би один нульовий елемент.
Характеристикою рядка цілочисельної матриці назвемо суму її від'ємних парних елементів. Переставляючи рядки заданої матриці, розташувати їх відповідно до убування характеристик.

**Варіант 15**
Упорядкувати рядки цілочисельної прямокутної матриці за збільшенням кількості однакових елементів в кожному рядку.
Знайти номер першого із стовпців, що не містять жодного від'ємного елементу.

**Варіант 16**
Шляхом перестановки елементів квадратної речової матриці добитися того, щоб її максимальний елемент знаходився в лівому верхньому кутку, наступний по величині - в позиції (2, 2), наступний по величині - в позиції (3, 3) і т. д., заповнивши всю головну діагональ.
Знайти номер першого з рядків, що не містить жодного додатного елементу.

**Варіант 17**
Дана цілочисельна прямокутна матриця. Визначити:
кількість рядків, що містять хоч би один нульовий елемент;
номер стовпця, в якому знаходиться щонайдовша серія однакових елементів.

**Варіант 18**
Дана цілочисельна квадратна матриця. Визначити:
суму елементів в тих рядках, які не містять від'ємних елементів;
мінімум серед сум елементів діагоналей, паралельних головній діагоналі матриці.

**Варіант 19**
Дана цілочисельна прямокутна матриця. Визначити:
- кількість від'ємних елементів в тих рядках, які містять хоч би один нульовий елемент;
- номери рядків і стовпців всіх седлових точок матриці (визначення сусідніх елементів див. у варіанті 9).

**Варіант 20**
Дана цілочисельна прямокутна матриця.
Упорядкувати елементи головної діагоналі.

## Лабораторна робота 7. Рядки

Теоретичний матеріал: розділ 6.

**Варіант 1**
Написати програму, яка прочитує з текстового файлу три речення і виводить їх в зворотному порядку.

**Варіант 2**
Написати програму, яка прочитує текст з файлу і виводить на екран тільки речення, що містять введене з клавіатури слово.

**Варіант 3**
Написати програму, яка прочитує текст з файлу і виводить на екран тільки рядки, що містять двозначні числа.

**Варіант 4**
Написати програму, яка прочитує англійський текст з файлу і виводить на екран слова, що починаються з голосних букв.

**Варіант 5**
Написати програму, яка прочитує текст з файлу і виводить його на екран, міняючи місцями кожні два сусідні слова.

**Варіант 6**
Написати програму, яка прочитує текст з файлу і виводить на екран тільки речення, що не містять ком.

**Варіант 7**
Написати програму, яка прочитує текст з файлу і визначає, скільки в нім слів, що складаються не більше ніж з чотирьох букв.

**Варіант 8**
Написати програму, яка прочитує текст з файлу і виводить на екран тільки цитати, тобто пропозиції, заключені в лапки.

**Варіант 9**
Написати програму, яка прочитує текст з файлу і виводить на екран тільки речення, що складаються із заданої кількості слів.

**Варіант 10**
Написати програму, яка прочитує англійський текст з файлу і виводить на екран слова тексту, що починаються і закінчуються на голосні букви.

**Варіант 11**
Написати програму, яка прочитує текст з файлу і виводить на екран тільки рядки, що не містять двозначних чисел.

**Варіант 12**
Написати програму, яка прочитує текст з файлу і виводить на екран тільки речення, що починаються з тире, перед яким можуть знаходитися тільки пробільні символи.

**Варіант 13**
Написати програму, яка прочитує англійський текст з файлу і виводить його на екран, замінивши прописною кожну першу букву слів, що починаються з голосної букви.

**Варіант 14**
Написати програму, яка прочитує текст з файлу і виводить його на екран, замінивши цифри від 0 до 9 словами “нуль”, “один”, “дев'ять”, починаючи кожне речення з нового рядка.

**Варіант 15**
Написати програму, яка прочитує текст з файлу, знаходить щонайдовше слово і визначає, скільки разів воно зустрілося в тексті.

**Варіант 16**
Написати програму, яка прочитує текст з файлу і виводить на екран спочатку питальні, а потім окличні речення.

**Варіант 17**
Написати програму, яка прочитує текст з файлу і виводить його на екран, після кожного речення додати слово яке  введене з клавіатури.

**Варіант 18**
Написати програму, яка прочитує текст з файлу і виводить на екран всі його речення в зворотному порядку.

**Варіант 19**
Написати програму, яка прочитує текст з файлу і виводить на екран спочатку речення, що починаються з однобуквених слів, а потім всі останні.

