МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний економічний університет

ОСНОВИ ПРОГРАМУВАННЯ.
Конспект лекцій
для студентів спеціальності 186 «Видавництво та поліграфія» усіх форм навчання

Укладачі: Браткевич В. В.

Відповідальний за випуск Пушкар О. І.

Харків, ХНЕУ імені С. кузнеця, 2019

Оновлено та затверджено на засіданні кафедри комп'ютерних систем і технологій

Протокол №1 від 27.08.2019 р.

Рецензент – канд. техн. наук, доцент кафедри комп'ютерних систем і технологій В. П. Молчанов

Наведено матеріал, що допомагає освоїти всі лекційні теми навчальної дисципліни «Основи програмування». В якості сучасної базової мови програмування розглядається мова C# та середовище програмування Microsoft Visual Studio .NET. Виклад теорії ілюстрований великою кількістю прикладів, більша частина з яких представлена в графічному вигляді, що істотно полегшує розуміння досліджуваних питань.

Методика викладу передбачає активне залучення студента в навчальний процес. Усі приклади програм, наведені в конспекті, можуть розглядатися в якості демонстраційних при підготовці до відповідних лабораторних робіт або як базові програми, що підлягають модифікації, при виконанні індивідуальних науково-дослідних завдань.

Вступ

У конспекті лекцій розглядаються всі лекційні теми за робочою програмою з навчальної дисципліни «Основи програмування». Структурно конспект складається з двох розділів, в кожному з яких наведений матеріал відповідних тем модулів «Організація програм» і «Організація даних».

Основна увага в першій частині конспекту приділяється опису етапів розробки програмного забезпечення, особливостям процесу компіляції в середовищі .NET, а також порівняльному аналізу процедурного та об’єктно-орієнтованого стилів програмування. Докладно розглянуті лексичні елементи й основні типи даних в C#.

Підкреслюється, що, незважаючи на те, що всі C#-додатки є об’єктно-орієнтованими, однак у рамках даної дисципліни розглядаються винятково консольні додатки. Це пояснюється тим, що в них для взаємодії з користувачем застосовується одне просте вікно. Тут немає вражаючих графічних інтерфейсів і анімації, а інформація виводиться в символьному режимі. При цьому програми виходять простішими, тоді як для насичених графікою додатків складність – звичайна справа. Відомо, що комерційних програм з консольним інтерфейсом не так багато, однак їх розробка – кращий спосіб вивчення мистецтва програмування. Він дозволяє зосередитися на мовних концепціях і допомагає швидко набути розуміння мови, необхідне для створення більш складних графічних програм і, що, мабуть, найважливіше для спеціальностей напрямку 6.092, – дозволяє відносно легко адаптуватися до процесу написання «C-подібних» скриптів, широко використовуваних у різних програмних середовищах розробки засобів мультимедіа (Flash, Director і т. п.).

Модуль 1. Організація програм

1. Введення в .NET Framework-технологію програмування

1.1. Основні концепції і термінологія

Базова технологія безпосередньо пов'язана з мовою C#, має назву .NET (вимовляється як "дот нет").

.NET – це загальний термін для багатьох служб, які надаються й використовуються під час створення та виконання програми на C#.

Особливості інфраструктури .NET-платформи

C# повністю залежить від .NET і тому походження багатьох концепцій C# уходить своїми коренями в. NET.

Нижче перераховані особливості інфраструктури .NET-платформи:

.NET надає засоби для виконання інструкцій, що містяться в програмі, написаної на C#. Ця частина .NET називається середовищем виконання (**execution engine**).

.NET допомагає реалізувати так зване середовище, безпечне до невідповідності типів даних (**type safe environment**).

.NET звільняє програміста від стомлюючого процесу і такого, що нерідко веде до помилок при керуванні комп'ютерною пам'яттю, яка використовується програмою.

До складу .NET-платформи входить **бібліотека**, котра містить масу готових програмних компонентів, які можна використовувати у власних програмах. Вона заощаджує чимало часу, тому що програміст може скористатися готовими фрагментами коду. Фактично він повторно використовує код, створений та ретельно перевірений професійними програмістами Microsoft.

У .NET спрощена підготовка програми до використання (**розгортання**).

.NET забезпечує перехресну взаємодію програм, написаних на різних мовах. Будь-яка мова, підтримувана .NET, може взаємодіяти з іншими мовами цієї платформи.

У даний момент на платформу .NET перенесено близько 20 мов. Оскільки для виконання коду, написаного на будь-якій мові, що підтримується платформою .NET, використовується те саме середовище виконання, його часто називають єдиним середовищем виконання (**Common Language Runtime, CLR**).

Програма, при створенні якої була передбачена можливість повторного використання, називається компонентом (**програмним компонентом**).

1.1.1. Мови програмування та компілятори

Програмувати на перших комп'ютерах, виготовлених у сорокових роках минулого сторіччя, було дуже непросто. Для цього використовувалася **машинна мова**, що складалася з послідовностей бітів, прямо керуючих простими діями процесора. Програмування на рівні машинної мови – заняття надзвичайно трудомістке та стомлююче.

Незабаром програмісти почали шукати альтернативи машинній мові, більш близькі людській мові, щоб підвищити продуктивність своєї праці.

Першим результатом пошуків стали так звані **асемблери** – мови, в яких використовувалися більш зрозумілі людині команди, наприклад move, getint або putint. Але, незважаючи на те, що асемблери були трохи простіші для читання й розуміння, їх поєднувала з машинним кодом одна важлива загальна риса: розроблювач при написанні програми повинен був мислити в термінах низькорівневих операцій процесора і пам'яті.

Однак еволюція комп'ютерів і зростаючі потреби у все більш складних програмах привели до появи повністю **машинно-незалежних мов** програмування. Першою з них стала FORTRAN, створена в середині п'ятидесятих років. Незабаром з'явилися інші мови високого рівня. Сьогодні їхня кількість за деякими оцінками перевищує дві тисячі.

Одним із останніх доповнень у родині мов високого рівня став C#. Однак як би високо ми не відходили від базових інструкцій процесора, нам, як і раніше, потрібний машинний код, який апаратне забезпечення комп'ютера може розуміти – а виходить, й виконувати.

Традиційно перетворення вихідного коду, написаного мовою високого рівня, в машинний код здійснювали системні програми, які називаються *компіляторами*.

На рис. 1.1. наведена ілюстрація того, як вихідний код типової мови високого рівня перетворюється у програму, що виконується.

Написаний текст, який містить інструкції мови високого рівня, називається *вихідним кодом*.

У випадку C# цей вихідний код зберігається в файлі з розширенням **.cs**.

Результатом компіляції стає *програма, що виконується*, яка складається з інструкцій машинної мови.

Рис. 1.1. **Традиційний процес компіляції**

1.1.2. Компіляція в .NET вихідного коду C#

Традиційний процес компіляції вихідного коду, написаного мовою програмування високого рівня, в програму, що виконується, мав кілька недоліків. Найбільш істотними з них є наступні.

Недолік 1. Для конкретної апаратної платформи, що характеризується типом процесора й т. д., потрібен свій компілятор, оскільки в кожної з них – своя машинна мова. Відповідно, якщо потрібно виконувати програму, написану на FORTRAN, на чотирьох комп'ютерах із процесорами різних виробників, буде потрібно чотири різних компілятори FORTRAN. Більш того, щораз, коли виробник апаратного забезпечення випускає нове покоління своїх процесорів, в компілятор доводиться вносити зміни та доповнення.

Недолік 2. У більшості програмістів є улюблена мова програмування, якої вони віддають перевагу перед усіма іншими. Можливо, кращим рішенням було б дозволити кожному члену команди писати на своїй улюбленій мові, але це нелегко, якщо додержуватися процесу компіляції, розглянутому на рис. 1.1.

Різні мови на машинному рівні реалізують ту саму функціональність різними способами – частково це залежить від особливостей компіляторів. В свою чергу, це унеможливлює взаємодію різних мов між собою.

Цю проблему були покликані вирішити так звані **компонентні системи** (такі, як **CORBA** і **COM**). Вони оговорювали стандарти взаємодії між різними частинами програм.

Програміст А писав компонент X, скажімо, мовою Visual Basic, і цей компонент міг взаємодіяти з компонентом Y, що був розроблений програмістом В на C++.

Компонентні системи мали комерційний успіх. Однак їхнє поширення викликало до життя інші проблеми, однією з яких стала неможливість забезпечити взаємодію "зовнішнього" компонента з іншими частинами програми на такому ж рівні, якби всі частини програми були написані на одній мові.

В C# і .NET реалізовані рішення описаних вище двох проблем. Розглянемо всі складові процесу компіляції програми в .NET (рис. 1.2).

Насамперед, варто звернути увагу на появу на рис. 1.2 ще двох мов – C++ і Visual Basic. І поза залежністю від того, на якій мові (з підтримуваних .NET) написана програма, це ніяк не впливає на процес її компіляції в .NET.

Після того як написаний вихідний код, його потрібно відкомпілювати в машинний код. Однак спочатку він компілюється в іншу мову, що називається **Microsoft Intermediate Language** (**MSIL**). Більш того, всі компілятори, орієнтовані на .NET-платформу, повинні генерувати на виході код даної проміжної мови MSIL.

Як ясно з назви, MSIL є проміжною ланкою між мовами високого рівня (вихідний код) і машинними мовами (називаними також **природним кодом**).

Код MSIL можна швидко та ефективно транслювати в машинну мову за допомогою **JIT-компілятора** (**Just in Time-Compiler**).

Код, що генерується JIT-компілятором, нічим не відрізняється від машинного коду, що генерується звичайним компілятором, однак JIT-компілятор використовує трохи іншу стратегію. Замість того, щоб інтенсивно використовуючи пам'ять і, затрачаючи значний час, перетворити в машинний код відразу весь код MSIL, він компілює в машинний код лише ті частини додатка, які реально необхідні в даний

момент. В результаті код компілюється "на ходу", безпосередньо перед виконанням, і JIT-компілятор не витрачає час на компіляцію MSIL-коду, що не використовується.

У чому ж переваги архітектури .NET?

1. Ввівши MSIL (рис.1.2) між мовою високого рівня та машинною мовою, ми фактично відокремили ці мови одну від одної.

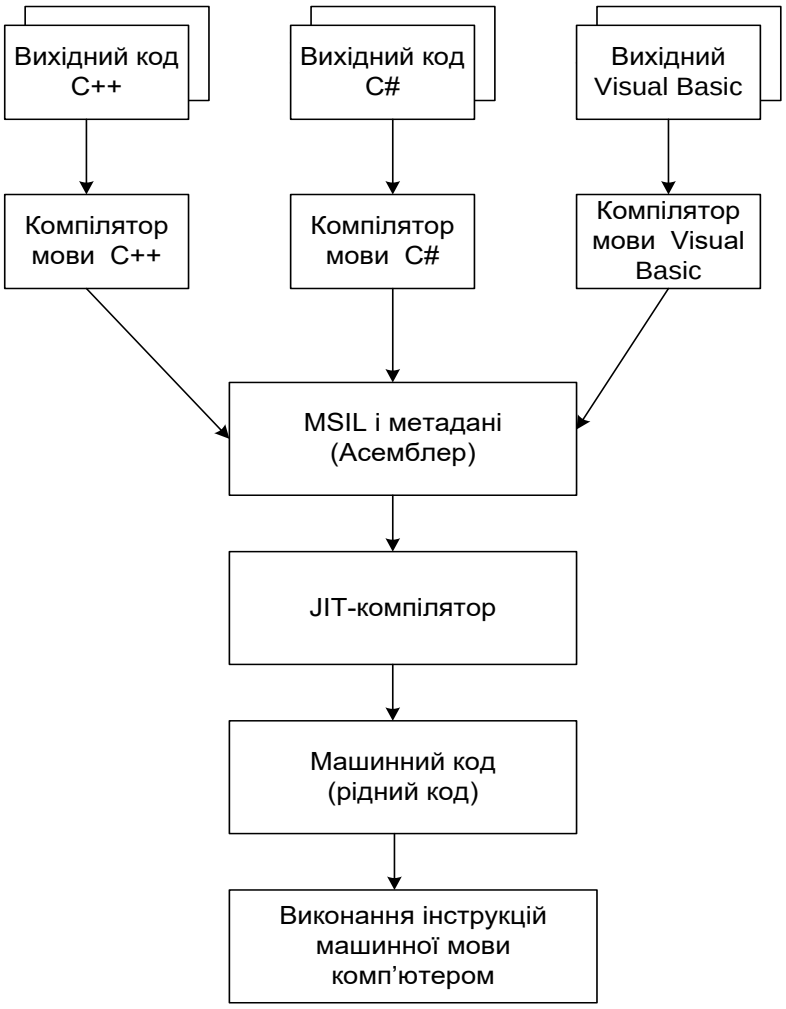


Рис. 1.2. **Процес компіляції в .NET**

Код MSIL залишається незмінним, на якій би апаратній платформі він не використовувався. Єдиним машинно-залежним елементом є JIT-компілятор, і при зміні обладнання лише він має потребу в модифікації.

На кожному комп'ютері застосовується свій JIT-компілятор, що перетворює код MSIL в машинний код, сумісний з даною конкретною конфігурацією. В результаті все, що потрібно, – це компілювати код, написаний мовою високого рівня, в універсальний код, мова якого залишається незмінною. Це і є рішенням згаданої вище першої проблеми.

Розглянемо блок «MSIL і метадані», наведений на рис. 1.2. Термін *"метадані"* можна перевести як "дані про дані". Метадані генеруються компілятором мови високого рівня і містять докладний опис елементів вихідного коду. Опис цей настільки докладний, що вихідний код інших мов зможе використовувати даний код так, якби він був написаний на тій же мові. Тепер програмісти, що пишуть на C++, C# и Basic, реально зможуть працювати в рамках одного проекту і, отже, у такий спосіб долається недолік 2.

Важливо знати про існування MSIL, але в повсякденному програмуванні стикатися з ним прямо не доводиться. Звичайно застосовуються дві команди – одна компіляції програми в MSIL-код і метадані, а інша – для виконання програми (при цьому буде викликатися JIT-компілятор). Фактично виконання програми – це виконання кінцевого результату роботи компіляторів. MSIL у цьому процесі залишається "невидимим" для користувача.

1.2. Розробка програмного забезпечення

Процес розробки програмного забезпечення (ПЗ) – це послідовність дій, кінцевим продуктом якої є комп'ютерна програма. За минулі 20 років були виділені чітко визначені етапи процесу розробки ПЗ. В даному розділі розглядаються лише найбільш важливі з них:

1. Створення специфікації програми**.** Визначення вимог до програми.

2. Проектування програми. Розробка концепції, що дозволяє втілити вимоги специфікації в працюючій програмі. Вона визначає шляхи реалізації функціональності, описаної в специфікації, засобами мови програмування високого рівня, наприклад C#.

3. Написання програми. Розробка та написання вихідного коду програми.

4. Тестування й налагодження програми. Необхідно перевірити, чи відповідає програма вимогам, визначеним у специфікації.

Ці етапи повинні виконуватися в зазначеній тут послідовності. Однак іноді виникають ситуації, коли для продовження роботи над програмою необхідне повернення до попереднього етапу.

Розглянемо кожний з етапів більш докладно.

Створення специфікації програми

Навіщо визначати вимоги до програми? По-перше, якщо чітко невідомо, що програма повинна робити, важко навіть почати думати про те, як її реалізувати. Визначення мети – перший крок до її досягнення. Часто рішення сховане в самій специфікації. (Відомо, що правильно сформульована проблема вже містить своє рішення.)

Ось приклад дуже простої специфікації програми: "Програма повинна вміти обчислювати середнє значення двох чисел".

Проектування програми

Проектування програми може включати множину різних дій залежно від розміру й характеру конкретного проекту. Великий проект може включати декілька стадій, описаних нижче:

а). розділення всього проекту на різні функціональні підсистеми. Прикладами підсистем можуть служити користувальницькі графічні інтерфейси, генератори звітів, інтерфейси до баз даних. Слід зазначити, що проекти, розглянуті далі, занадто малі, і їх не варто розбивати на підсистеми. Таким чином, наведені далі програми можна потенційно розглядати як частину функціональної підсистеми великого проекту;

б). розбивка кожної підсистеми на модулі. Термін "модуль" дуже гнучкий і приймає різні значення в різних контекстах. Тут *модуль* – це сукупність даних і функцій, які можуть працювати із цими даними. Функції, реалізовані в модулі, дозволяють йому у взаємодії з іншими модулями даної підсистеми виконувати вимоги, адресовані до неї.

Функція звичайно має одну дуже вузьку мету. Вона складається з набору конкретних інструкцій, що виконуються одна за другою. Прикладами функцій можуть бути "Знайти квадратний корінь числа" або "Знайти найбільше значення в списку". В різних високорівневих мовах цю концепцію називають по-різному: процедури, функції, підпрограми. В C# функції називаються *методами;*

в). визначення даних і методів у кожному модулі. Служби, пропоновані модулями, діляться на зручні частини, досить компактні, щоб їх можна було реалізувати в рамках одного методу. Кожній частині модуля призначається свій метод, і, відповідно, кожний метод одержує певну функціональність. І, нарешті, творці програми визначають дані, що можуть бути представлені цим модулем;

г). внутрішнє проектування методів. Проектуванням на цьому рівні звичайно займається програміст, що розробляє конкретний метод. На

даному етапі розробляються алгоритми виконання дуже вузьких, конкретних задач і пишуться перші оператори мовою високого рівня, – наприклад, C#.

Написання програми

Дана частина процесу створення ПЗ обов'язково присутня в проектах будь-якого масштабу.

Тестування й налагодження програми

Програма піддається різним тестам (як під час написання, так і після нього). Тести необхідні, щоб переконатися, що програма виконує саме те, що повинна виконувати. **Тестування**, зокрема, дозволяє виявити (і виправити) помилки, допущені в процесі проектування та написання програми.

Процес пошуку й усунення помилок називається **налагодженням** (англійський термін – debugging від bug).

Невеликі, неформальні проекти (наприклад, з лабораторного практикуму по даній дисципліні) часто включають лише рівні 3 і 4. Програміст може виконати їх, сидячи перед комп'ютером. Стадія проектування програми реалізується або в голові у програміста, або на листку паперу у формі декількох діаграм, або у вигляді пошуку декількох стандартних алгоритмів у підручнику.

1.3. Процедурно-орієнтоване програмування

Одним із традиційних підходів до проектування комп'ютерних програм був процедурно-орієнтований стиль. В ньому найвищий пріоритет належить діям, що виконує елемент програми, у той час як дані, з якими працює програма, залишаються ніби на другому плані.

Типова *процедурно-орієнтована програма* – це послідовність інструкцій, виконуваних одна за одною. В такій програмі, як правило, є численні точки розгалуження, в яких вибирається лише один із декількох можливих напрямків виконання, залежно від умов в програмі. Більша частина інструкцій маніпулює даними.

У старомодних, процедурно-орієнтованих програмах доступ до даних був можливий з будь-якої частини програми (як це показано на рис. 1.3.), а операції над ними – за допомогою будь-якої інструкції.

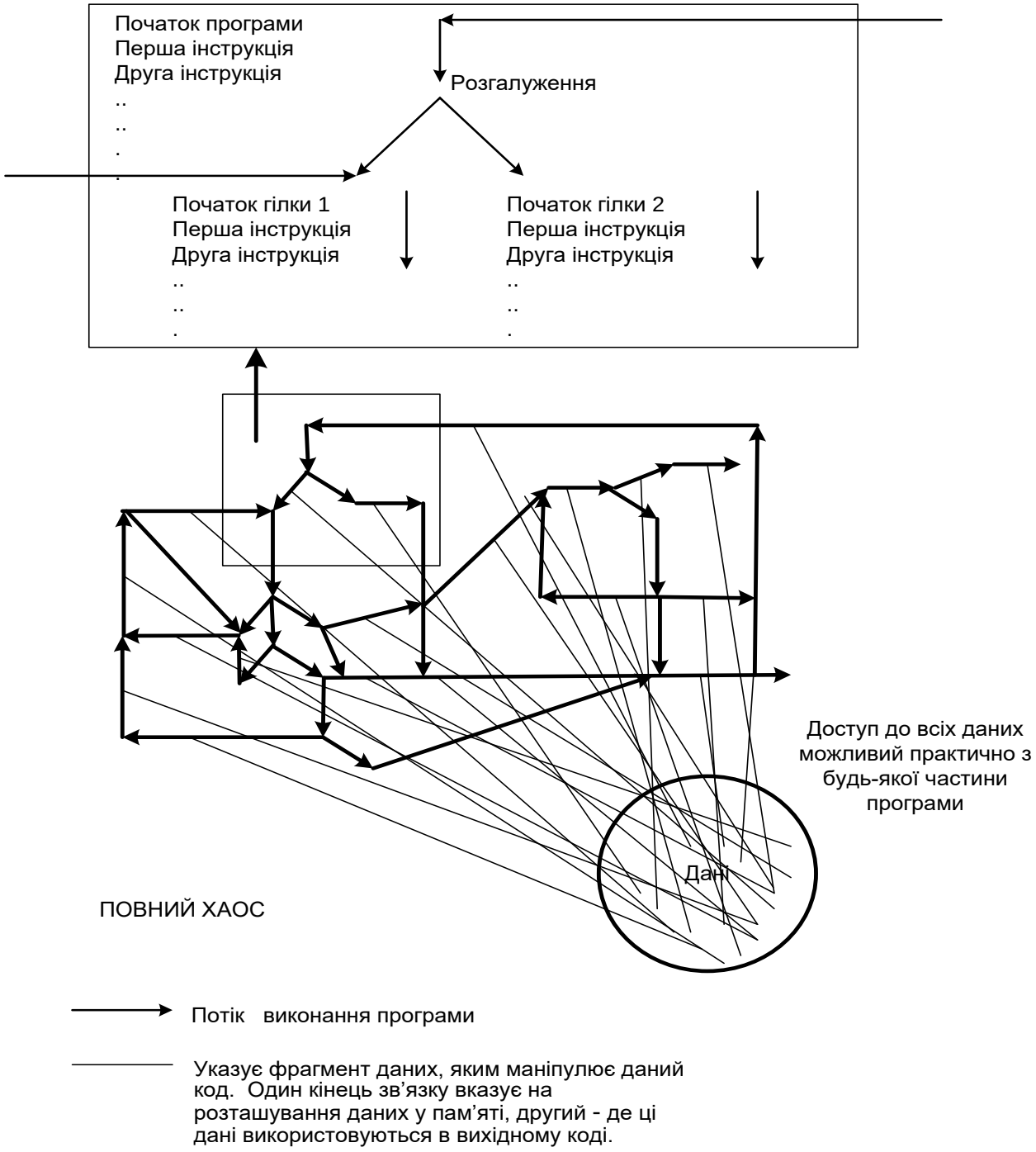


Рис. 1.3. **Структура процедурно-орієнтованої програми**

Слід зазначити, що багато з великих фрагментів вихідного коду, написаних на таких об'єктно-орієнтованих мовах, як C#, створено з використанням примітивних процедурно-орієнтованих мовних конструкцій.

Цей підхід гарний для дуже малих проектів, але при створенні великих та складних програм розроблювачеві доводиться відслідковувати всі можливі розгалуження в коді програми, оскільки всі частини її більш-менш взаємозв'язані. Гірше того, різні фрагменти коду програми можуть

одержати доступ до того ж самого значення даних – причому вони можуть не тільки читати, але й змінювати його.

При розробці однієї частини програми можна й не знати про те, що дані, з якими вона працює, можуть бути змінені іншими частинами програми.

Ситуація дуже швидко наближається до повного хаосу, коли над проектом спільно працюють декілька програмістів.

1.4. Об’єктно-орієнтоване програмування

Як же подолати розглянуту вище проблему процедурно-орієнтованого стилю програмування? Типовий людський підхід до подолання складної проблеми – розбити її на більш прості частини.

Спробуємо розбити попередній приклад на чотири менших, самостійних фрагменти. Результат показаний на рис. 1.4.

Тут весь набір інструкцій розділений на чотири окремих модулі. В об’єктно-орієнтованому світі такі модулі називаються **об'єктами**. Дані також розділені на чотири частини, так що кожний об'єкт містить лише ті, з якими він працює. Тепер при виконанні програми ці чотири об'єкти взаємодіють, **пересилаючи один одному повідомлення,** активізуючи інструкції та обмінюючись даними. Це не тільки значно знизило складність програми, але й дозволило створити чотири самодостатніх модулі, кожний з яких може бути "витягнутий" із програми й "вставлений" у неї знову.

Тепер не так уже складно добитися, щоб модулі створювалися й підтримувалися різними програмістами. Як можна бачити з рисунка, об'єкти використовуються для об'єднання даних з методами (інструкціями), що оперують над цими даними.

В об’єктно-орієнтованому світі дані і процедури мають рівне значення.

Але звідки ж знати, які об'єкти повинна містити програма? Які дані потрібні? Які інструкції необхідні?

Щоб відповісти на ці питання, насамперед необхідно з'ясувати, що таке об'єкт.

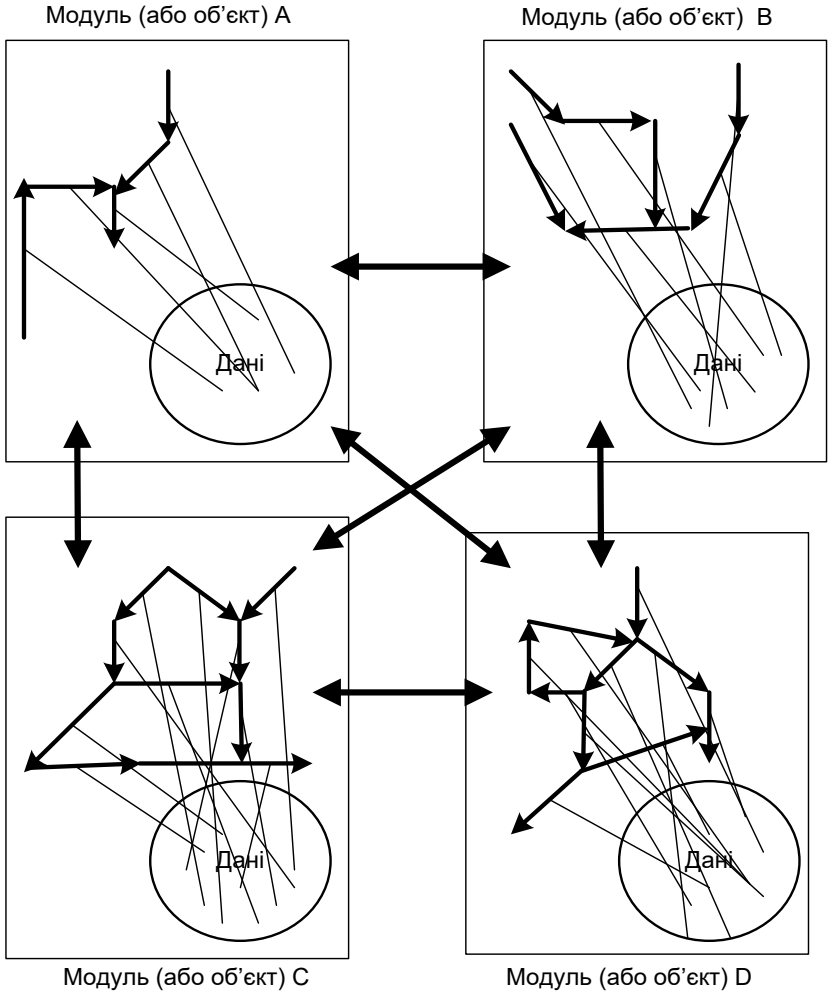


Рис. 1.4. **Розподіл процедурно-орієнтованої програми на самодостатні модулі**

Поняття об'єкта

В повсякденному житті нас оточують об'єкти: книги, будинки, машини, собаки, люди ... . Часто об'єкти здатні брати участь у певних діях. Автомобіль, наприклад, здатний "відкривати двері", "закривати двері", "запускати двигун", "рухатися вперед", "набирати швидкість", "розвертатися" та "гальмувати".

Кожний об'єкт взаємодіє зі своїм оточенням і впливає на інші об'єкти. Прикладами подібних взаємодій можуть бути об'єкт "людина", що підходить до об'єкта "автомобіль" та відкриває його двері, або об'єкт "автомобіль", який містить об'єкт "людина" та транспортує його із точки А в точку Б. При написанні програми за участю автомобілів і людей з використанням процедурно-орієнтованої методології вся увага буде приділена діям ("відкрити двері", "закрити двері" і т. д.). З іншого боку, при використанні об’єктно-орієнтованого програмування (ООП) програма буде будуватися навколо об'єктів "автомобіль" і "людина".

Таким чином, ООП – це методологія, в якій при моделюванні конкретних ситуацій використовуються об'єкти. **Комп'ютерне моделювання** намагається імітувати процеси, що відбуваються в реально існуючій або теоретичній системі. Збираючи й аналізуючи дані, отримані в цій штучній системі, можна одержати цінні відомості про внутрішні особливості системи реальної. Для успішного моделювання необхідно розробити модель та реалізувати її в комп'ютері. Часто модель є спрощенням реальної системи, однак вона цілком відповідає внутрішнім процесам і станам реальної системи, що модулюються.

Щоб заповнити пролом між реальним світом і моделюванням в комп'ютері, необхідно, насамперед, ідентифікувати об'єкти, що беруть участь у даному процесі. Звичайно, це один із перших кроків при розробці програми в ООП, і він прямо відповідає кроку б) (розбивка кожної підсистеми на модулі) процесу розробки програми, що обговорювався раніше.

Як приклад розглянемо ліфтову систему будівлі в дії. Які об'єкти в цьому беруть участь? Непоганий підхід: звертати увагу на те, які іменники використовуються при описі системи, оскільки іменники прямо відповідають об'єктам. Нижче приводиться опис системи, в якому реальні об'єкти виділені напівжирним шрифтом.

Опис системи: кілька **ліфтів** розташовано в **будівлі** з десятьма **поверхами**. Кожний ліфт має доступ до всіх десяти поверхів. Коли **людина** бажає викликати ліфт, їй необхідно нажати **кнопку** на поверсі, де вона перебуває в даний момент, і т. д.

Наступний етап відноситься до кроку в) (визначення даних і методів в кожному модулі) процесу розробки ПЗ. Він полягає у визначенні атрибутів (даних) і функцій (методів) кожного модуля (об'єкта).

Наприклад, об'єкт "Ліфт" може мати наступні атрибути: "максимальна швидкість", "поточне місце розташування", "поточна швидкість", "максимальне число людей, що може вмістити ліфт", і т. д. Функції об'єкта можуть бути наступними: "підніматися", "опускатися", "зупинитися" та "відкрити двері". Визначивши правильні атрибути і функції кожного об'єкта в реальному світі, їх можна представити в комп'ютерній моделі. Тут потрібно розширити запас термінології ООП ще декількома важливими термінами.

Кожна частина програми на C#, що представляє об’єкт реального світу, відповідно називається **об’єктом**.

Істотні атрибути об'єкта реального світу повинні бути представлені у відповідному об'єкті програми на *C#*. Ці атрибути називаються **змінними екземпляра** й відбивають поточний стан об'єкта. Змінні екземпляра еквівалентні даним на рис. 1.4.

Поведінка об'єкта реального світу представлена в об'єкті програми C# у формі методів. Кожний метод містить інструкції. Методи об'єкта виконують дії зі змінними екземпляра цього ж об'єкта. Методи еквівалентні потокам виконання на рис. 1.4.

Поняття класу

Ще один важливий термін ООП — клас. **Клас** визначає загальні риси (змінні екземпляра і методи) групи подібних один одному об'єктів. Отже, всі об'єкти одного класу мають ті ж змінні екземпляра і методи. Які змінні екземпляра і методи включати в створюваний клас, програміст вибирає сам: все залежить від потреб створюваної програми.

Як приклад класу, розглянемо автомобіль із концептуальної точки зору. В реальному житті люди використовують конкретні автомобілі. Прикладами можуть бути блакитний Volvo, що стоїть на стоянці, максимальна швидкість якого 100 миль на годину, або чорний BMW з максимальною швидкістю 150 миль на годину. Обидва ці реально існуючі відчутні автомобілі можна вважати об'єктами.

Щоб забезпечи опис конкретного автомобілю як об'єкта програми на C#, програміст приймає рішення про включення чотирьох змінних екземпляра в клас **Автомобіль: Марка, Поточне місце розташування, Максимальна швидкість** і **Поточна швидкість**. Слід зазначити, що атрибут **Колір** (Color) не включений у клас, оскільки програміст вважав його непотрібним для даної програми.

Далі програміст вирішує ввести до складу класу методи **Відкрити двері, Закрити двері, Рухатися вперед, Рухатися назад, Прискорюватися, Гальмувати** та **Повертати.**

Кожний екземпляр класу **Автомобіль** можна розглядати як порожню коробку, що має бути наповнена вмістом у конкретному об'єкті.

Вміст кожної змінної класу **Автомобіль** може в різних об'єктах мати як однакові, так і різні значення, але кожний метод, визначений класом **Автомобіль**, ідентичний у всіх об'єктах **Автомобіль**.

Концепції об'єкта і класу представлені на настільки ранній стадії вивчення C#, оскільки будь-яка, навіть найпростіша, C#-програма є об’єктно-орієнтованою.

1.5. Повторне використання програмного забезпечення

Повторне використання програмного коду закладене в основу програмування в .NET і C#. На ранній стадії написання кожна програма представляється зовсім не залежним проектом, розроблювальним з нуля. Однак це не зовсім раціональний спосіб створення програм, і сьогодні більшість із них створюється за іншою методологією, основу якої складає повторне використання заздалегідь розроблених і налагоджених програм або їхніх частин.

Високий ступінь повторного використання коду означає, що при створенні програми розроблювач написав тільки частину коду, а інша частина – це вставлені в програму компоненти, написані та налагоджені досвідченими програмістами. Навіть в найпростіших програмах на C# варто завжди дотримуватися концепції повторного використання коду.

Одна з найбільш привабливих сторін об'єктно-орієнтованих мов (в тому числі C#) – це ретельно продумана підтримки повторного використання коду. Зокрема, **class** виявився дуже гарним для повторного використання коду. Елементом повторного використання коду в .NET є також складання (**assembly**). Будь-яка програма в .NET і C# складається з одного або більше складань.

Складання – це логічний пакет, що містить свій опис. Він складається з коду MSIL, метаданих і, якщо необхідно, ресурсів, наприклад зображень.

Складанням є будь-яка програма, написана для .NET, будь то компонент для повторного використання або самодостатня программа, що виконується.

Складання можна розглядати з двох точок зору. З погляду розроблювачів складання, що розглядають його зсередини, на рівні вихідного коду і з погляду користувачів складання, що розглядають його зовні, коли потрібно підібрати підходящий компонент для повторного використання в тім або іншім проекті.

1.6. Бібліотека класів .NET Framework

Протягом багатьох років у різних типах програм постійно використовувалися ті самі функції та алгоритми. Прикладом може служити сортування списку, спеціалізовані інженерні розрахунки,

математичні обчислення й т. д. Цей факт усвідомило багато компаній і розроблювачів ПЗ, що почали створювати бібліотеки класів, в яких містилися подібні широко використовувані функції. Зараз важко знайти додаток, в якому б не використовувалися частини з повторно використовуваних бібліотек класів.

Компанія Microsoft включила до складу середовища .NET бібліотеку класів. Вона називається бібліотекою класів .NET Framework, або бібліотекою базових класів (**Base Class Library, BCL**). Бібліотека містить сотні класів і надає доступ до множини функцій.

В числі основних механізмів, використовуваних в BCL, – принципи ООП, технологія складань і пов'язані з нею концепції. В результаті BCL проста у використанні і забезпечує широкі можливості повторного використання коду.

У мові C# відсутня власна бібліотека класів – він повністю покладається на BCL і тісно інтегрований з нею. Відповідно, програму, написану на C#, неможливо запустити без BCL та середовища виконання .NET.

В основі всіх класів, написаних в C#, лежить один конкретный класс BCL, і багато конструкцій C# є лише представленням тих класів і їхніх функцій, що містяться в BCL. Наявність BCL значно спрощує доступ до служб операційної системи. Замість того, щоб використовувати малозрозумілі команди і складні вираження, служби ОС стають доступними у формі, набагато більш дружній користувачеві. Прикладом може служити надавана BCL підтримка віконних графічних інтерфейсів користувача.

1.7. Можливості мови C#

Як уже відмічалося вище, в теперішній час існує множина мов програмування. І, незважаючи на те, що будь-яку програму можна написати на практично будь-якій мові, багато мов було пристосовано для рішення більш-менш конкретного кола проблем.

C# и .NET працюють винятково в операційних системах Microsoft, але при цьому вони надають програмістові в рамках цих операційних систем дуже потужні засоби. Важливо, що C# – багатоцільова мова, і в рамках платформи Windows він може використовуватися для створення найрізноманітніших програм.

Нижче наведено короткий список лише декількох категорій програм, при створенні яких комбінація достоїнств C# і .NET забезпечить високу ефективність праці програміста C#.

Консольні додатки

У консольних додатках для взаємодії з користувачем застосовується одне просте вікно. В них немає вражаючих графічних інтерфейсів і анімації; інформація виводиться в символьному режимі. При цьому програми виходять більш простими, тоді як для насичених графікою додатків складність – звичайна справа.

Незважаючи на те, що комерційних програм з консольним інтерфейсом не так багато, створення їх – чудовий спосіб вивчення мистецтва програмування. Він дозволяє зосередитися на мовних концепціях і допомагає швидко набути розуміння мови, необхідне для створення більш складних графічних програм і, що, мабуть, найважливіше для спеціальностей напрямку 6.092, – дозволяє відносно легко адаптуватися до процесу написання «C-подібних» скриптів, широко використовуваних у різних програмних середовищах розробки засобів мультимедіа (Flash, Director і т. п.). Саме з цієї причини всі додатки, що вивчаються в рамках дисципліни «Основи програмування», є консольними.

Однак програмування для консолі аж ніяк не є уділом лише новачків. Професійні програмісти часто використовують вікно консолі для тестування додатків і компонентів.

