МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ ІМЕНІ СЕМЕНА КУЗНЕЦЯ

ПРОГРАМУВАННЯ ЗАСОБІВ МУЛЬТИМЕДІА

Конспект лекцій
для студентів спеціальності
186 "Видавництво та поліграфія"
першого (бакалаврського) рівня

Харків
ХНЕУ ім. С. Кузнеця
2023

УДК 004.4'27(075.034)
Б87

Рецензент – доцент кафедри інформаційних систем та технологій Полтавського державного аграрного університету, канд. с.-г. наук, доцент *Ю. В. Вакуленко*.

Затверджено на засіданні кафедри комп'ютерних систем і технологій.
Протокол № 12 від 17.05.2023 р.

Самостійне електронне текстове мережеве видання

Браткевич В. В.

Б87 Програмування засобів мультимедіа [Електронний ресурс] : конспект лекцій для студентів спеціальності 186 "Видавництво та поліграфія" першого (бакалаврського) рівня / В. В. Браткевич, І. О. Хорошевська. – Харків : ХНЕУ ім. С. Кузнеця, 2023. –176 с.

Подано матеріал, що допомагає засвоїти лекційні теми першого розділу навчальної дисципліни. Розглянуто мову C#, як сучасну базову мову програмування та середовище програмування Microsoft Visual Studio .NET. Виклад теорії проілюстровано великою кількістю прикладів, більшу частину з яких наведено в графічному вигляді, що істотно полегшує розуміння поточних тем.

Рекомендовано для студентів спеціальності 186 "Видавництво та поліграфія" першого (бакалаврського) рівня.

УДК 004.4'27(075.034)

© Браткевич В. В., Хорошевська І. О., 2023
© Харківський національний економічний університет імені Семена Кузнеця, 2023

Вступ

Навчальну дисципліну "Програмування засобів мультимедіа" вивчають студенти спеціальності 186 "Видавництво та поліграфія" всіх форм навчання протягом III і IV семестру. У результаті вивчення навчальної дисципліни студенти оволодіють навичками у складанні та налагоджуванні відповідних програм сучасними С-подібними мовами.

Структурно конспект складається із двох розділів, у кожному з яких наведено матеріал відповідних тем розділів "Організація процедурно-орієнтованих програм" і "Організація і обробка складених типів даних".

Розглянуто дві парадигми програмування: процедурну та об'єктно-орієнтовану. Навчальна дисципліна є базовою для подальшого вивчення сучасних вебтехнологій розроблення мультимедійних продуктів.

Мета навчальної дисципліни: надання здобувачам вищої освіти теоретичних знань і прикладних умінь у галузі застосування сучасних об'єктно-орієнтованих мов програмування для інструментального підтримання технологічного процесу виробництва видавничо-поліграфічних і мультимедійних продуктів.

Завданнями навчальної дисципліни є такі:

теоретичні та методологічні засади організації програм і даних;

опрацювання лінійних процесів і процесів із розгалуженням та ітераціями;

реалізація типових алгоритмів пошуку та сортування в одновимірних і двовимірних масивах;

використання вмонтованих функцій;

оголошення і використання функцій користувача;

користування раніше складеними програмами та здійснювання супроводу програм;

унесення змін до програми;

налагоджування програм за допомогою вбудованих інструментальних засобів.

Об'єктом вивчення навчальної дисципліни є технології розроблення програм мовою C# в середовищі платформи .NET Framework.

Предметом вивчення навчальної дисципліни є алфавіт, синтаксис і семантика мови C#, структури даних та типові алгоритми їхнього опрацювання.

Інструментальною базою вивчення навчальної дисципліни є платформа .NET Framework Microsoft Visual Studio.

Основною метою конспекту лекцій із навчальної дисципліни "Програмування засобів мультимедіа" є надання можливості закріплення студентами спеціальності 186 "Видавництво та поліграфія" таких компетентностей:

- здатність ухвалювати обґрунтовані рішення;
- здатність застосовувати відповідні математичні й технічні методи та комп'ютерне програмне забезпечення для вирішення інженерних завдань видавництва та поліграфії;
- здатність застосовувати принципи опрацювання інформації й особливостей її використання для виготовлення мультимедійних інформаційних продуктів та інших видів виробів видавництва і поліграфії.