**Варіант 20**
Написати програму, яка прочитує текст з файлу і виводить на екран речення, яке містить максимальну кількість знаків пунктуації.

## Лабораторна робота 8. Класи і операції

Теоретичний матеріал: розділ 7.

Кожен клас, що розробляється, винен, як правило, містити наступні елементи: приховані поля, конструктори з параметрами і без параметрів, методи; властивості, індексатори; перевантажені операції. Функціональні елементи класу повинні забезпечувати несуперечливий, повний, мінімальний і зручний інтерфейс класу. При виникненні помилок повинні викидатися виключення.

У програмі повинна виконуватися перевірка всіх розроблених елементів класу.

### Варіант 1

Описати клас для роботи з одновимірним масивом цілих чисел (вектором). Забезпечити наступні можливості:

- завдання довільних цілих меж індексів при створенні об'єкту;
- звернення до окремого елементу масиву з контролем виходу за межі масиву;
- виконання операцій поелементного складання і віднімання масивів з однаковими межами індексів;
- виконання операцій множення і ділення всіх елементів масиву на скаляр;
- виведення на екран елементу масиву по заданому індексу і всього масиву. Написати програму, що демонструє всі розроблені елементи класу.

### Варіант 2

Описати клас для роботи з одновимірним масивом рядків фіксованої довжини. Забезпечити наступні можливості:

- звернення до окремого рядка масиву по індексу з контролем виходу за межі масиву;
- виконання операцій поелементного зчеплення двох масивів з утворенням нового масиву;
- виконання операцій злиття двох масивів з виключенням елементів, що повторюються;
- виведення на екран елементу масиву по заданому індексу і всього масиву.

Написати програму, що демонструє всі розроблені елементи класу.

### Варіант 3

Описати клас многочленів від однієї змінної, які задаються ступенем многочлена і масивом коефіцієнтів. Забезпечити наступні можливості:

- обчислення значення многочлена для заданого аргументу;
- операції складання, віднімання і множення многочленів, з отриманням нового об'єкту-многочлена;
- отримання коефіцієнта, заданого по індексу;
- виведення на екран опису многочлена.

Написати програму, яка демонструє всі розроблені елементи класу.

**Варіант 4**
Описати клас, що забезпечує представлення матриці довільного розміру з можливістю зміни числа рядків і стовпців, виводу на екран підматриці будь-якого розміру і всієї матриці, доступу по індексах до елементу матриці.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 5**
Описати клас для роботи з восьмиричним числом, що зберігається у вигляді рядка символів. Реалізувати конструктори, властивості, методи і наступні операції:

- операції привласнення, що реалізовують значущу семантику;
- операції порівняння;
- перетворення в десяткове число;
- виведення формату;
- доступ до заданої цифри числа по індексу.

Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 6**
Описати клас “домашня бібліотека”. Передбачити можливість роботи з довільним числом книг, пошуку книги за якою-небудь ознакою (по авторові, по року видання або категорії), додавання книг в бібліотеку, видалення книг з неї, доступу до книги по номеру.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 7**
Написати клас “записник”. Передбачити можливість роботи з довільним числом записів, пошуку запису за якою-небудь ознакою (наприклад, по прізвищу, даті народження або номеру телефону), додавання і видалення записів, сортування по прізвищу і доступу до запису по номеру.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 8**
Описати клас “студентська група”. Передбачити можливість роботи із змінним числом студентів, пошуку студента за якою-небудь ознакою (наприклад, по прізвищу, імені, даті народження), додавання і видалення записів, сортування по різних полях, доступу до запису по номеру.
Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 9**
Описати клас, що реалізовує тип даних «речова матриця» і роботу з ними. Клас повинен реалізовувати наступні операції над матрицями:

- складання, віднімання (як з іншою матрицею, так і з числом);
- комбіновані операції привласнення ( + =, -=);
- операції порівняння на рівність/нерівність;
- операції обчислення зворотної і транспонованої матриці;
- доступ до елементу по індексах.

Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 10**

Описати клас, що реалізовує тип даних “дійсна матриця” і роботу з ними. Клас повинен реалізовувати наступні операції над матрицями:

- множення, ділення (як на іншу матрицю, так і на число);
- комбіновані операції привласнення (* =, / = );
- операцію піднесення до ступеня;
- методи обчислення детермінанта і норми;
- доступ до елементу по індексах.

Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 11**

Описати клас, що реалізовує тип даних “дійсна матриця” і роботу з ними. Клас повинен реалізовувати наступні операції над матрицями:

- методи, що реалізовують перевірку типу матриці (квадратна, діагональна, нульова, одинична, симетрична, верхня трикутна, нижня трикутна);
- операції порівняння на рівність/нерівність;
- доступ до елементу по індексах.

Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 12**

Описати клас “множина”, що дозволяє виконувати основні операції: додавання і видалення елементу, перетин, об'єднання і різниця множин.

Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 13**

Описати клас “наочний покажчик”. Кожен компонент покажчика містить слово і номери сторінок, на яких, це слово зустрічається. Кількість номерів сторінок, що відносяться до одного слова, від одного до десяти. Передбачити можливість формування покажчика з клавіатури і з файлу, виведення покажчика, виведення номерів сторінок для заданого слова, видалення елементу з покажчика.

Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 14**

Описати клас “автостоянка” для зберігання відомостей про автомобілі. Для кожного автомобіля записуються госномер, колір, прізвище власника і ознака присутності на стоянці. Забезпечити можливість пошуку автомобіля по різних критеріях, виведення списку присутніх і відсутніх на стоянці автомобілів, доступу до наявних відомостей по номеру місця.

Написати програму, що демонструє всі розроблені елементи класу.

**Варіант 15**

Описати клас “колода карт”, що включає закритий масив елементів класу “карта”. У карті зберігаються масть і номер. Забезпечити можливість виведення карти по номеру, виведення всіх карт, перемішування колоди і видачі всіх карт з колоди поодинці і по 6 штук у випадковому порядку.

Написати програму, що демонструє всі розроблені елементи класів.

**Варіант 16**

Описати клас “поїзд”, що містить наступні закриті поля:

назва пункту призначення;

номер поїзда (може містити букви і цифри);

час відправлення.

Передбачити властивості для отримання стану об'єкту.

Описати клас “вокзал”, що містить закритий масив поїздів. Забезпечити наступні можливості:

виведення інформації про поїзд по номеру за допомогою індексу;

виведення інформації про поїзди, що відправляються після введеного з клавіатури часу;

перевантажену операцію порівняння, що виконує порівняння часу відправлення двох поїздів;

виведення інформації про поїзди, що відправляються в заданий пункт призначення.

Інформація має бути відсортована за часом відправлення. Написати програму, що демонструє всі розроблені елементи класів

**Варіант 17**

Описати клас “товар”, що містить наступні закриті поля:

- назва товару;
- назва магазина, в якому продається товар;
- вартість товару в гривнах.

Передбачити властивості для отримання стану об'єкту.

Описати клас “склад”, що містить закритий масив товарів. Забезпечити наступні можливості:

- виведення інформації про товар по номеру за допомогою індексу;
- виведення на екран інформації про товар, назва якого введена з клавіатури; якщо таких товарів немає, видати відповідне повідомлення;
- сортування товарів по назві магазина, по найменуванню і за ціною;
- перевантажену операцію складання товарів, що виконує складання їх цін. Написати програму, що демонструє всі розроблені елементи класів.

**Варіант 18**

Описати клас “літак”, що містить наступні закриті поля:

- назва пункту призначення;
- шестизначний номер рейса;
- час відправлення.

Передбачити властивості для отримання стану об'єкту.

Описати клас “аеропорт”, що містить закритий масив літаків. Забезпечити наступні можливості:

- виведення інформації про літак по номеру рейса за допомогою індексу;
- виведення інформації про літаки, що відправляються протягом години після введеного з клавіатури часу;
- виведення інформації про літаки, що відправляються в заданий пункт призначення;
- перевантажену операцію порівняння, що виконує порівняння часу відправлення двох літаків.

Інформація має бути відсортована за часом відправлення. Написати програму, що демонструє всі розроблені елементи класів.

**Варіант 19**

Описати клас “запис”, що містить наступні закриті поля:

- прізвище, ім'я;
- номер телефону;
- дата народження (масив з трьох чисел).

Передбачити властивості для отримання стану об'єкту.

Описати клас “записник”, що містить закритий масив записів. Забезпечити наступні можливості:

- виведення на екран інформації про людину, номер телефону які введений з клавіатури; якщо такого немає, видати на дисплей відповідне повідомлення;
- пошук людей, день народження яких сьогодні або в заданий день;
- пошук людей, день народження яких буде наступного тижня;
- пошук людей, номер телефону яких починається на три задані цифри.