Віконні додатки на базі WinForms

На відміну від консольних в них застосовується графічний користувальницький інтерфейс користувача, де команди користувача передаються шляхом клацань миші на екранних кнопках і піктограмах, а введення інформації здійснюється через різні текстові вікна і вікна списків.

Щоб написати віконний додаток з нуля, не використовуючи ніяких готових компонентів, доведеться витратити на розробку місяці й навіть роки. Бібліотека класів .NET Framework містить великий набір компонентів за назвою **WinForms**. Ці компоненти застосовуються для створення складних віконних додатків. **WinForms** забезпечує програмісту на C# простий доступ до віконних служб операційної системи Windows.

Додатки ASP.NET

ASP.NET (Active Server Pages .NET) – це група компонентів, використовуваних для спрощення створення додатків на базі браузера.

Браузер – це додаток, що дозволяє користувачеві в зручній формі переглядати документи в форматі HTML (HyperText Marcup Language). Браузери широко використовуються для відображення інформації в Internet.

Web-служби

Web-служби – це важлива частина нової технології, що обіцяє змінити шляхи використання Internet, а з ними – представлення про те, як розробляти додатки й використовувати їх. Web-служби представляють просто компоненти або додатки, доставлені на комп'ютер з Internet. З Web-служб, розміщених на різних комп'ютерах, зв'язаних з Internet, можуть бути сформовані нові Web-служби.

Це відкриває для програмістів різні цікаві можливості. Зокрема, з'являється можливість збирати компоненти і додатки не тільки з класів .NET Framework і вихідного коду, написаного самим програмістом, але й з Web-служб, знайдених в Internet.

Питання для самоконтролю

1. Що означає .NET? У чому особливість .NET-платформи?
2. Перерахуйте основні етапи одержання программи, що виконується.
3. Навіщо необхідний етап компіляції?
4. У чому особливість компіляції в .NET вихідного коду C#?
5. У чому суть процедурно-орієнтованого стилю програмування? Яка структура процедурно-орієнтованої програми?
6. Укажіть недоліки процедурного стилю програмування та шляхи їхнього подолання.
7. Дайте визначення об'єкта і наведіть приклад з описом його властивостей і поводження.
8. Що таке клас?
9. Навіщо необхідно повторне використання програмного забезпечення? Приведіть приклади повторного використання.
10. Призначення бібліотеки класів .NET Framework.
11. Опишіть можливі типи C#-додатків і області їхнього застосування.

2. Основні типи даних C#

2.1. Лексичні елементи мови (термінологія)

Всяка алгоритмічна мова містить три складові частини: **алфавіт** (кінцева множина відмінних між собою символів, використовуваних у даній мові); **синтаксис** (сукупність правил, що визначають припустимі (правильні) конструкції даної мови. Синтаксис мови C# визначений у спеціальній граматиці); **семантика** (сукупність правил, що визначають значеннєвий зміст окремих конструкцій. Семантика забезпечує однозначність тлумачення всіх понять мови).

Алфавіт – це символи, що використовуються в мові C# при написанні программ. Кожний файл – це текст. Для запису програм використовуються знаки у відповідному кодуванні. Російські букви можна використовувати в коментарях і літералах.

Коментарі. Будь-який текст, починаючи із двох знаків ділення \\ і до кінця рядка є коментарем, ніяк не аналізується комп'ютером і служить лише для пояснень. Крім того, будь-який текст, розміщений між символами /* і */ також є коментарем. Три знаки \\\ також є ознакою коментарю, що може бути використаний при компіляції програми для виділення фрагментів документації до програми у форматі XML.

Ідентифікатори. Послідовність символів з латинських букв, символів підкреслення і арабських цифр, що починається з букви та служить для іменування різних елементів програми.

Програма – це запис алгоритму на одній із мов програмування. Програма містить розділ команд і розділ опису даних.

Дані – це формалізоване представлення всіх тих об'єктів (предметів, фактів, ідей), з якими може оперувати ПК. Включають в себе змінні та константи. Перш ніж задавати в програмі дії над даними, змінні та константи повинні бути визначені.

Змінна – символічне позначення величини в програмі. З погляду архітектури ПК, змінна – це символічне позначення комірки ОП, в якій зберігаються дані. Безпосередньо записати величину в програмі можна за допомогою літерної константи (як константа використовуються символи відповідного коду).

Вираження – це послідовність операндів, знаків операцій, круглих дужок, що задає обчислювальний процес одержання результату певного типу.

Операнд – це елемент-учасник операції. Операндами можуть бути: *константи* (це лексема, що представляє зображення фіксованого числового, строкового або символьного (літерного) значення); *змінні*; *виклики функцій* – вказівка ім'я викликуваної функції, за яким у круглих дужках вказується список аргументів (можливо, порожній). Під час виконання програми результат, що повертається викликаною функцією, заміняє виклик функції; вираження.

Приклади перерахованих вище лексичних елементів C# розглянемо, аналізуючи базову структуру C#-програми.

2.2. Базова структура C#-програми

Лістинг, наведений нижче, незважаючи на свою простоту, містить основні компоненти типової програми на C#.

Лістинг 2.1.

```
01:// Проста програма на C#
02:class Hello
03:{
04:    // Програма починається з виклику методу Main()
05:    public static void Main()
06:    {
07:            string answer;
08:
09:            System.Console.WriteLine("Do you want me to write the two words?");
10:            System.Console.WriteLine("Type y for yes; n for no. Then <enter>");
11:            answer = System.Console.ReadLine();
12:            if (answer == "y")
13:                    System.Console.WriteLine("Hello World!");
14:            System.Console.WriteLine("Bye Bye!");
15:    }
16:}
```

Короткий аналіз кожного рядка лістингу 2.1 наведений на лістингу 2.2 (рядки двох лістингів точно відповідають один одному).

Лістинг 2.2. Аналіз вихідного коду Hello.cs

01:Коментар: Проста програма на C#

02:Початок визначення класу Hello

03:Початок блоку класу Hello

04:Коментар: Програма починається з виклику методу Main( )

05:Початок визначення методу Main()

06:Початок блоку методу Main()

07:Оголошення змінної answer для зберігання тексту

08:Порожній рядок

09:Вивести: Do you want me to write the two words? Перейти на рядок нижче

10:Вивести: Type y for yes; n for no. Then <enter> Перейти на рядок нижче

11:Зберегти відповідь користувача в змінній answer. Перейти на рядок нижче.

12,13:Якщо в answer зберігається 'y', вивести: Hello World!

Якщо в answer не зберігається 'y,' пропустити рядок 13 і продовжити виконання з рядка 14.

14:Вивести: Bye Bye! Перейти на рядок нижче.

15:Кінець блоку методу Main()

16:Кінець блоку класу Hello

Приклад виведення 1, коли користувач відповідає y (yes):

Do you want me to write the two words?

Type y for yes; n for no. Then <enter>

y <enter>

Hello World!

Bye Bye!

Приклад виведення 2, коли користувач відповідає n (no):

Do you want me to write the two words?

Type y for yes; n for no. Then <enter>

n <enter>

Bye Bye!

Розглянута програма містить важливі компоненти типової програми на C#. Розглянемо кожну частину програми докладно. Номера рядків відповідають номерам рядків з лістингу 2.1.

Коментарі

Рядок 1 містить коментар, вміст якого ігнорується компілятором. Він використовується для опису дій, які виконує програма. В даному випадку він просто повідомляє про те, що програма написана на C#.

01: // Проста програма на C#

Подвійний символ косої риски (//) змушує компілятор ігнорувати текст до кінця рядка. Рядок 1 містить тільки коментар, однак останній можна розмістити й у рядку з кодом. Рядки 1 та 2 можна об'єднати в такий спосіб:

class Hello // Проста програма на C#

Інший варіант коду некоректний:

// Проста програма на C# class Hello

тому що весь рядок, включаючи й class Hello, розглядається компілятором як коментар.

Визначення класу

Для пояснення рядка 2 необхідно звернутися до концепції ключового, або зарезервованого, слова. Ключове слово має спеціальне значення в мові C# і розпізнається компілятором.

У рядку 2 для визначення класу використовується ключове слово class.

02: class Hello

Hello – це ім'я класу, що розташовується безпосередньо за class.

У лістингу 2.1 представлено кілька ключових слів: class, public, static, void, string та if. Ключові слова мають для компілятора спеціальне значення. Їх не можна використовува для інших цілей в C# (на що вказує термін "зарезервовані"). Слід зазначити, що ключове слово може бути частиною ім'я, тому назва classVariable цілком коректна.

Ідентифікатори (імена)

Імена у вихідному коді часто називають ідентифікаторами. Багато елементів – класи, об'єкти, методи, змінні екземпляра – повинні завжди мати ідентифікатори. На відміну від ключових слов C# вибір усіх ідентифікаторів залишається за програмістом. Тут існує декілька правил.

Ідентифікатор може складатися тільки з букв, цифр (0 – 9) і символу підкреслення (_). Ідентифікатор не може починатися з цифри та збігатися з одним із ключових слів.

Приклади припустимих ідентифікаторів:

Elevator

_elevator
My2Elevators
My_Elevator
MyElevalor

Приклади неприпустимих ідентифікаторів:
Ele vator
6Elevators

В C# враховується регістр, тому прописні та малі літери вважаються різними символами.

Фігурні дужки та блоки вихідного коду

Рядок 3 містить фігурну дужку ({), що вказує на початок блоку.

Блок – це фрагмент вихідного коду C#, поміщений у фігурні дужки. Блок є логічною одиницею коду. Фігурні дужки завжди застосовуються в парах. Коли в коді зустрічається {, це значить, що десь далі обов'язково знаходиться }, що їй відповідає. Дужка }, що відповідає рядку 3,

03: {

знаходиться в рядку 16. Ще одна пара фігурних дужок перебуває в рядках 6 та 15.

Оскільки { в 3 розташована відразу після визначення в рядку 2, компілятор знає, що визначення всього класу Hello міститься між { в рядку 3 і } в рядку 16.

У блоці визначення класу (рис. 2.1) тепер можна розмістити методи та змінні екземпляра за умови, що всі оголошення перебувають усередині блоку.

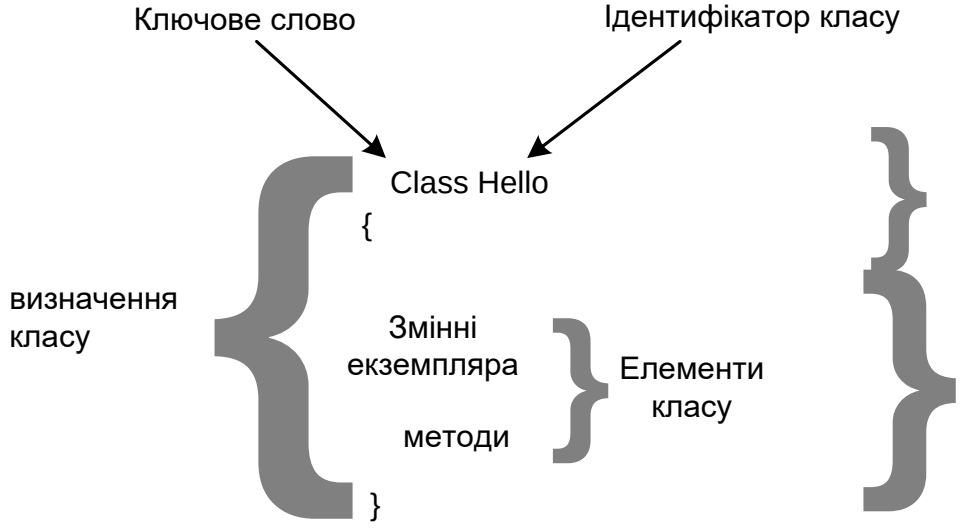


Рис. 2.1. **Визначення класу**

У рядку 4

04: // Програма починається з виклику методу Main()

знаходиться вже знайомий символ коментарю //.

Метод Main() та його визначення

У рядку 5 починається визначення методу під назвою Main. В C# немає ключового слова на зразок "method", що вказує на те, що конструкція є методом. Компілятор розпізнає метод за круглими дужками, наступними за його ім'ям, зокрема, ( ) після Main.

05: public static void Main( )

Метод Main має в C# спеціальне значення. З цього методу починає виконання кожний додаток на C# – він викликається середовищем виконання при запуску програми.

Наприклад, складний додаток для робіт із електронними таблицями, написаний на C#, может містити тисячі методів з різними ідентифікаторами, але тільки метод Main викликається середовищем виконання .NET при запуску програми.

Точне значення всіх елементів рядка 5 поки що не буде обговорюватися, оскільки вимагає більш детального розуміння певних об’єктно-орієнтованих принципів C#. Отже, клас складається з інтерфейсу, реалізованого за допомогою відкритих методів і схованої частини, що складається з закритих методів і змінних екземпляра.

Ключове слово public в рядку 5 є специфікатором доступу, воно дозволяє управляти видимістю елемента класу. В даному випадку (перед методом Main) воно вказує, що Main є відкритим методом і, таким чином, частиною інтерфейсу класу Hello. В результаті метод Main можна викликати ззовні об'єкта Hello. Основні елементи визначення методу ілюструються на рис. 2.2.

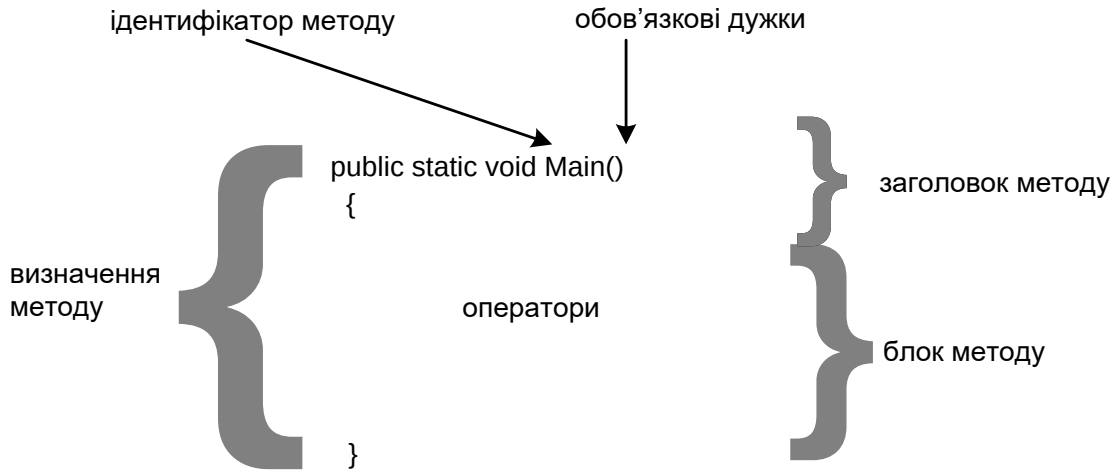


Рис. 2.2. **Визначення методу**

Кожна програма на C# повинна містити метод Main(). При її запуску середовище виконання .NET, у першу чергу, шукає цей метод. Якщо він знайдений, з нього починається виконання, якщо ні – виводиться повідомлення про помилку.

Main() на лістингу 2.1 розташований усередині класу Hello, а середовище виконання -.NET – зовні. При спробі запуску Main() середовище буде розглядатися як ще один об'єкт, що запитує доступ до методу класу. Тому його необхідно відкрити, зробивши частиною інтерфейсу класу. Щоб середовище .NET могло отримати доступ до Main(), його необхідно завжди оголошувати як public. Звичайно Main() викликає методи інших об'єктів, але в цьому простому прикладі є тільки один клас з одним методом.

Для початкового розгляду ключового слова static звернемося знову до обговорення розходжень між класом та об'єктом.

Клас є специфікацією того, як створити об'єкт, так само, як креслення є просто планом реального будинку. Клас звичайно не може робити будь-яких дій. Ключове слово static дозволяє відійти від цієї схеми і скористатися методами класу, не створюючи конкретного екземпляра об'єкта.

Коли static включено в заголовок методу, це повідомляє клас про те, що для використання методу не потрібно створювати екземплярів за межами класу. Таким чином, метод Main() може використовуватися до створення певного об'єкта класу Hello. В даному випадку це обов'язково, тому що Main() викликається середовищем виконання .NET до того, як створюються які-небудь об'єкти.

Щоб зрозуміти значення ключового слова void в рядку 5, необхідно звернутися безпосередньо до того, як працюють методи. У цьому розділі буде наведене лише коротке пояснення: void означає, що Main() не повертає значення в точку виклику.

У рядку 6 дужка { вказує на початок блоку Main(), де міститься тіло методу. Блок закінчується дужкою } в рядку 15.

06: {

Для поліпшення читаності коду варто вибирати значущі імена змінних і уникати абревіатур.

Змінні

Змінна є іменованою позицією в пам'яті, що представляє збережений блок даних. Ключове слово string вказує, що answer належить типу string

07: string answer;

Ідентифікатор (answer) програміст вибирає за своїм розсудом, а string є зарезервованим словом.

Розміщення answer після string в рядку 7 означає, що оголошена змінна answer типу string.

Кожна змінна, що використовується в програмі на C#, повинна бути оголошена.

Змінна answer застосовується в рядках 11 та 12.

Змінна типу string може містити текст. "Привіт!", "Buenos dias!", “Julian is a boy”, "y", "n" є прикладами тексту, що може зберігатися в answer.

В C# рядки тексту позначаються " " (подвійні лапки). Складові частини визначення змінної показані на рис. 2.3.

З рисунку видно, що змінна складається з трьох елементів:

ідентифікатора (в даному випадку, answer);

типу, тобто виду інформації, що вона може зберігати (в даному випадку – string, тобто послідовність символів);

значення, тобто збереженої інформації. Поточне значення на рисунку дорівнює "Julian is a boy".

Рис. 2.3. **Тип, ідентифікатор та значення змінної**

Будь-яку задачу, що виконується програмою на C#, можно розбити на послідовність інструкцій. Найпростіша інструкція називається оператором. Усі оператори закінчуються символом крапки з комою.

Рядок 7 містить оператор оголошення змінної, тому він, як і інші, закінчується крапкою з комою.

Рядок 8 є порожнім. Компілятор C# ігнорує пусті рядки. Однак вони можуть бути вставлені у вихідний код для поліпшення його читаності.

Запуск методів .NET-платформи

Оператор в рядку 9

09: System.Console.WriteLine("Do you want me to write the two words?");

змушує програму вивести па екран наступне:

Do you want me to write the two words?

На даний момент досить розглядати виклик System. Console.WriteLine( ) як просто спосіб виведення, що має сенс: "вивести все, що міститься в дужках після WriteLine на екран та перейти на один рядок нижче".

Ось що коротенько відбувається в рядку 9.

System.Console – це клас .NET Framework. .NET Framework є бібліотекою, яка містить множину корисних класів, створених розроблювачами з Microsoft. Таким чином, для виведення тексту на екран повторно використовується клас System.Console. Він містить метод WriteLine( ), що й викликається командою System.Console.WriteLine( ).

Коли метод виконує певну задачу в програмі, це називають викликом. Елемент усередині круглих дужок (текст "Do you want me to write the famous words?" в прикладі) називається аргументом. Аргумент містить інформацію, необхідну викликуваному методу для виконання завдання. Аргумент передається методу WriteLine при виклику. Після цього метод звертається до даних вже за допомогою своїх внутрішніх операторів. Рядок 9, як і 7, містить оператор і тому закінчується крапкою з комою.

Розглянемо, як саме в рядку 9 використовується метод класу System.Console. Як уже підкреслювалося, класи є "схемами", а об'єкти – "виконавцями". Метод класу можна використовувати в тому випадку, коли в його оголошенні присутнє ключове слово static, яке згадувалося раніше. Таким чином, метод WriteLine доступний без створення конкретного екземпляра об'єкта System.Console.

Загальний механізм виклику методу

Інструкції методу містяться усередині його визначення у формі операторів. Викликати метод – означає виконати його інструкції. Виконання відбуваються послідовно в тім порядку, в якому вони написані у вихідному коді.

Метод можна визначити тільки усередині класу. Він є дією, яку здатен виконати об'єкт.

Виклик методу має наступний синтаксис: ім'я об'єкта (або класу, якщо метод оголошений як static), точка, ім'я методу та завершальна пара круглих дужок ( ), в яких можуть міститися аргументи. Останні представляють собою дані, передані методу.

Виклик нестатичного методу:

ІмяОб'єкта.ІмяМетоду(Необов'язкові_аргументи)

Виклик статичного методу:

ІмяКласу.ІмяМетоду (Необов'язкові_аргументи)

Замінивши загальні елементи реальними іменами, легко отримати оператор з рядка 14 лістингу 2.1.

Syslem.Console.WriteLine("Bye Bye!");

По завершенні методу потік керування вертається в точку, з якої відбувся виклик (див. рис. 2.4).

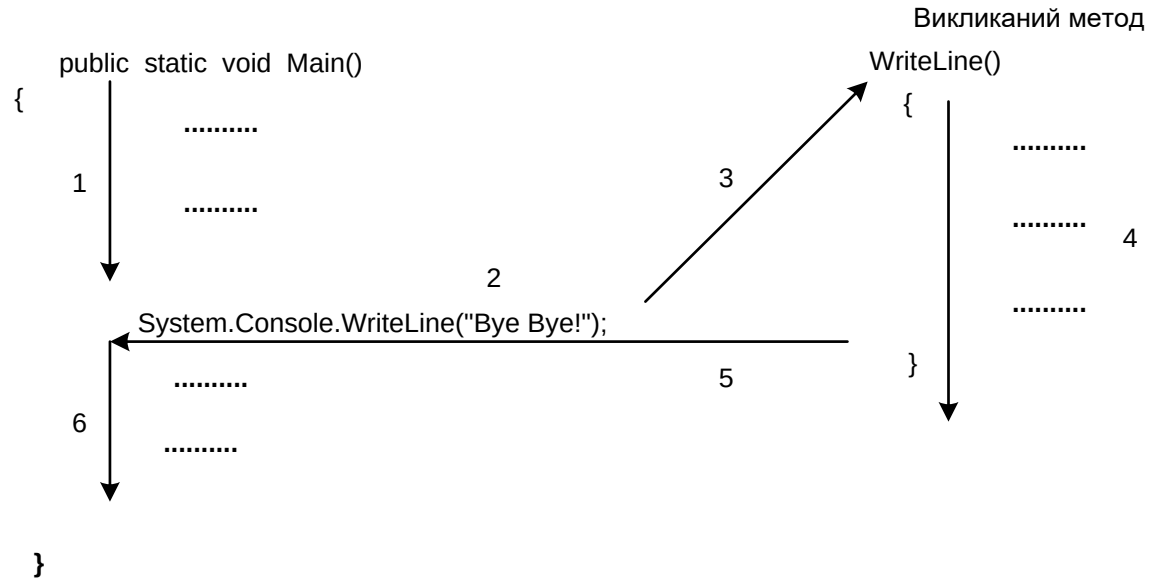


Рис. 2.4. **Рух потоку виконання програми при виклику метода**

На рисунку можна виділити наступні кроки:

1. Виконати оператори в рядках, що передують 14.
2. Запустити рядок 14.

3. Викликати System.Console.WriteLine з аргументом "Bye Bye!".

4. Виконати оператори усередині System.Console WriteLine (...).

5. Повернути керування операторові в рядку після 14.

6. Виконати інші оператори методу Main.

У рядку 10 міститься ще один виклик WriteLine

10: System.Console.WriteLine(“Type y for yes; n for no. Then <enler>");

У результаті на екран виводиться рядок:

Type y for yes; n for no. Then <enter>

і курсор перемішається на один рядок нижче.

Повідомлення

Розглянемо рядок 9 лістингу 2.1. У ньому міститься оператор, що викликає метод WriteLine. В об’єктно-орієнтованому програмуванні для позначення такого виклику часто застосовується ще один термін – “повідомлення”.

Коли метод об'єкта А містить оператор, що викликає метод об'єкта В, говорять, що А посилає повідомлення В. В рядку 10 клас Hello посилає повідомлення класу System.Console. Повідомленням є

WrileLine(“Type y for yes; n for no. Then <enter>");

Загальна схема ООП припускає, що об'єкти виконують дії, які запускаються при одержанні повідомлень. У розглянутому прикладі дією є виведення на консоль

Type y for yes; n for no. Then <enter>

Присвоювання значення змінної

У рядку 11 знову повторно використовується клас System.Console. Цього разу застосовується інший з його статичних методів – ReadLine, що призупиняє виконання програми, очікуючи введення від користувача. Відповіддю може бути введений текст, який завершується натисканням клавіші Enter. Як виходить із назви, метод ReadLine читає введення:

11: answer = System.Console.ReadLine();

При натисканні Enter текст, введений користувачем, зберігається в змінній answer. Очевидно, коли користувач вводить 'y', answer містить "y", коли 'n' , – "n" і т. д. За це відповідає знак рівності (=), розташований після answer.

В C# знак рівності (=) використовується трохи інакше, ніж у стандартній арифметиці.

Механізм завдання нового значення змінній answer називається присвоюванням. Говорять, що введений текст присвоюється змінній

answer. Загальне вираження в рядку називають оператором присвоювання, а сам знак рівності (=) називають операцією присвоювання (в даному контексті). Якщо знак “рівність” використовується в інших контекстах, він має інші назви.

Розгалуження за допомогою оператора if

Одне із застосувань знака «рівність» показано в рядку 11. В рядку 12 він використовується в зовсім іншому контексті: цього разу, – в стандартному арифметичному:

```
12:    if (answer == "y")
13:                    System.Console.WriteLine("Hello World”);
```

У C# два послідовних знака рівності (==) позначають операцію рівності, використовувану для порівняння виражень ліворуч і праворуч від нього.

У рядку 12 запитується: answer == "y" ?, тобто "Чи дорівнює значення answer "y" ?" Відповіддю може бути істина (true) або неправда (false).

Вираження, що може приймати тільки одне із двох значень (true або false), називається логічним.

Ключове слово з наступним логічним вираженням answer == "y", розміщеним у дужках, має наступний сенс: тільки якщо answer == "y" дорівнює true (істинно), потрібно виконати оператор в рядку, що є наступним за 12 (тобто (13)).

Якщо ж answer == "y", дорівнює false (неправда), потік виконання повинен перейти до рядка 14. На рис. 2.5 за допомогою стрілок показаний хід потоку виконання в рядках 12 – 14.

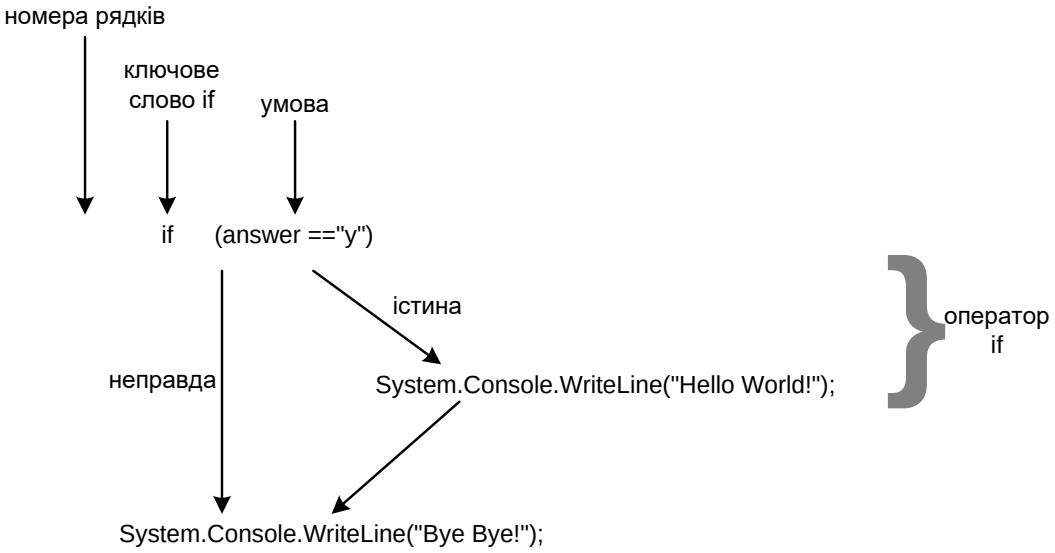


Рис. 2.5. **Послідовність виконання програми з оператором if**

Незалежно від того, дорівнює answer y або n, програма виводить наприкінці рядок "Bye Bye!".

Тільки answer рівне “y” змушує програму вивести Hello World!. Будь-яка інша відповідь користувача робить логічне вираження рівним false, в результаті чого рядок 13 не виконується.

У рядках 12 та 13 міститься оператор if. Він керує потоком виконання, дозволяючи вибрати два різних напрямки. Такі оператори, як if, називаються операторами розгалуження.

Завершення методу Main() і класу Hello

Дужка } в рядку 15 завершує блок методу Main(), початий в рядку 6.

15: }

У рядку 16 дужка } завершує блок класу Hello.

16: }

Формат вихідного коду C#

Порожні рядки, символи пробілу, табуляції та повернення каретки називають єдиним терміном "порожній символ". Компілятор C# ігнорує їх. Тому всі ці символи можна використовувати еквівалентно.

Неподільні елементи в рядку вихідного коду називаються лексемами. Вони повинні відділятися один від одного порожніми символами, комами або крапками з комою. Самі лексеми розділяти порожніми або іншими символами не можна. Лексема (token) – це слово, що має певне значення. Цей термін часто використовується в логіці та лінгвістиці. Спроба розриву лексем приводить до некоректного коду.

Хоч C# надає певну свободу у форматуванні коду, гарний стиль може значно поліпшити його читаність. Стиль, наведений у лістингу 2.1, прийнятий більшістю програмістів.

2.3. Поняття типу даних

2.3.1. Концепція типу даних

Кожний конкретний тип даних визначається двома факторами: множиною значень, які можуть приймати об'єкти даного типу; набором операцій, що можна застосовувати до даного типу.

В описі даних повинна міститися (для компілятора) наступна інформація, що задається типом даних:

ім'я змінної або константи;

розмір пам'яті, необхідної для зберігання значень;

які дії можна виконувати зі змінною або константою;

вид та спосіб виділення пам'яті;

початкове значення змінної або значення константи.

C# є жорстко типізованою мовою. При його використанні програміст повинен оголошувати тип кожного об'єкта, що він створює (наприклад, цілі числа, числа із плаваючою точкою, рядки, вікна, кнопки і т. д.).

Класифікація типів даних

C# підрозділяє типи на два види: вбудовані типи (або прості типи), які визначені в мові, і типи, визначені користувачем (типи, які вибирає програміст).

C# також підрозділяє типи на дві інші категорії: типи-значення (або розмірні) та посилальні типи.

Основна відмінність між ними – це спосіб, яким їхні значення зберігаються в пам'яті.

Змінна типу-значення містить значення, збережене безпосередньо в ній (тобто у відповідних комірках пам'яті комп'ютера). Прикладом може служити тип int. Механізм оголошення змінної myNumber типу int і присвоювання їй літерала 345 можна зобразити, як показано на рис. 2.6.

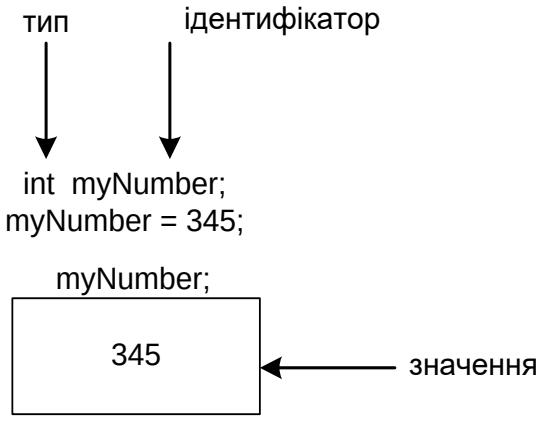


Рис. 2.6. **Тип int є типом-значенням**

Змінна посилального типу містить в пам'яті посилання на об'єкт, а не сам об'єкт безпосередньо.

Посилання становить позицію (адресу) об'єкта в пам'яті. Для ілюстрації звернемося до вже вивченого типу string (який, як з'ясується надалі, є посилальним). Змінна типу string не містить рядок, а оголошується для зберігання посилання на рядок. Сам рядок розміщується за визначеною адресою в пам'яті.

Розглянемо наступний фрагмент коду:

string myText; // оголошення змінної myText типу string

Цей оператор оголошення можна передати словами так: "Нехай myText містить посилання на рядок".

Після цього myText можна застосувати в наступному операторі присвоювання:

myText = "Lets go to the C to catch a #";

В результаті чого змінної myText присвоюється адреса рядка "Lets go to the C to catch a #". Однак в комірках пам'яті, де міститься змінна myText, немає ніякого тексту, там міститься тільки адреса пам'яті, звана також посиланням або покажчиком.

Сказане ілюструє рис. 2.7, на якому текст розміщений за довільною адресою 4027, що міститься в змінній myText.

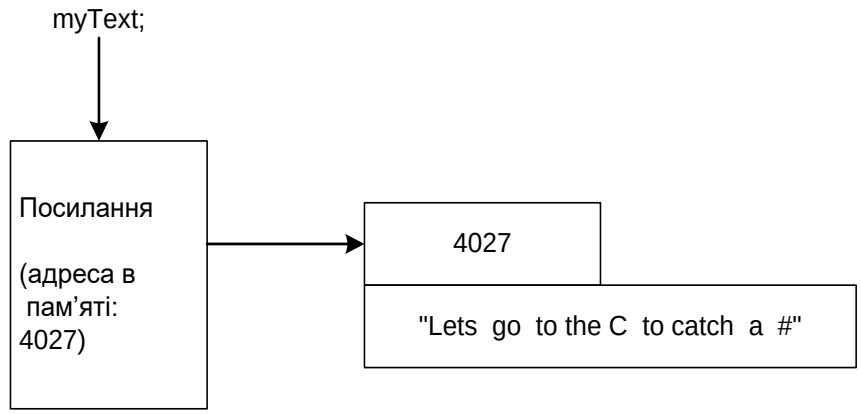


Рис. 2.7. **Тип string є посилальним**

Важливо відзначити, що фактична адреса при використанні посилань у вихідному коді програми ніколи не застосовується.

Усі класи є посилальними типами.

Адреса місця в комп'ютерній пам'яті, де зберігається об'єкт, називається посиланням, або покажчиком на цей об'єкт.

У більшості випадків відмінності в роботі з типами-значеннями та посилальними типами незначні. Приклад цього – можливість застосування рядків в попередніх програмах без використання поняття посилання.

2.3.2. Огляд основних типів C#

Ієрархічна структура основних типів C# приведена в табл. 2.1.

Таблиця 2.1.

Огляд основних типів C#

Типи-значення	Посилальні типи
Прості типи (наприклад, int)	Типи-класи Тип string
Типи-перерахування	Типи-масиви
Типи-структури	Типи-інтерфейси Типи-делегати

Прості типи. Тип int є одним з 13 простих типів C#. Прості типи використовуються для зберігання числових значень і окремих символів.

Типи-перерахування. Ці типи забезпечують засоби для створення символічних констант, зокрема, їхніх наборів: дні тижня (понеділок, вівторок і т. д.), місяці (січень, лютий, березень і т. д.), кольори (зелений, червоний, синій і т. д.) і багатьох інших.

Типи-структури. Ці типи містять методи та дані так само, як і класи. Будучи подібними до класів, вони мають і декілька відмінностей. Одна з найбільш важливих полягає в тому, що типи-класи є посилальними в той час, як структури – це типи-значення. Фактично, всі прості типи – типи-структури.

Типи-класи. Клас визначає категорію об'єктів і є "кресленням" для їхнього створення. Класи містять елементи, які можуть бути змінними екземпляра, що описують стан об'єкта, і методами, котрі визначають його поводження. Класи можуть успадковувати елементи інших класів. Поняття класу є центральним в ООП.

Типи-масиви. Змінні типу масив – це об'єкти, призначені для зберігання колекцій даних. Усі елементи масиву належать тому самому типу.

Типи-інтерфейси. Інтерфейс передбачає абстрактне поводження, визначаючи один або декілька заголовків методу без супровідної їх внутрішньої реалізації, що присутня у неабстрактних методах класу. Класи можуть реалізовувати інтерфейси, конкретизуючи абстрактне поводження, встановлене інтерфейсом. Інтерфейси дозволяють програмістові реалізувати найбільш складні концепції ООП.

Типи-делегати. Подібно інтерфейсам, делегати використовуються для визначення поводження, але задають заголовок тільки для одного методу. Екземпляр типу “делегат” містить один метод, а сам делегат є

посиланням на нього. Делегати (або посилання на методи) передаються в програмі як звичайні посилання й виконуються як звичайні методи. Делегати життєво важливі для виконання керованих подіями програм на мові C#.

Єдина система типів .NET (Common Type System – CTS)

Єдина система типів (CTS) є складовою частиною .NET Framework. Вона визначає всі типи, описані в цьому розділі, і містить правила їхнього застосування в додатках, що виконуються в середовищі .NET. Як виходить із назви, всі реалізовані на платформі .NET мови програмування, включаючи C#, засновані на типах, визначених CTS.

2.4. Характеристика та особливості застосування простих типів

Прості типи належать до групи вбудованих типів C#. Прикладами їхніх значень є окремі числа (тип int) і окремі символи.

При виборі змінної певного типу програміст фактично задає вид величини, що змінна може зберігати, і набір операцій, в яких вона може брати участь. Кожний простий тип характеризується наступними властивостями:

Форма подання змінної. Приклади – цілі числа, числа з плаваючою точкою та одиночні символи.

Діапазон значень змінної. Приміром, діапазон типу int:

від -2147483648 до 2147483647.

Обсяг використовуваної внутрішньої пам'яті. Для представлення однієї змінної залежно від її типу використовується від 8 до 64 бітів. Наприклад, змінна типу int займає 32 біта пам'яті.

Типи операцій, які можна виконувати зі змінної. Попередні приклади показують, що тип int підходить для додавання, а строкові значення – для конкатенації.

В C# визначено 13 простих типів, які перераховані в табл. 2.2.

Хоча тип bool (останній рядок табл. 2.2.) розглядається тут як простий тип, він пов'язаний з керуванням потоком виконання програм. Ключове слово відноситься до символу, що використовується у вихідному коді C# при оголошенні змінної.