Розділ 1
Організація процедурно орієнтованих програм

1. Теоретичні та методологічні засади організації програм і даних

Передумовою для опанування поточного матеріалу лекції є перелік попередньо прослуханих навчальних дисциплін: "Інформатика і комп'ютерна техніка" та "Інформаційні технології".

1.1. Основні концепції й термінологія

Принцип програмного управління комп'ютером. Комп'ютер є універсальним інструментом для вирішення різноманітних завдань із перетворення інформації, але його універсальність визначено не стільки апаратним забезпеченням, скільки встановленими програмними засобами, інакше кажучи, усі "знання" комп'ютера зосереджено в програмах, які становлять точну й детальну послідовність інструкцій, поданих зрозумілою для комп'ютера мовою, з опрацювання інформації. Змінюючи програми, можна перетворити комп'ютер на робоче місце дизайнера, бухгалтера, конструктора, використовувати його для прослуховування музики, перегляду кінофільмів та інших розваг.

Основні принципи побудови комп'ютерів, описані Джоном фон Нейманом, досі є стандартом практично для всіх комп'ютерів. Одним із них є програмне управління.

В основі принципу програмного управління лежить уявлення алгоритму розв'язання будь-якої задачі у формі програми обчислень.

Алгоритм – це точне розпорядження, що визначає процес перетворення вихідних даних на кінцевий результат.

Програма – це впорядкована послідовність команд, що підлягає опрацюванню; описує операції, які потрібно виконати процесору комп'ютера для вирішення поставленого завдання.

Команда – це інструкція машині на виконання елементарної операції. Набір операцій, які може виконувати комп'ютер, і правил їхнього запису утворюють машинну мову.

Сутність принципу програмного управління полягає в такому [1 – 3]:

- усі обчислення, запропоновані алгоритмом вирішення завдання, мають бути поданими у формі програми, що складається з послідовності керівних слів-команд;
- кожна команда містить указівки на конкретну виконувану операцію, місце знаходження (адресу) операндів і ряд службових ознак. **Операнди** – це змінні, значення яких беруть участь в операціях перетворення даних. Список усіх змінних (вхідних і даних, проміжних значень і результатів обчислень) є невід'ємним елементом будь-якої програми;
- для доступу до програм, команд і операндів використовують їхні адреси, якими є номери комірок пам'яті комп'ютера, призначених для зберігання об'єктів;
- команди програми розташовано в пам'яті одна за одною, що дозволяє процесору організовувати вибірку ланцюжка команд із послідовно розміщених комірок пам'яті та виконувати команду за командою;
- для переходу до виконання наступної за порядком команди, використовують команди умовного або безумовного переходів. Вибірка команд із пам'яті припиняється після досягнення кінця програми або виконання команди "стоп". Отже, процесор виконує програму автоматично, без утручання людини.

Програмне забезпечення (англ. Software) – це сукупність програм, що забезпечують функціонування комп'ютерів і вирішення завдань предметних галузей.

Програмне забезпечення (ПЗ) становить невід'ємну частину комп'ютерної системи, є логічним продовженням технічних засобів і визначає сферу застосування комп'ютера.

ПЗ сучасних комп'ютерів містить безліч різноманітних програм, які можна умовно розподілити на три групи:

Системне програмне забезпечення (системні програми) – управляє роботою комп'ютера і виконує різні допоміжні функції, наприклад, управління ресурсами комп'ютера, створення копій інформації, перевірку працездатності пристроїв комп'ютера, видачу довідкової інформації про комп'ютер та ін.

Прикладне програмне забезпечення (прикладні програми) – призначено для вирішення конкретних завдань користувача в предметних галузях.