Написати програму, що демонструє всі розроблені елементи класів.

**Варіант 20**

Описати клас “англо-російський словник”, що забезпечує можливість зберігання декількох варіантів перекладу для кожного слова. Реалізувати доступ по рядковому індексу - англійському слову. Забезпечити можливість виведення всіх значень слів по заданому префіксу.

## Лабораторна робота 9. Спадкоємство

Теоретичний матеріал: розділ 8.
У програмах потрібно описати базовий клас (можливо, абстрактний), в якому за допомогою віртуальних або абстрактних методів і властивостей задається інтерфейс для похідних класів. Метою лабораторної роботи є максимальне використання спадкоємства, навіть якщо для конкретного завдання воно не дає виграшу в об'ємі програми. У всіх класах слід перевизначити метод Equals, щоб забезпечити порівняння значень, а не посилань.
Функція *Main* повинна містити масив з елементів базового класу, заповнений посиланнями на похідні класи. У цій функції повинне демонструватися використання всіх розроблених елементів класів.

### Варіант 1
Створити клас Point (крапка). На його основі створити класи *ColoredPoint* і *Line* (лінія) . На основі класу *Line* створити класи *Coloredline* і *Polyline* (багатокутник) . У класах описати наступні елементи:

- конструктори з параметрами і конструктори за умовчанням;
- властивості для установки і набуття значень всіх координат, а також для зміни кольору і отримання поточного кольору;
- для ліній - методи зміни кута повороту ліній щодо першої крапки;
- для багатокутника - метод масштабування.

### Варіант 2
Створити абстрактний клас Vehicle (транспортний засіб). На його основі реалізувати класи Plane (літак), Car (автомобіль) і Ship (корабель). Класи повинні мати можливість задавати і отримувати координати і параметри засобів пересування (ціна, швидкість, рік випуску і т. п.) за допомогою властивостей. Для літака має бути визначена висота, для літака і корабля - кількість пасажирів, для корабля - порт приписки. Динамічні характеристики задати за допомогою методів.

### Варіант 3
Описати базовий клас *Рядок*. Обов'язкові поля класу:

- поле для зберігання символів рядка;
- значення типу *word* для зберігання довжини рядка в байтах.

Реалізувати обов'язкові методи наступного призначення:

- конструктор без параметрів;
- конструктор, що приймає як параметр рядковий літерал;
- конструктор, що приймає як параметр символ;
- метод отримання довжини рядка;
- метод очищення рядка (зробити рядок порожнім).

Описати похідний від *Рядок* клас *Комплексне_число*.
Рядки даного класу складаються з двох полів, розділених символом *i*.
Перше поле задає значення дійсної частини числа, друге, - значення уявної.

Кожне з полів може містити тільки символи десяткових цифр і символи - і +, задаючи знак числа. Символи - або + можуть знаходитися тільки в першій позиції числа, причому символ + може бути відсутнім, в цьому випадку число вважається додатним. Якщо у складі ініціалізованого рядка будуть зустрінуті будь-які символи, відмінні від допустимих, клас *Комплексне_число* приймає нульове значення. Приклади рядків:
33i12
-7i100
+5i - 21
Для класу Комплексне_число визначити наступні методи:

- перевірка на рівність;
- складання чисел;
- множення чисел.

**Варіант 4**
Описати базовий клас Рядок відповідно до варіанту 3. Описати похідний від Рядок клас Десятичний_ рядок.
Рядки даного класу можуть містити тільки символи десяткових цифр і символи - і +, задаючи знак числа. Символи - або + можуть знаходитися тільки в першій позиції числа, причому символ + може бути відсутнім, в цьому випадку число вважається додатним. Якщо у складі рядка будуть зустрінуті будь-які символи, відмінні від допустимих, клас Десятичний_рядок приймає нульове значення. Вміст даних рядків розглядається як десяткове число.
Для класу визначити наступні методи:

- конструктор, що приймає як параметр число;
- арифметична різниця рядків;
- перевірка на більше (за значенням);
- перевірка на менше (за значенням).

**Вариант 5**
Описати базовий клас *Рядок* відповідно до варіанту 3.
Описати похідний від *Рядок* клас *Бітовий_ рядок*.
Рядки даного класу можуть містити тільки символи '0' або ' 1'. Якщо у складі рядка будуть зустрінуті будь-які символи, відмінні від допустимих, клас *Бітовий_ рядок* приймає нульове значення. Вміст даних рядків розглядається як двійкове число. Від'ємні числа зберігаються в додатковому коді.
Для класу *Бітовий_ рядок* визначити наступні методи:

- конструктор, що приймає як параметр рядковий літерал;
- деструктор;
- зміна знаку на протилежний (переклад числа в додатковий код);
- привласнення;
- обчислення арифметичної суми рядків;
- перевірка на рівність.