Простір імен System .NET Framework містить всі прості типи. Кожне ключове слово, показане в першому стовпці, – це псевдонім типу, визначеного в CTS. Наприклад, ключове слово int позначає System.Int32 в

CTS. Таким чином, у вихідному коді можна використовувати як короткий псевдонім типу, так і його довге повне ім'я.

Таблиця 2.2

Прості типи в мові C#

Ключове слово мови C#	Тип .NET CTS	Вид значення	Використовувана пам'ять	Діапазон і точність
sbyte	System.SByte	Ціле число	8 бітів	Від -128 до 127
byte	System.Byte	Ціле число	8 бітів	Від 0 до 255
short	System.Int16	Ціле число	16 бітів	Від -32768 до 32767
ushort	System.Ulnt16	Ціле число	16 бітів	Від 0 до 65535
int	System.Int32	Ціле число	32 біта	Від -2147483648 до 2147483647
uint	System.Ulnt32	Ціле число	32 біта	Від 0 до 4294967295
long	System.Int64	Ціле число	64 біта	Від -9223372036854775808 до 9223372036854775807
ulong	System.Ulnt64	Ціле число	64 біта	Від 0 до 18446744073709551615
char	System.Char	Ціле число (один символ)	16 бітів	Всі символи Unicode
float	System.Single	Число з плаваючою точкою	32 біта	Від (+/-)1.5 * 10-45 до (+/-)3.4 * 1038 Приблизно 7 значущих цифр
double	System.Double	Число з плаваючою точкою	64 біта	Від (+/-) 5.0 * 10-324 до (+/-)3.4 * 1030 15-16 значущих цифр
decimal	System.Decimal	Десяткове число (високої точності)	128 бітів	Від (+/-) 1.0 * 10-28 до (+/-)7.9 * 1028
bool	System.Boolean	true або false	1 біт	Немає

Два наступні вираження ідентичні:

int myVariable;

System.Int32 myVariable;

В 3-ому стовпці визначено чотири різних групи простих типів, що містяться в C# – ціле число, число з плаваючою точкою, true / false і число високої точності.

Величини типу bool можуть містити тільки два значення – true або false.

Стовпець "Діапазон і точність" відображає діапазон і точність, забезпечувані величинами відповідного типу. Слід зазначити, що хоча тип char розроблений для окремих символів, він розглядається як цілочисельний.

У розглянутій таблиці наведено дев'ять цілочисельних типів. Вони відрізняються один від одного за трьома параметрами – діапазоном, обсягом займаної пам'яті та здатністю зберігати негативні числа.

У таблиці містяться і три типи з плаваючою точкою – float, double і decimal, використовувані для зберігання чисел, що містять дробову частину (приміром, 6.87, 9.0 і 100.01). Основні відмінності – діапазон, використовувана пам'ять і точність.

2.4.1. Синтаксис оголошення змінних

Дотепер для зображення синтаксису C# використовувався його опис (на основі розмовної мови) та приклади. Однак синтаксис вимагає дуже точного запису (аж до крапки з комою), тому його опис розширений до точної форми.

Обрана тут форма запису – це спрощена версія часто використовуваної для опису синтаксису комп'ютерної мови нотації, називаною формою Бекуса-Наура (Backus-Naur), або BNF. Вона була розроблена Дж. Бекусом і П. Науром для опису мови Алгол 60.

Форма запису синтаксису складається з наступних елементів:

Символ **:: =**, що означає "визначається як".

Метазмінні (розміщені в кутових дужках) в формі **<Слово>.**

Символ, що складається із двох квадратних дужок **[ ]** та позначає необов'язкові елементи [<це необов'язково>].

Три крапки **...** які вказують на необмежену кількість елементів.

Вертикальна риса **|**, що вказує на можливі альтернативи.

Як приклад розглянемо оператор оголошення змінної:

int myNumber;

Оператор_оголошення_змінної :: =

<Тип> <Ідентифікатор_змінної>;

У даному випадку **:: =** вказує, що далі йде визначення змінної, оголошеної в операторі.

Тут **<Тип>** — це синтаксична змінна, яка може бути замінена на int, string або будь-яке припустиме в C# ім'я типу.

Друга синтаксична змінна <Ідентифікатор_змінної> може бути замінена ім'ям змінної, приміром myNumber.

Оператор оголошення повинен закінчуватися крапкою з комою.

На рис. 2.8. розглядається приклад, коли для точного визначення конструкції C# необхідні три крапки «**...**» і символ необов'язкового елемента [ ].

Нарешті, потрібна можливість виразити випадок, коли кілька альтернативних елементів можуть використовуватися в одній позиції.

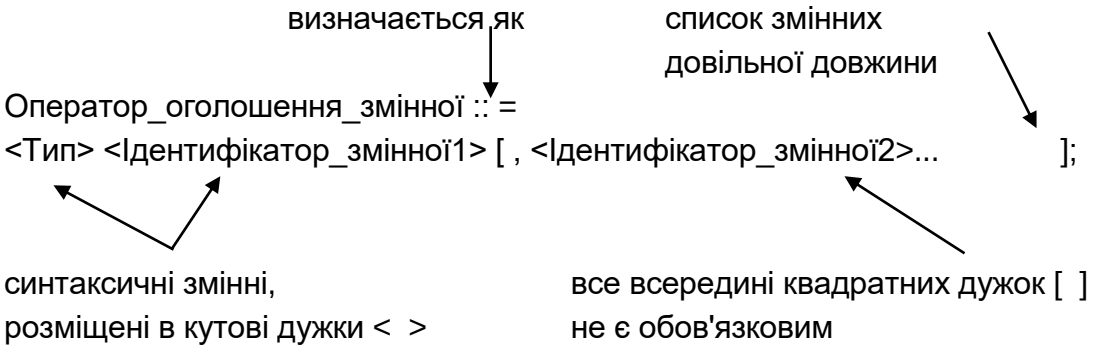


Рис. 2.8. **Приклад, коли для точного визначення конструкції C# необхідні три крапки «...» і символ необов'язкового елемента [ ]**

Приміром, при оголошенні методів і змінних екземпляра класу застосовуються ключові слова public і private:

private int currentFloor;

Хоча це можна розцінювати як невірну методику програмування, але змінну екземпляра можна оголосити і public:

public int currentFloor;

Можна взагалі опустити специфікатор доступу:

int currentFloor;

При цьому змінна currentFloor буде мати специфікатор доступу за замовчуванням – private.

Можливість використання або public, або private позначається вертикальною рисою (|): public | private. Оскільки тут специфікація необов'язкова, вона має вигляд: [public | private]. Повне оголошення змінної тепер визначене в такий спосіб:

```
Оператор_оголошення_змінної :: =
[public | private]  <Тип> <Ідентифікатор_змінної1> [ ,
                                    <Ідентифікатор_змінної2>...  ];
```

Рис. 2.9. **Повне оголошення змінної**

2.4.2. Цілочисельні типи

Як показано в табл. 2.2, в C# визначено дев’ять цілочисельних типів. Цілі числа – це числа без дробової частини, наприклад 34, 0 або -7653. Оскільки множина цілих чисел не обмежена, а комп'ютерна пам'ять має кінцевий обсяг, можна представити лише деяку її підмножину.

Кожний цілочисельний тип використовує певну кількість бітів пам'яті. Чим більше діапазон числа, тим більше, природно, потрібно пам'яті.

Деякі типи (такі, як int), звані знаковими типами, можуть зберігати і негативні, і позитивні значення. Інші типи – беззнакові – можуть представляти тільки позитивні значення (включаючи нуль). Є чотири знакових (sbyte, short, int і long) і чотири беззнакових типи (byte, ushort, uint і ulong).

Тип char можна також розглядати як цілочисельний тип, незважаючи на його спеціальні властивості, пов'язані з можливістю представляти символи Unicode. Але тому що це специфічний тип, безпосередньо пов'язаний з типом String, спочатку тут представлені лише вісім раніше згаданих цілочисельних типів.

Біти

Один біт представляє тільки два значення – 0 і 1. Відповідно, два біти можуть представляти 2 x 2 = 4 різні значення, 3 біти – 2 x 2 x 2 = 8, а x бітів – 2x значень:

8 бітів – 256 значень;

16 бітів – 65 536 значень;

32 біта – 4 294 967 296 значень;

64 біта – 18 446 744 073 709 551 616 значень.

Знакові та беззнакові цілочисельні типи

Кожному з чотирьох знакових цілочисельних типів відповідає беззнаковий, що використовує той же обсяг пам'яті. Тому що негативна частина відсутня, беззнаковий тип дозволяє зберігати у два рази більші позитивні числа. Наприклад, тип sbyte може представляти числа в діапазоні від -128 до 127, а його беззнаковий еквівалент byte – від 0 до 255. Знакові та беззнакові цілочисельні типи представлені в табл. 2.3.

Таблиця 2.3

Знакові та беззнакові цілочисельні типи

Знаковий	Беззнаковий	Використовувана пам'ять
sbyte	byte	8 бітів
short	ushort	16 бітів
int	uint	32 біта
long	ulong	64 біта

Приклад: sbyte myNumber;

Цілочисельні літерали

На відміну від змінних літерали не можуть змінювати своє значення: 5 завжди дорівнює 5 і ніколи – 3 або 8. Числа типу 3, 1009 і -487 – приклади літералів.

Усі цілочисельні літерали мають певний тип точно так же, як і всі оголошені змінні. Компілятор C# при визначенні типу літералу дотримується точних правил: він розглядає розмір літерала і наступний за ним необов’язковий суфікс.

Якщо суфікс відсутній, компілятор вибирає перший з наступних типів: int, uint, long і ulong. Якщо об'єднати цей факт із діапазонами, зазначеними в табл. 2.2, можна обчислити тип кожного літерала. Наприклад, літерали:

43200 типу int

2507493742 типу uint

-25372936858775201 типу long

270072036654375827 типу long

17016748093204541685 типу ulong

Суфіксом літерала може бути U, L або UL.

Мова C# дозволяє використовувати цілочисельні літерали з двома різними основами – 10 (десяткові числа) і 16 (шістнадцятирічні числа).

Літерал з основою 10 починається з будь-якої із цифр 1 – 9, а 0x перед числом указує, що це шістнадцятирічний літерал. Наприклад, 99 має основу 10, а 0x99 – основу 16 і дорівнює 153 (по основі 10). Приклади:

```
int  aNumber;
aNumber = 99;              // Присвоювання 99 по основі 10
aNumber = 0x99;            // Присвоювання 99 по основі 16
```

Цілочисельні літерали не можуть містити ком (32,000 – невірно) або десяткових точок (3.0 і 76.97 – невірно).

2.4.3. Оператори присвоювання.

Використовуючи форму запису синтаксису, представлену вище, можна записати оператор присвоювання для простого типу в загальній формі:

<Ідентифікатор_змінної>:: = <Вираження> ;
де
<Вираження>
::= <Літерал>
::= <Ідентифікатор_змінної>
::= <Числове_вираження>

Слід зазначити, що тут <Числове_вираження> – це припустима комбінація числових значень і операцій: приміром, (count * 4) + (distance - 100).

Коли комп'ютер виконує оператор присвоювання, спочатку він обчислює його праву частину. Знайдене в результаті значення потім присвоюється змінній, що знаходиться в лівій частині операції.

2.4.4. Перетворення вбудованих типів

Об'єкти одного типу можуть бути перетворені в об'єкти іншого типу неявно або явно.

Неявні перетворення відбуваються автоматично, компілятор робить це замість програміста (рис. 2.10).

Явні перетворення здійснюються, коли програміст “приводить” значення до іншого типу.

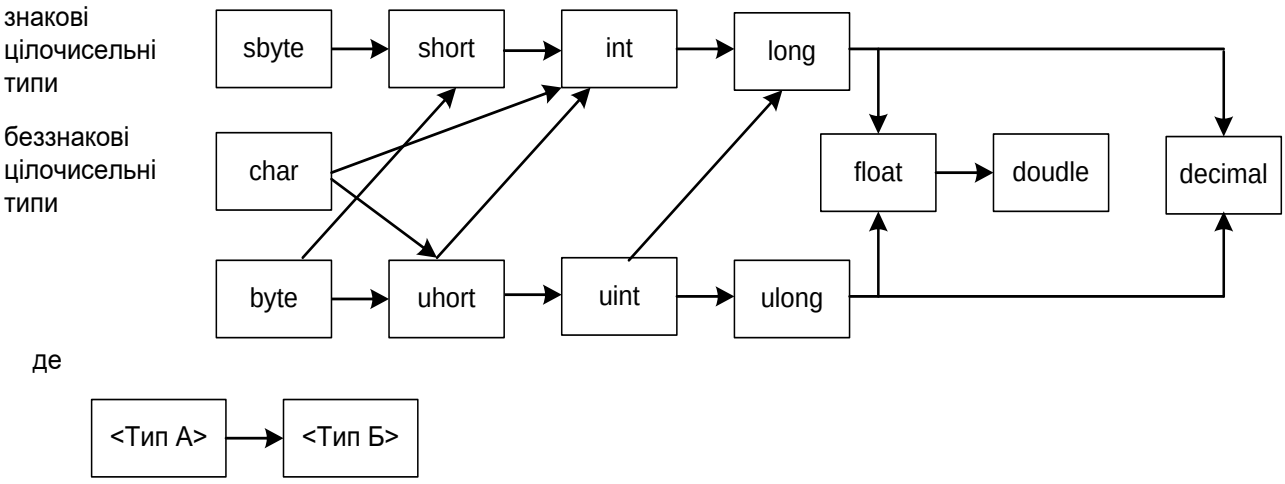


Рис. 2.10. **Шляхи неявного перетворення числових типів**

Неявні перетворення гарантують також, що дані не будуть загублені. Наприклад, ви можете неявно приводити від short (2 байти) до int (4 байти). Незалежно від того, яке значення находиться в short, воно не втратиться при перетворенні до int:

```
short x = 1;
int y = x;              //      неявне перетворення
```

Якщо ви робите зворотне перетворення, то, звичайно ж, можете втратити інформацію. Якщо значення в int більше, ніж 32.767, воно буде усічене при перетворенні. Компілятор не стане виконувати неявне перетворення від int до short:

```
short   x;
int y  =  5;
x  =   y;    //       не   скомпілюється
```

Програміст повинен виконати явне перетворення, використовуючи оператор приведення:

```
short   x;
int   y  =  5;
x  =   (short)   y; //       OK
```

2.4.5. Типи з плаваючою точкою

Змінні з плаваючою точкою дозволяють зберігати числа з дробовою частиною, як, наприклад, число 2.99 або 3.1415926535897931.

Числа з плаваючою точкою дозволяють представити значно ширший діапазон величин, ніж найбільш об'ємний цілий тип long. При написанні чисел такої величини зручно використовувати нотацію, звану науковою нотацією, або е-нотацією (експонентною нотацією) і нотацією з плаваючою точкою.

Ця нотація виражає число 756 000 000 000 000 так, як показано на рис. 2.11, але її не можна використовува у вихідному коді C#.

Мова C# дозволяє писати числа з плаваючою точкою в одній з двох нотацій.

Можна використовувати форму, яка знайома нам з повсякденного життя – зі знаком десяткового дробу, супроводжуваним цифрами (наприклад, 134.87 і 0.0000345), або використовувати показаний тут експонентний формат. Однак знак множення й 10 вилучені із запису і замінені буквою E (або e) з показником, написаним після нього.

Наприклад, число 0.000000456 (рівне 4.56* 10-7) записується в C# як +4.56E-7.

На рис. 2.11 та 2.12 показані всі елементи цієї форми нотації.

Нижче наведено декілька чисел з плаваючою точкою, заданих в C# із застосуванням звичайної форми запису:

```
56.78          // звичний запис з плаваючою точкою
0.645          // число з плаваючою точкою
7.0            // також число з плаваючою точкою
456E-7         // те ж, що і 4.56e-7
8e3            // те ж, що і +8.0e+3
-8.45e8        // негативна величина
```

Варто звернути увагу, що, хоча дробова частина в останньому рядку дорівнює 0, знак десяткового дробу змушує компілятор поводитися з 7.0 як з числом з плаваючою точкою.

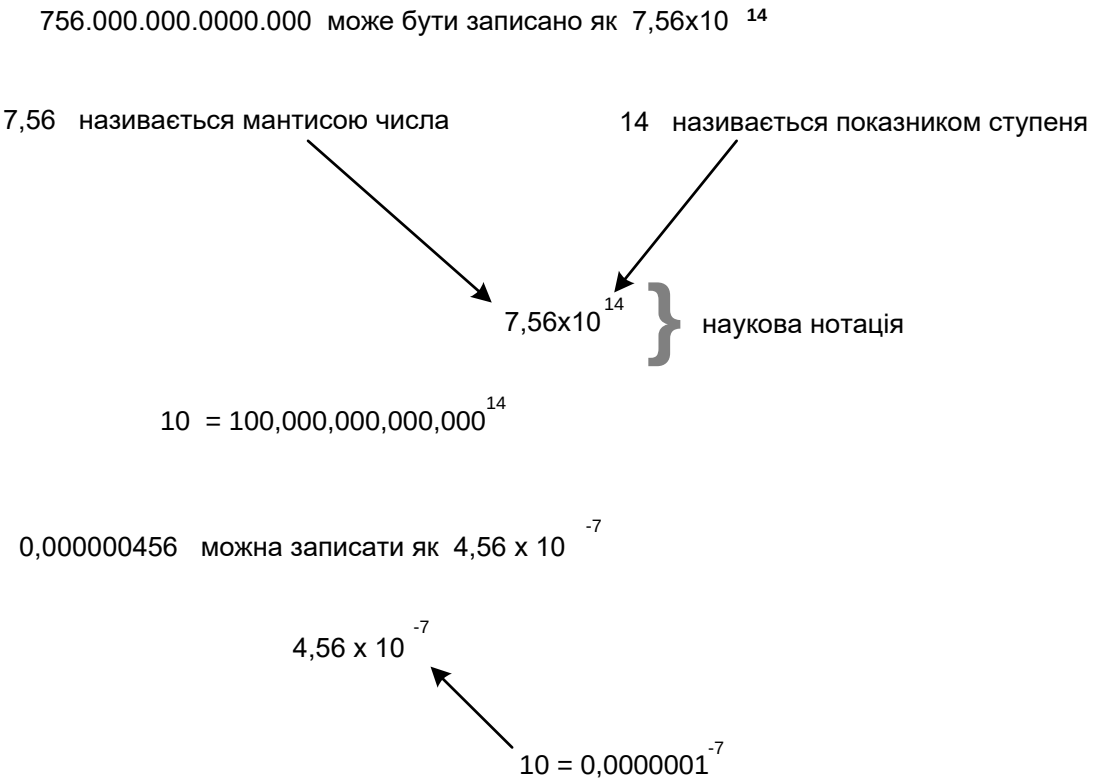


Рис. 2.11. **Наукова нотація чисел з плаваючою точкою**

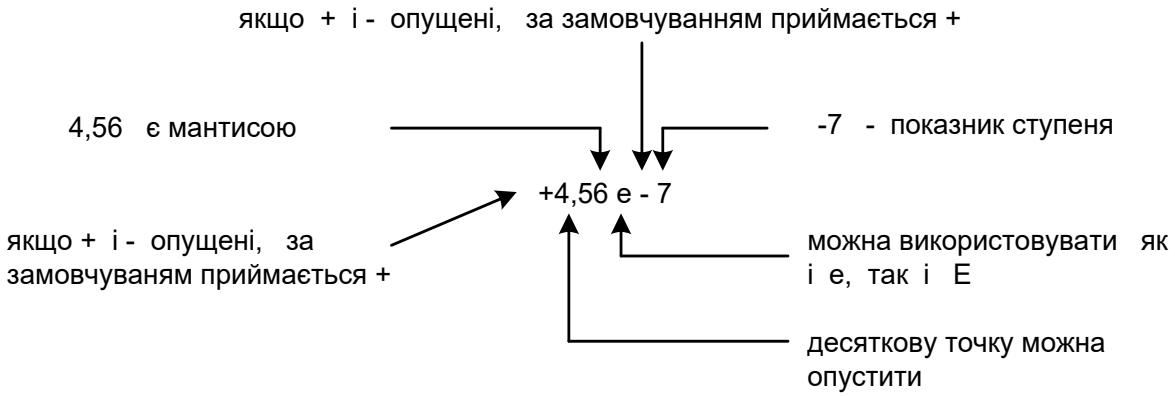


Рис. 2.12. **Деталізація наукової нотації чисел з плаваючою точкою**

Інтерпретація експонентної нотації

Запис 4.56E+7 означає: "Взяти мантису 4.56 і перемістити знак десяткової точки на 7 розрядів вправо. Якщо знак десяткової точки виходить за межі набору цифр, використовувати нулі як заповнювачі".

Запис 4.56E-7 означає: "Взяти мантису 4.56 і перемістити знак десяткової точки на 7 позицій вліво. Якщо знак десяткової точки виходить за межі набору цифр, використовувати нулі як заповнювачі".

В C# є два типи з плаваючою точкою – float та double. Їх відрізняють два основних атрибути – кількість значущих цифр, які вони можуть представляти (це пов'язане з поняттям мантиси, що вже

обговорювалося), і діапазон зміни величини показника ступеня (пов'язане з поняттям коефіцієнта масштабування).

Змінні типу float можуть містити величини від -3.4E38 до 3.4E38. Вони можуть бути близькі до нуля (не будучи рівними йому), як 1.5e-45 або -1,5e-45. Величини представляються за допомогою приблизно 7 значущих цифр. Тип float займає 32 біта (4 байти) пам'яті.

Змінні типу double можуть містити величини від -1.7E308 до 1.7E308. Вони можуть бути близькі до нуля (не будучи рівними йому), як 5.0E-324 або -5.0E-324. Величини представляються за допомогою 15 - 16 значущих цифр. Тип double займає 64 біта (8 байт) пам'яті.

Особливості операцій над числами з плаваючою точкою

Операції за участю величин з плаваючою точкою ніколи не видають переривання в аварійних ситуаціях. Замість цього залежно від виконуваної операції виходить один з наступних результатів:

позитивний (+0) або негативний (-0) нуль,

позитивна (+°°) або негативна (-°°) нескінченність,

NaN (Not a Number – не число).

Якщо робиться спроба присвоїти величину x змінній типу float, де x належить інтервалу -1.5E-45 > x > 1.5E-45 і x != 0 (інакше кажучи, x дуже близько до нуля, але ним не є), результатом операції буде позитивний (якщо x позитивно) або негативний (якщо x негативно) нуль.

Якщо робиться спроба присвоїти величину x змінній типу double, де x належить інтервалу -50E-324 > x > 5.0E-324 і x != 0 (інакше кажучи, x дуже близько до нуля, але ним не є), результатом операції буде позитивний (якщо x позитивно) або негативний (якщо x негативно) нуль.

Якщо робиться спроба присвоїти величину x змінній типу float, де x належить діапазону -3.4E38 > x або x > 3.4E38 (інакше кажучи, x занадто велика по модулю величина), результат операції буде: негативна (позначувана у виводі C# як -Infinity) або позитивна (позначувана у виводі Infinity) нескінченність.

Якщо робиться спроба присвоїти величину x змінній типу double, де x належить діапазону -1.7E308 > x або x > 1.7E308 (інакше кажучи, x занадто велика за модулем величина), результат операці буде негативна (позначувана у виводі C# як -Infinity) або позитивна (позначувана у виводі Infinity) нескінченність.

Якщо робиться спроба виконати некоректну операцію над типами з плаваючою точкою, результатом буде NaN. Приклад невірної операції: 0.0/0.0.

Літерали з плаваючою точкою

Числа з плаваючою точкою на зразок 5.87 або 8.24E8 у програмі на C# за замовчуванням відносяться до типу double. Щоб визначити величину як float, потрібно додати суфікс f (або F). Таким чином, числа 5.87f і 8.24E8F – літерали типу float. І навпаки, можна також явно визначити літерал як double, використовуючи суфікс d (або D), як у числах 5.87d або 8.24E8D.

Багато виражень, розглянуті в математиці як рівні, не завжди рівні при обчисленнях з плаваючою точкою. Це викликано тим фактом, що величина з плаваючою точкою, обчислена одним способом, часто відрізняється від цієї ж величини, обчисленої іншим способом.

Розглянемо приклад програми. Що можна чекати при виведенні?

```
01: using System;
02:
03: class NonEquality
04:{
05:    public static void Main()
06:    {
07:       double mySum;
08:
09:       mySum = 0.2f + 0.2f + 0.2f + 0.2f + 0.2f;
10:       if (mySum == 1.0)
11:          Console.WriteLine("mySum is equal to 1.0");
12:       Console.WriteLine("mySum holds the value " + mySum);
13:    }
14: }
```

Результат:

```
mySum holds the value 1.0000000149011612
```

У рядку 9 лістингу число 0.2f додається п'ять разів, і результат присвоюється змінній типу double mySum. У математиці 5 х 0.2 в точності дорівнює 1. Проте при порівнянні mySum з 1.0 операцією рівності == в

рядку 10 ця умова виявляється рівною false, і тому рядок 11 не виконується.

Змінна mySum дорівнює 1.0000000149011612, як показано в прикладі виведення.

Бажано уникати використання операцій порівняння для величин з плаваючою точкою. Якщо необхідно виконати таке порівняння, потрібно враховувати діапазон можливої погрішності.

Дотримуючись попередньої поради, можна вказати припустиму точність, скажемо +/- 0.0001, і переписати рядок 10 лістингу в такий спосіб:

```
10:  if (Math.Abs(mySum – 1.0) < 0.001)
```

Припустима помилка Модуль різниці mySum і 1.0

Тип decimal

Багато в чому тип decimal подібний до типів з плаваючою точкою, оскільки він дозволяє представляти числа з дробовою частиною й використовує концепцію величини, значущих цифр та коефіцієнта масштабування. Однак цей тип значно більш точний, має менший діапазон і займає більше пам'яті. Таким чином, тип decimal корисний для обчислень, де потрібна висока точність.

Хоча значення типу decimal використовуються для представлення чисел з дробовою частиною на зразок типів з плаваючою точкою, вони не відносяться до останніх.

Тип decimal не підтримує позитивні/негативні нулі, позитивні/негативні нескінченності або NaN, як типи з плаваючою точкою. Замість цього величини в інтервалі -1.0E-28 і 1.0E-28 вважаються нулем, а величини поза припустимим діапазоном генерують переривання переповнення зверху/знизу.

На відміну від цілочисельних типів числа з плаваючою точкою і типу decimal дозволяють представляти числа з дробовою частиною й у більшому діапазоні, але ціною цьому є більші вимоги до обсягу пам'яті та уповільнення продуктивності.

Запис чисел типу decimal. Будь-яке число з десятковою точкою або записане в експонентній формі відноситься за замовчуванням до типу double. Таким чином, щоб літерал мав тип decimal, необхідно додати до числа суфікс m (або M).

Величини цілочисельних типів (при необхідності) неявно перетворюються в decimal. Отже, літералам цих типів, приміром 10, 756 і -963, суфікс m не потрібен.

2.5. Константні величини

Часто у вихідній програмі використовуються фіксовані числа, наприклад: 3.141592 або інші величини, які не змінюються протягом виконання програми. Мова C# дозволяє оголошувати імена для літералів або інших виражень і використовувати їх замість запису фактичного значення.

Наприклад, замість

distance = secondsTraveled * 186000;

можна дати значенню 186000 ім'я SpeedOfLight і привести вираження до вигдялу:

distance = secondsTraveled * SpeedOfLight;

Константа SpeedOfLight схожа на змінну тим, що вона має ім'я і певне значення. Фактично її можна було б оголосити як змінну:

int SpeedOfLight = 186000;

Це дозволяє застосовувати SpeedOfLight при обчисленні відстані. Але не слід забувати, що значення SpeedOfLight може бути випадково змінене в програмі. Тому, щоб залишити величину SpeedOfLight незмінною, при її визначенні потрібно вказати ключове слово const:

const int SpeedOfLight = 186000;

Ім'я, що представляє постійну величину, в C# називається **константою**.

Класифікація константних величин

Константи діляться на наступні групи:

- Числові
  - Цілі
  - Речовинні
- Перераховуємі
- Символьні (літерні)
  - Клавіатурні
  - Кодові (керуючі або розділові символи)
  - Кодові числові
- Строкові (рядки або літерні рядки)
- Іменовані (символічні)

Синтаксичний блок оголошення константи.

Оператор_оголошення_константи ::=

[public | private] const <Тип> <Ідентифікатор_Константи> =
<Константне_вираження>

Примітка: тут <Константне_вираження> може складатися лише з одного літерала, як в наступному рядку коду:

public const double Pi = 3.14159;

Це вираження може також складатися з комбінації літералів, інших констант і операцій за умови, що воно може бути обчислене під час компіляції:

public const double Pi = 3.14159;

public const double TwicePi = 2 * Pi; // Коректно, тому що значення 2 і Pi відомі під час компіляції.

Оголошення константи може знаходитися у визначенні класу – при цьому константа стає його елементом. Константи завжди мають специфікацію static і тому доступні іншим об'єктам за межами класу тільки з використанням ім'я класу, за яким йде операція точки та ім'я константи, а не конкретного екземпляра об'єкта. Сказане демонструється в наведеному нижче прикладі.

Приклад. Обчислення маси 100 електронів.

```
using System;
class Constants
{
   public const decimal MassOfElectron = 9.0E-28m;
}

class MassCalculator
{
   public static void Main()
   {
      decimal totalMass;
```

```
14:        totalMass = 100 * Constants.MassOfElectron;
        Console.WriteLine(totalMass);
    }
}
```

Результат:
9E-26

У наведеному лістингу містяться два класи – Constants і MassCalculator. Клас Constants містить константу MassOfElectron. Для обчислення маси 100 електронів класу MassCalculator потрібен доступ до цієї константи.

У рядку 14 міститься ім'я класу Constants, що супроводжується операцією уточнення (.) та ім'ям константи MassOfElectron.

Спроба звертання до константи MassOfElectron шляхом створення екземпляра класу Constants, як показано у двох наступних рядках, виявляється некоректною.

```
Constants  myConstants = new Constants;     // Створення нового об'єкта
// myConstants  класу Constants
……
totalMass = 100 *  myConstants.MassOfElectron; // Некоректно.
```

Доступ до MassOfElectron можливий тільки за допомогою ім'я класу

Символьні (літерні) константи

Розрізняють наступні символьні константи.

Клавіатурні: ‘1’, ‘t’, ‘y’ – клавіатурний символ задається в апострофах;

Наприклад, char t = 'd'; // d

Кодові – для завдання деяких керуючих і розділових символів, наприклад, ‘\n’, ‘\t’;

В англійській термінології для керуючих послідовностей застосовується термін "escape" – "бігти", "уникати", що підкреслює їхню здатність уникати стандартної конкатенації.

Таблиця 2.4

Керуючі послідовності

Керуюча послідовність	Призначення	Представлення Unicode
\’	Одинарні лапки	\u0027
\”	Подвійні лапки	\u0022
\\	Зворотна коса риска	\u005C
\0	Символ Null	\u0000
\a	Дзвінок	\u0007
\b	Символ забою	\u0008
\f	Подача сторінки	\u000C
\n	Переведення рядка	\u000A
\r	Повернення каретки	\u000D
\t	Горизонтальна табуляція	\u0009
\v	Вертикальна табуляція	\u0008

Кодові числові – для завдання будь-яких кодів символів, наприклад,

```
char t = '\x1A' ;  // код символу «←»
```

2.6. Форматування числових значень

Дотепер числа виводилися з використанням простого вбудованого формату, наприклад,

```
Console.WriteLine("Distance  traveled:" + 10000000.432);
```

При цьому на консоль відображається наступне:

Distance traveled: 10000000.432

Однак, змінюючи зовнішній вигляд числа за допомогою ком (в закордонній нотації комою прийнято відокремлювати розряди, кратні тисячам), наукової нотації та обмеженої кількості десяткових цифр, можна поліпшити його читаність і зробити більше компактним при виведенні на екран. В табл. 2.5. наведені відповідні приклади.

Таблиця 2.5

Приклади форматованих чисел

Ім'я змінної	Число	Відформатоване число
distance	7000000000000000	7.00E+015
Mass	3.87839029837898773 62	3.8784
Length	20000000	20,000,000

Стандартне форматування

Кожний числовий тип в.NET Framework представлений структурою struct, що дозволяє йому містити корисну вбудовану функціональність. Одним із її прикладів є метод ToString. Він дозволяє перетворити будь-який із простих типів в рядок і, крім того, вказати необхідний формат.

На рис. 2.13 показане використання методу ToString для перетворення числа 20000000.45965981m типу decimal в рядок типу string з комами (для розділення тисяч) і лише двома десятковими розрядами.

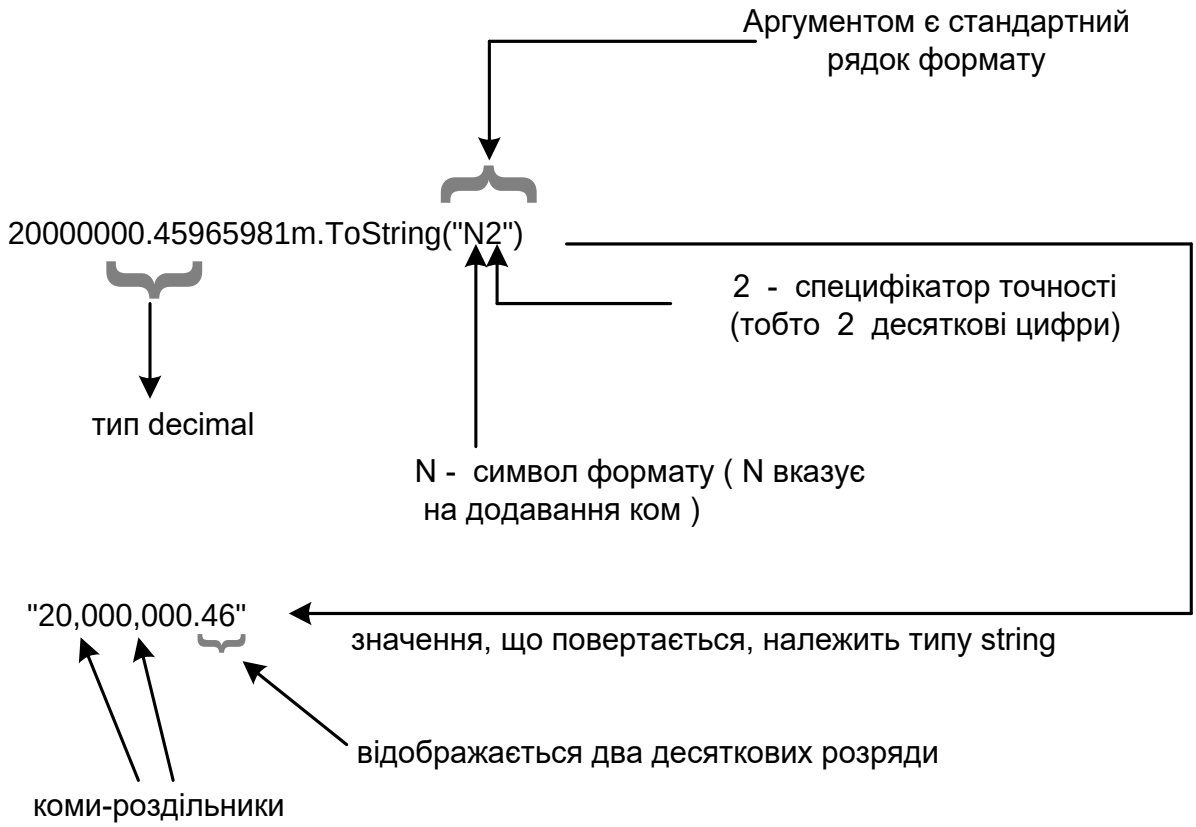


Рис. 2.13. **Форматування літерала типу decimal**

Метод ToString (у використовуваній тут формі) має один аргумент типу string, що задає формат.

Аргумент складається з символу, званого символом формату (в даному випадку N), що задає формат, і необов'язкового числа специфікатора точності (в даному випадку 2), яке має різний сенс для різних символів формату.

Використовувані символи й відповідні їм формати наведені в табл. 2.6.

Таблиця 2.6.

Використовувані в C# символи формату

Символ	Опис	Приклад
1	2	3
C, c	Валюта. Форматування, специфічне для настроювань локалізації. Вони містять інформацію про тип грошової одиниці й інших параметрів, які можуть змінюватися залежно від країни	2000000.456m.ToString("C") Повертає "$2,000,000,46" (Якщо операційна система настроєна відповідно до американських стандартів)
D, d	Ціле число. Специфікатор точності встановлює мінімальне число цифр. Виведення доповнюється ведучими нулями, якщо кількість цифр фактичного числа менше, чим специфікатор точності. (Примітка: цей символ формату застосовується тільки для цілочисельних типів)	45687.ToString("D8") Повертає: "00045678"
E, e	Експонентна (наукова) нотація. Специфікатор точності визначає кількість десяткових цифр, за замовчуванням рівне 6	345678900000.ToString("E3") Повертає "3.457E+011" 345678912000.ToString("e") Повертає "3.456789e011"
F, f	Фіксована точка. Специфікатор точності вказує кількість десяткових цифр	3.7667892.ToString("F3") Повертає "3.767"

Закінчення табл. 2.6

1	2	3
G, g	Загальний. Найбільш компактний формат при виборі E або F. Специфікатор точності встановлює максимальну кількість цифр в представленні числа	65432.98765.ToString("G") Повертає "65432.98765" 65432.98765.ToString("G7") Повертає "65432.99" 65432.98765.ToString("G4") Повертає "6.543E4"
N, n	Число. Число з комами-роздільниками. Специфікатор точності встановлює кількість десяткових цифр	1000000.123m.ToString("N2") Повертає "1,000,000.12"
X, x	Шістнадцятирічне число. Специфікатор точності встановлює мінімальну кількість цифр, що представляються в рядку. Для досягнення певної ширини додаються ведучі нулі	950.ToString("x") Повертає "3b6" 950.ToString("X6") Повертає "0003B6"

Метод ToString є потужним засобом для здійснення процесу форматування числових величин. Але його застосування виявляється незручним, якщо в рядок типу string вставляється декілька по-різному відформатованих чисел.