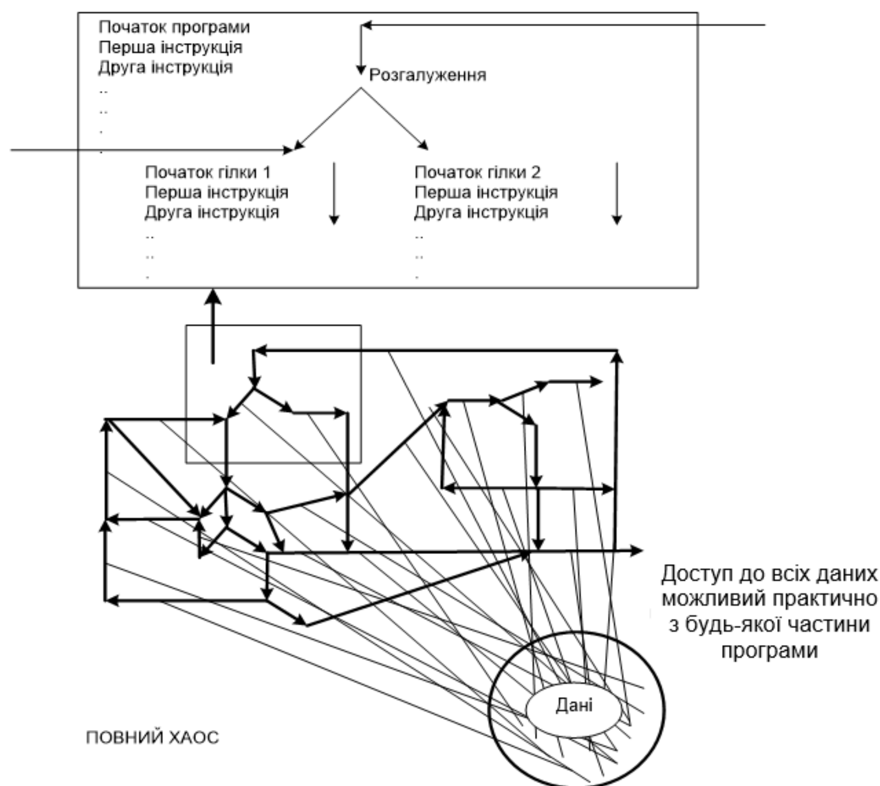
Інструментальне програмне забезпечення (інструментальні системи). До нього належать системи програмування (для розроблення

нових програм), інструментальні середовища (для розроблення застосунків) і системи моделювання.

Процедурно орієнтоване програмування. Одним із традиційних підходів до проєктування комп'ютерних програм був процедурно орієнтований стиль. У ньому найвищий пріоритет належить діям, що виконує елемент програми, тоді як дані, із якими працює програма, залишаються ніби на другому плані.

Типова *процедурно орієнтована програма* – це послідовність інструкцій, виконуваних одна за одною. У такій програмі, зазвичай, є численні точки розгалуження, у яких вибирають лише один із декількох можливих напрямів виконання, залежно від умов у програмі. Більша частина інструкцій маніпулює даними.

У застарілих процедурно орієнтованих програмах доступ до даних був можливим із будь-якої частини програми (рис. 1.1), а операції над ними – за допомогою будь-якої інструкції.



Потік виконання програми
Указує фрагмент даних, яким маніпулює код. Один кінець зв'язку вказує на розташування даних у пам'яті, другий – де ці дані використовують у вихідному коді

Рис. 1.1. **Структура процедурно орієнтованої програми**

Слід зазначити, що багато з великих фрагментів вихідного коду, написаних такими об'єктно орієнтованими мовами, як C#, створено з використанням примітивних процедурно орієнтованих мовних конструкцій.

Цей підхід є прийнятним для дуже малих проєктів, але під час створення великих і складних програм розроблювачеві доводиться відстежувати всі можливі розгалуження в коді програми, оскільки всі її частини більш-менш є взаємозв'язаними. Навіть більше, різні фрагменти коду програми можуть дістати доступ до того самого значення даних – причому вони можуть не тільки читати, але й змінювати його.

Під час розроблення однієї частини програми можна й не знати про те, що дані, із якими вона працює, може бути змінено іншими частинами програми. Ситуація дуже швидко наближається до повного хаосу, коли над проєктом спільно працюють декілька програмістів.