У разі потреби коротший бітовий рядок розширюється вліво знаковим розрядом.

**Варіант 6**

1. Описати базовий клас *Елемент*.

Закриті поля:

- ім'я елементу (рядок символів);
- кількість входів елементу;
- кількість виходів елементу.

Методи:

- конструктор класу без параметрів;
- конструктор, який задає ім'я і встановлює рівним 1 кількість входів і виходів;
- конструктор, який задає им’я значення всіх полів елементу.

Властивості:

- ім'я елементу (тільки читання);
- кількість входів елементу;
- кількість виходів елементу.

2. На основі класу *Елемент* описати похідний клас *Комбінаційний*, такий, що є комбінаційним елементом (двійковий вентиль), який може мати декілька входів і один вихід.

Поле - масив значень входів.

Методи:

- конструктори;
- метод, який задає значення на входах екземпляра класу;
- метод, що дозволяє опитувати стан окремого входу екземпляра класу;
- метод, що обчислює значення виходу (по варіанту завдання).

3. На основі класу *Елемент* описати похідний клас *Пам'ять*, що є тригером. Тригер має входи, відповідні типу тригера (див. далі варіант завдання), і входи установки і скидання. Всі тригери вважаються за синхронні, сам синхровхід до складу тригера не включається.

Поля:

- масив значень входів об'єкту класу, в масиві враховуються всі входи (що управляють і інформаційні);
- перебування на прямому виході тригера;
- перебування на інверсному виході тригера.

Методи:

- конструктор (за умовчанням скидає екземпляр класу);
- конструктор копіювання;
- метод, який задає значення на входах екземпляра класу;
- методи, що дозволяють опитувати стани окремого входу екземпляра класу;
- метод, що обчислює стан екземпляра класу (по варіанту завдання) залежно від поточного перебування і значень на входах;
- метод, для перевантаження операції = = .

4 . Створити клас *Регістр*, використовуючи клас *Пам'ять* як вкладений клас.

Поля:

- стан входу “Скидання” - один для екземпляра класу;
- стан входу “Установка” - один для екземпляра класу;
- масив типу *Пам'ять* заданої у варіанті розмірності;
- масив (масиви), що містить значення на відповідних входах елементів масиву типу *Пам'ять*.

Методи:

- метод, що задає значення на входах екземпляра класу;
- метод, що дозволяє опитувати стан окремого виходу екземпляра класу;
- метод, що обчислює значення нового стану екземпляра класу.

Всі поля класів *Елемент*, *Комбінаційний* і *Пам'ять* мають бути описані з ключовим словом *private*.

У завданні перераховані тільки обов'язкові члени і методи класу. Можна задавати додаткові члени і методи, якщо вони не відміняють обов'язкові і забезпечують додаткові зручності при роботі з даними класами, наприклад, описати функції обчислення виходу/стану як віртуальні.

5. Для перевірки функціонування створених класів написати програму, що використовує ці класи. У програмі мають бути продемонстровані всі властивості створених класів.

Конкретний тип комбінаційного елементу, тип тригера і розрядність регістра вибираються відповідно до варіанту завдання:

| Варіант | Комбінаційний елемент | Число входів | Тригер | Розрядність регістра |
|---|---|---|---|---|
| 1 | И-НЕ | 4 | RS | 8 |
| 2 | ИЛИ | 5 | RST | 10 |
| 3 | МОД2-НЕ | 6 | D | 12 |
| 4 | И | 8 | T | 8 |
| 5 | ИЛИ-НЕ | 8 | V | 9 |
| 6 | И | 4 | RS | 10 |
| 7 | ИЛИ-НЕ | 5 | JK | 11 |
| 8 | МОД2 | 5 | D | 8 |
| 9 | И | 4 | T | 10 |
| 10 | ИЛИ | 3 | JK | 8 |
| 11 | И-НЕ | 3 | RS | 12 |
| 12 | ИЛИ-НЕ | 4 | RST | 4 |
| 13 | МОД2 | 5 | D | 10 |
| 14 | МОД2-НЕ | 6 | T | 10 |
| 15 | ИЛИ-НЕ | 8 | V | 10 |
| 16 | И | 8 | JK | 6 |
| 17 | И-НЕ | 8 | RS | 10 |
| 18 | ИЛИ | 8 | T | 10 |
| 19 | МОД2 | 6 | JK | 8 |
| 20 | МОД2-НЕ | 5 | V | 10 |

## Лабораторна робота 10. Структури

Теоретичний матеріал: розділ 9 .