Наступний фрагмент ілюструє, яким заплутаним стає виклик WriteLine і як важко зрозуміти, яка частина є текстом, а яка – форматованим числом:

Console.WriteLine(“The length is: " + 10000000.4324ToString("N2") + "The widlh is: " + 65476356278.098746.ToString("N2") + "The height is: " + 4532554432.45684.ToString("N2"));

На екран виводиться наступне:

The length is: 10,000,000.43 The width is: 65,476,356,278.10 The height is: 4,532,554,432.46

Вираження було б більш чітким, якби можна було розділити статичний текст і числа та вказати (за допомогою невеликого числа специфікаторів) позицію і формат кожного числа.

Мова C# надає витончене рішення цієї проблеми.

Якщо на час відкинути форматування і використовувати специфікатор {<N>}, де <N> задає позицію числа в списку чисел, що йдуть після статичного тексту у виклику WriteLine, попередні рядки коду можна записати в такому видгляді:

Console.WriteLine(“The length is: {0} The width is: {1} The height is: {2}", 10000000.4324, 65476356278.098746, 4532554432.45684);

де {0} відноситься до першої величини (100000000.4324) в списку чисел після рядка тексту, {1} – до другої (65476356278.098746), а {2} – до третьої.

Ще один варіант організації виведення. Console.WriteLine є перевантаженим методом, що має кілька різних версій. Одна з них підтримує наступні аргументи:

Console.WriteLine(<Формат>, <Значення0>, <Значення1>...)

де <Формат> представляє статичний текст зі специфікаторами формату (див. табл. 2.6);

<Значення0>, <Значення1>... представляють значення, що вставляються в рядок <Формат> в позиціях, заданих специфікаторами.

На специфікатор формату вказують фігурні дужки ({ }). Він завжди повинен містити індекс, що посилається на одне з наступних значень <Значення> і показує, що вставляється в рядок на його місці.

Наприклад, виклик методу Console.WriteLine (рис. 2.14), приводить до наступного виведення на консоль:

Mass: 100 Distance 50 Energy 35

Специфікатор формату дозволяє також вказати ширину простору, що відводиться під виведення <Значення>. Для цього після індексу через кому задається ширина в символах.

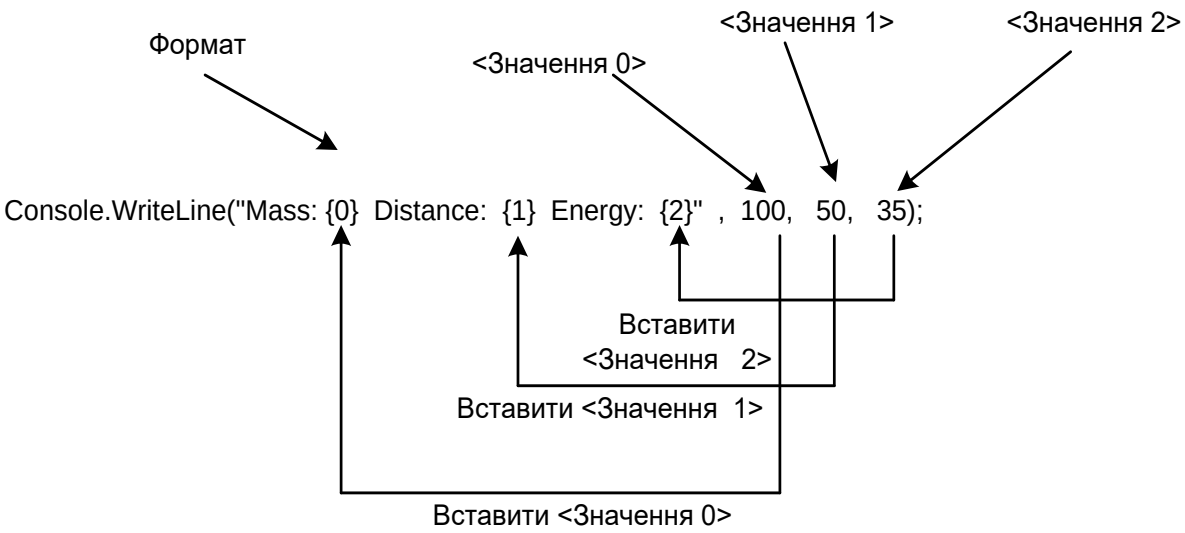


Рис. 2.14. **Приклад виклику методу Console.WriteLine**

Позитивне число означає вирівнювання <Значення> праворуч, а негативне – ліворуч.

Наприклад:

Console.WriteLine(“Distance: {0,10} miles”, 100);

виводить

Distance: 100 miles

ширина дорівнює 10 символів і
число вирівняне праворуч,

тоді, як код

Console.WriteLine(“Distance: {0,-10} miles”, 100);

негативне число вказує
на вирівнювання ліворуч.

Виводить:

Distance: 100 miles

ширина дорівнює 10 символів і
число вирівняне ліворуч.

На закінчення можна визначити і формат значення, включивши в специфікатор додатковий символ формату з наступним необов'язковим специфікатором точності.

Символ формату йде після крапки з комою. Він має таке ж значення, як при використанні з методом ToString.

Наприклад, вираження

Console.WriteLine("Distance:{0,-12E2} Mass: {1,-12N}", 100000000, 5000000);

спричиняє наступне виведення:

Distance: 1.00E+008 Mass: 5,000,000.00

Для друку на консолі рядка з фігурними дужками досить просто опустити <Значення0>, <Значення1> і т. д. у виклику Console.WriteLine. Компілятор C# вибере іншу версію (завдяки перевантаженню) метода Console.WriteLine, де дужки { } ігноруються та друкуються, як є.

Наприклад,

Console.WriteLine(“Number {0} is black, number {1} is red");

надрукує наступне: Number {0} is black, number {1} is red

Питання для самоконтролю

1. Перерахуйте лексичні елементи мови C#.
2. Опишіть базову структуру C#-програми.
3. Дайте огляд основних типів C#.
4. Що означає Єдина система типів .NET (Common Type System - CTS)?
5. Опишіть синтаксис оголошення змінних.
6. Як здійснюються перетворення вбудованих типів даних?
7. Коли доцільно використовувати типи з плаваючою точкою?
8. Опишіть класифікацію константних величин.
9. Приведіть синтаксичний блок оголошення константи.
10. У чому суть форматування числових значень. Приведіть приклади стандартного форматування.

3. Програмування лінійних обчислювальних процесів

3.1. Основні операції мови C#

Будь-яка програма може бути складена за допомогою **чотирьох основних структур**:

послідовність (або структура проходження) – це група команд, виконуваних одна за другою. Програми, що складаються тільки зі структури проходження, називаються ***лінійними програмами***;

рішення (або структура вибору) – це конструкція, що дозволяє даним впливати на хід виконання програми (організовувати розгалуження в програмах);

повторення (цикл або структура повторення) – дозволяє багаторазово виконувати команду або групу команд;

метод (процедура, функція) – дає можливість замінити групу команд однією командою.

Структура проходження вбудована в C#. Поки не зазначено інше, комп'ютер виконує інструкції C# одну за другою в тій послідовності, в якій вони записані.

3.1.1. Огляд основних операцій C#

В C# доступна велика кількість операцій. За винятком тих, що призначено для спеціальних цілей, що залишилися, можна розділити на чотири основних категорії: арифметичні, відношення, логічні та побітові.

Арифметичні операції застосовуються до значень простих числових типів і становлять основу для обчисленнь в C#.

Операції відношення дозволяють порівнювати значення, наприклад **sum < 100.**

Операції відношення дають вираження логічного типу, що, за визначенням, завжди має значення або **true**, або **false**.

Логічні операції дозволяють об'єднати два або більше логічних виражень (будь-які значення типу **bool**). Вони тісно пов'язані з операціями відношення і дають можливість за допомогою оператора **if** та операторів циклів керувати потоком виконання програми.

Побітові операції дозволяють працювати з окремими бітами операнда.

Арифметичні операції (додавання (+), віднімання (-), множення (*) і ділення (/)) в сполученні з числами (званими операндами) формують арифметичні вираження.

Арифметичні вираження на C# досить легко інтерпретувати, тому що вони (більш-менш) відповідають усім відомим арифметичним правилам. Усім чотирьом базовим операціям, згаданим у попередньому розділі, потрібен один операнд ліворуч і один праворуч. Арифметична операція, що поєднує у вираженні два операнда, наприклад

distance1 + distance2,

називається ***бінарною***. Операнд у такому контексті може приймати різні форми. Це може бути просто число (наприклад, 345.23), змінна, константа, що представляє числове значення, або ж числове вираження, наприклад:

distance1 * 100 + distance2 * 300.

Операндом може бути й виклик методу. В цьому випадку, природно, метод повинен повертати значення. Як ілюстрація наведено виклик методу **Average** (рис. 3.1).

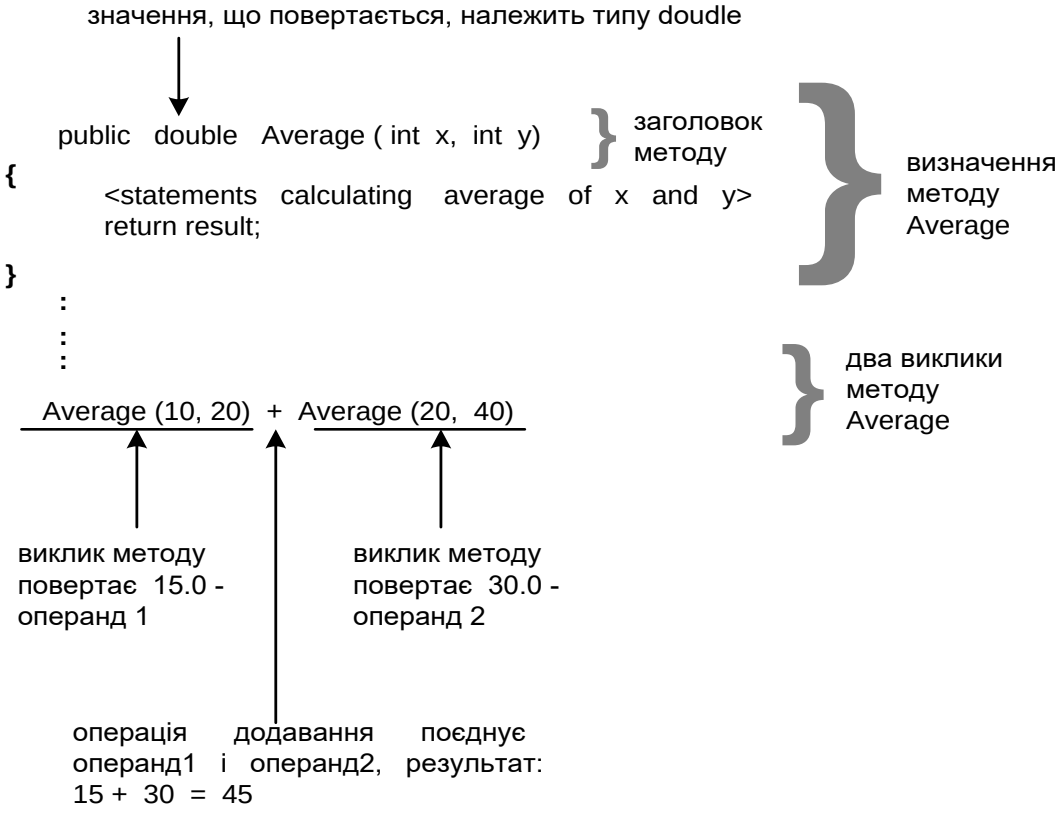


Рис. 3.1. **Виклик методу Average**

Обидва виклики відбуваються до того, як виконується операція додавання. По їх завершенні відповідні конструкції можна розглядати як

числа (15.0 і 30.0) типу **double**, який указаний у заголовку методу **Average**).

3.1.2. Синтаксичні блоки

Усі п'ять форм операнду можна лаконічно виразити, використовуючи наступний синтаксичний блок.

Синтаксичний блок «Операнд»

Операнд можна представити одним із п'яти різних способів:

Операнд
::= <Літерал>
::= <Ідентифікатор_числової_змінної>
::= <Ідентифікатор_числової_константи>
::= <Числове_вираження>
::= <Виклик_методу>

Поняття числового вираження можна також формалізувати, як показано в синтаксичному блоці «Числове вираження»

Числове вираження
::= <Операнд> <Бінарна_операція> <Операнд>
::= <Числове_вираження> <Бінарна_операція>
<Числове_вираження>

3.1.3. Логічні операції

Мова C# містить три логічних операції, семантично еквівалентних словам «**І**», «**АБО**» та «**НЕ**» звичайної мови (табл. 3.1).

Таблиця 3.1

Три часто використовувані логічні операції

Логічна операція	Назва	Позна-чення	Число операндів	Приклад
І	Кон'юнкція	&&	2	(5>3) && (10<20)
АБО	Диз'юнкція	||	2	(mass<800) || (distance>1000)
НЕ	Заперечення	!	1	!(mass>8000)

Логічне І в C# позначається як **&&,** логічне АБО — **||**, логічне НЕ — **!** (знак оклику). Таблиця істинності (табл. 3.2) для типових логічних операцій наведена нижче.

Табліця 3.2

Таблиця істинності для типових логічних операцій

x	y	X & Y І (кон'юнкція)	X | Y АБО (диз'юнкція)	X ^ Y (АБО, що виключає)	!X (заперечення)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Коли програма обробляє логічне вираження з операцією **&&,** наприклад **(distance == 1000) && (mass == 3000)**, спочатку визначається значення **(distance == 1000)**.

Якщо воно дорівнює **false**, компілятор уже знає, що і все вираження буде дорівнювати **false**, незалежно від другої частини **((mass == 3000)).** Тому (**mass == 3000**) не обробляється. Такий механізм називається "*коротким замиканням".*

Аналогічно, якщо програма містить вираження з операцією ||, наприклад,

(distance == 1000) || (mass == 3000)

і перша частина **((distance == 1000))** дорівнює **true**, друга частина ігнорується, тому що незалежно від її значення все вираження дорівнює **true**.

Мова C# містить ще три логічних операції, звані побітове І **(&)**, побітове АБО **(|)** та АБО, що виключає **(^).** Таблиці істинності **&** та **|** ідентичні таблицям **&&** та **||** відповідно (див. табл. 3.2). Але **&** та **|** не використовують "коротке замикання" при обробці виражень, тому вони виконуються трохи повільніше.

У більшості випадків застосовуються **&&** та **||,** але в рідких випадках необхідно, щоб програма завершила обробку всіх логічних підвиражень, навіть якщо це й потрібно для визначення значення загального вираження.

3.1.4. Пріоритети операцій

Будь-яке вираження, де використовується більше двох операндів, завжди можна розбити на підвираження, кожне з яких складається тільки з двох операндів і однієї бінарної операції. Після обчислення результатів підвиражень вони використовуються на наступному рівні.

Розглянемо вираження:

4 * 5 + 40 * 10 – 20 * 40/10 + 70

Легко побачити, що воно складається з декількох підвиражень. На рис. 3.2. показано, як, користуючись визначенням числового вираження, можна виконати обчислення.

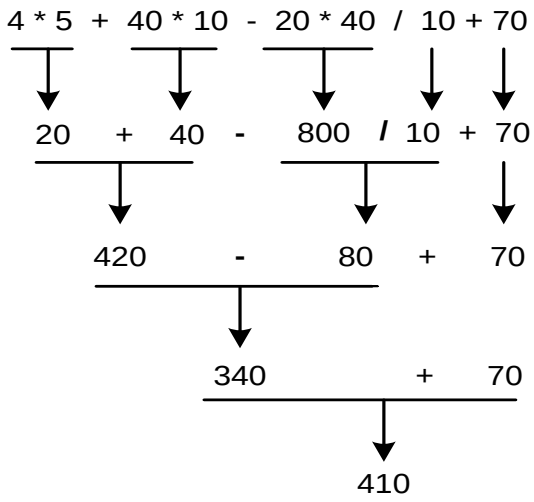


Рис. 3.2. **Черговість виконання операцій при обчисленні вираження**

Відповідно до відомих правил, множення відбувається до додавання, як показано на рис. 3.3:

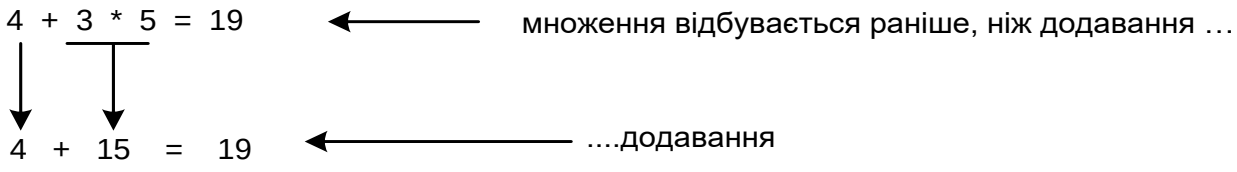


Рис. 3.3. **Множення відбувається до додавання**

Отже, коли операнд (наприклад, число 3 на рис. 3.3) може бути оброблений більш ніж однією операцією (+ або * в даному випадку), мова C# діє за правилами старшинства (пріоритетів) операцій.

Для простоти в числових вираженнях на рисунках використовувалися числа в явній формі. Очевидно, що замість них можна було б застосувати комбінацію будь-яких визначень операндів, наведених в синтаксичному блоці «Операнд».

Крім чотирьох бінарних операцій, C# містить і інші арифметичні операції. Порядок виконання кожної з них стосовно інших чітко визначений у таблиці пріоритетів.

Огляд операцій та їхніх пріоритетів наведений у табл. 3.3.

Таблиця 3.3

Пріоритети операцій

Катюго-рія	Операція	Асоціатив-ність	Значення
1	2	3	4
Угрупу-вання	(<Вираження>)	Зліва направо	Круглі дужки для угрупування
Первин-ні	<Ім'я_об'єкта>.<Ім'я_елемента>	Доступ до елемента	
	<Ім'я_методу>(<Аргументи>)	Зліва направо	Виклик методу
	<Ім'я_масиву>[Індекс]		Доступ до масиву
	<Змінна>++	Справа наліво	Постфіксна операція інкременту
	<Змінна> --	Справа наліво	Постфіксна операція декременту
	new <Ім'я_класу> (<Аргументи>)		Створення екземпляра класу
	typeof(<Тип>)		Визначення типу
	sizeof(<Тип>)		Визначення розміру структури
Унарні	+<Вираження>	Справа наліво	Унарний плюс
	-<Вираження>	Справа наліво	Унарний мінус
	!<Логічне_вираження>	Справа наліво	Логічне заперечення
	++<Змінна>	Справа наліво	Префіксна операція інкременту
	--<Змінна>	Справа наліво	Префіксна операція декременту
	(<Тип>) <Вираження>	Справа наліво	Приведення типу
Мульти-пліка-тивні	<Вираження1> * <Вираження2>	Зліва направо	Множення
	<Вираження1> / <Вираження2>	Зліва направо	Ділення
	<Вираження1> % <Вираження2>	Зліва направо	Ділення по модулі

Закінчення табл. 3.3

1	2	3	4
Адитив-ні	<Вираження1> + <Вираження2>	Зліва направо	Додавання
	<Вираження1> - <Вираження2>	Зліва направо	Віднімання
Відно-шення	<Вираження1> < <Вираження2>	Зліва направо	Менше
	<Вираження1> <= <Вираження2>	Зліва направо	Менше або дорівнює
	<Вираження1> > <Вираження2>	Зліва направо	Більше
	<Вираження1> >= <Вираження2>	Зліва направо	Більше або дорівнює
	<Об'єкт> is <Тип>	Зліва направо	Приналежність типу
Рівність	<Вираження1> == <Вираження2>	Зліва направо	Дорівнює
	<Вираження1> != <Вираження2>	Зліва направо	Не дорівнює
Логічні побітові	<Логічне_вираження1> & <Логічне_вираження2>	Зліва направо	Побітове І
	<Логічне_вираження1> ^ <Логічне_вираження2>	Зліва направо	Побітове АБО, що виключає
	<Логічне_вираження1> | <Логічне_вираження2>	Зліва направо	Побітове АБО
Логічні	<Логічне_вираження1> && <Логічне_вираження2>	Зліва направо	Логічне І
	<Логічне_вираження1> || <Логічне_вираження2>	Зліва направо	Логічне АБО
	<Логічне_вираження1> ? <Вираження1> : <Вираження2>	Зліва направо	Умовна операція
Присво-ювання	<Змінна> = <Вираження>	Справа наліво	Просте присвоювання
	<Змінна> *= <Вираження>	Справа наліво	Множення і присвоювання
	<Змінна> /= <Вираження>	Справа наліво	Ділення і присвоювання
	<Змінна> % = <Вираження>	Справа наліво	Ділення по модулі і присвоювання
	<Змінна> += <Вираження>	Справа наліво	Додавання і присвоювання
	<Змінна> - = <Вираження>	Справа наліво	Вирахування і присвоювання

3.1.5. Простір імен

Простір імен, використовуваний в .NET, дозволяє створювати *контейнери* для коду додатків, щоб і код, і його складові частини були однозначно ідентифіковані.

Простір імен використовується також як засіб категоризації об'єктів в .NET Framework. Більшість таких об'єктів є визначеннями типів, наприклад, визначенням простих типів.

Код C# за замовчуванням міститься в глобальному просторі імен. Це означає, що до об'єктів в коді C# можна звернутися з будь-якого іншого коду в глобальному просторі імен просто за їх іменем.

Можна скористатися ключовим словом **namespace** для того, щоб явно задати простір імен для будь-якого блоку коду, що розміщено у фігурних дужках. Імена, які перебувають в такому просторі імен, якщо до них звертаються не з даного простору імен, повинні кваліфікуватися.

Кваліфікованим ім'ям називається ім'я, в якому міститься вся інформація відносно його ієрархії. Це означає, що якщо в нас є код, котрий перебуває в одному просторі імен, і необхідно скористатися ім'ям, визначеним в іншому просторі імен, то нам необхідно використовувати посилання на цей простір імен. У кваліфікованих іменах для розділення рівнів просторів імен використовується символ точки.

Наприклад:

```
namespace LevelOne
{
// програма, що перебуває в просторі імен LevelOne
// в ній описується ім'я "NameOne”
}
// програма, що перебуває в глобальному просторі імен
```

У цій програмі описується єдиний простір імен — **LevelOne**. (В даному випадку в програмі не міститься ніякого коду, що виконується.)

Код, що міститься усередині простору імен **LevelOne**, може просто посилатися на ім'я **LevelOne**, і ніякої класифікації не потрібно. Однак в коді, що перебуває в глобальному просторі імен, необхідно використовувати класифіковане ім'я **LevelOne.NameOne** для того, щоб на нього послатися.

Усередині будь-якого простору імен можна описувати вкладені простори імен, використовуючи те ж саме ключове слово **namespace**. При звертанні до вкладених просторів імен необхідно вказувати всю їхню ієрархію, відокремлюючи один рівень ієрархії від іншого за допомогою точки. Розглянемо наступні простори імен:

```
namespace LevelOne
{
// програма, що перебуває в просторі імен LevelOne
namespace LevelTwo
{
// програма, що перебуває в просторі імен LevelOne.LevelTwo
// в ній описується ім'я "NameTwo”
}
}
// програма, що перебуває в глобальному просторі імен
```

У даному випадку звертання до ім'я **NameTwo** з глобального простору імен повинне мати вигляд **LevelOne.LevelTwo.NameTwo,** з простору імен **LevelOne** — **LevelTwo.NameTwo**, а з простору імен **LevelOne.LevelTwo** — **NameTwo**.

Після того, як простір імен визначений, з'являється можливість використовувати оператор **using** для спрощення доступу до імен, що в ньому містяться.

У наступній програмі передбачається, що код, який перебуває в просторі імен **LevelOne**, повинен мати доступ до імен простору імен **LevelOne.LevelTwo**, без якого б не було класифікації:

```
namespace LevelOne
{
using LevelTwo;

namespace LevelTwo
{
// тут описується ім'я "NameTwo"
}
}
```

Код, що перебуває в просторі імен **LevelOne**, тепер може звертатися до **LevelTwo.NameTwo** просто як **NameTwo**.

Оператор **using** сам по собі не забезпечує доступу до імен, що перебувають в інших просторах імен. Доти, доки код із простору імен не буде яким-небудь способом прив'язаний до нашого проекту (наприклад, описаний у вихідному файлі проекту або описаний в якому-небудь коді, прив'язаному до цього проекту), ми не отримаємо доступу до імен, що містяться в ньому.

Більше того, якщо код, у якому міститься якийсь простір імен, прив'язаний до нашого проекту, то ми маємо доступ до імен, що містяться в ньому, незалежно від використання оператора **using**.

Оператор **using** всього лише спрощує звертання до цих імен і дозволяє скоротити код, котрий сильно подовжується, роблячи його більш зрозумілим.

3.1.6. Область видимості змінних

У більшості програм, представлених раніше, для позначення контексту класу або методу, зокрема методу **Main()**, використовувалися блокові конструкції з фігурними дужками.

Надалі важливо представити нову концепцію, звану областю видимості. Вона визначає, які сегменти (а значить, і змінні) вихідного коду видні (і, таким чином, доступні) з інших його частин.

Область видимості становить сегмент вихідного коду, де може використовуватися певний ідентифікатор. Змінні можна використовувати лише усередині їхньої області видимості. Фактично областю видимості змінної є той блок вихідного коду, де вона оголошена.

Дві основні області видимості в C# – блок классу та блок методу – показані на рис. 3.4.

Хоча відмінність між ними, скоріше, штучна, область визначення класу має кілька характеристик, відсутніх в області визначення методу. Наприклад, оператори **if-else** можна розмістити тільки в області визначення методу.

Подальше обговорення присвячене області визначення методу і блокам усередині неї.

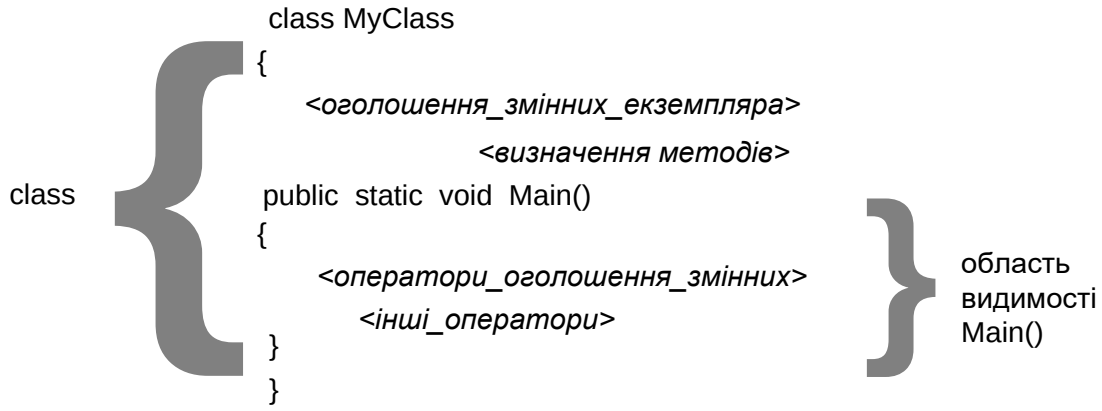


Рис. 3.4. **Блок класу і блок методу**

Новий блок (а значить, і нову область видимості) можна створити, поставивши пару фігурних дужок (рис. 3.5).

```
…
public static void Main()
{
        // Блок Main()
        …
        {
                // Блок A ←——— внутрішній для Main(), зовнішній для B
                  …
                  {
                       // Блок B
                          …
                  }
                  …
        }
        …
}
…
```

Рис. 3.5. **Блоки можуть знаходитися в будь-якій області видимості методу**

Коли блок **B** перебуває усередині іншого блоку **A**, область видимості першого називається *внутрішньою*, а другого – *зовнішньою*.

Ці терміни несуть відносний сенс, оскільки область блоку **A** є внутрішньою стосовно області блоку **Main()**.

Слід зазначити, що імена **A** і **B** обрані лише з метою ілюстрації. Блоки, створені додаванням фігурних дужок, не мають явних імен, і посилатися на них не можна.

Змінна, оголошена усередині методу, називається *локальною*. Такі змінні недоступні за межами блоків, в яких вони оголошені.

Змінні, оголошені усередині блоку, доступні тільки в ньому (це стосується й до змінних інших внутрішніх блоків, вкладених в нього) і тільки після оголошення змінної. Змінну можна оголосити в будь-якому місці блоку.

Дію цих правил ілюструє рис. 3.6.

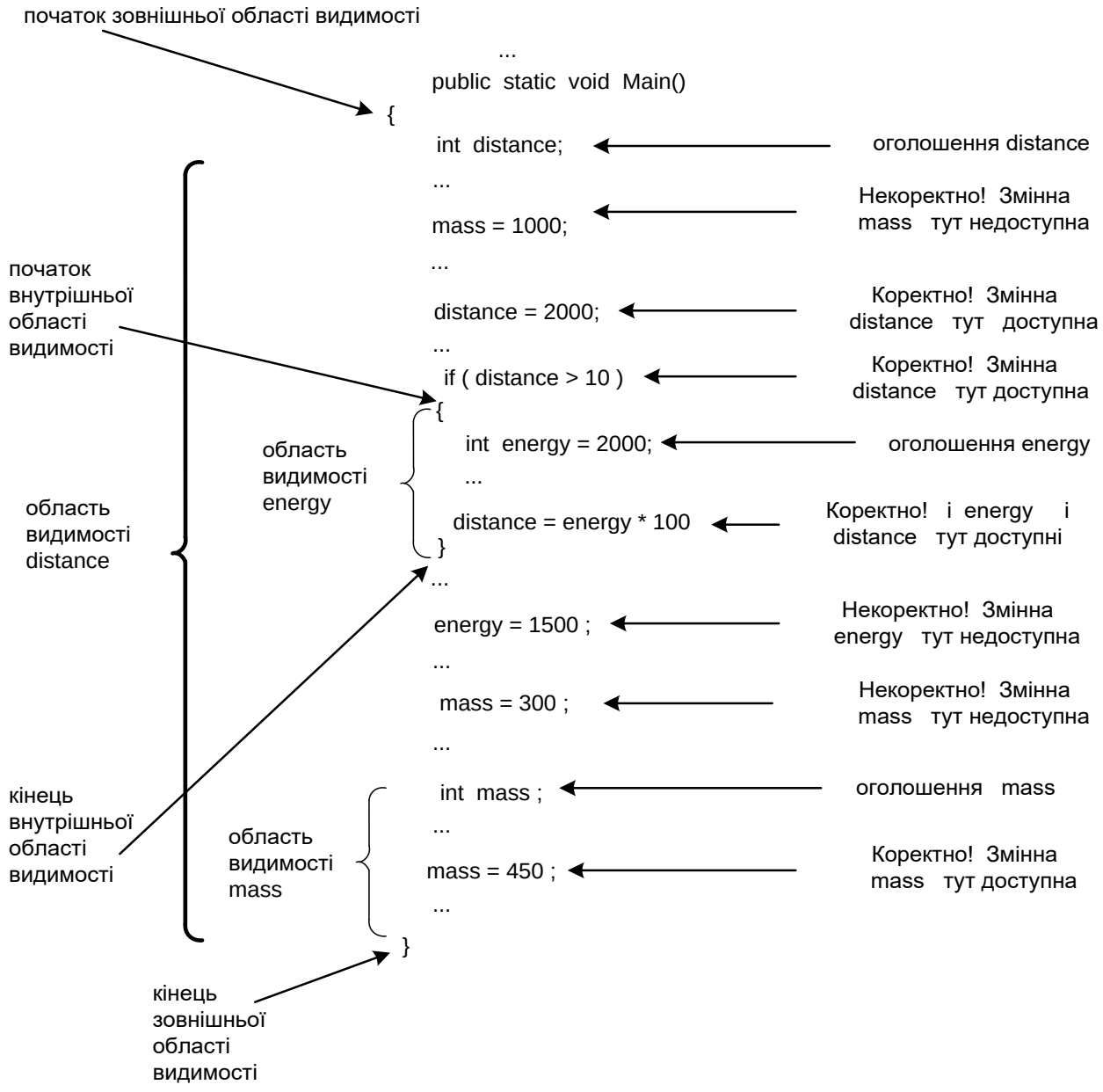


Рис. 3.6. **Блоки, області видимості і доступ до змінних**

Тут показані два блоки коду: один належить методу **Main()**, а інший, розміщений усередині **Main()**, – операторові **if**.

Оскільки змінна **distance** оголошена на початку зовнішньої області видимості, вона доступна у всьому блоці **Main(),** включаючи й код внутрішнього блоку оператора **if**.

Змінна **mass** оголошена ближче до закінчення методу **Main(),** тому вона доступна у відносно невеликому сегменті коду, між її оголошенням і закінченням області видимості **Main().**

Змінна **energy** оголошена усередині блоку **if** і доступна лише усередині нього.

Оголошення змінної у внутрішній області з ім'ям, ідентичним імені змінної в зовнішній, є помилкою. Причина полягає в тім, що це надавало б іншого значення імені змінної з зовнішньої області.

3.1.7. Область видимості і час існування змінних

В обговоренні імен змінних (ідентифікаторів) говорилося, що це зручний спосіб посилатися на певні області пам'яті.

При розгляді концепції області видимості важливо розрізняти можливість використовувати задане ім'я для посилання на дані в пам'яті (область видимості) і час, протягом якого ці дані в пам'яті існують. Останнє називається *часом існування* змінної.

Різниця між областю видимості і часом існування ілюструється на рис. 3.7.

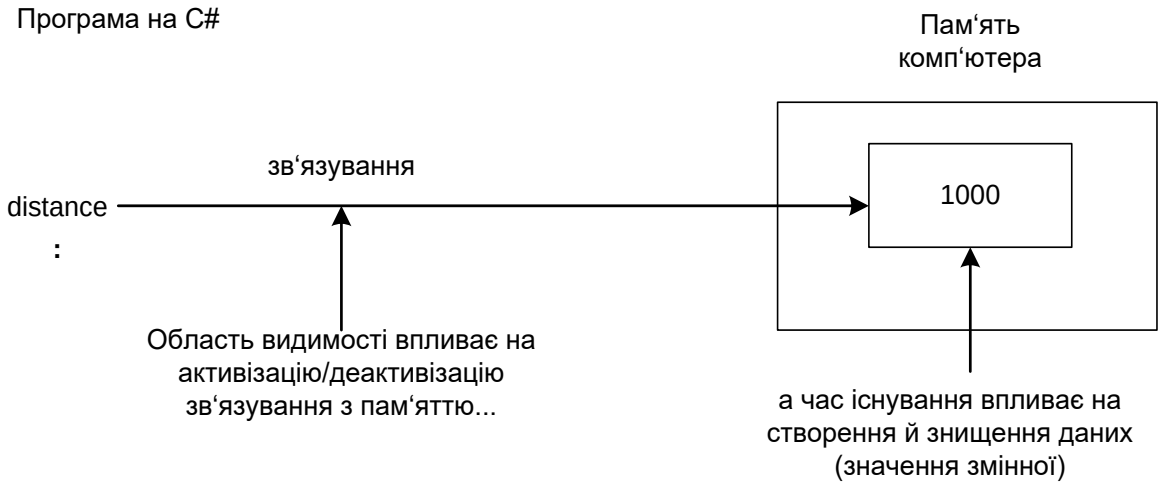


Рис. 3.7. **Область видимості і час існування**

Час існування змінної – це час між її створенням і знищенням. З моменту створення змінної й до знищення її значення зберігається в пам'яті комп'ютера.

Посилання на дані в пам'яті за ім'ям змінної часто називається зв'язуванням.

Відповідно до загального правила, локальні змінні в C# створюються тоді, коли потік керування програми входить в область їхньої видимості, і знищуються, коли потік залишає її. В результаті час існування змінної обмежений областю її видимості.

Хоча наведене правило здається очевидним, воно виконується не для всіх мов програмування. Деякі з них дозволяють потоку програм входити й залишати область видимості змінної декілька разів, зберігаючи при цьому значення останньої. При кожному наступному входженні змінна має те ж значення, що вона мала, коли потік керування залишав область видимості.

3.2. Приклади основних операцій C#

3.2.1. Приклад 1. Арифметичні та логічні операції, операції відношення

```
using System;
class Operadores
{
	static void Main(string[ ] args)
	{
/*
	//=================================================
		int x;
		x = -3 + 4 * 5 - 6;
		Console.WriteLine("x = {0}\n",x);     // 11   Операція 1.1
		x =  3 + 4 % 5 - 6;
		Console.WriteLine("x = {0}\n",x);     // 1    Операція 1.2
		x = -3 * 4 % -6;
		Console.WriteLine("x = {0}\n",x);     // 0    Операція 1.3
		x = (7 + 6) % 5 / 2;
		Console.WriteLine("x = {0}\n",x);     // 1    Операція 1.4
	//=================================================
```

```
        int x = 2, y, z;

        x *= 3 + 2;
        Console.WriteLine("x = {0}\n",x);   // 10    Операція 2.1
        x *= y = z = 4;
        Console.WriteLine("x = {0}\n",x);   // 40    Операція 2.2
//=====================================================
        bool x,y,z;

        x = true; y = true; z = false;

        x = x && y || z;
        Console.WriteLine("x = {0}\n",x);   //  true    Операція 3.1
        x = x || y && z;
        Console.WriteLine("x = {0}\n",x);   //  true    Операція 3.2

        int xx,yy,zz;
        xx = yy = 1;
        zz = xx++ - 1;
        Console.WriteLine("xx = {0}\n",xx);       //  2     Операція 3.3
        Console.WriteLine("zz = {0}\n",zz);       //  0     Операція 3.4

        zz += - xx++ + ++yy;
        Console.WriteLine("xx = {0}\n",xx);       //  3     Операція 3.5
        Console.WriteLine("zz = {0}\n",zz);       //  0     Операція 3.6

        zz = xx / ++xx;
        Console.WriteLine("zz = {0}\n",zz);       //  0     Операція 3.7
//=====================================================

        int x = 5, y = 3, z = 12, t;
        t = x & y | z;                            //       (& -> |)  13
        Console.WriteLine("x & y | z  = {0}\n",t );

        x = 10; y = 5; z = 7;
        t = x | y & z;                            //       (& -> |) 15
        Console.WriteLine("x | y & z  = {0}\n",t );
```

```
            x = 10; y = 5; z = 4;
            t = x ^ y & z | 6;                              //      (& -> ^ -> |) 14
            Console.WriteLine("x | y & z  = {0}\n",t );
        //=============================================
*/
        Console.WriteLine("Введіть ціле число A:");
        int myInt = Convert.ToInt32 (Console.ReadLine()); // 7
        Console.WriteLine("Число A менше ніж 10? {0}", myInt < 10); // true
        Console.WriteLine("Число знаходиться в діапазоні між 0 and 5? {0}",
            (0 <= myInt) && (myInt <= 5)); // false
        Console.WriteLine("Логічна операція A & 10  = {0}", myInt & 10); // 2

        }
}
```

Лістинг містить закоментований блок, розділений символами «========» на окремі фрагменти, кожний з яких ілюструє виконання однієї з операцій, і поточний виконуваний код, що демонструє операції відношень.