Об'єктно орієнтоване програмування (ООП). Як же подолати розглянуту раніше проблему процедурно орієнтованого стилю програмування? Типовий підхід до подолання складної проблеми – розподілити її на більш прості частини.

Спробуймо розподілити попередній приклад на чотири менших, самостійних фрагменти. Результат показано на рис. 1.2.

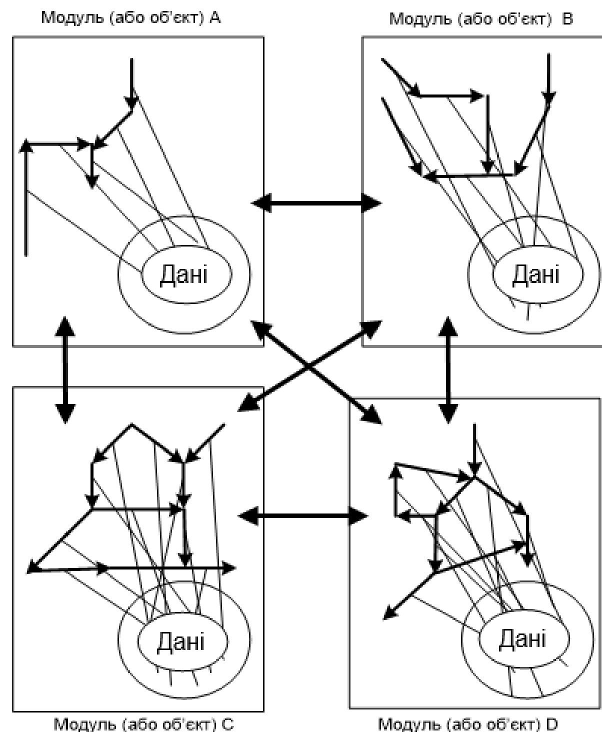


Рис. 1.2. **Розподіл процедурно орієнтованої програми на самодостатні модулі**

Тут весь набір інструкцій розподілено на чотири окремих модулі. В об'єктно орієнтованому світі такі модулі називають **об'єктами**. Дані також розподілено на чотири частини, оскільки кожний об'єкт містить лише ті, із якими він працює. Тепер під час виконання 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 із яких може бути "вилучено" із програми та "вставлено" в неї знову.

Тепер не так уже складно досягти, щоб модулі створювали й підтримували різні програмісти. Як можна бачити з рис. 1.2, об'єкти використовують для об'єднання даних із методами (інструкціями), що оперують над цими даними.

В об'єктно орієнтованому світі дані та процедури мають однакове значення.

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

Щоб відповісти на ці запитання, насамперед, слід з'ясувати, що таке об'єкт.

Поняття об'єкта. У повсякденному житті нас оточують об'єкти: книги, будинки, машини, собаки, люди ... . Часто об'єкти є здатними брати участь у певних діях. Автомобіль, наприклад, є здатним відчиняти двері, зачиняти 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 відчиняє його двері, або об'єкт "автомобіль", який містить об'єкт "людина" та транспортує його з точки А до точки Б. Під час написання програми за участю автомобілів і людей із використанням процедурно орієнтованої методології всю увагу буде приділено діям ("відчинити двері", "зачинити двері" тощо).З іншого боку, під час використання об'єктно орієнтованого програмування (ООП) програму буде побудовано навколо об'єктів "автомобіль" і "людина".

Отже, ООП – це методологія, у якій під час моделювання конкретних ситуацій використовують об'єкти.

Комп'ютерне моделювання намагається імітувати процеси, що відбуваються в реально наявній або теоретичній системі. Збираючи й аналізуючи дані, визначені в цій штучній системі, можна дістати цінні відомості

про внутрішні особливості системи реальної. Для успішного моделювання потрібно розробити модель і реалізувати її в комп'ютері. Часто модель є спрощенням реальної системи, однак вона повністю відповідає внутрішнім процесам і станам реальної системи, що модулюють.

Щоб заповнити розрив між реальним світом і моделюванням у комп'ютері, потрібно, насамперед, ідентифікувати об'єкти, що беруть участь у цьому процесі. Звичайно, це один із перших кроків під час розроблення програми в ООП, і він прямо відповідає кроку б) (розподіл кожної підсистеми на модулі) процесу розроблення програми, що обговорювали раніше.