**Варіант 1**

Описати структуру з ім'ям *STUDENT*, що містить наступні поля:

- прізвище і ініціали;
- номер групи;
- успішність (масив з п'яти елементів).

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з десяти структур типу *STUDENT* (записи мають бути впорядковані за збільшенням номера групи);
- виведення на екран прізвищ і номерів груп для всіх студентів, включених, в масив, якщо середній бал студента більше 4,0 (якщо таких студентів немає, вивести відповідне повідомлення).

**Варіант 2**

Описати структуру з ім'ям *STUDENT*, що містить наступні поля:

- прізвище і ініціали;
- номер групи;
- успішність (масив з п'яти елементів).

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з 10 структур типу *STUDENT* (записи мають бути впорядковані за збільшенням середнього балу) ;
- виведення на екран прізвищ і номерів груп для всіх студентів, що мають оцінки 4 і 5 (якщо таких студентів немає, вивести відповідне повідомлення).

**Варіант 3**

Описати структуру з ім'ям *STUDENT*, що містить наступні поля:

- прізвище і ініціали;
- номер групи;
- успішність (масив з п'яти елементів).

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з 10 структур типу STUDENT (записи мають бути впорядковані за алфавітом);
- виведення на екран прізвищ і номерів груп для всіх студентів, що мають хоч би одну оцінку 2 (якщо таких студентів немає, вивести відповідне повідомлення).

**Варіант 4**

Описати структуру з ім'ям AEROFLOT, що містить наступні поля:

- назва пункту призначення рейса;
- номер рейса;
- тип літака.

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з 7 елементів типу *AEROFLOT* (записи мають бути впорядковані за збільшенням номера рейсу);
- виведення на екран номерів рейсів і типів літаків, що вилітають в пункт призначення, назва якого збіглася з назвою, введеною з клавіатури (якщо таких рейсів немає, вивести відповідне повідомлення).

**Варіант 5**

Описати структуру з ім'ям *AEROFLOT*, що містить наступні поля:

- назва пункту призначення рейса;
- номер рейса;
- тип літака.

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з семи елементів типу *AEROFLOT* (записи мають бути розміщені в алфавітному порядку по назвах пунктів призначення);
- виведення на екран пунктів призначення і номерів рейсів, що обслуговуються літаком, тип якого введений з клавіатури (якщо таких рейсів немає, вивести відповідне повідомлення).

**Варіант 6**

Описати структуру з ім'ям *WORKER*, що містить наступні поля:

- прізвище і ініціали працівника;
- назва посади;
- рік вступу на роботу.

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з 10 структур типу *WORKER* (записи мають бути впорядковані за алфавітом);
- виведення на екран прізвищ працівників, стаж роботи яких перевищує значення, введене з клавіатури (якщо таких працівників немає, вивести відповідне повідомлення).

**Варіант 7**

Описати структуру з ім'ям *TRAIN*, що містить наступні поля:

- назва пункту призначення;
- номер поїзда;
- час відправлення.

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з восьми елементів типу *TRAIN* (записи мають бути розміщені в алфавітному порядку по назвах пунктів призначення);
- виведення на екран інформації про поїзди, що відправляються після введеного з клавіатури часу (якщо таких поїздів немає, вивести відповідне повідомлення).

**Варіант 8**

Описати структуру з ім'ям *TRAIN*, що містить наступні поля:

- назва пункту призначення;
- номер поїзда;
- час відправлення.

Написати програму що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з 6 елементів типу *TRAIN* (записи мають бути впорядковані за часом відправлення поїзда);
- виведення на екран інформації про поїзди, що прямують в пункт, назва якого введена з клавіатури (якщо таких поїздів немає, вивести відповідне повідомлення).

**Варіант 9**

Описати структуру з ім'ям *TRAIN*, що містить наступні поля:

- назва пункту призначення;
- номер поїзда;
- час відправлення.

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з 8 елементів типу *TRAIN* (записи мають бути впорядковані по номерах поїздів);
- виведення на екран інформації про поїзд, номер якого введений з клавіатури (якщо таких поїздів немає, вивести відповідне повідомлення).