Рекомендується набрати наведений вище лістинг програми на клавіатурі, відкомпілювати й запустити на виконання. Далі послідовно заміняйте поточний виконуваний код на один із фрагментів закоментованого коду. Проаналізуйте результати, отримані на екрані.

3.2.2. Приклад 2. Обчислення суми, добутку, максимуму й мінімуму двох чисел, введених користувачем

У вихідному коді представлена програма **SimpleCalculator.cs**, що обчислює суму, добуток, максимум й мінімум двох чисел, введених користувачем. Відповідь виводиться на консоль.

```
01: using System;
02:
03: /*
04:  * Даний клас визначає суму, добуток,
05: * мінімум й максимум двох цілих чисел
06:  */
```

```
07: public class SimpleCalculator
08: {
09:    public static void Main()
10:    {
11:           int x;
12:           int y;
13:
14:           Console.Write("Введіть перше число: ");
15:           x = Convert.ToInt32(Console.ReadLine());
16:           Console.Write("Введіть друге число: ");
17:           y = Convert.ToInt32(Console.ReadLine());
18:           Console.WriteLine("Сума чисел дорівнює: " + Sum(x, y));
19:           Console.WriteLine("Добуток чисел дорівнює: " + Product(x, y));
20:           Console.WriteLine("Максимальне число дорівнює: " + Math.Max(x, y));
21:           Console.WriteLine("Мінімальне число дорівнює: " + Math.Min(x, y));
22:    }
23:
24:    // Метод Sum обчислює суму двох чисел типу  int
25:    public static int Sum(int a, int b)
26:    {
27:           int sumTotal;
28:
29:           sumTotal = a + b;
30:           return sumTotal;
31:    }
32:
33:    // Метод Product обчислює добуток двох чисел типу  int
34:    public static int Product(int a, int b)
35:    {
36:           int productTotal;
37:
38:           productTotal = a * b;
39:           return productTotal;
40:    }
41: }
```

Результат роботи програми:

Введіть перше число: 3

Введіть друге число: 8

Сума чисел дорівнює: 11

Добуток чисел дорівнює: 24

Максимальне число дорівнює: 8

Мінімальне число дорівнює: 3

Press any key to continue

Аналіз вихідного коду програми SimpleCalculator.cs

Програма SimpleCalculator.cs містить декілька важливих елементів, властивих більшості програм, написаних мовою C#.

Щоб зрозуміти призначення рядка 1, необхідно звернутися до концепції простору імен:

01: using System;

.NET-платформа містить важливий простір імен під назвою **System**. У ньому містяться класи, фундаментальні для будь-якої програми на C#. **System** включає й уже розглянутий клас **Console**, використовуваний раніше.

Є два варіанти доступу до класів простору імен **System** (або будь-якого іншого) у вихідному коді:

Без інструкції **using System.** Тоді при всякому звертанні до класів простору імен **System** ідентифікатор останнього необхідно вказувати явно, як, наприклад,

System.Console.WriteLine("Bye Bye!");

З інструкцією **using System;**, як у рядку 1 лістингу прикладу 2 (або в лістингу прикладу 1). У даному випадку будь-який клас простору імен **System** можна викликати без явної вказівки префікса **System**. Попередній приклад коду можна, таким чином, урізати до наступного

Console.WriteLine("Bye Bye!");

Команда **using** звільняє програмістів від необхідності постійно вказувати простір імен у вихідному коді. Крім того, вона поліпшує читаність коду, скорочуючи посилання.

Перетворення значення типу string у тип int. Введення користувача, завершується натисканням **Enter**. Отриманий результат методом **Console.ReadLine( )**

15: x = Convert.ToInt32(Console.ReadLine());

належить типу **string**. Будь-яке число, що вводиться, наприклад **432**, споконвічно розглядається як набір символів – таких же, як **ABC** або **#@$.**

Число, представлене рядком, не можна зберегти як тип **int**. Спочатку необхідно перетворити змінну типу **string** в змінну типу **int**.

Щоб перетворити введення користувача в число, в рядку 15 застосовується метод **ToInt32** класу **Convert**. Результат перетворення (праворуч від знака рівності в рядку 15) є числом і присвоюється змінній **x**. Природно, значення, введене користувачем, повинне збігатися з цілим числом. Введення **57,53** або **three hundred** викликає помилку, а значення **109** або **64732** припустимі.

Створення та виклик користувальницьких методів. Щоб зрозуміти зміст рядка 18, необхідно звернутися спочатку до рядків 25-31.

```
25:   public static int Sum(int a, int b)
26:   {
27:         int sumTotal;
28:
29:         sumTotal = a + b;
30:         return sumTotal;
31:   }
```

До цього моменту в програмах використовувалися готові методи на зразок **Console.ReadLine()** і **Console.WriteLine().** Незважаючи на величезну кількість вбудованих методів .**NET Framework**, а також доступність інших комерційних бібліотек класів, при створенні власного коду будуть потрібні й власні, визначені користувачем методи.

В рядках 25 – 31 міститься визначення користувальницького методу **Sum().**

При виклику метод **Sum()** одержує два числа. Він складає їх і повертає результат в точку виклику.

Визначення методу **Sum()** (заголовок, фігурні дужки { } і тіло методу) відповідає тій же загальній структурі, якій підпорядковується метод **Main**().

У загальному випадку класи представляють об'єкти, а методи — дії. Саме це варто відображати в їхніх назвах: для іменування класів (**Car**, **Airplane** і т. д.) використовують іменники, а для іменування методів (**DriveForward**, **MoveLeft**, **TakeOff** і т.д.) – дієслова.

Заголовок методу в рядку 25 багато в чому схожий на заголовок методу **Main** в рядку 9 (нагадаємо, що специфікатор доступу **public** включає метод в інтерфейс класу, якому той належить).

Далі, ключове слово **void** замінене на **int**, а в дужках після **Sum** перебувають конструкції, схожі на оголошення змінних. Це вимагає деякого пояснення.

Рядок 25 становить інтерфейс методу **Sum**. Він відповідає за взаємодію між методом, що викликає **Sum** (в даному випадку **Main**), і тілом **Sum**.

Інакше кажучи, інтерфейс – це властивість, що дозволяє окремим (і часто несумісним) елементам ефективно взаємодіяти.

На рис. 3.8. показаний процес виклику методу **Sum(int a, int b)**, оголошеного в рядку 25, з рядка 18 методу **Main()** як **Sum(x, y).**

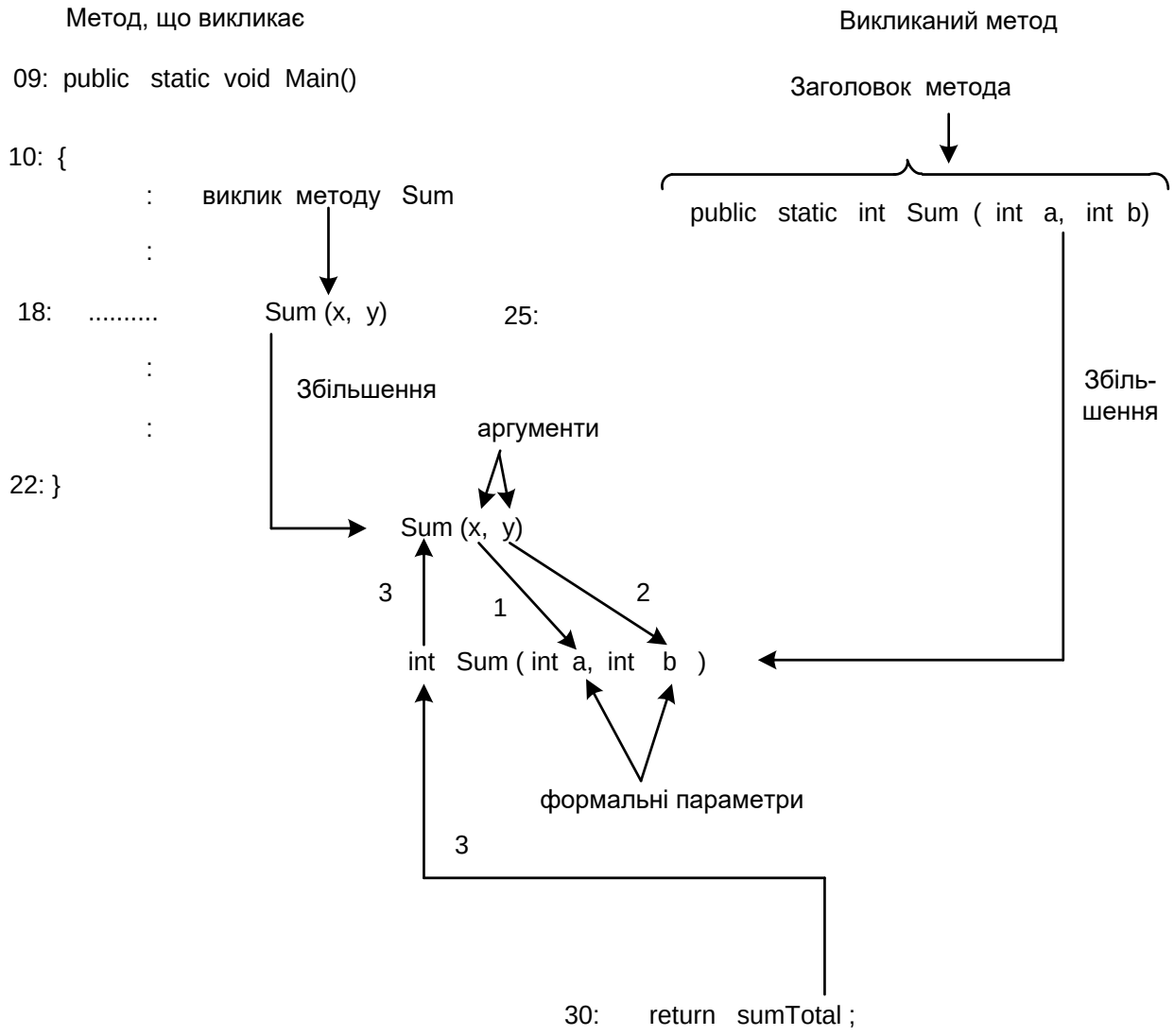


Рис. 3.8. **Виклик методу, визначеного користувачем**

Дві стрілки, позначені як "Збільшення", вказують, які частини показані докладніше.

Змінні **a** та **b** у заголовку методу **Sum** називаються формальними параметрами.

Формальний параметр застосовується в тілі методу для звертання до того значення, що було збережено в ньому в момент виклику методу. У даному випадку аргументи **x** та **y** в рядку 18 присвоюються параметрам **a** і **b** (це показано стрілками 1 і 2) так же, якби були виконані оператори присвоювання **a = x**; і **b = y**;.

Змінні **a** і **b** використовуються в тілі методу (в рядку 29) як звичайні оголошені змінні. Їхні значення рівні **x** та **y**.

Ключове слово **int** в рядку 25, що замінило **void**, указує, що метод **Sum** тепер повертає значення, яке належить типу **int**. Цьому відповідає ключове слово **return** в рядку 30.

30: return sumTotal;

Ключове слово **return** належить до операторів повернення керування. Коли запускається такий оператор, виконання методу зупиняється. Потік виконання вертається в точку виклику й підставляє вираження, зазначене після **return** (в даному випадку, **sumTotal**, див. стрілку 3 на рис. 3.8). Тому тип **sumTotal** повинен бути сумісним з типом, оголошеним у заголовку методу (тут **int**).

Коли перед ім'ям методу стоїть слово **void**, це значить, що по його завершенні в точку виклику не вертається ніяких даних.

Об'єднання рядків і викликів методів. Розглянемо в рядку 18 блок **Sum(x, y),** який, як уже показано, викликає метод **Sum**, визначений в рядках 25 – 31, і пересилає йому два аргументи **x** та **y**.

18: Console.WriteLine("Сума чисел дорівнює: " + Sum(x, y));

Після того як метод **Sum** повертає керування, можна фактично замінити **Sum(x, y)** значенням **sumTotal** з рядка 30.

Хоча конструкція **Sum(x, y)** перебуває усередині виклику іншого методу – **Console.WriteLine(),** — C# визначає, в якій послідовності повинні відбуватися виклики.

Тут виклик здійснюється тоді, коли потрібне значення **Sum(x. y)**. Після того як виконується код з рядків 25 – 31 і потік виконання вертається в рядок 18, його можна представити у вигляді

Console.WriteLine ("The sum is: " + 11);

Оскільки 11 (припускається, що раніше були введені числа a = 3, b = 8) перебуває усередині виклику методу **WriteLine**, це значення автоматично перетвориться в рядок. Таким чином, одержуємо наступний вид оператора:

Console.WriteLine("Сума чисел дорівнює: " + "11");

Коли + розташований між двома арифметичними операндами, він складає їх стандартним способом. Якщо ж він оточений рядками, компілятор C# змінює його функціональність. Він поєднує два рядки разом, даючи в результаті **("Сума чисел дорівнює: 11").**

Об'єднання двох рядків разом називається *конкатенацією*, а символ + в даному випадку – *операцією конкатенації*.

По завершенню, рядок 18 приймає вигляд:

Console.WrileLine ("Сума чисел дорівнює: 11").

Рядки 34 – 40 дуже схожі на рядки 25 – 31, єдине розходження полягає в імені методу та змінних. Крім того, метод **Product** обчислює не суму, а добуток двох чисел.

```
34:    public static int Product(int a, int b)
35:    {
36:            int productTotal;
37:
38:            productTotal = a * b;
39:            return productTotal;
40:    }
41: }
```

Концептуально, рядки 18 та 19 ідентичні.

```
18:            Console.WriteLine("Сума чисел дорівнює: " + Sum(x, y));
19:            Console.WriteLine("Добуток чисел дорівнює: " + Product(x, y));
```

Порядок, в якому визначені методи класу, є довільним і не пов'язаний з тим, як вони викликаються з вихідного коду. Наприклад, можна змінити порядок визначень методів **Sum** і **Product** в класі **SimpleCalculator**.

Клас Math

Рядок 20 схожий з 18 та 19, однак в ньому для знаходження найбільшого з двох чисел застосовується метод **Max** класу **Math** простору імен **System** бібліотеки класів NET.Framework.

20: Console.WriteLine("Максимальне число дорівнює: " + Math.Max(x, y));

Перелік часто використовуваних методів класу **Math** наведений нижче:

```
public static decimal Abs(
   decimal value
);
public static double Acos(
   double d
);
public static double Asin(
   double d
);
public static double Atan(
   double d
);
public static double Cos(
   double d
);
public static double Cosh(
   double value
);
public static double Exp(
   double d
);
public static double Floor(
   double d
);
public static decimal Max(decimal, decimal);
public static double Max(double, double);
public static int Max(int, int);
public static decimal Min(decimal, decimal);
```

```
public static double Min(double, double);
public static int Min(int, int);
public static float Min(float, float);
public const double Pi;
public static double Pow(
  double x,
  double y
);
public static decimal Round(decimal);
public static double Round(double);
public static decimal Round(decimal, int);
public static double Sin(
  double a
);
public static double Sinh(
  double value
);
public static double Sqrt(
  double d
);
public static double Tan(
  double a
);
public static double Tanh(
  double value
);
```

Призначення кожного з методів очевидно – воно визначається абревіатурою його назви й тому тут не приводиться.

Питання для самоконтролю

1. Опишіть відомі вам алгоритмічні структури.
2. Дайте огляд основних операцій C#.
3. Розкрийте сутність використання синтаксичних блоків.
4. Навіщо потрібні логічні операції. Наведіть приклади.
5. Що таке пріоритети операцій? Де і коли вони використовуються?

6. Розкрийте суть понять: простір імен, область видимості змінних, область видимості і час існування змінних.

7. Напишіть програму обчислення суми, добутку, максимуму і мінімуму трьох чисел.

8. Як перетворити значення типу string в тип int?

9. Опишіть загальну схему створення і виклику призначених для користувача методів.

10. Охарактеризуйте клас Math. Наведіть приклад програми, де використовуються вбудовані математичні методи.

4. Оператори керування програмою

4.1. Структури вибору альтернатив

4.1.1. Поняття потоку керування програмою

В розглянутих лінійних програмах звичайно всі оператори в методі виконувалися послідовно. Програма, заснована лише на послідовному виконанні, завжди виконує ті самі дії. Вона не здатна реагувати на поточні умови.

Однак часто програмам потрібно змінювати потік керування, реагуючи на якісь зовнішні події.

Потік керування становить порядок, в якому виконуються оператори програми. Крім того, часто використовуються терміни “порядок виконання” і “керуючий потік”.

Приклад програми з лістингу 2.2 (див. розділ 2) дозволяв користувачеві впливати (введенням **yes** або **no**) на потік керування й здійснювати виведення повідомлення "Hello World!". Така логіка вимагає можливості робити вибір між двома або декількома гілками на основі умови. Оператор **if**, представлений в програмі лістингу 2.2, разом з оператором **switch**, про який буде сказано далі, формують групу виражень, призначених саме для цієї мети.

Гілкою називають сегмент програми, що містить один оператор або їх групу. Оператор розгалуження дозволяє запускати потрібний блок операторів. Вибір здійснюється за умовою. Оператори розгалуження часто називають *операторами вибору.*

C# забезпечує три типи структур вибору альтернатив:

єдиний вибір – структура **if** (ЯКЩО);
подвійний вибір – структура **if / else** (ЯКЩО / ІНАКШЕ);
множинний вибір – структура **switch**.

4.1.2. Структура вибору if

Синтаксис оператора **if** відображений у наступному синтаксичному блоці:

Оператор_if ::=
if(<Умова>)
<Оператор>;

Вираження логічного типу **(<Умова>)** завжди дає одне з двох значень: **true** (істина) або **false** (неправда).

<Оператор>, що слідує за логічним вираженням, виконується лише в тому випадку, якщо останнє істинно.

Графічна схема оператора наведена на рис. 4.1.

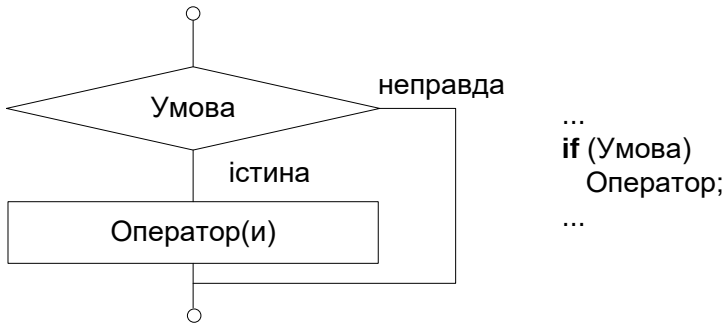


Рис. 4.1. **Графічна схема оператора if**

Приклад. Перевірка правильності введення змінної, яка може містити числа від 1 до 31.

```
using System;

    class Class1
    {
        static void Main( )
        {
            int valor;
```

```
                Console.WriteLine("Введіть число місяця");
                valor = Convert.ToInt32(Console.ReadLine());
                if (valor < 1  ||  valor >31)
                        Console.WriteLine("Помилка введення!");
                Console.WriteLine("Ви ввели число, рівне {0}", valor);
        }
    }
```

Результат роботи програми:

```
Введіть число місяця
17
Ви ввели число, рівне 17
Press any key to continue
Введіть число місяця
32
Помилка введення!
Ви ввели число, рівне 32
Press any key to continue
```

Як оператори не можна використовувати оголошення і визначення. Однак тут можуть бути складені оператори і блоки:

Складений_оператор ::=
{
<Оператори>
}

4.1.3. Структура вибору if / else

Синтаксичний блок оператора **if / else**:

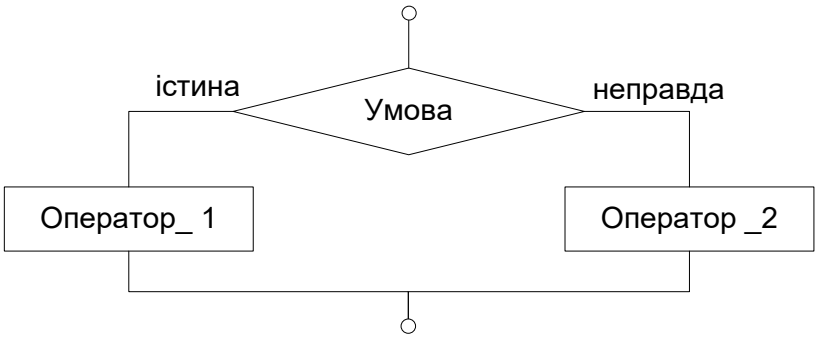
Оператор if / else ::=
if (<Умова>)
<Оператор_1>; | <Складений_оператор_1>
else
<Оператор_2>; | <Складений_оператор_2>

<Оператор_1>; | <Складений_оператор_1> виконується лише в тому випадку, коли логічне вираження **(<Умова>)** дорівнює **true**.

<Оператор_2>; | <Складений_оператор_2> виконується лише тоді, коли **(<Умова>)** дорівнює **false**.

Перед **else** обов'язково ставиться крапка з комою.

Графічна схема оператора наведена на рис. 4.2.




```
...
if (Умова)
  Оператор_1;
else
  Оператор_2;
...
```

Рис. 4.2. **Графічна схема оператора if / else**

***Приклад*.** Знайти мінімум із двох чисел.

```
using System;
    class Class1
    {
        static void Main()
        {
            int x, y, min;
            Console.WriteLine("Послідовно введіть два цілих числа");
            x = Convert.ToInt32(Console.ReadLine());
            y = Convert.ToInt32(Console.ReadLine());
            if  (x<y)
                min = x;
            else
                min = y;
            Console.WriteLine("Мінімальне число дорівнює {0}", min);
        }
    }
```

Оператори **if** можуть бути вкладені друг у друга.

Приклад. Знайти максимальне число із трьох чисел a, b, c.

```
using System;
    class Class1
    {
        static void Main()
        {

            int a, b, c, max;
            Console.WriteLine("Послідовно введіть три цілих числа");
            a = Convert.ToInt32(Console.ReadLine());
            b = Convert.ToInt32(Console.ReadLine());
            c = Convert.ToInt32(Console.ReadLine());
            if  (a > b  &&  a > c)
                max = a;
            else
                if  (b > c)
                max = b;
            else
                max = c;
            Console.WriteLine("Максимальне число дорівнює {0}", max);
        }
    }
```

Обидві гілки повного умовного оператора можуть бути складеними.

4.1.4. Множинний вибір – структура switch

Оператор **switch** дозволяє програмі обрати одну з кількох дій на основі значення заданого вираження. Логіка, реалізована **switch**, подібна до логіки оператора **if / else**.

Синтаксичний блок оператора **switch:**

```
switch (<вираження_switch >)
{
    case <Константне_вираження>:
        [ <Оператор>;
```

<Оператор>; <Оператор>;
<Оператор_break>; | <Оператор_goto>]

case <Константне_вираження>:
[<Оператор>;
<Оператор>; <Оператор>;
<Оператор_break>; | <Оператор_goto>]

<Будь-яка кількість блоків case>

[default:
[<Оператор>;
<Оператор>; <Оператор>;
<Оператор_break>; | <Оператор_goto>]

}

Графічна схема оператора **switch** наведена на рис. 4.3.

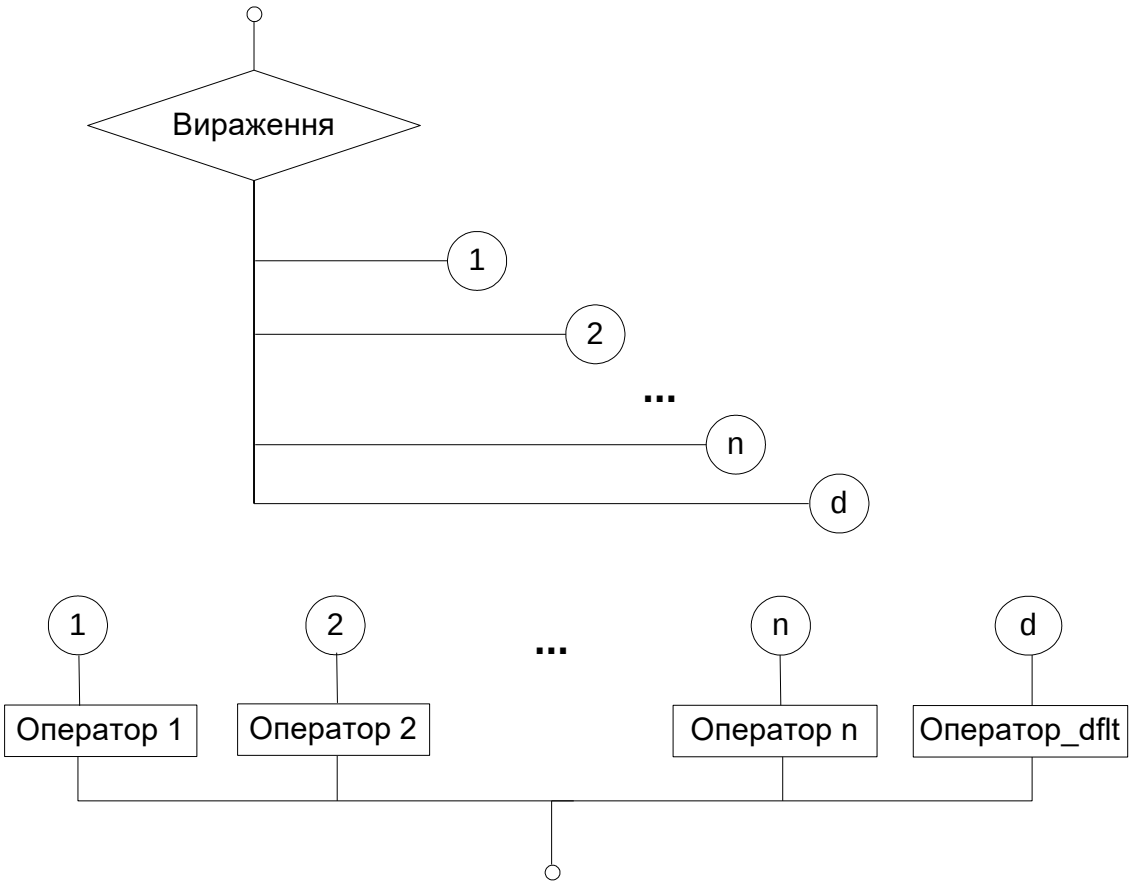


Рис. 4.3. **Графічна схема оператора switch**

Може бути один або не бути жодного блоку **default**.

Термін (<**вираження_switch**>), використовуваний разом із ключовим словом **switch**, – це загальноприйнята назва керуючого вираження.

Розділи **case** і **default** називають розділами вибору.

<**Константне_вираження**>, що йде услід за ключовим словом **case**, називають значенням або **case**-міткою. Кожна з них повинна бути унікальною.

Найпоширеніший спосіб завершення розділу вибору – застосування <**Оператор_break**> або <**Оператор_goto**>.

Хоча розділи **case** і **default** можна розміщати в будь-якій послідовності, гарний стиль програмування передбачає, що розділ **default** розміщується наприкінці оператора **switch**.

Коли потік керування переходить від одного розділу **switch** до іншого, таке виконання називається *провалом*. Для його запобігання застосовується оператор **break** або **goto**.

Приклад. Необхідно проаналізувати значення змінної **nota**, що є виставленою оцінкою.

```
using System;
class Nota
{
    public static void Main()
    {
        int nota;

        Console.WriteLine("Ваша оцінка (від 2 до 5)? ");
        nota = Convert.ToInt32(Console.ReadLine());

        switch(nota)
        {
                    case 2:
                            Console.WriteLine("Ви вибрали 2 ");
                            Console.WriteLine("Оцінка - незадовільно");
                            break;
                    case 3:
```

```
                    Console.WriteLine("Ви вибрали 3 ");
                    Console.WriteLine("Оцінка - задовільно");
                    break;
                case 4:
                    Console.WriteLine("Ви вибрали 4 ");
                    Console.WriteLine("Оцінка - добре");
                    break;
                case 5:
                    Console.WriteLine("Ви вибрали 5 ");
                    Console.WriteLine("Оцінка - відмінно");
                    break;
                default:
                    Console.WriteLine("Помилка вибору. Ви повинні
вибрати число " +
                    "між 2 and 5");
                    break;
            }
        }
}
```

Результат:

```
Ваша оцінка (від 2 до 5)?
4
Ви вибрали 4
Оцінка - добре
Press
```

Приклад. Створення простого меню.

```
using System;
    class Class1
    {
        static void Main()
        {
            char vibor;
            Console.Write("МЕНЮ:\t A(dd) D(elete) S(ort) Q(uit) \n");
            Console.Write("Ваш вибір? ");
```

```
                vibor = Convert.ToChar(Console.Read());
                switch (Char.ToUpper(vibor))
                {
                        case 'A':
                                Console.Write("Обрано Add\n");
                                break;
                        case 'D':
                                Console.Write("Обрано Delete\n");
                                break;
                        case 'S':
                                Console.Write("Обрано Sort\n");
                                break;
                        case 'Q':
                                Console.Write("Обрано Quit\n");
                                break;
                        default:
                                Console.Write("Введений помилковий символ\n");
                                break;
                }
        Console.Write("\nДля завершення програми натисніть <Enter>");
        Console.ReadLine();  // для паузи
        Console.ReadLine();  // для паузи
        }
}
```

Один з можливих результатів роботи програми:

```
МЕНЮ:    A(dd) D(elete) S(ort) Q(uit)
Ваш вибір? a
Обрано Add

Для завершення програми натисніть <Enter>
```

4.1.5. Умовне вираження

В C# є ще одна скорочена форма умовного оператора – так зване умовне вираження. Його синтаксис:

<Логічне_вираження_1> ? <Вираження_2> : <Вираження_3>;

Сенс цієї конструкції полягає в наступному:

обчислюється **<Логічне_вираження_1>**; якщо воно істинне (**true**), то результатом буде **<Вираження_2>**, у противному випадку результатом буде **<Вираження_3>**.

По суті, умовне вираження еквівалентне наступному умовному операторові:

```
if ( Логічне_вираження_1 )
      Вираження_2;
else
      Вираження_3;
```

Приклад. Знайти максимум із двох чисел.

```
…
int x = 13, y = 7, max;
max  =  (x > y) ? x : y;
Console.WriteLine("max = {0}", max);
…
```

4.2. Структури повторення

4.2.1. Термінологія

Оператори циклу використовуються для організації багаторазово повторюваних обчислень.

Будь-який цикл складається з:

тіла циклу, тобто тих операторів, які виконуються кілька разів;

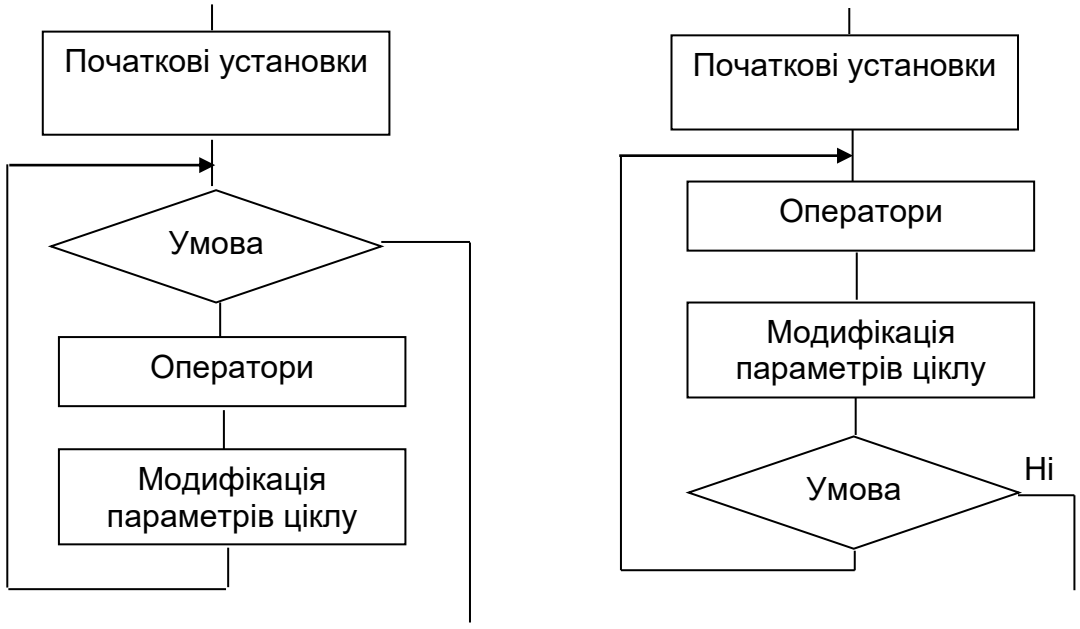
початкових установок;

модифікації параметра циклу;

перевірки умови продовження виконання циклу.

Один прохід циклу називається *ітерацією*. Перевірка умови виконується на кожній ітерації або до тіла циклу (*цикл із передумовою*), або після тіла циклу (*цикл із постумовою*).

Графічні схеми, що демонструють роботу циклу з перед- (а) та постумовою (б) показані на рис. 4.4.



а) цикл із передумовою б) цикл із постумовою

Рис. 4.4. **Графічні схеми, що демонструють роботу циклів**

Різниця між ними полягає в тому, що тіло циклу з постумовою завжди виконується хоча б один раз, після чого перевіряється, чи треба його виконувати ще раз. Перевірка необхідності виконання циклу з передумовою робиться до тіла циклу, тому можливо, що він не виконається жодного разу.

Змінні, які змінюються в тілі циклу та використовуються при перевірці умови продовження, називаються *параметрами циклу*.

Цілочисельні параметри циклу, що змінюються з постійним кроком на кожній ітерації, називаються *лічильниками циклу*.

Початкові установки можуть явно не бути присутніми в програмі, їх сенс полягає в тому, щоб до входу в цикл задати значення змінним, які в ньому використовуються.

Цикл завершується, якщо умова його продовження не виконується.

Можливо примусове завершення як поточної ітерації, так і циклу в цілому. Для цього служать оператори **break**, **continue, return, goto**. Передавати керування ззовні усередину циклу не рекомендується.

В C# є чотири різних оператори циклу – **while, do while, for, foreach.**

4.2.2. Цикл із передумовою (while)

Цикл із передумовою реалізує графічну схему (без блоку початкових установок), наведену на рис. 4.4-а, і має наступний синтаксис:

Оператор_while ::=
while (<Умова_циклу>)
<Тіло_циклу>

де:

<Умова_циклу> ::= <Логічне_вираження>
<Тіло_цикпу> ::= <Оператор>;
::= <Складений_оператор>

Тіло циклу повторюється доти, доки **<Умова_циклу>** істинне (**true**).

Виконання оператора починається з обчислення логічного вираження **(<Умова_циклу>)**. Якщо воно істинне (не дорівнює **false**), виконується **<Тіло_циклу>**.

Якщо при першій перевірці вираження дорівнює **false**, цикл не виконується жодного разу. Логічне вираження обчислюється перед кожною ітерацією циклу.

Приклад. Виведення алфавіту.

```
using System;
    class Class1
    {
        static void Main()
        {

            char c;
            Console.WriteLine("Виведення алфавіту:\n");
            c = 'A';
            while ( c <= 'Z' )
            {
                Console.Write("{0} ", c );
                c++;
            }
```

```
            Console.WriteLine(); // Переведення на новий рядок
        }
}
```

Результат:
Виведення алфавіту:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Press any key to continue

Розповсюджений прийом програмування – організація безкінечного циклу з заголовком **while (true)** і примусовим виходом з тіла циклу після виконання якої-небудь умови.