**Варіант 10**

Описати структуру з ім'ям *MARSH*, що містить наступні поля:

- назва початкового пункту маршруту;
- назва кінцевого пункту маршруту;
- номер маршруту.

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з восьми елементів типу *MARSH* (записи мають бути впорядковані по номерах маршрутів);
- виведення на екран інформації про маршрут, номер якого введений з клавіатури (якщо таких маршрутів немає, вивести відповідне повідомлення).

**Варіант 11**

Описати структуру з ім'ям *MARSH*, що містить наступні поля:

- назва початкового пункту маршруту;
- назва кінцевого пункту маршруту;
- номер маршруту.

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з 8 елементів типу *MARSH* (записи мають бути впорядковані по номерах маршрутів);
- виведення на екран інформації про маршрути, які починаються або закінчуються в пункті, назва якого введена з клавіатури (якщо таких маршрутів немає, вивести відповідне повідомлення).

**Варіант 12**

Описати структуру з ім'ям *NOTE*, що містить наступні поля:

- прізвище, ім'я;
- номер телефону;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з восьми елементів типу *NOTE* (записи мають бути впорядковані по даті народження);
- виведення на екран інформації про людину, номер телефону якого введений з клавіатури (якщо такого немає, вивести відповідне повідомлення).

**Варіант 13**

Описати структуру з ім'ям *NOTE*, що містить наступні поля:

- прізвище, ім'я;
- номер телефону;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з 8 елементів типу *NOTE* (записи мають бути розміщені за алфавітом);
- виведення на екран інформації про людей, чиї дні народження доводяться на місяць, значення якого введене з клавіатури (якщо таких немає, вивести відповідне повідомлення).

**Варіант 14**

Описати структуру з ім'ям *NOTE*, що містить наступні поля:

- прізвище, ім'я;
- номер телефону;
- дата народження (масив з 3 чисел).

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з 8 елементів типу *NOTE* (записи мають бути впорядковані по трьом першим цифрам номера телефону);
- виведення на екран інформації про людину, чиє прізвище введене з клавіатури (якщо такого немає, вивести відповідне повідомлення).

**Варіант 15**

Описати структуру з ім'ям *ZNAK*, що містить наступні поля:

- прізвище, ім'я;
- знак Зодіаку;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з 8 елементів типу *ZNAK* (записи мають бути впорядковані по даті народження);
- виведення на екран інформації про людину, чиє прізвище введене з клавіатури (якщо такого немає, вивести відповідне повідомлення).

**Варіант 16**

Описати структуру з ім'ям *ZNAK*, що містить наступні поля:

- прізвище, ім'я;
- знак Зодіаку;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з 8 елементів типу *ZNAK* (записи мають бути впорядковані по даті народження);
- виведення на екран інформації про людей, що народилися під знаком, назва якого введена з клавіатури (якщо таких немає, вивести відповідне повідомлення).

**Варіант 17**

Описати структуру з ім'ям *ZNAK*, що містить наступні поля:

- прізвище, ім'я;
- знак Зодіаку;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з восьми елементів типу *ZNAK* (записи мають бути впорядковані по знаках Зодіаку);
- виведення на екран інформації про людей, що народилися в місяць, значення якого введене з клавіатури (якщо таких немає, вивести відповідне повідомлення).

**Варіант 18**

Описати структуру з ім'ям *PRICE*, що містить наступні поля:

- назва товару;

- назва магазина, в якому продається товар;
- вартість товару в гривнах.

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з 8 елементів типу PRICE (записи мають бути впорядковані в алфавітному порядку по назвах товарів);
- виведення на екран інформації про товар, назва якого введена з клавіатури (якщо таких товарів немає, вивести відповідне повідомлення).

**Варіант 19**

Описати структуру з ім'ям *PRICE*, що містить наступні поля:

- назва товару;
- назва магазина, в якому продається товар;
- вартість товару в гривнах.

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з восьми елементів типу PRICE (записи мають бути впорядковані в алфавітному порядку по назвах магазинів);
- виведення на екран інформації про товари, що продаються в магазині, назва якого введена з клавіатури (якщо такого магазина немає, вивести відповідне повідомлення).

**Варіант 20**

Описати структуру з ім'ям *ORDER*, що містить наступні поля:

- розрахунковий рахунок платника;
- розрахунковий рахунок одержувача;
- перерахована сума в гривнах.