У круглих дужках після ключового слова **while** можна вводити опис змінної. Областю її дії є цикл, наприклад:

while (int x = 0) { ... /* область дії x */ }

Приклад. Обчислення вираження f(x) = 1 + 1/2 + 1/3 + ... + 1/n.

```
using System;
    class Class1
    {
        static void Main()
        {
            int n;
            double f;
            Console.WriteLine("Введіть n (ціле)");
            n = Convert.ToInt32(Console.ReadLine());
            f = 0;
        while (n > 0)
            {
                f +=1.0 / n;
                n--;
            }
        Console.WriteLine("Значення функції дорівнює {0:N3}", f);
        }
}
```

Результат:
Введіть n (ціле)
10
Значення функції дорівнює 2,929
Press any key to continue

Приклад. Організація лічильника повторень.

```
using System;
class Exelent
{
	public static void Main()
	{
		int counter;
		Console.WriteLine("Я буду відмінником!");
		Console.Write("Скільки разів повторити цю фразу?");
		counter = Convert.ToInt32(Console.ReadLine());
		while(counter > 0)
		{
			Console.WriteLine("Я буду відмінником!");
			counter--;
		}
		Console.WriteLine("Буде так!!!");
	}
}
```

Результат:
Я буду відмінником!
Скільки разів повторити цю фразу? 5
Я буду відмінником!
Я буду відмінником!
Я буду відмінником!
Я буду відмінником!
Я буду відмінником!
Буде так!!!
Press any key to continue

Приклад. Угадай число. Користувачеві дається 10 спроб для вгадування заданого в програмі числа 25**.**

```
using System;
class De_que_numero
{
	public static void Main()
	{

		int	i = 1,  respuesta = 1;
		Console.WriteLine("Введіть ціле число в діапазоні від 1 до 50");
		while ( i++ <=10   &&  respuesta  !=25 )
				{
				Console.WriteLine("{0} спроба...", i-1);
				respuesta=Convert.ToInt32(Console.ReadLine());
				}
		if  ( i  ==  12 )
			Console.WriteLine("На жаль..., не вгадали.");
		else
			Console.WriteLine("Поздоровляємо, Ви вгадали число!");
	}
}
```

Результат:

```
Введіть ціле число в діапазоні від 1 до 50
1 спроба...
23
2 спроба...
5
3 спроба...
6
4 спроба...
5
5 спроба...
6
6 спроба...
25
```

Поздоровляємо, Ви вгадали число!
Press any key to continue

4.2.3. Цикл із постумовою (do while)

Цикл із постумовою реалізує графічну схему (без блоку початкових установок), наведену на рис. 4.4-б, і має вигляд:

Оператор_do_while ::=
do
<Тіло_цикпу>
while (<Умова_циклу>);

де:
<Тіло_циклу> ::= <Оператор>;
::= <Складений_оператор>
<Умова_циклу>::= <Логічне_вираження>

<Тіло_циклу> повторюється до тих пір, поки **<Умова_циклу>** дорівнює **true**.

Спочатку виконується простий або складений оператор, що складають тіло циклу, а потім обчислюється логічне вираження ***(<Умова_циклу>)***. Якщо воно істинно (не дорівнює **false**), тіло циклу виконується ще раз.

Цикл завершується, коли **(<Умова_циклу>)** стане рівною **false** або в тілі циклу буде виконаний який-небудь оператор передачі керування.

Приклад. Виведення алфавіту.

```
using System;
class Class1
{
    static void Main()
    {
        char c = 'A';
        Console.WriteLine(" Виведення алфавіту:\n");
        c = Convert.ToChar(Convert.ToInt32('A')-1);
        do
```

```
            {
                c++;
                Console.Write("{0} ", c );
            } while ( c < 'Z' );
            Console.WriteLine(); // Переведення на новий рядок
        }
}
```

Результат:
Виведення алфавіту:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Press any key to continue

Приклад. Угадай букву.

```
using System;
class AlphabetGame
{
        public static void Main()
        {
            string secretLetter;
            string letterGuess;
            Random Randomizer = new Random();
            // Випадковий вибір secretLetter з повного
            // алфавіту  від Unicode 65 ('A') до  Unicode 91 ('Z')

            secretLetter = ((char)Randomizer.Next(65, 91)).ToString();

Console.WriteLine("Угадай мою букву \n" +
"Я буду повідомляти, якщо твоя буква буде НИЖЧЕ задуманої мною \n" +
                    "або ВИЩЕ (перед моєю) за абеткою");

do
            {
                letterGuess = Console.ReadLine().ToUpper();
```

```
                // secretLetter до або після letterGuess?
                if (secretLetter.CompareTo(letterGuess) < 0)
                        Console.WriteLine("   шукай ВИЩЕ\n");
                if (secretLetter.CompareTo(letterGuess) > 0)
                        Console.WriteLine("   шукай НИЖЧЕ\n");
        } while (secretLetter != letterGuess);

        Console.WriteLine("УГАДАВ!!! \n\nКінець гри");
    }
}
```

Результат:

```
Угадай мою букву
Я буду повідомляти, якщо твоя буква буде НИЖЧЕ задуманої мною
або ВИЩЕ (перед моєю) за абеткою
m
  шукай ВИЩЕ

g
  шукай ВИЩЕ

d
  шукай НИЖЧЕ

e
УГАДАВ!!!

Кінець гри
Press any key to continue
```

4.2.4. Оператор циклу for

Цей оператор використовується в тих випадках, коли кількість повторень тіла циклу заздалегідь відома або може бути визначена програмним шляхом.

Його формат:

for([<Оператори_ініціалізації>];
[<Умова_циклу>]; [<Оператори_оновлення>])
<Тіло_циклу>
де:
<Тіло_циклу> ::= <Оператор>;
::= <Складений_оператор>
<Оператор_ініціалізації> ::= <Оператор_ініціалізації1>,
<Оператор_ініціалізації2>...
<Умова_циклу> :: = <Логічне_вираження>
<Оператори_оновлення> ::= <Оператор_оновлення1>,
<Оператор_оновлення2>...

<Оператор_ініціалізації> використовується для оголошення та присвоєння початкових значень величинам, що використовуються в циклі. В цій частині можна записати декілька операторів, розділених комами.

Областю дії змінних, оголошених в частині ініціалізації циклу, є цикл.

Ініціалізація виконується один раз на початку виконання циклу.

<Умова_циклу> визначає умову виконання циклу: якщо його результат дорівнює **true**, цикл виконується.

<Оператори_оновлення> виконуються після кожної ітерації циклу й служать звичайно для зміни параметрів циклу. В частині модифікацій можна записати декілька операторів через кому.

Простий або складений оператор становить тіло циклу. Будь-яка з частин оператора **for** може бути опущена (але крапки з комою треба залишити на своїх місцях!).

Цикл з параметром реалізований як цикл із передумовою (див. рис. 4.4-а).

Приклад. Розрахунок площі.

```
using System;
class AreaCalculator
{
        public static void Main()
        {
                int i;
```

```
            int width;
            int height;

            Console.WriteLine("Висота  Ширина    Площа\n");
            for (height = 1000, width = 100, i=0; i <= 10;
                  height = height + 100, width = width + 10, i++)
            {
                  Console.WriteLine("{0}     {1}      {2}",
                        height, width, height * width);
            }
      }
}
```

Результат:

```
Висота  Ширина    Площа

1000       100       100000
1100       110       121000
1200       120       144000
1300       130       169000
1400       140       196000
1500       150       225000
1600       160       256000
1700       170       289000
1800       180       324000
1900       190       361000
2000       200       400000
Press any key to continue
```

4.3. Керуючі оператори в циклах

Крім умовних операторів і операторів циклу, існують ще три оператори, призначених для цих же цілей. Вони застосовуються рідше, тому що часте їх використання погіршує наочність програми й збільшує ймовірність помилок.

4.3.1. Оператор break

Може використовуватися в циклах усіх трьох типів.

Виконання оператора **break** призводить до виходу із циклу, в якому він міститься, і переходу до оператора за циклом, що є наступним.

Якщо оператор **break** перебуває усередині вкладених циклів, то його дія поширюється тільки на той цикл, безпосередньо в якому він знаходиться.

Приклад. Угадай число.

Потрібно визначити задумане (визначене в програмі) число з 3-х спроб.

У цьому прикладі при вгадуванні числа 12 відбувається припинення виконання циклу за допомогою оператора **break**.

```
using System;
class De_que_numero
{
	public static void Main()
	{
		int	i = 1,  respuesta = 1;
		int kol = 3; // кількість спроб = 3
		Console.WriteLine("Введіть ціле число в діапазоні від 1 до 15. " +
                                        "У вас є {0} спроби", kol);
		while ( i <= kol )
		{
			Console.WriteLine("{0} спроба...", i);
			respuesta=Convert.ToInt32(Console.ReadLine());
			if  ( respuesta  ==  12 )
				break;
			Console.WriteLine("На жаль..., не вгадали.");
			i++;
		}
		if(i<=kol)
			Console.WriteLine("Поздоровляємо, Ви вгадали число!");
			Console.WriteLine("Гра закінчена. Удачі!");
	}
}
```

Один із можливих результатів:

```
Введіть ціле число в діапазоні від 1 до 15. У вас є 3 спроби
1 спроба...
4
На жаль..., не вгадали.
2 спроба...
6
На жаль..., не вгадали.
3 спроба...
8
На жаль..., не вгадали.
Гра закінчена. Удачі!
Press any key to continue
```

4.3.2. Оператор continue

Оператор **continue** може використовуватися тільки серед операторів тіла циклу. Він викликає пропуск частини ітерації усередині циклу, що залишилася, й перехід до наступної ітерації.

Приклад. Контроль введення значень дня місяця.

Вводяться числа місяця для обробки. Необхідно здійснити перевірку правильності введення. Число 31 означає кінець обробки.

```
using System;
class Class1
{
	static void Main()
	{
		int dia = 1;
		while( dia !=31 )
			{
				Console.WriteLine("Введіть день місяця");
				dia = Convert.ToInt32(Console.ReadLine());
				if( dia < 1  ||  dia >=31 )
					continue;
				Console.WriteLine("Обробка введеної дати  {0}",dia);
			}
```

```
            Console.WriteLine("Кінець обробки");
        }
}
```

Результат:

```
Введіть день місяця
3
Обробка введеної дати  3
Введіть день місяця
35
Введіть день місяця
2
Обробка введеної дати  2
Введіть день місяця
78
Введіть день місяця
9
Обробка введеної дати  9
Введіть день місяця
31
Кінець обробки
Press any key to continue
```

У даному прикладі неправильне введення значення призводить до пропуску частини ітерації, призначеної для обробки цього значення.

Помітимо, що така зміна умови

if (dia < 1 || dia > 31) на зворотне

if (dia > 0 && dia < 32)

дозволяє виключити використання оператора **continue**.

З допомогою оператора **continue** іноді можна скоротити деякі програми, особливо якщо вони містять у собі вкладені оператори **if else.**

Якщо **continue** використовується в циклі **for**, то треба мати на увазі наступне: перед початком нової ітерації спочатку обчислюється вираження модифікації (знаходиться нове значення параметра циклу) і лише потім керування передається на заголовок.

Приклад. Застосування операторів **continue** і **break** в циклі **for.**

```
using System;
        class Class1
        {
                static void Main()
                {
                        int  i;
                        Console.WriteLine("Стартує цикл із continue");
                        for  ( i=1; i<=10; i++ )
                        {
                                if  ( i<5)  continue;
                                        Console.WriteLine("{0} ",i);
                        }
                        Console.WriteLine("Значення і після циклу дорівнює {0}",i);
                        Console.WriteLine("Стартує цикл із break");
                        for  ( i=1; i<=10; i++ )
                        {
                                if  ( i>5)  break;
                                Console.WriteLine("{0} ",i);
                        }
                        Console.WriteLine("Значення і після виходу дорівнює {0}",i);
                }
}
```

Результат:

```
Стартує цикл із continue
5
6
7
8
9
10
Значення і після циклу дорівнює 11
Стартує цикл із break
1
2
3
4
5
```

Значення і після виходу дорівнює 6
Press any key to continue

4.3.3. Оператор goto (перехід на задану мітку)

Оператор goto є поганим засобом. Його використання призводить до значного ускладнення логіки програми.

Існує лише один випадок, коли програмісти – професіонали допускають використання **goto**, – це вихід із вкладеного набору циклів при виявленні помилок (**break** дає можливість виходу лише з одного циклу).

4.3.4. Вкладені цикли

Вкладеним циклом називають конструкцію, в якій один цикл виконується всередині іншого.

Внутрішній цикл виконується повністю під час кожної з ітерацій зовнішнього циклу.

Приклад. Потрібно заповнити заданий прямокутник символами “*”.

```
using System;
class StarsTwoDimensions
{
	public static void Main()
	{
		for (int i = 0; i < 5; i++)
		{
			for (int j = 0; j < 7; j++)
			{
				Console.Write("*");
			}
			Console.WriteLine();
		}
	}
}
```

Результат:

```
*******
*******
*******
*******
*******
Press any key to continue
```

У програмі можна використовувати будь-які комбінації вкладених циклів всіх типів: **for, do / while, while**, якщо цього вимагає логіка побудови програми.

Приклад. Здійснити введення п'яти значень днів місяця з перевіркою правильності введення.

```
using System;
class Class1
{
	static void Main()
	{
		int dia;
		for  (int i=1; i <= 5; i++ )
		{
			do
			{
				Console.WriteLine("Введіть день місяця");
				dia = Convert.ToInt32(Console.ReadLine());
			}
			while  ( dia < 1  ||  dia >31 );
				Console.WriteLine("Обробка введеної дати  {0}",dia);
		}
		Console.WriteLine("Кінець обробки");
	}
}
```

Результат:

```
Введіть день місяця
3
```

```
Обробка введеної дати  3
Введіть день місяця
17
Обробка введеної дати  17
Введіть день місяця
32
Введіть день місяця
1
Обробка введеної дати  1
Введіть день місяця
0
Введіть день місяця
7
Обробка введеної дати  7
Введіть день місяця
9
Обробка введеної дати  9
Кінець обробки
Press any key to continue
```

Зовнішній цикл (for) виконується 5 разів, а внутрішній (do / while) буде виконуватися доти, доки не буде введене правильне значення дня місяця.

4.3.5. Рекомендації з вибору циклів

Оператори циклу взаємозамінні. Нижче наведені деякі рекомендації з вибору найкращого оператора циклу в кожному конкретному випадку:

оператор **do while** звичайно використовують, коли цикл потрібно обов'язково виконати хоча б раз (наприклад, якщо в циклі здійснюється введення даних);

оператор **for** переважніше в більшості інших випадків (однозначно — для організації циклів з лічильниками);

оператором **while** зручніше користуватися у випадках, коли число ітерацій заздалегідь не відомо, очевидних параметрів циклу немає, або модифікацію параметрів зручніше записувати не наприкінці тіла циклу.

Оператор **foreach** буде розглянутий при обробці масивів.

Питання для самоконтролю

1. Дайте поняття потоку управління програмою.
2. Що становить собою структура вибору if, коли її треба використовувати?
3. Опишіть структуру вибору if / else.
4. У чому полягає специфіка множинного вибору і як цей вибір доцільно реалізувати за допомогою структури “switch”?
5. Умовний вираз. Наведіть його синтаксис і відповідний приклад використання.
6. Дайте загальний огляд структур повторення.
7. Опишіть особливості використання циклу з передумовою (while). Наведіть приклади.
8. Коли доцільно використовувати цикл з постумовою (do while). Наведіть приклади.
9. Дайте синтаксис оператора циклу for.
10. Які управляючі оператори можуть використовуватися в циклах?
11. Опишіть оператори break і continue.
12. У чому полягають особливості застосування оператора continue в циклі for?
13. Охарактеризуйте оператор goto.
14. Що таке вкладені цикли і коли їх доцільно використовувати?
15. Наведіть загальні рекомендації щодо вибору циклів.
16. Розробити алгоритм (графічну схему) і написати програму для обчислення наступного вираження:

$Y=1 + x/1! + x2/2! + \ldots + x_n/n!$ с заданою точністю eps = 0.1 e-3

Модуль 2. Організація даних

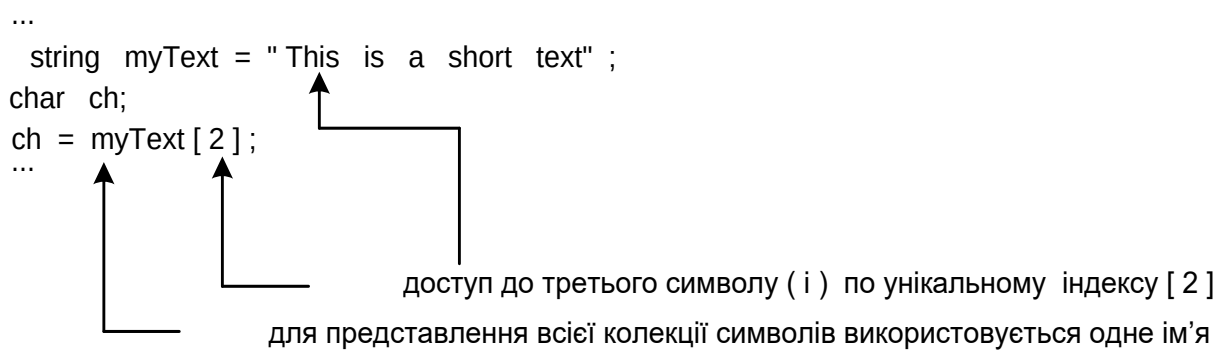
5. Масиви

5.1. Загальні відомості про масиви

5.1.1. Оголошення й визначення масиву

Тип масиву дуже схожий на тип **string** — саме тому тип **string** часто іменується як масив символів.

Змінна типу **string** може містити набір чітко розташованих і проіндексованих символів. Доступ до кожного з них здійснюється наступним чином (рис. 5.1):



або:

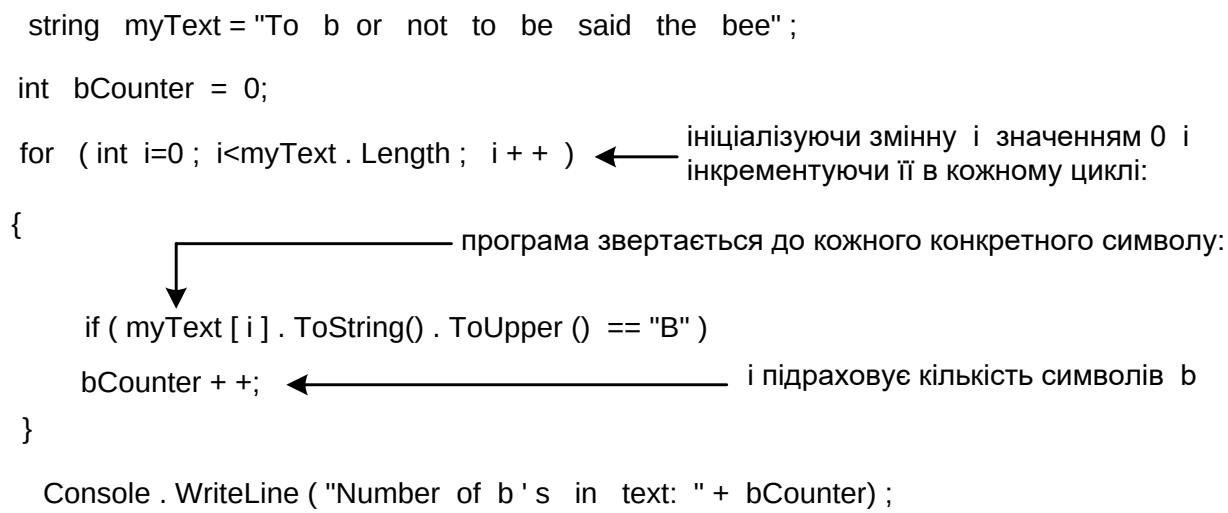


Рис. 5.1. **Інтерпретація типу string як масиву символів**

У цьому фрагменті коду підраховується кількість елементів **b** в рядку **myText** і виводиться повідомлення:

Number of b's in text: 3

Рядок **string** і масив є як *типами-класами*, так і *посилальними* типами.

Як **System.String** є базовим класом для типу **string**, так і **System.Array** представляє собою основу для масиву.

Отже, масив — це об'єкт, що, подібно об'єкту **string**, має багато вбудованих методів, використовуваних для доступу до елементів колекцій даних та їхньої обробки.

Між рядками **string** і масивами є чимало подібності, але в той час як тип **string** обмежується лише представленням колекцій символів, *масив може містити колекцію величин будь-якого типу*, включаючи який-небудь із простих типів **int**, **long**, **decimal** і т. д., а також будь-яку групу об'єктів.

У загальному випадку **масив** є іменованою структурою даних (або об'єктом), для якої задана відповідність між множиною індексів і комірок, званих **елементами масиву**.

Усі елементи масиву повинні належати тому самому типу. Тип елементів масиву називається типом масиву, або **базовим типом масиву.**

Елементи масиву іноді називають **індексованими змінними**, або просто елементами.

Змінна типу **array** оголошується у вихідній програмі за допомогою зазначення масиву, що супроводжується порожньою парою квадратних дужок і ім'ям масиву. В наступному рядку

decimal [ ] accountBalances;

оголошується змінна **accountBalances**, що здатна зберігати посилання на об'єкт масиву, що містить колекцію чисел типу **decimal**.

Попереднім оголошенням створюється порожній контейнер, здатний зберігати посилання на об'єкт масиву. Сам об'єкт масиву з колекцією величин типу **decimal** ще не створений, і пам'ять під нього не виділена.

Об'єкт типу **array**, подібно будь-якому іншому класу (за винятком **string**), повинен створюватися із застосуванням ключового слова **new**.

В рядку коду на рис. 5.2. створюється об'єкт масиву (класу **System.Array**), що містить колекцію з п'яти величин типу **decimal**. Посилання на нього присвоюється змінній **accountBalances**.

Рис. 5.2. **Створення об'єкта масиву з п'ятьома елементами**

Оператори оголошення й присвоювання можна об'єднати в одному рядку, як показано нижче:

decimal [ ] accountBalances = new decimal [5];

Іноді можна зустріти синтаксис оголошення змінних масиву і створення нових об'єктів, що відрізняється від розглянутого тут.

У ньому прямо використовуються посилання на імена класів і методів .NET Framework, наприклад:

```
System.Array rainfall = System.Array.CreateInstance(
                        Type.GetType("System.Decimal"), 5 );
```

де оголошується змінна **rainfall**, якій присвоюється посилання на новий об'єкт базового типу **decimal**.

Після того як оголошена змінна масиву **accountBalances** і їй присвоєний новий об'єкт масиву, одержуємо поточний стан, показаний на рис. 5.3.

Тепер **accountBalances** містить посилання, що вказує на конкретний об'єкт класу **System.Array**, розташований в оперативній пам'яті. Він може містити колекцію з п'яти величин типу **decimal**. Оскільки кожна з них вимагає 128 біт (або 16 байт), ця частина об'єкта масиву займе в пам'яті 5 x 128 = 640 біт (або 80 байт). Саме посилання вимагає 4 байти пам'яті.

Розмір масиву (або довжина масиву) – це загальне число елементів, які масив може містити.

При створенні масиву можна визначати елементи і присвоювати їм початкові значення.

Коли масив створюється без будь-яких початкових значень (див. попередні приклади) елементи *автоматично ініціалізуються* значеннями за замовчуванням. Значення залежать від типу елементів масиву:

Числовим величинам типів **short**, **int**, **float**, **decimal** і т. д. присвоюється значення **нуль**.

Величинам типу **char** присвоюється символ **\u0000**.

Величини типу **bool** ініціалізуються значенням **false**.

Величини посилальних типів ініціалізуються значенням **null**.

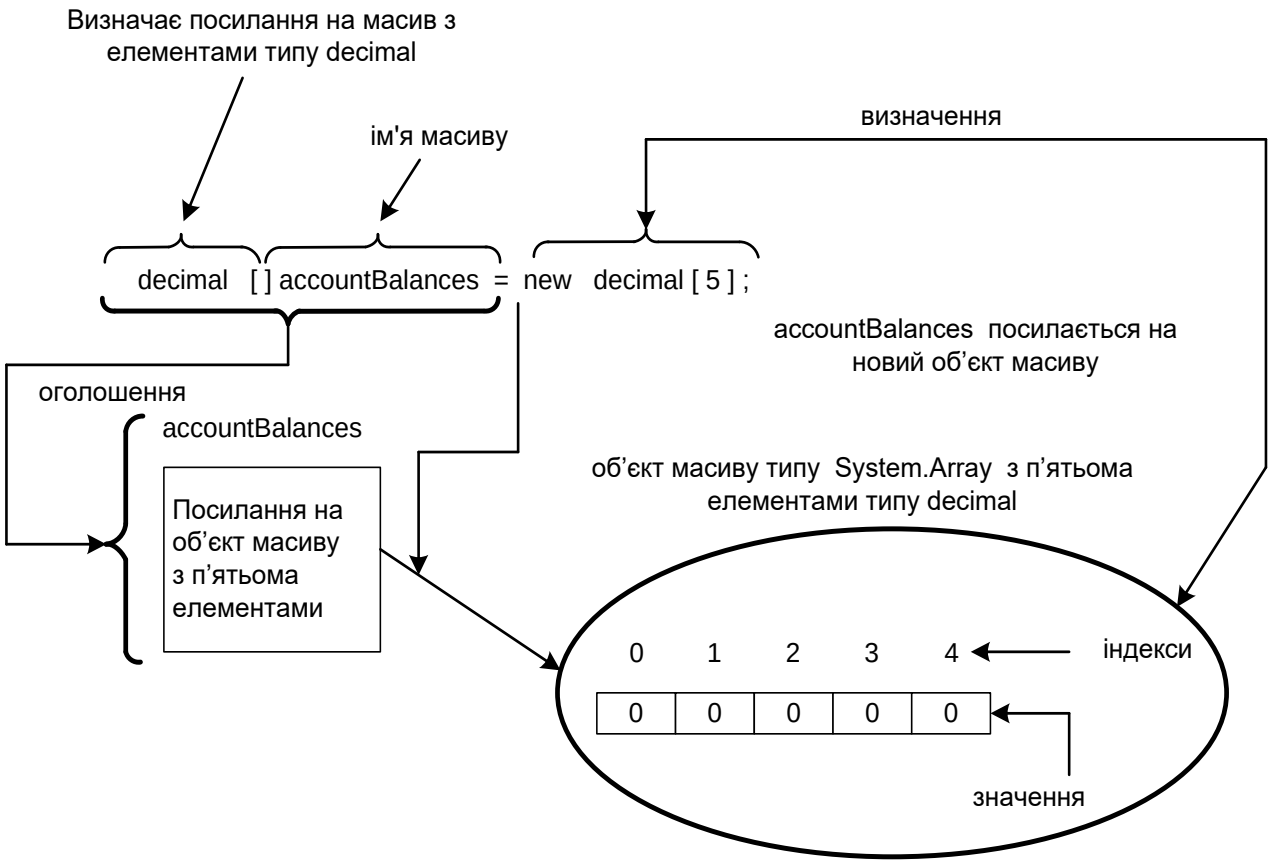


Рис. 5.3. **Посилання на об'єкт масиву**

У наведеному прикладі тип елемента масиву — **decimal**, тому всі величини ініціалізуються нулем, як показано на рис. 5.3.

Гарна методика програмування передбачає явну ініціалізацію в програмі, а не на етапі компіляції.

Значення null

Коли посилання не вказує на конкретний об'єкт, воно має значення **null**. Воно сумісне з усіма посилальними типами, представлено в мові C# ключовим словом **null**.

Якщо потрібне посилання, що ні на що не вказує, значення **null** можна присвоїти йому прямо: **eievator3 = null;**

Загальний синтаксис для оголошення й створення масиву наведений в наступному синтаксичному блоці.

Оголошення масиву й присвоювання йому значення

Оголошення_масиву ::=

<Базовий_тип> [ ] <Ідентифікатор масиву>;

Створення_масиву ::= new <Базовий_тип> [<Довжина_масиву>];

Присвоювання_посилання_на_масив ::= <Ідентифікатор_масиву> =
new <Базовий_тип> [<Довжина_масиву>];

Присвоювання_посилання_на_масив ::= <Оголошення_масиву>
new <Базовий_тип> [<Довжина_масиву>];

<Базовий_тип> в оголошенні повинен бути ідентичним **<Базовий_тип>** в операторі породження об'єкта.

<Довжина_масиву> повинна бути позитивною й належати типу, що неявно перетворюється до **int**. Це може бути літерал, константа або змінна.

Квадратні дужки **[ ]** в даному випадку є частиною синтаксису; вони не вказують на необов'язковість заключених в них елементів синтаксису.

Коли масив оголошений і присвоєне посилання на його об'єкт, можна звертатися і до його окремих елементів.

5.1.2. Доступ до окремих елементів масиву

Доступ до індивідуальних елементів масиву здійснюється так само, як до окремих символів змінної типу **string** — за ім'ям змінної з наступною парою квадратних дужок, що містять індекс. В наступному рядку коду

accountBalances[0] = 1000m;

величина **l000** типу **decimal** присвоюється першому елементу масиву, на який посилається **accountBalances**. Індекс може бути будь-яким числовим вираженням, що дає в результаті ненегативну величину типу, який неявно перетворюється в **int**.

Індекс повинен мати значення не більше, ніж довжина масиву мінус 1.

Рядки (типу **string**) є *незмінними*: будь-який символ в рядку не можна замінити. Масив же дозволяє присвоювати різні значення своїм елементам.

Елемент **accountBalances[0]** можна розглядати як звичайну змінну типу **decimal**: присвоювати та читати її значення, а також використовувати її в будь-якому вираженні.

Наприклад, щоб вивести суму десятивідсоткового нарахування на **accountBalances[0**], можна скористатися оператором:

Console.WriteLine(accountBalances[0] * 0.1m);

Приклад. Нарахування відсотків.

У вихідному коді, представленому нижче, оголошений масив з п'яти балансів рахунків. Суми присвоюються двом першим елементам, потім по них нараховується десять відсотків, і результат виводиться на екран. Інші три елементи масиву **accountBalances[2]**, **accountBalances[3]** і **accountBalances[4]** в цій програмі не використовуються.

```
1: using System;
2:
3: class SimpleAccountBalances
4: {
5:     public static void Main()
6:     {
7:         const decimal interestRate = 0.1m;
8:         decimal [ ] accountBalances;
9:
10:        accountBalances = new decimal [5];
11:
12:        Console.WriteLine("Please enter two account balances: ");
13:        Console.Write("First balance: ");
14:        accountBalances[0] = Convert.ToDecimal(Console.ReadLine());
15:        Console.Write("Second balance: ");
16:        accountBalances[1] = Convert.ToDecimal(Console.ReadLine());
17:
18:        accountBalances[0] = accountBalances[0] + accountBalances[0] *
interestRate;
19:        accountBalances[1] = accountBalances[1] + accountBalances[1] *
interestRate;
20:
21:        Console.WriteLine("New balances after interest: ");
22:        Console.WriteLine("First balance: {0:N2}", accountBalances[0]);
23:        Console.WriteLine("Second balance: {0:N2}", accountBalances[1]);
24:    }
25:}
```

Результат роботи програми:

```
Please enter two account balances:
First balance: 1034,5
```

Second balance: 667,3
New balances after interest:
First balance: 1 137,95
Second balance: 734,03
Press any key to continue

У рядку 8 оголошується змінна **accountBalances**, що містить посилання на масив з елементами типу **decimal**.

У рядку 10 створюється об'єкт класу **System.Array**, виділяється пам'ять під нього, а посилання на нього присвоюється змінній **accountBalances**.

У рядках 14 і 16 елементам масиву з індексами 0 і 1 присвоюються задані користувачем значення. Якщо користувач вводить суми 1000 і 2000, об'єкт масиву має вигляд як на рис. 5.4, де представлений синтаксис доступу до кожного елемента масиву.

У рядках 18 і 19 на два рахунки нараховується відсоток, визначенний **const interestRate** в рядку 7.

Нові значення виводяться в рядках 22 і 23.

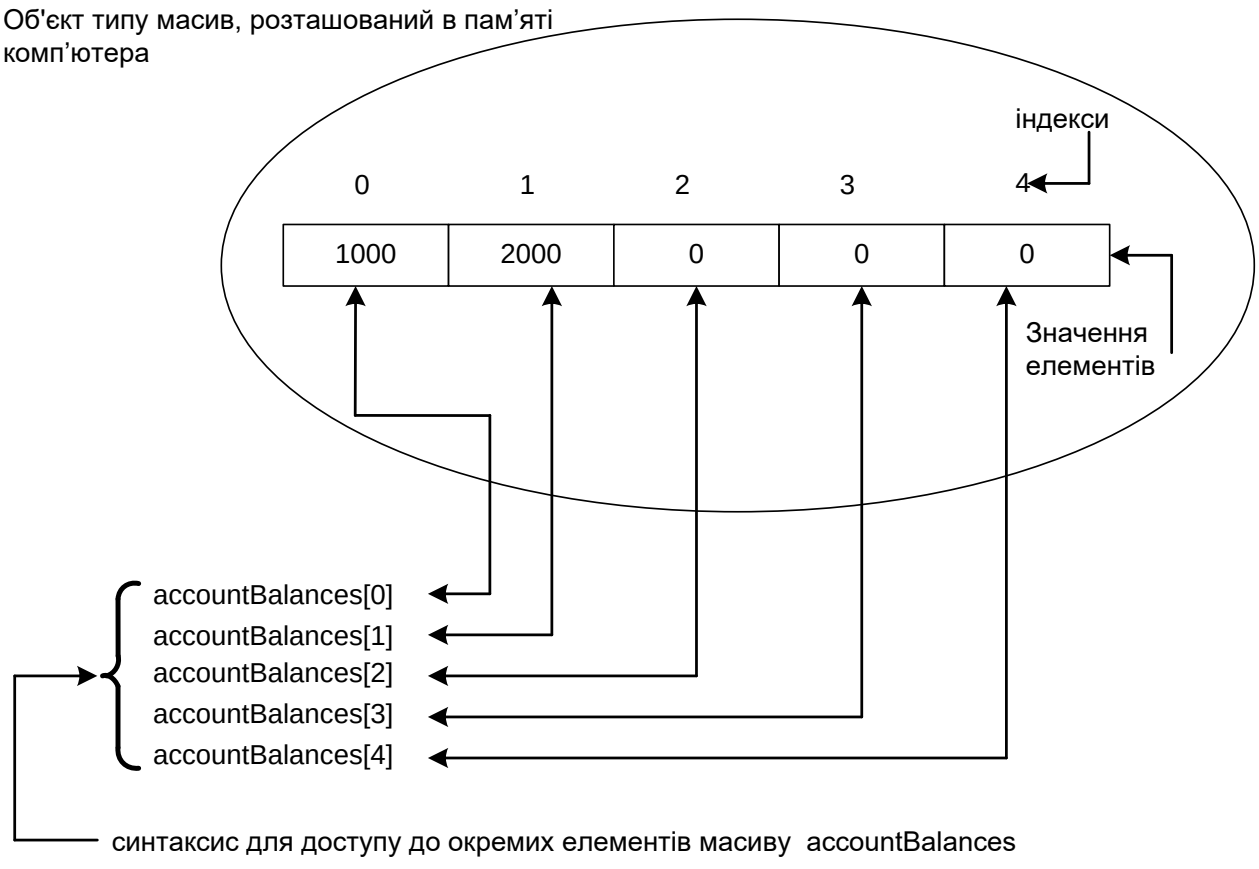


Рис. 5.3. **Два значення, що присвоюються accountBalances**

Синтаксис для доступу до окремих елементів масиву формалізований в наступному синтаксичному блоці.

Синтаксичний блок доступу до елементів масиву

Доступ_до_елемента_масиву ::=

<Ідентифікатор_масиву> [<Числове_вираження>]

Квадратні дужки [ ] в даному випадку не вказують на необов'язковість заключених в них елементів синтаксису, фактично в них розміщується індекс у формі **<Числове_вираження>.**

<**Числове_вираження**> може бути будь-яким числовим вираженням з типом, що неявно перетворюється в **int**. Тут можна використовувати вираження типу **sbyte**, **byte**, **short**, **ushort** і **char**. Будь-який інший тип повинен або мати визначений користувачем шлях явного перетворення, або бути тим, що приводиться до потрібного типу неявно.

Важливо розрізняти три випадки, в яких квадратні дужки використовуються з масивами:

1). для оголошення змінної типу масив:

decimal [ ] accountBalances;

2). для зазначення довжини масиву при створенні об'єкта типу масив:

new decimal [5];

3). для доступу до індивідуальних елементів масиву:

accountBalances[0]

Приклад. Нарахування відсотків (ускладнений варіант).

Розглянемо застосування масиву в сполученні з оператором циклу. Функціональні властивості цієї програми аналогічні програмі з попереднього прикладу, однак їхні реалізації розрізняються. Об'єднання в програмі циклу **for** з масивом **accountBalances** дозволяє користувачеві:

ввести п'ять балансів рахунків;

додавати відсотки до кожного рахунку;

вивести на екран п'ять результатів.

```
1: using System;
2:
3 :class AccountBalanceTraversal
```

```
4: {
5:     public static void Main()
6:     {
7:         const decimal interestRate = 0.1m;
8:
9:             decimal [ ] accountBalances;
10:
11:        accountBalances = new decimal [5];
12:
13:        Console.WriteLine("Please enter {0} account balances:",
accountBalances.Length);
14:        for (int i = 0; i < accountBalances.Length; i++)
15:        {
16:            Console.Write("Enter balance with index {0}: ", i);
17:            accountBalances[i] = Convert.ToDecimal(Console.ReadLine());
18:        }
19:
20:        Console.WriteLine("\nAccount balances after adding interest\n");
21:        for (int i = 0; i < accountBalances.Length; i++)
22:        {
23:            accountBalances[i] = accountBalances[i]
24:                + (accountBalances[i] * interestRate);
25:            Console.WriteLine("Account balance with index {0}: {1:N2}",
26:                i, accountBalances[i]);
27:        }
28:    }
29:}
```

Результат роботи програми:

```
Please enter 5 account balances:
Enter balance with index 0: 10000
Enter balance with index 1: 20000
Enter balance with index 2: 15000
Enter balance with index 3: 50000
Enter balance with index 4: 100000

Account balances after adding interest
```

```
Account balance with index 0:11 000,00
Account balance with index 1:22 000,00
Account balance with index 2:16 500,00
Account balance with index 3:55 000,00
Account balance with index 4:110 000,00
Press any key to continue
```

Тут (як і раніше) **accountBalances** посилається на об'єкт масиву, що містить 5 величин типу **decimal** (рядки 9 і 11).

Цикл **for**, розташований в рядках 14 – 18, повторюється п'ять разів. Значення змінної **i** починається з 0 і збільшується на 1 при кожному повторенні тіла циклу.

Виконання зупиняється, коли умова циклу стає рівною **false**.

Вираження **accountBalances.Length** еквівалентне властивості **Length** класу **string** і є одним із багатьох корисних вбудованих властивостей і методів об'єкта **System.Array**.

Властивість **Length** повертає довжину масиву, в даному випадку 5, і робить вираження **i < accountBalances.Length** рівним **false**, коли **i** стає рівним 5. Отже, коли тіло циклу виконується востаннє, **i** дорівнює 4.

Це добре узгоджується з функціональністю, що реалізується в тілі циклу в рядках 16 і 17:

отримати нову суму від користувача і присвоїти її елементу масиву з індексом, що починається з 0 і збільшується на 1 для кожної нової суми.

Кожному з усіх 5-ти елементів масиву користувач присвоює нову суму.

Цикл **for** в рядках 21 – 27 містить ті ж оператори ініціалізації, умови і оновлення циклу, як і попередній оператор **for**.

Нарахування відсотків здійснює оператор в рядках 23 і 24. Виведення нового балансу відбувається в рядках 25 і 26.

Слід зазначити, що область видимості змінної **i**, оголошеної в рядку 14, обмежується рядками 14 – 18. Ця змінна відрізняється від **i**, оголошеної в рядку 21, яка має область видимості в межах рядків 21 – 27.

У визначенні довжини масиву в програмі варто використовувати властивість **Length**, а не літерали або константи. Це дозволяє програмі самостійно контролювати зміну розміру масиву й вимагає зміни коду лише в єдиному місці.

5.1.3. Ініціалізація масивів

Елементи масиву можна ініціалізувати будь-якими значеннями під час його створення.

Спочатку потрібно оголосити змінну масиву. Але замість створення об'єкта з ключовим словом **new**, як в наступному прикладі:

new int [6];

потрібно явно визначити список ініціалізуємих величин, указуючи їх в дужках після оператора присвоювання (рис. 5.4).

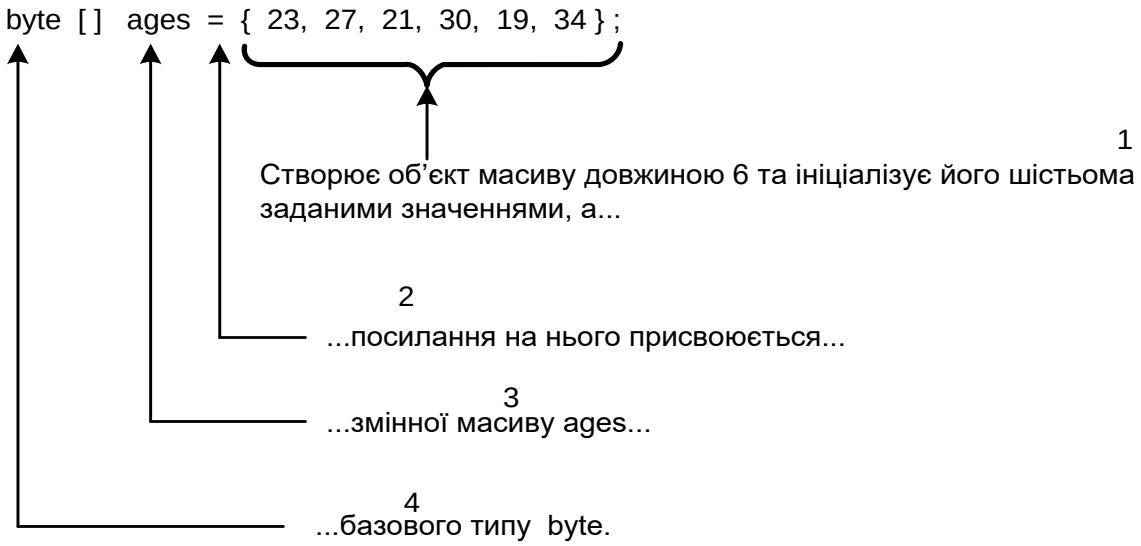


Рис. 5.4. **Приклад явної ініціалізації елементів масиву**

Об'єкт масиву автоматично створюється з кількістю елементів (в цьому прикладі 6), необхідних для зберігання значень у фігурних дужках. У даному випадку немає необхідності визначати довжину масиву явно. В дужках задаються значення тільки того типу, що може бути явно перетворений в базовий тип, визначений в оголошенні масиву (в даному випадку **byte**).

Довжину масиву можна визначити і явно (хоча за замовчуванням вона задається кількістю елементів в фігурних дужках). Наступний рядок еквівалентний попередньому (див. рис. 5.4):

byte [ ] ages = new byte [6] { 23, 27, 21, 30, 19, 34} ;

new byte [6] тут не є обов'язковим, однак ця конструкція має дві переваги:

довжина масиву видна відразу при читанні коду – це зручно, коли список значень в дужках досить довгий;

компілятор порівнює довжину масиву, зазначену в квадратних дужках, з кількістю елементів в фігурних дужках. Будь-які невідповідності приводять до повідомлення про помилку (це дозволяє автоматично знайти пропущені або зайві значення).

При використанні фігурних дужок для ініціалізації масиву необхідно задавати значення всіх елементів. Якщо деякі елементи потрібно пропустити, можна скористатися одним із наступних прийомів:

задати ініціалізуючі значення в дужках, вказавши при цьому для деяких елементів значення за замовчуванням (приміром, 0). Якщо, скажімо, в прикладі, наведеному раніше, значення трьох останніх елементів невідомі, можна використовувати такий оператор:

byte [ ] ages = { 23, 27, 21, 0, 0, 0};

Синтаксичний блок оголошення і явної ініціалізації масиву

Оголошення_і_явна_ініціалізація масиву ::=

<Базовий_тип> [ ] <Ідентифікатор_масиву> =
new <Базовий_тип> [<Довжина_масиву>]
{ <Значення1>, <Значення2>,...<ЗначенняN> } ;

<Значення1>, **<Значення2>** і т. д. повинні явно приводитися до **<Базовий_тип>**.

Кількість значень, що присвоюється масиву (позначена тут **N**), дорівнює довжині масиву.

5.1.4. Перебір елементів масиву за допомогою оператора foreach

Крім циклу **for**, для проходження по всьому масиву можна скористатися оператором **foreach**.

Наприклад, для виведення значення кожного елемента масиву **childbirths,** оголошеного й визначеного як

uint [ ] childbirths = {1340, 3240, 1003, 4987, 3877};

може використовуватися оператор **foreach**:

```
foreach ( uint  temp   in   childbirths )
{
        Console.WriteLine(temp);
}
```

який дає виведення:

1340 3240 1003 4987 3877

Розглянемо докладно цей спосіб.

Оператор **foreach** складається із заголовка й тіла (рис. 5.5). Тіло циклу може бути одиночним або складеним оператором.

Перші два слова в круглих дужках заголовка є, відповідно, типом і ідентифікатором. Разом вони оголошують ітераційну змінну оператора **foreach**. У даному випадку вона називається **temp** (від слова **temporary** – тимчасовий).

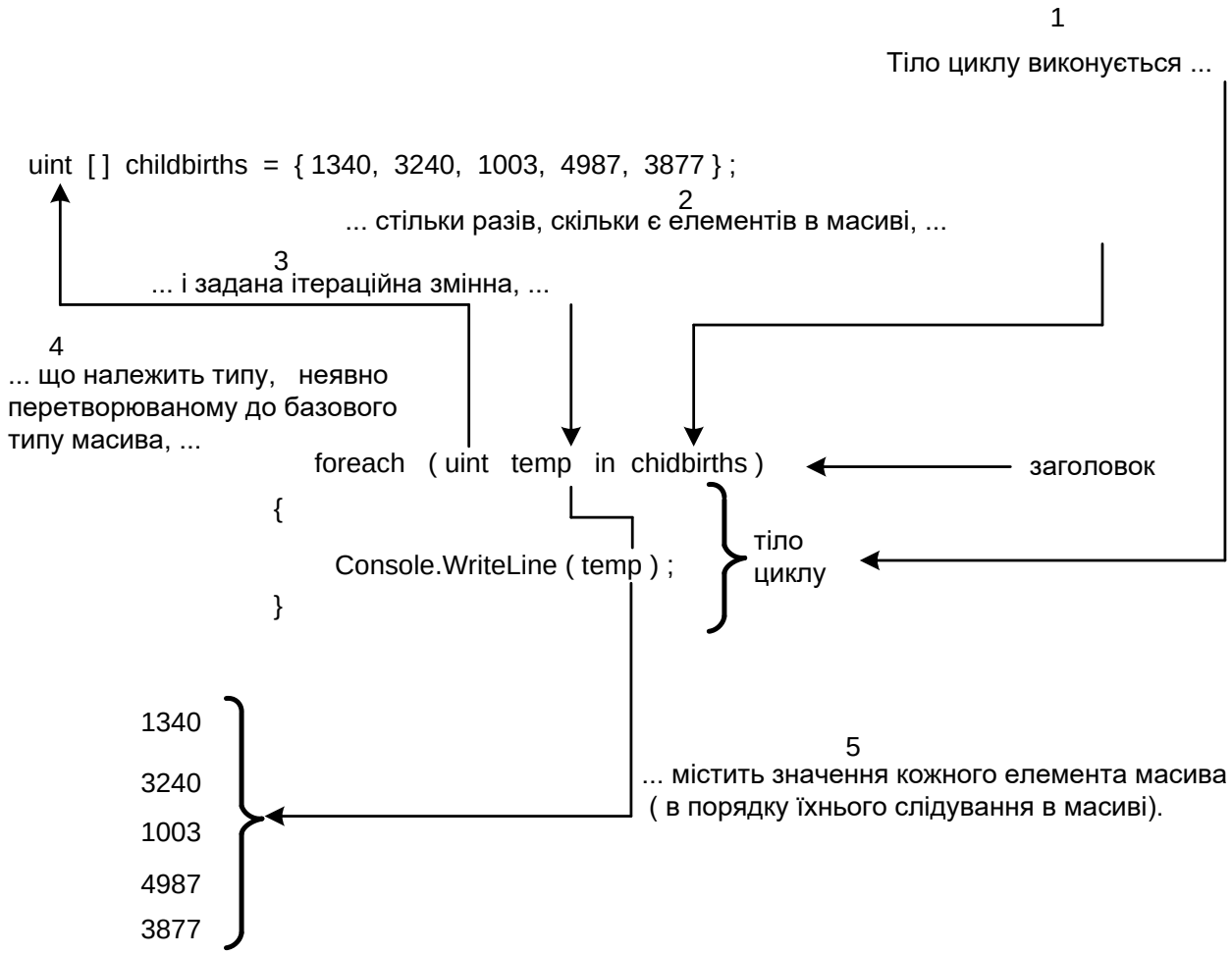


Рис. 5.5. **Оператор foreach**

Праворуч від ітераційної змінної знаходиться ключове слово **in**, за яким іде масив (у даному випадку **childbirths**). Важливо, щоб тип ітераційної змінної міг бути неявно перетворений в базовий тип масиву.

Тіло циклу виконується один раз для кожного елемента. Ітераційну змінну можна використовувати й у тілі циклу. При першому проході (виконанні) вона дорівнює першому елементу масиву й т. д.

Синтаксичний блок оператора foreach

Оператор foreach ::=
foreach (<Тип> <Ідентифікатор_ітераційної_змінної>
in <Ідентифікатор_масиву>)
[< Оператор > | <Складений_оператор>]

Виконуючи оператор **foreach**, C# автоматично визначає кількість ітерацій. При цьому кожний елемент масиву присвоюється ітераційній змінній без необхідності явної індексації.

Лічильник, умова і оновлення циклу (необхідні в стандартному циклі **for**), в даному випадку непотрібні, що забезпечує простоту та ясність коду.

5.2 Приклади обробки одномірних масивів

*5.2.1. Обчислення функції y = a*x^2 + sin(x)*

```
using System;
        class Class1
        {
                static void Main()
                {
                        const double a = 10.5;
                        double [ ] mas = new double[7];
//              double [ ] mas = {-1, -0.93, -0.49, 0, 1.13, 0.96, 1.75};
//              double [ ] mas = new double[7]{-1, -0.93, -0.49, 0, 1.13, 0.96, 1.75};
                        double y;

                        // Введення масиву
                        Console.WriteLine("Введіть значення елементів масиву");
                        for ( int i=0; i < mas.Length; i++)
                        {
                                Console.Write("mas[{0}] = ",i);
                                mas[i] = Convert.ToDouble(Console.ReadLine());
                        }
```

```
            // Контрольне виведення масиву
            Console.WriteLine("Вихідний масив:");
            for ( int i=0; i < mas.Length; i++)
            {
                  Console.Write("{0} ",mas[i]);
            }
            Console.WriteLine(); // переведення рядка

            //  Обчислення функції
            for ( int i=0; i < mas.Length; i++)
            {
            y = a*mas [ i ] * mas [ i ] - Math.Sin ( mas [ i ] );
            Console.WriteLine("При значенні x={0}, y={1:N4}",mas[i], y);
            }
      }
}
```

Результат роботи програми:

```
Введіть значення елементів масиву
mas[0] = -1
mas[1] = -0,93
mas[2] = -0,49
mas[3] = 0
mas[4] = 1,13
mas[5] = 0,96
mas[6] = 1,75
Вихідний масив:
-1 -0,93 -0,49 0 1,13 0,96 1,75
При значенні x=-1, y=11,3415
При значенні x=-0,93, y=9,8831
При значенні x=-0,49, y=2,9917
При значенні x=0, y=0,0000
При значенні x=1,13, y=12,5030
При значенні x=0,96, y=8,8576
При значенні x=1,75, y=31,1723
Press any key to continue
```

5.2.2. Пузиркове сортування

```
using System;
class Class1
{
	static void Main()
	{
		const double a = 10.5;
		double [ ] mas = new double[7];
		// double [ ] mas = {-1, -0.93, -0.49, 0, 1.13, 0.96, 1.75};
		// double [ ] mas = new double[7] {-1, -0.93, -0.49, 0, 1.13, 0.96, 1.75};
		double y;

		// Введення масиву
		Console.WriteLine("Введіть значення елементів масиву");
		for ( int i=0; i < mas.Length; i++)
		{
			Console.Write("mas[{0}] = ",i);
			mas[i] = Convert.ToDouble(Console.ReadLine());
		}

		// Контрольне виведення масиву
		Console.WriteLine("Вихідний масив:");
		for ( int i=0; i < mas.Length; i++)
		{
			Console.Write("{0}  ",mas[i]);
		}
		Console.WriteLine(); // переведення рядка
		//  Обчислення функції
		double t;
		for ( int i=0; i < mas.Length-1; i++)
			for ( int j=0; j < mas.Length-1; j++)
				if(mas[j] < mas[j+1])      // варіант if(mas[j] > mas[j+1])
				{
					t=mas[j];
					mas[j] = mas[j+1];
					mas[j+1]=t;
				}
```

```
            // Контрольне виведення масиву
            Console.WriteLine("Масив після сортування:");
            for ( int i=0; i < mas.Length; i++)
            {
                  Console.Write("{0}  ",mas[i]);
            }
            Console.WriteLine(); // переведення рядка
      }
}
```

Результат роботи програми:

```
Введіть значення елементів масиву
mas[0] = 1,3
mas[1] = -34,12
mas[2] = 5,23
mas[3] = 6
mas[4] = 0,23
mas[5] = -0,56
mas[6] = 65,3
Вихідний масив:
1,3  -34,12  5,23  6  0,23  -0,56  65,3
Масив після сортування:
65,3  6  5,23  1,3  0,23  -0,56  -34,12
Press any key to continue
```

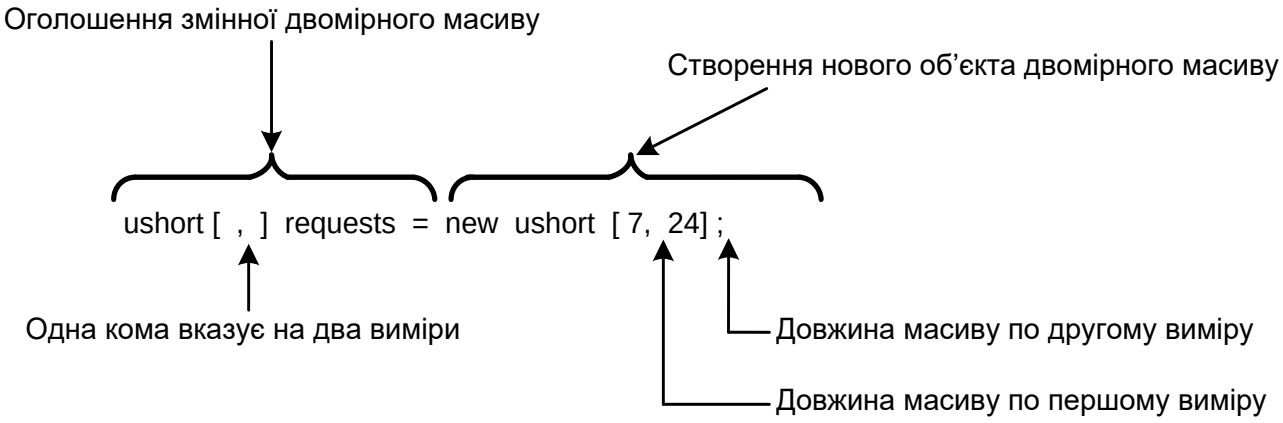
5.3. Багатомірні масиви

Оголошення й визначення двомірного масиву

Мова C# дозволяє визначати двомірні масиви, де для ідентифікації елемента потрібні два індекси. (Фактично в C# можна визначати масиви будь-якої розмірності).

На рис. 5.6 наведений приклад оголошення й визначення двомірного масиву.

Оголошення змінної масиву, створення нового об'єкта і присвоювання змінній посилання на об'єкт (перший рядок на рис. 5.6) можна (як і у випадку одномірного масиву) розбити на два оператори. Результат показаний в нижній частині рис. 5.6.



Попередній оператор, як і у випадку одномірних масивів, можно розбити на два рядки:

```
ushort [ , ]  requests ;
requests  =  new  ushort  [  7 , 24 ] ;
```

Рис. 5.6. **Приклад оголошення й визначення двомірного масиву**

Індекс першого виміру змінюється від 0 до 6, а другого – від 0 до 23.

Доступ до елементів двомірного масиву

Після оголошення змінної масиву і присвоювання їй посилання на двомірний об'єкт можна звертатися до його окремих елементів.

За винятком того, що для звертання до елементів двомірного масиву потрібен додатковий індекс, вони використовуються аналогічно елементам одномірного масиву. Отже, будь-який з них можна використовувати так само, як і окрему змінну базового типу. Наприклад:

requests[0,0] = (ushort) 89;

Тут застосовується операція приведення до типу (ushort), оскільки він є базовим типом requests. Воно необхідно, тому що літерал 89 належить до типу int, що не може бути неявно перетворений в ushort.

Представлення двомірного масиву як масиву масивів

Два виміри масиву requests можна розглядати як масив масивів. Якщо звернутися до першого виміру requests (що представляє, наприклад, дні тижня), сім днів можна вважати одномірним масивом (див. рис. 5.6). Якщо тепер включити в розгляд години, то кожний елемент "день" можна вважати складеним з одномірного масиву "години".

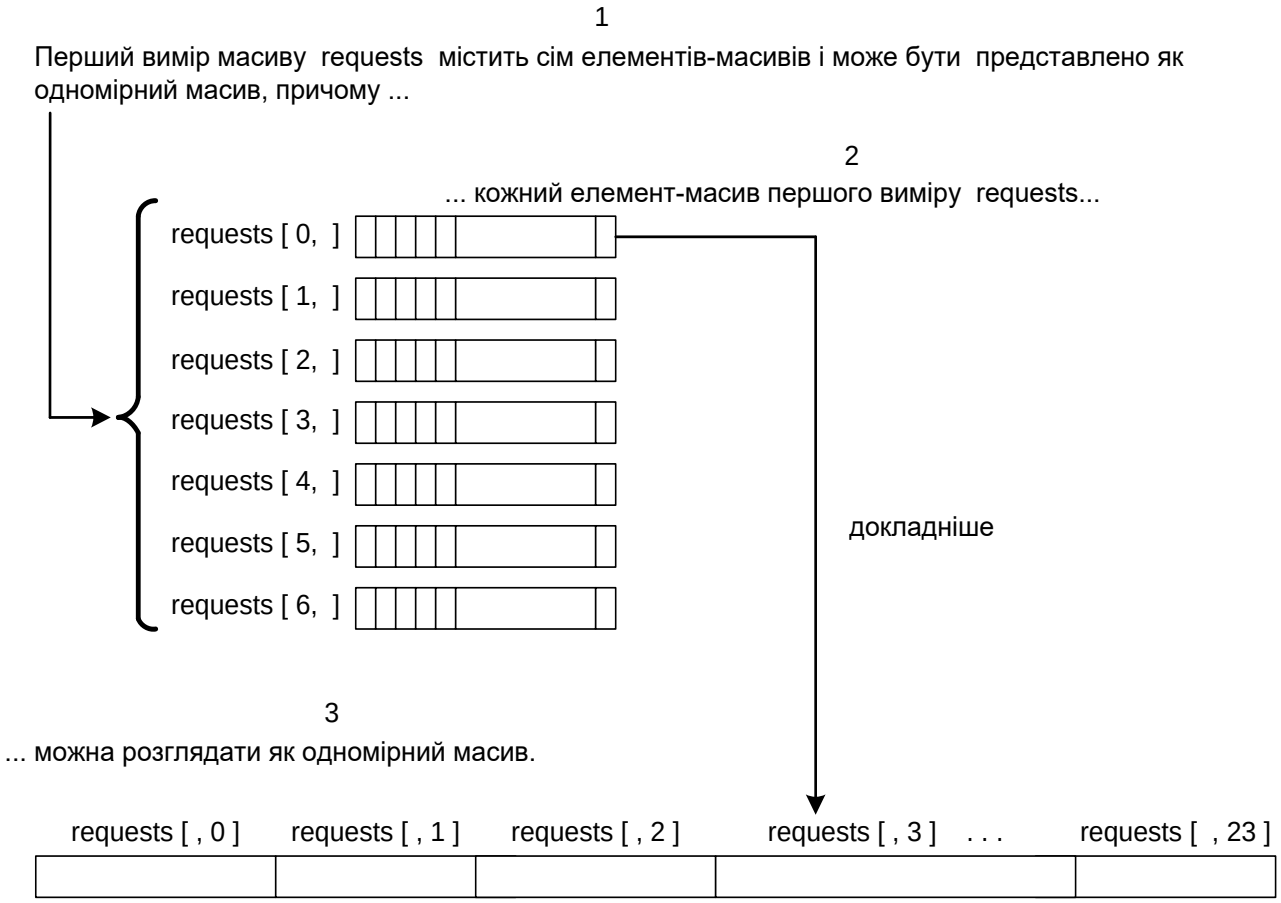


Рис. 5.7. **Двомірний масив можна представити як два одномірних масиви**

Приклад. Обробка матриці зарплати.

```
// Обробка матриці зарплати
using System;
class Class1
{
	static void Main()
	{
		int kol_rab;		// кіл-ть робітників в бригаді
		int kol_brigad;	// кіл-ть бригад
		char vibor;		// для організації діалогу
		do
		{
			//	Введення матриці із клавіатури
			Console.WriteLine("Введіть кількість робітників в бригаді...");
			kol_rab=Convert.ToInt32(Console.ReadLine());
```

```
                Console.WriteLine("Введіть кількість бригад...");
                kol_brigad=Convert.ToInt32(Console.ReadLine());

                double  [,] Matr = new double[kol_rab,kol_brigad];

                for( int i=0; i < kol_rab;  i++ )
                {
                        for (int j=0; j < kol_brigad;  j++)
                        {
                        Console.WriteLine("Введіть зарплату {0} робітника з
{1} бригади", i+1,j+1);
                        Console.WriteLine("Matr[{0},{1}]=?",i,j);
                        Matr[i,j]=Convert.ToDouble(Console.ReadLine());
                        }
                }
                // Контрольне виведення матриці
                Console.WriteLine("Матриця зарплат:");
                for( int i=0; i < kol_rab;  i++ )
                {
                        for (int j=0; j < kol_brigad;  j++)
                        {
                                Console.Write("{0}  ",Matr[i,j]);
                        }
                        Console.WriteLine();  // переведення рядка
                }
                // Обробка матриці
                Console.WriteLine("Розрахунок максимальної зарплати
по бригадах:\n");
                Console.WriteLine("Номер бригади      Номер робітника
Зарплата");
                int  n_rab = 0,       // номер робітника
                        n_brigad = 0;        // номер бригади
                double max_zarplata = 0;
                for( int j=0; j < kol_brigad;  j++ )
                {
                        for (int i=0; i < kol_rab ;  i++)
                        {
```

```
                        if(Matr[i,j]>max_zarplata)
                        {
                            max_zarplata = Matr[i,j];
                            n_rab = i;
                            n_brigad = j;
                        }
                    }
                    Console.WriteLine("{0,7}{1,15}{2,15}",
n_brigad+1,n_rab+1,max_zarplata );
                    max_zarplata = 0;  // підготовка до наступної ітерації
                }
                // Діалог для продовження роботи
                Console.WriteLine(" Будете продовжувати роботу? Так -
<Y>, Hi - <N> " );
                vibor=Convert.ToChar(Console.ReadLine());
            } while( (vibor == 'y') || (vibor == 'Y') );
            Console.WriteLine("Роботу завершено!" );
            }
        }
```

Робота програми:

```
Введіть кількість робітників в бригаді...
4
Введіть кількість бригад...
2
Введіть зарплату 1 робітника з 1 бригади
Matr[0,0]=?
345,7
Введіть зарплату 1 робітника з 2 бригади
Matr[0,1]=?
654,1
Введіть зарплату 2 робітника з 1 бригади
Matr[1,0]=?
123,9
Введіть зарплату 2 робітника з 2 бригади
Matr[1,1]=?
765,1
Введіть зарплату 3 робітника з 1 бригади
```

```
Matr[2,0]=?
342,1
Введіть зарплату 3 робітника з 2 бригади
Matr[2,1]=?
765,3
Введіть зарплату 4 робітника з 1 бригади
Matr[3,0]=?
943,3
Введіть зарплату 4 робітника з 2 бригади
Matr[3,1]=?
342,6
Матриця зарплат:
345,7  654,1
123,9  765,1
342,1  765,3
943,3  342,6
Розрахунок максимальної зарплати по бригадах:

Номер бригади   Номер робітника   Зарплата
     1                  4              943,3
     2                  3              765,3
Будете продовжувати роботу? Так - <Y>, Ні - <N>
n
Роботу завершено!
Press any key to continue
```

Питання для самоконтролю

1. У чому полягають особливості призначення, оголошення й визначення масиву.

2. Як відтворюється доступ до окремих елементів масиву?

3. Наведіть приклад двох варіантів ініціалізації масиву.

4. Опишіть загальну схему перебору елементів масиву за допомогою оператора foreach.

5. Що таке розмір і ранг масиву?

6. Наведіть приклади алгоритмів пошуку заданих елементів масиву.

7. Наведіть приклади алгоритмів перетворення масиву.

8. У чому суть алгоритму сортування елементів масиву методом «Пузирку»?

9. Як реалізується доступ до елементів двомірного масиву?

10. У чому суть представлення двомірного масиву як масиву масивів?

11. Наведіть приклади обробки матриць.

6. Структури

6.1. Загальні відомості про структури

Структури – це складені типи даних, побудовані з використанням інших типів. Вони представляють собою об'єднаний загальним ім'ям набір даних різних типів.

Окремі дані структури називаються *елементами* або *полями*. Елементи однієї й тієї ж структури повинні мати унікальні імена, але дві різні структури можуть містити неконфліктуючі елементи з однаковими іменами.

Синтаксичний блок визначення структури

```
[ <Специфікатор_доступності> ] struct <Ідентифікатор_структури>
                                        [ <Список_інтерфейсів> ]
{
        <Елементи_структури>
};
```

Відповідно до синтаксису мови, опис структури починається зі службового слова struct, услід за яким міститься обране користувачем ім'я типу. Елементи, що входять у структуру, розміщуються в фігурних дужках, після яких ставиться крапка з комою. Елементи структури можуть мати вбудований або похідний тип.

Опис структури не резервує ніякого простору в пам'яті, він тільки створює новий тип даних, що може використовуватися для визначення змінних. У структурі обов'язково повинен бути вказаний хоча б один компонент.

Припустимо, що необхідно створити тип для опису характеристики викладача університету. Цей тип повинен містити ім'я викладача, його

кваліфікацію (гарна, задовільна й т. д.), стаж роботи і поточну якість викладання (за 12-бальною оцінкою). Нижче наведений опис структури, що задовольняє цим вимогам:

```
struct Profesor
{
        public string Nombre;            // ім'я
        public string Calificacion;       // кваліфікація
        public int  Aprendizaje;          // стаж
        public double Calidad;            // якість
} ;
```

Ключове слово struct указує на те, що код визначає тип структури. Ідентифікатор Profesor – назва для цього типу. Таким чином, тепер можна створювати змінні типу Profesor так само, як змінні будь-якого базового типу, наприклад int або char.

Між фігурними дужками знаходиться список полів структури. Кожний елемент списку – це оператор визначення. Тут можна використовувати будь-який з типів даних C#, включаючи масиви та інші стуктури. В цьому прикладі використовуються два масиви типу **string**, зручні для збереження рядків з атрибутами “Ім'я” і “Кваліфікація”, а також змінні int і double – для зберігання відповідних числових значень.

Тепер, коли структура оголошена, її можна використовувати. Для цього спочатку потрібно створити (визначити) екземпляр структури.

Екземпляр структури створюється за допомогою ключового слова new:

```
Profesor  P_Econom_Inform = new Profesor( );
```

але, на відміну від класу, екземпляр структури можна створити і без new. Це виглядає в такий спосіб:

```
Profesor  P_Econom_Inform;
```

При створенні структури без ключового слова new її конструктори не викликаються. При цьому значення всім її елементам слід присвоїти явно, звернувшись до них через ім'я структури, як показано нижче:

```
P_Econom_Inform.Nombre = "Браткевич В'ячеслав";
        P_Econom_Inform.Calificacion = "задовільна";
        P_Econom_Inform.Aprendizaje = 32;
        P_Econom_Inform.Calidad = 7.59;
```

Ініціалізацію не можна виконати через методи або властивості, оскільки жоден з елементів-функцій не може бути викликаний, поки не будуть ініціалізовані елементи-дані. Тому останні потрібно оголошувати як public**.**

6.2. Приклади елементарної обробки структур

Приклад. Оголошення, визначення (без **new**), ініціалізація та виведення на екран структури **Profesor**.

```
using System;
      // Опис структури Profesor
struct Profesor
{
      public string Nombre;            // ім'я
      public string Calificacion;      // кваліфікація
      public int  Aprendizaje;         // стаж
      public double Calidad;           // якість
} ;

class Class1
{
      static void Main()
      {
            Profesor  P_Econom_Inform; // Оголошення екземпляра структури
            //      Роздільна ініціалізація полів структури
            P_Econom_Inform.Nombre = "Браткевич В'ячеслав";
            P_Econom_Inform.Calificacion = "задовільна";
            P_Econom_Inform.Aprendizaje = 32;
            P_Econom_Inform.Calidad = 7.59;

            // Контрольне виведення
            Console.WriteLine("Викладач {0}: \nКваліфікація - {1};"+
            "\nСтаж - {2};\nЯкість - {3}", P_Econom_Inform.Nombre,
            P_Econom_Inform.Calificacion, P_Econom_Inform.Aprendizaje,
            P_Econom_Inform.Calidad);
      }
}
```

Результат роботи програми:
Викладач Браткевич В'ячеслав:
Кваліфікація - задовільна;
Стаж - 32;
Якість - 7,59
Press any key to continue

Приклад. Оголошення, визначення (з **new**), ініціалізація та виведення на екран структури **Profesor**.

```
using System;
        // Опис структури Profesor
struct Profesor
{
        public string Nombre;            // ім'я
        public string Calificacion;      // кваліфікація
        public int  Aprendizaje;         // стаж
        public double Calidad;           // якість
} ;

class Class1
{
        static void Main()
        {
                // Ініціалізації елементів структури за замовчуванням
                Profesor  P_Econom_Inform = new Profesor();

                // Контрольне виведення
                Console.WriteLine("Викладач {0}: \nКваліфікація - {1};"+ "\nСтаж
- {2};\nЯкість - {3}\n", P_Econom_Inform.Nombre,
P_Econom_Inform.Calificacion, P_Econom_Inform.Aprendizaje,
                                                        P_Econom_Inform.Calidad);
                // Роздільна ініціалізація полів структури

                P_Econom_Inform.Nombre = "Браткевич В'ячеслав";
                P_Econom_Inform.Calificacion = "задовільна";
                P_Econom_Inform.Aprendizaje = 32;
                P_Econom_Inform.Calidad = 7.59;
```

```
            // Контрольне виведення
            Console.WriteLine("Викладач {0}: \nКваліфікація - {1};"+ "\nСтаж
- {2};\nЯкість - {3}\n", P_Econom_Inform.Nombre,
P_Econom_Inform.Calificacion, P_Econom_Inform.Aprendizaje,
                                                P_Econom_Inform.Calidad);
            Console.Read();  // для паузи
        }
}
```

Результат роботи програми:

Викладач :
Кваліфікація - ;
Стаж - 0;
Якість - 0

Викладач Браткевич В'ячеслав:
Кваліфікація - задовільна;
Стаж - 32;
Якість - 7,59
Press any key to continue

6.3. Масиви структур

Приклад. Обробка масиву структур.

```
using System;
struct Stroka
{
        public string name;              // Автор книги
        public double stoimost;          // Вартість виданої книги
        public int kolich;               // Кількість виданих книг одного автора
        public double sum_stoimost;    // Вартість виданих книг
};
class Class1
{
        static void Main()
        {
```

```
        //      Введення вихідних даних
        Console.WriteLine("Введіть кількість рядків в документі");
        int kol = Convert.ToInt32(Console.ReadLine());
        Stroka[ ] Tabl = new Stroka[kol];
        for( int i=0; i < Tabl.Length; i++)
        {
                Console.WriteLine("Автор книги?");
                Tabl[i].name = Console.ReadLine();

                Console.WriteLine("Вартість книги?");
                Tabl[i].stoimost = Convert.ToDouble(Console.ReadLine());

                Console.WriteLine("Кількість книг?");
                Tabl[i].kolich = Convert.ToInt32(Console.ReadLine());
        }
        //  Виконання розрахунків:
        double s1=0, s2=0, s3=0;
        for( int i=0; i < Tabl.Length; i++)
        {
                Tabl[i].sum_stoimost = Tabl[i].stoimost * Tabl[i].kolich;
                s1 += Tabl[i].stoimost;
                s2 += Tabl[i].kolich;
                s3 += Tabl[i].sum_stoimost;
        }
//  Побудова "шапки" таблиці
Console.WriteLine("\nВідомості про вартість виданих книг\n");
Console.WriteLine("|-------------------------------------------------------------|");
Console.WriteLine("|  n/n |  Автор  |  Вартість  | Видано |  Витрати  |");
Console.WriteLine("|-------------------------------------------------------------|");
        // Заповнення таблиці даними:
        for( int i=0; i < Tabl.Length; i++)
        {
                Console.WriteLine("|{0,5}{1,20}{2,14}{3,9}{4,10:N2}  |",
                        i+1, Tabl[i].name, Tabl[i].stoimost, Tabl[i].kolich,
                        Tabl[i].sum_stoimost);
        }
Console.WriteLine("|-------------------------------------------------------------|");
Console.WriteLine("| Разом: {0,31} {1,8} {2,9:N2}  |",s1, s2, s3);
```

```
        Console.WriteLine("|-----------------------------------------------------------|");
        }
}
```

Результат роботи програми:

```
Введіть кількість рядків в документі
3
Автор книги?
Рубіна
Вартість книги?
34,45
Кількість книг?
3
Автор книги?
Улицька
Вартість книги?
45,78
Кількість книг?
10
Автор книги?
Пушкін
Вартість книги?
75,23
Кількість книг?
3

Відомості про вартість виданих книг

|-------------------------------------------------------------------------|
|  n/n  |     Автор     |   Вартість   |  Видано  |    Витрати   |
|-------------------------------------------------------------------------|
|    1          Рубіна          34,45           3             103,35  |
|    2        Улицька           45,78          10             457,80  |
|    3          Пушкін          75,23           3             225,69 |
|-------------------------------------------------------------------------|
| Разом:                      155,46          16             786,84 |
|-------------------------------------------------------------------------|
Press any key to continue
```

Питання для самоконтролю

1. Коли доцільно використовувати структури?
2. Як реалізується призначення, оголошення й визначення структур.
3. Наведіть приклади оголошення, визначення (без операції **new** та з операцією **new)**, ініціалізації та виведення на екран елементів заданої структури.
4. Які особливості обробки елементів структур?

7. Функції

7.1. Загальні відомості про функції

Досить часто виникають ситуації, коли виконання певних задач – наприклад, пошук максимального елемента масиву – необхідно здійснювати в різних місцях програми.

Рішенням такого роду проблем є застосування *функцій***.**

Функції в C# – це засіб, що дозволяє виконувати деякі ділянки коду в довільному місці додатка.

Функції володіють тією перевагою, що вони дозволяють робити програму більш зручночитаною, і ми отримуємо можливість групувати разом логічно зв'язані між собою частини коду. Поступаючи таким чином, можна зробити тіло самого додатка невеликим, оскільки рішення внутрішніх задач додатку буде здійснюватися окремо.

Функції можуть також використовуватися для створення *багатоцільових* програм, які виконують ті самі операції над різними даними.

Ми маємо можливість передавати функціям інформацію, з якою вони повинні працювати, у вигляді параметрів і одержувати результати роботи функції у вигляді *значень, що повертаються.* Наприклад, можна передати функції як параметр масив, в якому здійснюється пошук, і одержати елемент масиву з максимальним значенням, як значенням, що повертається. Звідси виходить, що можна щораз використовувати ту саму функцію для роботи з різними масивами.

Параметри функції і значення, що повертається, разом називаються *сигнатурою* функції.

В рамках даної лекції ми розглянемо способи опису і використання простих функцій, які не одержують і не повертають ніяких значень. Після цього перейдемо до способів передачі й одержання інформації від функцій.

Потім ми звернемося до області дії змінних. Яким чином дані додатки на C# локалізуються в різних ділянках коду – це питання виявляється особливо важливим, коли програма розбивається на множину окремих функцій.

На закінчення лекції ми зосередимося на двох більш складних темах: на *перевантаженні функцій* і на *делегатах*.