Написати програму, що виконує наступні дії:

- введення з клавіатури даних в масив, що складається з восьми елементів типу *ORDER* (записи мають бути розміщені в алфавітному порядку по розрахункових рахунках платників);
- виведення на екран інформації про суму, зняту з розрахункового рахунку платника, введеного з клавіатури (якщо такого розрахункового рахунку немає, вивести відповідне повідомлення).

## Лабораторна робота 11. Інтерфейси і параметризовані колекції

Теоретичний матеріал: розділи 9, 13.

Виконати завдання лабораторної роботи 9, використовуючи для зберігання екземплярів розроблених класів стандартні параметризовані колекції.

У всіх класах реалізувати інтерфейс *IComparable* і перенавантажувати операції відношення для реалізації значущої семантики порівняння об'єктів по якому-небудь полю на розсуд студента.

## СПИСОК ЛІТЕРАТУРИ

1. Павловская Т.А. C#. Программирование на языке высокого уровня. Учебник для вузов. – СПб.: Питер, 2009. – 432 с.
2. Биллиг В. А. Основы программирования на C#. - М. : Изд-во «Интернет-университет информационных технологий - ИНТУИТ.ру» , 2006. - 488 с.
3. Брукс Ф. Мифический человеко-месяц, или как создаются программные комплексы. - М. : Символ-Плюс, 2000. - 304 с.
4. Ватсон К. C#. - М. : Лори, 2004. - 880 с.
5. Вирт Н. Алгоритмы и структуры данных. - СПб: Невский диалект, 2001. - 352 с.
6. Гиббонз П. Платформа .NET для Java-программистов. - СПб.: Питер, 2003. - 336 с.

# ДОДАТКИ

## Додаток 1. Специфікатори формату для рядків C#

Специфікатори використовуються для форматування арифметичних типів при їх перетворенні в рядкове уявлення.

| Специ-фікатор | Опис |
|---|---|
| C або c | Виведення значень в грошовому (currency) форматі. За умовчанням перед значенням, що виводиться, підставляється символ долара ($). Цей символ, а також задану за умовчанням кількість позицій можна змінити за допомогою об'єкту NumberFormatInfo. Безпосередньо після специфікатора можна задати ціле число, що визначає довжину дробової частини |
| D або d | Виведення цілих значень. Безпосередньо після специфікатора можна задати ціле число, що визначає ширину поля виводу. Пусті місця заповнюються нулями, наприклад, виведення числа 12 по формату D3 виглядає як 012 |
| E або e | Виведення значень в експоненціальному форматі, тобто у вигляді d.ddd..E+ddd або d.ddd...e+ddd. Безпосередньо після специфікатора можна задати ціле число, що визначає довжину дробової частини. Мінімальна довжина порядку - 3 символи |
| F або f | Виведення значень з фіксованою точністю. Безпосередньо після специфікатора можна задати ціле число, що визначає довжину дробової частини, наприклад, число 0,12, виведене по формату F3, виглядає як 0,120 |
| G або g | Формат загального вигляду. Застосовується для виведення значень з фіксованою точністю або в експоненціальному форматі залежно від того, який формат вимагає меншої кількості позицій. Для різних типів величин за умовчанням використовується різна ширина виводу, наприклад; для single - 7 позицій, для byte і sbyte - 3, для decimal - 29 |
| N або n | Виведення значень у форматі d,ddd,ddd.ddd..., тобто групи розрядів розділяються роздільниками, відповідними регіональним налаштуванням. Безпосередньо після специфікації можна задати ціле число, що визначає довжину дробової частини |
| P або p | Виведення числа в процентному форматі (число, помножене на 100, після якого виводиться знак % ) |
| R або r | Відміна округлення числа при перетворенні в рядок. Гарантує, що при зворотному перетворенні в значення того ж типу вийде те ж саме |
| X або x | Виведення значень в шістнадцятиричному форматі. Якщо використовується прописна буква X, то буквені символи в шістнадцятиричних символах також будуть прописними |

Приклад застосування специфікаторів:

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

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
         int i = 1234;
         Console.WriteLine(i.ToString("C"));
         Console.WriteLine(i.ToString("D5"));
         Console.WriteLine(i.ToString("E"));
         Console.WriteLine(i.ToString("G"));
         Console.WriteLine("{0,9:n2}", i);
         Console.WriteLine("{0,1:p3}", i);
         Console.WriteLine("{0,1:x}", i);


      }
   }
}
```

Результати роботи програми:

```
1 234.000р.
01234
1.234000E+003
1234
 1 234.00
123 400.000%
4d2
```