Перевантаження функцій – це спосіб, що дозволяє мати декілька функцій з однаковим ім'ям, але із сигнатурами, що відрізняються.

Делегат – тип змінної, який припускає непряме використання функції. Той самий делегат може бути використаний для виклику будь-якої функції, що має зазначену сигнатуру, а це дозволяє здійснювати вибір з декількох різних функцій у процесі виконання.

7.2. Опис і використання функцій

У даному розділі розповідається, яким чином можна включати функції до складу додатків, а потім використовувати (викликати) їх з коду.

Розглянемо наступний приклад.

```
using System;

namespace Function_1
{
        class Class1
        {

    static string myString;

    static void Write()
    {
      string myString = "String defined in Write()";
      Console.WriteLine("Now in Write()");
      Console.WriteLine("Local myString = {0}", myString);
```

```
    Console.WriteLine("Global myString = {0}", Class1.myString);
  }

  static void Main(string[ ] args)
  {
    string myString = "String defined in Main()";
    Class1.myString = "Global string";
    Write();
    Console.WriteLine("\nNow in Main()");
    Console.WriteLine("Local myString = {0}", myString);
    Console.WriteLine("Global myString = {0}", Class1.myString);
  }
}
}
```

Результат роботи програми:

```
Now in Write()
Local myString = String defined in Write()
Global myString = Global string

Now in Main()
Local myString = String defined in Main()
Global myString = Global string
Press any key to continue
```

Аналіз структури програми

Наступні чотири рядки коду описують просту функцію з ім'ям Write()

```
static void Write()
    {
      string myString = "String defined in Write()";
      Console.WriteLine("Now in Write()");
      Console.WriteLine("Local myString = {0}", myString);
      Console.WriteLine("Global myString = {0}", Class1.myString);
    }
```

Код, що міститься в даній функції, виводить деякий текст в консольному вікні.

Опис функції складається з:

двох ключових слів: static і void;

ім'я функції, за яким розташовуються параметри Write();

ділянки виконуваного коду, розміщеного у фігурних дужках.

Код, що використовується для опису функції Write(), виглядає майже так само, як і код самого додатка:

```
static  void  Main(string[ ]  args)
{
. . .
}
```

Це пояснюється тим, що весь код, що ми створювали дотепер (не враховуючи опису типів), являє собою частину деякої функції.

Функція Main( ) забезпечує, як згадувалося в попередніх лекціях, точку входу в консольний додаток. Коли запускається додаток, написаний на C#, то відбувається виклик функції точки входу, що міститься в ньому, а коли ця функція закінчує свою роботу, виконання додатка переривається. Будь-який виконуваний код, написаний на C#, повинен мати точку входу.

Єдина відмінність між функцією Main( ) і функцією Write( ), не враховуючи тих рядків коду, які в них містяться, полягає в тім, що в круглих дужках, розташованих за ім'ям функції Main, знаходиться деякий код. Цей код служить для завдання параметрів.

Як уже згадувалося раніше, обидві функції – і Main( ) і Write( ) – описуються з використанням ключових слів static (статичний) і void (відсутній).

Ключове слово static має відношення до понять об’єктно-орієнтованого програмування (до його розгляду ми перейдемо нижче). На даному етапі потрібно запам'ятати тільки те, що всі функції, які будуть задіяні в додатках даної лекції, обов'язково повинні використовувати це ключове слово.

Ключове слово void указує, що функція не повертає ніякого значення. Далі буде розказано, як необхідно писати в тих випадках, коли у функції є значення, котре повертається.

Продовжуючи розглядати наш додаток, ми виявляємо код, що здійснює виклик функції Write( ).

Він складається з ім'я функції, за яким містяться порожні круглі дужки. Коли виконання програми досягне цієї точки, почне виконуватися код, який міститься в функції Write( ).

Зверніть увагу, що використання круглих дужок, як при описі функції, так і при її виклику, є обов'язковим.

Значення, що повертаються

Найпростіший спосіб обміну даними з функціями – використання значення, що повертається. Функції, в яких застосовуються значення, що повертаються, точно так само володіють чисельним значенням, як і будь-які змінні, використовувані при обчисленні виражень. Аналогічно змінним значення, що повертаються, володіють типом.

Наприклад, можна описати функцію з ім'ям getString( ), значення, що повертається, якої буде мати тип string, і використовувати її у своїй програмі:

```
string = myString;
myString = getString( );
```

З іншого боку, можна описати функцію з ім'ям getVal( ), що буде повертати значення типу double, і використовувати її в математичному вираженні:

```
double myVal;
double multiplier = 5.3;
myVal = getVal( ) * multiplier;
```

Якщо функція повинна мати значення, що повертається, то необхідно внести дві зміни:

в описі функції замість ключового слова void вказати тип значення, що повертається;

по завершенні всіх обчислень у функції використовувати ключове слово return і передавати значення, що повертається, коду, що викликає.

Синтаксис коду для розглянутого типу функцій консольного додатка буде виглядати наступним чином:

```
static   <Тип, що повертається>   <ім’яФункції>(  )
{
. . .
        return
}
```

Єдиним обмеженням у даному випадку є вимога стосовно **<Значення, що повертається>**, яке повинне мати тип **<Тип, що повертається>** або ж повинна існувати можливість його неявного перетворення до цього типу.

Тип **<Тип, що повертається>** може бути будь-яким, включаючи самі складні типи з числа розглянутих раніше.

Значення, що повертаються, звичайно є результатом деяких обчислень. Коли при виконанні функції досягається оператор return, керування негайно передається назад у визивний код. Ніякі рядки коду після цього оператора виконуватися не будуть. Звідси, однак, зовсім не значить, що в тілі функції оператор return обов'язково повинен бути останнім. Він може бути використаний раніше, наприклад, при розгалуженні по якій-небудь умові.

Включення оператора return в цикл for, в блок if або в яку-небудь іншу структуру приведе до негайного закінчення виконання як цієї структури, так і всієї функції в цілому. Наприклад:

```
static double getVal( )
{
double checkVal;
// присвоювання змінної checkVal деякого значення,
// отриманого в результаті деяких обчислень.
if (checkVal < 5)
return 4.7;
return 3.2;
}
```

У даному випадку буде повернуто одне із двох значень – залежно від значення змінної checkVal.

Є єдине обмеження: оператор return повинен виконуватися до того, як буде досягнута закриваюча фігурна дужка } даної функції. Наступний код не є припустимим:

```
static double getVal( )
{
double checkVal;
// присвоювання змінної checkVal значення,
// отриманого в результаті деяких обчислень.
if (checkVal < 5)
return 4.7;
}
```

Якщо checkval >= 5, то не зустрінеться жодного оператора return, а це заборонено. Всі гілки повинні закінчуватися цим оператором.

Оператор return також може застосовуватися у функціях, оголошених з використанням ключового слова void (в них відсутнє яке-небудь значення, що повертається). В таких випадках функція просто припиняє роботу. Тому при використанні оператора return буде помилкою розміщати значення, що повертається, між ключовим словом return і наступною за ним крапкою з комою.

Параметри

Якщо функція повинна одержувати параметри, то необхідно задати:

список приймаємих функцією параметрів в її описі, а також типи цих параметрів;

співпадаючий список параметрів при кожному виклику функції.

Це передбачає використання наступного коду:

static <Тип, що повертається> <ім'яФункції> {<типПараметра> <ім'яПараметра>, . . .)

{

return <Значення, що повертається>;

}

Тут може бути довільне число параметрів, для кожного з яких указуються тип та ім'я. Як роздільник між параметрами ставляться коми.

Кожний з параметрів доступний усередині даної функції в якості змінної.

Наприклад, наступна функція приймає два параметри типу double і повертає їхній добуток:

```
static double product (double  param1, double  param2)
{
return param1 * param2;
}
```

Розглянемо більш складний приклад, в якому визначається максимальний елемент заданого в програмі масиву.

```
using System;
namespace Fandorin
{
    class Class1
{
```

```
    static  int  MaxValue(int[ ] intArray)
    {
      int maxVal = intArray[0];
      for (int i = 1; i < intArray.Length; i++)
      {
        if (intArray[i] > maxVal)
          maxVal = intArray[i];
      }
      return maxVal;
    }

    static void Main(string[ ] args)
    {
      int[ ] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
      int maxVal = MaxValue(myArray);
      Console.WriteLine("The maximum value in myArray is {0}", maxVal);
    }

}
}
```

Результат роботи програми:

```
The maximum value in myArray is 9
Press any key to continue
```

Розглянутий код містить функцію, що приймає як параметр масив цілих чисел і повертає найбільше з них. Її опис має наступний вигляд:

```
    static int MaxValue(int[ ] intArray)
    {
      int maxVal = intArray[0];
      for (int i = 1; i < intArray.Length; i++)
      {
        if (intArray[i] > maxVal)
          maxVal = intArray[i];
      }
      return maxVal;
    }
```

Функція MaxValue ( ) – має один параметр, описаний як масив типу int з ім'ям intArray. Значення, що повертається, також має тип int.

Визначення максимального значення становить нескладну задачу. Наведемо словесний опис алгоритму пошуку.

Локальній цілій змінній з ім'ям maxVal як початкове значення присвоюється перший елемент масиву, а потім здійснюється порівняння цього значення послідовно з усіма іншими елементами.

Якщо поточний елемент більше, ніж значення змінної maxVal, то поточне значення maxVal заміняється на це значення. Коли виконання циклу завершене, змінна maxVal містить найбільше значення даного масиву, що й вертається оператором return.

Код, розташований в Main (), оголошує та ініціалізує простий цілий масив, що буде використовуватися разом з функцією MaxValue( ):

```
int[ ] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
```

При виклику функції функцією MaxValue( ) значення присвоюється змінній maxVal типу int:

```
int maxVal = MaxValue(myArray);
```

Потім це значення виводиться на екран за допомогою оператора

```
Console.WriteLine("The maximum value in myArray is {0}", maxVal);
```

Відповідність параметрів

При виклику функції її параметри повинні в точності відповідати її опису. Необхідний збіг типів параметрів, їхньої кількості та порядку їх слідування. Це означає, що, наприклад, функція:

```
static void myFunction (string myString, double myDouble)
{
. . .
}
```

не може бути викликана з використанням рядка:

```
myFunction(2.6, " Hello ");
```

У даному випадку ми намагаємося передати значення типу double як перший параметр, а значення типу string – у якості другого, що не відповідає порядку, в якому ці параметри містяться в описі функції.

Не можна використовувати й такий рядок:

myFunction("Hello");

оскільки в ньому передається тільки один параметр типу string замість двох обов'язкових.

Спроба скористатися будь-яким із цих двох рядків для виклику функції призведе до помилки при компіляції, тому що компілятор накладає вимогу точної відповідності сигнатурам викликуваних функцій.

Повертаючись до попереднього прикладу, бачимо що така вимога означає: функція MaxValue( ) може використовуватися тільки для одержання максимального цілого з масиву цілих чисел. Якщо ми змінимо код в Main( ) наступним чином:

```
static void Main (string [ ] args)
{
double [ ] myArray = {1.3, 8.9, 3.3, 6.5, 2.7, 5.3};
double maxVal = MaxValue (myArray);
Console.WriteLine(“The maximum value in myArray is {0}", maxVal);
}
```

то такий код не пройде процедуру компіляції через невірний тип параметра.

Нижче – в розділі, присвяченому перевантаженню операторів – буде розглянутий дуже корисний спосіб, що дозволяє вирішити дану проблему.

Масиви параметрів

В C# передбачено можливість завдання одного (і тільки одного) спеціального параметра функції. Цей параметр, що обов'язково повинен розташовуватися останнім, відомий за назвою *масиву параметрів*. Він дозволяє при звертанні до функцій використовувати змінну кількість параметрів.

Масив параметрів описується за допомогою ключового слова params і служить одним зі способів спрощення коду, оскільки, завдяки їм, не доводиться передавати масиви з визивного коду. Навпроти, можна передати кілька параметрів того самого типу, які розміщуються в масив для подальшого використання усередині функції.

При описі функції з масивом параметрів застосовується наступний код:

static <Тип, що повертається> <ім’яФункції> (<params>
<тип>[ ] <ім'я>)
{
return <Значення, що повертається>;
}

Для того щоб викликати цю функцію, потрібен наступний код:

<ім’яФункції> (<значення1>, <значення2>, . . .) ;

У даному випадку **<значення1>**, **<значення2>** і т. д. – це значення типу **<тип>**.

Вони використовуються для ініціалізації масиву з ім'ям **<ім'я>**. Ніяких обмежень на кількість параметрів, які можуть бути тут задані, не існує.

Єдина пропонована до них вимога – вони всі повинні бути одного типу **<тип>**.

Приклад. Визначення суми елементів масиву.

```
using System;

namespace Bondar_Irina
{
    class Class1
    {
  static int sumVals(params int[ ] vals)
  {
    int sum = 0;
    foreach (int val in vals)
    {
      sum += val;
    }
    return sum;
  }
```

```
    static void Main(string[ ] args)
    {
      int sum = sumVals(1, 5, 2, 9, 8);
      Console.WriteLine("Summed Values = {0}", sum);
    }

      }
}
```

Результат роботи програми:

```
Summed Values = 25
Press any key to continue
```

У цьому прикладі функція sumVals( ) описана з використанням ключового слова params, що дозволяє їй приймати довільну кількість параметрів типу **int** (і ніякого іншого):

```
static int sumVals(params int[ ] vals)
```

Код у цій функції проходить в циклі всі значення масиву vals і підсумовує їх, повертаючи одержаний результат.

Ми викликаємо цю функцію з Main( ) з п'ятьома цілими параметрами:

```
int sum = sumVals (1, 5, 2, 9, 8);
```

Однак можна з тим же успіхом викликати цю функцію, передаючи їй нуль, один, два або сто параметрів цілого типу: ніяких обмежень на кількість параметрів, що задаються, не існує.

7.3. Обмін інформацією з функцією

Передача параметрів по посиланню і за значенням

В усіх попередніх функціях застосовувалися параметри, передані за *значенням***.** Інакше кажучи, ми передавали значення в змінну, яка потім використовувалася усередині функції. Будь-які зміни, які ця змінна перетерплювала усередині функції, *не робили ніякого впливу* на параметр, використаний при виклику функції. Як приклад розглянемо функцію, що подвоює переданий їй параметр і виводить результат на екран:

```
static  void  showDouble(int  val)
{
        val *= 2;
Console.WriteLine("val doubled = { 0 } " , val);
}
```

У цій функції відбувається подвоєння переданого параметра. Якщо ж ми викличемо цю функцію в такий спосіб:

```
int my Number = 5;
Console.WriteLine("myNumber = {0}", myNumber);
showDouble(myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

то вихідний потік, виведений на консоль, буде таким:

```
myNumber = 5
val doubled = 10
myNumber = 5
```

Виклик функції showDouble( ) зі змінної myNumber як параметр не оказує ніякого впливу на змінну myNumber, описану в Main( ), незважаючи на те, що параметр, якому присвоюється її значення, подвоюється.

Однак часто потрібно, щоб значення змінної myNumber було змінено, виникне проблема. Звичайно, можна скористатися функцією, яка повертає нове значення змінній myNumber, і викликати її в такий спосіб:

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
myNumber = showDouble(myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

Однак, в такому коді досить важко розібратися; крім того, такий спосіб не дозволяє відобразити зміни значень змінних, використовуваних як параметри (оскільки функції завжди повертають тільки одне значення).

Замість цього варто передавати параметр за *посиланням*. Це означає, що функція буде працювати саме з тією змінною, яка використовувалася при її виклику, а не зі змінною, що має те ж значення. Отже, будь-які зміни змінною, що використовувалася функцією, відіб'ються на значенні змінної, що використовувалася як параметр. Для цього при описі параметра необхідно скористатися ключовим словом ref:

```
static void showDouble(ref int val)
{
val *= 2;
Console.WriteLine(“val doubled = {0}", val);
}
```

Вдруге ключове слово ref необхідно використовувати при виклику функції (це обов'язкова вимога, оскільки параметр описаний як ref, є невід'ємною частиною сигнатури функції):

```
int myNumber = 5;
Console.WriteLine(“myNumber = {0}", myNumber);
showDouble(ref myNumber);
Console.WriteLine(“myNumber = {0}", myNumber);
```

У цьому випадку на консоль буде виведений наступний текст:

```
myNumber = 5
val doubled = 10
myNumber =10
```

Для змінних, використовуваних як параметри типу ref, є два обмеження. По-перше, оскільки існує ймовірність, що в результаті виклику функції значення викликуваного по посиланню параметра буде змінено, то при виклику функції *забороняється використовувати змінні типу const.*

Звідси виходить, що наступний виклик є неприпустимим:

```
const int myNumber = 5;
Console.WriteLine("mvNumber = {0}", myNumber);
showDouble(ref myNumber);
```

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

По-друге, необхідно використовувати заздалегідь ініціалізовану змінну.

C# не гарантує, що параметр типу ref буде ініціалізований тією функцією, в якій він використовується. Наступний код також є неприпустимим:

```
int myNumber;
showDouble(ref myNumber);
Console.WriteLine(“myNumber = {0}”, myNumber);
```

Вихідні параметри

На додаток до можливості передавати параметри за посиланням ми можемо вказати, що даний параметр є вихідним. Для цього в його опис включається ключове слово out, використовуване так само, як і ключове слово ref (як модифікатор параметра в описі функції й при виклику функції).

Фактично цей спосіб надає майже такі ж можливості, що й передача параметрів за посиланням, в тому розумінні, що значення параметра після виконання функції попадає в змінну, що використовувалася при виклику цієї функції.

Є, однак, і істотні відмінності.

Хоча використовувати змінну, якій не присвоєне початкове значення, як параметр типу ref неприпустимо, вона може застосовуватися як параметр типу out.

Параметр типу out буде розглядатися функцією, в якій він використовується як не маючий початкового значення. Це означає, що хоча передача змінної, якій присвоєне деяке значення, як параметр типу out є припустимою, однак у процесі виконання функції значення, що зберігається в цій змінній, буде втрачено.

Як приклад розглянемо розширення функції maxValue( ), що повертає елемент масиву з максимальним значенням. Модифікуємо цю функцію так, щоб одержувати індекс елемента масиву, який містить найбільше значення (у випадках, коли максимальне значення міститься в декількох елементах, ми будемо одержувати індекс першого максимального елемента). Для цього додамо вихідний параметр:

```
static int MaxValue(int[ ] intArray, out int maxlndex)
{
int maxVal = intArray[0];
maxlndex = 0;
for (int i = 1; i < int Array.Length; i++)
{
if (intArray[i] > maxVal)
{
maxVal = intArray[i];
maxlndex = i;
}
}
return maxVal;
}
```

Ця функція може бути використана в такий спосіб:

```
int[ ] myArray = {I, 8, 3, 6, 2, 5, 9, 3, 0, 2};
int maxlndex;
Console.WriteLine (“The maximum value in myArray is {0}”,
maxValue(myArray, out maxlndex));
Console.WriteLine(“The first occurrence of this value is at element {0}”,
maxlndex + 1);
```

Результатом її виконання буде наступне:
The maximum value in myArray is 9
(Максимальне значення, що міститься в масиві myArray, – 9)
The first occurrence of this value is at element 7
(Перше входження з таким значенням знайдено в елементі 7)

Важливим моментом, на який варто звернути увагу, є необхідність використовувати ключове слово out при виклику функції (так само, як і ключове слово ref).

Область дії змінних

Навіщо потрібно обмінюватися даними з функціями? Відповідь полягає в тому, що в C# доступ до змінних може здійнюватися тільки з

певних ділянок коду. Прийнято говорити, що змінна має певну *область дії*, в рамках якої вона є доступною.

Повернемося до розгляду програми з розділу 7.2 (простір імен namespace Function_1).

Дана програма додатково виконує наступні дії:

у функції Main ( ) описується та ініціалізується змінна з ім'ям myString;

функція Main( ) передає керування функції Write( );

у функції Write( ) описується та ініціалізується змінна з ім'ям myString, відмінна від змінної з ім'ям myString, описаної в функції Main( );

функція Write ( ) виводить на консоль рядок, що містить те значення змінної mystring, що їй присвоєне в функції Write ( );

функція Write( ) повертає керування функції Main( );

функція Main ( ) виводить на консоль рядок, що містить те значення змінної myString, що їй присвоєне в функції Main ( ).

Змінні, область дії яких поширюється тільки на одну функцію, називаються *локальними*.

Існує можливість опису глобальних змінних, чия область дії охоплює відразу кілька функцій.

У розглянутій програмі така змінна описана в такий спосіб:

static string myString;

Зверніть увагу, що нам знову треба було використовувати ключове слово static. Ми не будемо вдаватися в деталі: на даному етапі досить знати, що для консольних додатків даного типу в описі глобальних змінних *обов'язково використання* або ключового слова static, або ключового слова const. Якщо ми плануємо змінювати значення глобальної змінної, то обов'язковим є ключове слово static, тому що const забороняє будь-які зміни значення змінної.

Для того щоб відрізняти глобальну змінну від локальних змінних з тим же ім'ям в функціях Main( ) і Write( ), нам необхідно класифікувати ім'я глобальної змінної, використовуючи повністю кваліфіковане ім'я.

У даному випадку ми звертаємося до глобальної змінної за ім'ям class1.myString. Зверніть увагу, що це необхідно робити тільки в тому випадку, якщо існують глобальна й локальна змінні з однаковим ім'ям. Якби локальної змінної з ім'ям myString не існувало, то ми зовсім вільно могли б використовувати для звертання до глобальної змінної ім'я myString замість Class1.myString.

У тому випадку, коли використовується локальна змінна, ім'я якої збігається з ім'ям глобальної змінної, говорять, що глобальна змінна є *схованою*.

Значення глобальної змінної задається в функції Main ( ):

Class1.myString = "Global string";

і використовується в функції Write ( ):

Console.WriteLine("Global myString = {0}", Class1.myString);

Чи варто застосовувати глобальні змінні, залежить від способу використання кожної конкретної функції. Проблема глобальних змінних полягає в тім, що вони, за великим рахунком, виявляються непридатними для функцій “загального призначення”, які здатні працювати з будь-якими переданими їм даними і не обмежуються даними, що задаються конкретною глобальною змінною.

Функції і структури

Тип структур призначений для зберігання в одному місці декількох елементів інформації. Насправді можливостей у структур набагато більше, і однією з таких мождивостей є здатність містити не тільки дані, але й функції.

Як простий приклад розглянемо наступну структуру:

```
struct customerName
{
public string firstName, lastName;
}
```

Якщо в нас є змінні типу customerName і нам необхідно вивести на консоль повне ім'я, ми будемо змушені конструювати це ім'я з його складових частин. Наприклад, для змінної типу customerName з ім'ям customer можна використовувати наступний синтаксис:

```
customerName  myCustomer;
myCustomer.firstName = “Ivan”;
myCustomer.lastName = “Jana”;
Console.WriteLine(“{0} {1}”, customer.firstName, customer.lastName);
```

Маючи можливість включати в структури функції, ми можемо спростити цю процедуру за рахунок централізованого виконання задач, що часто зустрічаються.

У даному випадку це можна зробити в такий спосіб:

```
struct customerName
{
public string firstName, lastName;
public string Name ( )
{
return firstName + “  “ + lastName;
}
}
```

Ця функція дуже схожа на інші функції, розглянуті раніше, за винятком того, що в ній відсутній модифікатор static. Причини цієї відсутності стануть зрозумілі пізніше, а поки досить просто запам'ятати, що це ключове слово для функцій структур не потрібне.

Ми можемо скористатися цією функцією в такий спосіб:

```
customerName myCustomer;
myCustomer.firstName = “Ivan”;
myCustomer.lastName = “Jana”;
Console.WriteLine(customer.Name());
```

Цей синтаксис набагато простіше й зрозуміліше, ніж попередній.

Необхідно відзначити, що функція Name( ) має безпосередній доступ до полів структури firstName і lastName. В рамках структури customerName вони можуть вважатися глобальними.

Перевантаження функцій

Оператор перевантаження надає можливість створювати одночасно декілька функцій, що мають однакові імена, але працюють із параметрами різних типів.

Наприклад, раніше ми використовували наступну програму, в якій містилася функція з ім'ям MaxValue ( ).

```
class Class1
{
    static  int  MaxValue(int[ ] intArray)
    {
```

```
        int maxVal = intArray[0];
        for (int i = 1; i < intArray.Length; i++)
        {
          if (intArray[i] > maxVal)
            maxVal = intArray[i];
        }
        return maxVal;
    }

    static void Main(string[ ] args)
    {
        int[ ] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
        int maxVal = MaxValue(myArray);
        Console.WriteLine("The maximum value in myArray is {0}", maxVal);
    }

}
```

Ця функція може використовуватися тільки для масивів значень типу int.

Ми могли б створити функції з іншими іменами, призначені для роботи з параметрами інших типів, перейменувавши вищенаведену функцію як-небудь накшталт IntArrayMaxValue ( ) і додавши функції на зразок DoubleArrayMaxValue ( ) для роботи з іншими типами.

Як альтернатива ми можемо просто включити в нашу програму наступну функцію:

```
static  double  MaxValue(double  [ ]  doubleArray)
    {
        double  maxVal = doubleArray[0];
        for (int i = 1; i < doubleArray.Length; i++)
        {
          if (doubleArray [i] > maxVal)
            maxVal = doubleArray [i];
        }
        return maxVal;
    }
```

Різниця між двома функціями полягає в тім, що ця функція працює зі значеннями типу double. Ім'я функції – MaxValue( ) – виявляється тим же самим, однак сигнатура (це принципово) відрізняється. Було б помилкою описати дві функції з однаковим ім'ям і однаковою сигнатурою, однак оскільки в даному випадку сигнатури різні, то все нормально.

Тепер у нас є дві версії функції MaxValue( ), які приймають масиви типу int і масиви типу double і повертають максимальне значення типу int або типу double відповідно. Отже, не потрібно явно вказувати, яку з цих двох функцій ми збираємося використовувати. Ми просто задаємо масив-параметр, і це приводить до виконання того варіанта, що відповідає типу використовуваного параметра.

На даному етапі варто відзначити ще одну рису Visual Studio. Якщо в нашому додатку є дві однойменні функції, описані вище, і ми наберемо це ім'я, наприклад в Main( ), то Visual Studio виведе у вигляді довідки доступні варіанти перевантаження даної функції.

При перевантаженні функцій враховуються всі аспекти, що стосуються їхніх сигнатур.

Існує можливість, наприклад, описати дві різні функції, одна з яких приймає параметри за значенням, а інша, відповідно, за посиланням:

```
static void showDouble(ref int val)
{
…….....
}
static void showDouble(int val)
{
……
}
```

Вибір використовуваної версії здійснюється винятково на підставі того, чи є у зверненні до функції ключове слово ref. При наступному виклику буде використаний варіант, у якому параметр передається за посиланням:

showDouble(ref val);

А такий виклик дозволить передати параметр за значенням:

showDouble(val);

Аналогічним чином можна описувати функції, що відрізняються числом параметрів, що ними вимагаються, і т. п.

Делегати

Делегатом називається тип, що дозволяє зберігати посилання на функції.

Основне призначення делегатів навряд чи вдасться зрозуміти доти, доки ви не познайомитеся з подіями та з їхньою обробкою, однак розгляд самого поняття делегатів принесе величезну користь. Коли пізніше наступить час їх використовувати (при вивченні дисципліни «Основи об’єктно-орієнтованого програмування»), вони будуть вам уже знайомі, що зробить деякі складні теми набагато більш легкими для розуміння.

Оголошення делегатів багато в чому нагадує оголошення функцій; при цьому відсутнє само тіло функції, але додається ключове слово delegate.

Оголошення делегата визначає сигнатуру функції, яка складається з типу, що повертається, і списку параметрів. Після оголошення делегата ми одержуємо можливість оголосити змінну типу цього делегата й потім ініціалізувати цю змінну, присвоївши їй посилання на довільну функцію, що володіє сигнатурою, яка збігається із сигнатурою делегата.

У результаті ми одержуємо можливість викликати цю саму функцію за допомогою даної змінної-делегата так, ніби остання сама була цією функцією.

Тепер, коли є змінна, що посилається на функцію, ми також одержуємо можливість виконувати й деякі інші операції, які не можуть бути виконані з використанням інших засобів. Наприклад, з'являється можливість передавати змінну-делегат іншої функції як параметр, що дозволить цієї функції використовувати даного делегата для виклику функції, на яку він посилається, без необхідності визначати викликувану функцію до початку виконання програми.

Приклад. Використання делегата для виклику функції.

```
using System;
namespace Fand_delegate
{
    class Class1
    {
  delegate double processDelegate(double param1, double param2);

  static double Multiply(double param1, double param2)
```

```
    {
      return param1 * param2;
    }

    static double Divide(double param1, double param2)
    {
      return param1 / param2;
    }

    static void Main(string[ ] args)
    {
      processDelegate process;
      Console.WriteLine("Enter 2 numbers separated with a space:");
      string input = Console.ReadLine();
      int spaceaPos = input.IndexOf(' ');
      double param1 = Convert.ToDouble(input.Substring(0, spaceaPos));
      double param2 = Convert.ToDouble(input.Substring(spaceaPos + 1,
        input.Length - spaceaPos - 1));
      Console.WriteLine("Enter M to multiply or D to divide:");
      input = Console.ReadLine();
      if (input == "M")
        process = new processDelegate(Multiply);
      else
        process = new processDelegate(Divide);
      Console.WriteLine("Result: {0}", process(param1, param2));
    }

      }
}
```

Результат роботи програми:

```
Enter 2 numbers separated with a space:
15,5 0,5
Enter M to multiply or D to divide:
M
Result: 7,75
Press any key to continue
============================
```

```
Enter 2 numbers separated with a space:
15,5 0,5
Enter M to multiply or D to divide:
D
Result: 31
Press any key to continue
```

Цей код описує делегата (processDelegate), сигнатура якого збігається із сигнатурою двох функцій: Multiply( ) і Divide( ). Делегат має наступне визначення:

```
delegate double processDelegate(double param1, double param2);
```

Ключове слово delegate вказує, що дане визначення є визначенням делегата, а не функції (це визначення розташовується в тім же місці програми, де могло б перебувати й визначення функції).

Далі приводиться сигнатура, в якій задаються значення типу double, що повертається, і два параметри типу double. Використовувані в сигнатурі імена є довільними, тому тип делегата та імена параметрів ми можемо вибирати на свій розсуд.

У даному випадку ми вибрали ім'я делегата processDelegate, а параметри назвали param1 і param2.

Код функції Main( ) починається з того, що ми оголошуємо змінну, скориставшись описаним типом делегата:

```
static void Main(string[ ] args)
{
  processDelegate  process;
```

Потім іде абсолютно стандартний для C# код, який запитує два числа, розділені комою, і присвоює їх двом змінним типу double:

```
Console.WriteLine("Enter 2 numbers separated with a space:");
  string input = Console.ReadLine();
  int spaceaPos = input.IndexOf(' ');
  double param1 = Convert.ToDouble(input.Substring(0, spaceaPos));
  double param2 = Convert.ToDouble(input.Substring(spaceaPos + 1,
  input.Length - spaceaPos - 1));
```

Зверніть увагу, що для наочності в програму не включені ніякі перевірки допустимості інформації, що вводиться користувачем. У реальній програмі нам довелося б витратити велику кількість часу, перевіряючи допустимість тих значень, які ми одержуємо для локальних змінних param1 і param2.

Потім ми запитуємо користувача, чи варто множити або ділити ці числа:

```
Console.WriteLine("Enter M to multiply or D to divide:");
input = Console.ReadLine();
```

Залежно від отриманої відповіді ми ініціалізуємо змінну-делегат:

```
if (input == "M")
   process = new processDelegate(Multiply);
else
   process = new processDelegate(Divide);
```

Для присвоєння змінної-делегату посилання на функцію нам доводиться використовувати синтаксис, що дивно виглядає. Так само, як і у випадку присвоювання значень масиву, для створення нового делегата використовується ключове слово new. Після цього слова вказується тип створюваного делегата й задається параметр, що визначає ту функцію, яку ми хочемо використовувати, а саме – функцію Console.WriteLine( ). Зверніть увагу, що цей параметр не збігається ні з параметрами типу делегата, ні з параметрами використовуваної функції: це унікальний синтаксис, застосовуваний винятково для здійснення присвоювання делегатові. Як параметр використовується просто ім'я тієї функції, яку необхідно використовувати, без дужок.

Нарешті, ми викликаємо обрану нами функцію за допомогою делегата. Тут застосовується той самий синтаксис незалежно від того, на яку саме функцію посилається делегат:

```
Console.WriteLine("Result: {0}", process(param1, param2));
```

У даному випадку ми поводжуємося зі змінною-делегатом так, якби вона представляла собою ім'я функції. Однак, на відміну від функцій, над цією змінною можна виконувати деякі додаткові операції, наприклад,

передачу її іншій функції як параметр. Простий приклад такого використання може виглядати приблизно таким чином:

```
static void executeFunction(processDelegate process)
{
process(2,2 3,3);
}
```

Це означає, що ми одержуємо можливість керувати поводженням функцій, передаючи їм делегати функцій (як параметри).

Наприклад, може бути функція, використовувана для сортування строкового масиву за алфавітом. Існують різні алгоритми сортування списків; вони мають різну швидкість, котра залежить від характеристик списку, який сортується. За допомогою делегатів ми отримуємо можливість визначити метод, що необхідно використовувати, передаючи функції, яка виконує сортування, делегата тієї функції, в якій реалізований відповідний алгоритм сортування.

Варіантів застосування делегатів існує дуже багато, але, як відзначалося вище, найбільш плідно вони використовуються при обробці подій.

7.4. Приклад використання функцій

Приклад обробки одномірного масиву, що приводиться нижче, не має потреби в коментарях. Функціональність програми добре видна з назви відповідних змінних і функцій.

Приклад. Обробка одномірного масиву з використанням функцій.

```
using System;
    class Class1
    {
        static void pech_mas(int[ ] m)
        {
            for (int i = 0; i < m.Length;  i++)
                Console.Write("{0} ",m[i]);
            Console.WriteLine(); Console.WriteLine();
        }
```

```
static void sum_otr(int[ ] m)
{
	double sum_otr = 0.0;
	for(int i=0; i<m.Length; i++)
		if(m[i]<0)
			sum_otr += m[i];
	Console.WriteLine( "sum_otr = {0}", sum_otr);
}

static void proisv_maxMod_minMod(int[ ] m)
{
	int proisv_maxMod_minMod = 1;

	int maxMod,minMod;
	maxMod = minMod = Math.Abs(m[0]);

	int index_maxMod, index_minMod;
	index_maxMod = index_minMod = 0;

	for(int i=0; i<m.Length; i++)
	{
		if( Math.Abs(m[i]) < minMod )
		{
			minMod = Math.Abs(m[i]);
			index_minMod=i;
		}

		if( Math.Abs(m[i])> maxMod )
		{
			maxMod = Math.Abs(m[i]);
			index_maxMod=i;
		}
	}
	for(int i=0; i<m.Length; i++)
		if(	i>index_minMod	&&	i<index_maxMod	||
i<index_minMod && i>index_maxMod)
			proisv_maxMod_minMod *= m[i];
```

```
            Console.WriteLine("index_minMod = {0}, index_maxMod = {1}",
index_minMod, index_maxMod);
            Console.WriteLine("minMod = {0}, maxMod = {1}", minMod,
maxMod);
            Console.WriteLine("index_minMod = {0}, index_maxMod = {1}",
index_minMod, index_maxMod);

            Console.WriteLine("proisv_maxMod_minMod = {0}",
proisv_maxMod_minMod);

            }
            static void Main(string[ ] args)
            {
                  int[ ] mas = {-1,5,2,3,5,0,3,9,2,0,1,6,0,-3,2};
                  pech_mas(mas);
                  sum_otr(mas);
                  proisv_maxMod_minMod(mas);
                  Console.WriteLine();
            }

      }
```

Результат роботи програми:

```
-1 5 2 3 5 0 3 9 2 0 1 6 0 -3 2

sum_otr = -4
index_minMod = 5, index_maxMod = 7
minMod = 0, maxMod = 9
index_minMod = 5, index_maxMod = 7
proisv_maxMod_minMod = 3

Press any key to continue
```

Підведемо підсумки темам, які обговорювалися в даному розділі:

Визначення й використання функцій в консольних додатках.

Обмін даними з функціями за допомогою параметрів і значень, що повертаються.

Масиви параметрів.

Передача параметрів за посиланням і за значенням.

Використання вихідних параметрів для додаткових значень, що повертаються.

Поняття області дії змінної.

Використання функцій у типах структур.

Перевантаження функцій.

Делегати.

Питання для самоконтролю

1. Дайте призначення функції. Чому функція може служити прикладом коду, який повторно виконується?

2. Перерахуйте основні етапи виконання функції.

3. Який синтаксис запису значень, що повертаються з функції?

4. Охарактеризуйте параметри функцій. У чому полягає відповідність параметрів?

5. Як обробляти масиви параметрів?

6. У чому полягають основні ідеї реалізації процесу обміну інформацією з функцією.

7. Що таке область дії змінних? Охарактеризуйте параметри і значення, що повертаються, за порівнянням із глобальними даними.

8. У чому полягають особливості передачі параметрів за посиланням і за значенням.

9. Які особливості використання функції і структури?

10. Які особливості використання функції і масиву?

11. Що таке перевантаження функцій? Коли його доцільно вживати?

12. Що розуміється під делегатом (посиланням на функції).

Рекомендована література

Основна

1. Си Шарп. Создание приложений для Windows. / В. В. Лабор –, СПб.: Мн. Харвест. – 2003. – 384 с.

2. Программист – программисту. C#. / Карли Ватсон и др. – Пер. с англ. – М.: Издательство «Лори», 2005. – 862 с.

3. C# без лишних слов. / Робинсон У. – Пер. с англ. – М.: ДМК Пресс, 2002. – 352 с.

4. Петцолъд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 1. / Пер. с англ. — М.: Издательско-торговый дом «Русская редакция», 2002. – 576 с.

5. http://www.natahaus.ru/

6. http://www.intuit.ru/department/pl/visualcsharp/1/2.html

Зміст