Як приклад розгляньмо ліфтову систему будівлі в дії. Які об'єкти в цьому беруть участь? Непоганий підхід: звертати увагу на те, які іменники використовують під час опису системи, оскільки іменники прямо відповідають об'єктам. Далі наведено опис системи, у якому реальні об'єкти виділено 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 натиснути **кнопку** на поверсі, де вона перебуває.

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

Наприклад, об'єкт "ліфт" може мати такі атрибути: "максимальна швидкість", "поточне місце розташування", "поточна швидкість", "максимальна кількість людей, що може вмістити ліфт", тощо. Функції об'єкта можуть бути такими: "підніматися", "опускатися", "зупинитися" та "відчинити двері". Визначивши правильні атрибути та функції кожного об'єкта в реальному світі, їх можна подати в комп'ютерній моделі. Тут потрібно розширити запас термінології ООП ще декількома важливими термінами.

Кожну частину програми на C#, що становить об'єкт реального світу, відповідно називають **об'єктом**.

Істотні атрибути об'єкта реального світу мають бути поданими у відповідному об'єкті програми на *C#*. Ці атрибути, які називають **змінними екземпляра**, відображають поточний стан об'єкта. Змінні екземпляра є еквівалентними даним на рис. 1.2.

Поведінку об'єкта реального світу подано в об'єкті програми C# у формі методів. Кожний метод містить інструкції. Методи об'єкта виконують дії зі змінними екземпляра цього самого об'єкта. Методи є еквівалентними потокам виконання на рис. 1.2.

Поняття класу. Ще один важливий термін ООП – клас. **Клас** визначає загальні риси (змінні екземпляра і методи) групи подібних один одному об'єктів. Отже, усі об'єкти одного класу мають ті самі змінні екземпляра та методи. Які змінні екземпляра й методи додавати у створюваний клас, програміст вибирає сам: усе залежить від потреб створюваної програми.

Як приклад класу, розгляньмо автомобіль із концептуального погляду. У 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 наявні автомобілі можна вважати об'єктами.

Щоб забезпечити опис конкретного автомобіля як об'єкта програми на C#, програміст ухвалює рішення про додавання чотирьох змінних екземпляра у клас **Автомобіль: Марка, Поточне місце розташування, Максимальна швидкість** і **Поточна швидкість**. Слід зазначити, що атрибут **Колір** (Color) не додано у клас, оскільки програміст уважав його непотрібним для цієї програми.

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

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

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

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

Можливості мови C#. Як уже зазначалося раніше, нині є безліч мов програмування. І, незважаючи на те, що будь-яку програму можна написати практично будь-якою мовою, багато мов було пристосовано для розв'язання більш-менш конкретного кола проблем.

C# і .NET працюють винятково в операційних системах Microsoft, але водночас вони надають програмістові в межах цих операційних систем дуже потужні засоби. Важливо, що C# – багатоцільова мова, і в межах платформи Windows її можна використовувати для створення найрізноманітніших програм.

Далі наведено короткий список лише декількох категорій програм, під час створення яких комбінація переваг C# і .NET забезпечить високу ефективність праці програміста C#.

Консольні застосунки. У консольних застосунках для взаємодії з користувачем використовують одне просте вікно. У них немає вражаючих графічних інтерфейсів і анімації; інформацію виводять у символьному режимі. Водночас, програми виходять більш простими, тоді як для насичених графікою застосунків складність – це звичайна справа.

Незважаючи на те що комерційних програм із консольним інтерфейсом не так багато, створення їх – це чудовий спосіб вивчення мистецтва програмування. Він дає змогу зосередитися на мовних концепціях і допомагає швидко набути розуміння мови, необхідного для створення більш складних графічних програм і, що, мабуть, є найважливішим для спеціальності 186, – дозволяє відносно легко адаптуватися до процесу написання "С-подібних" скриптів, широко використовуваних у різних програмних середовищах розроблення засобів мультимедіа.

Програмування для консолі аж ніяк не є долею лише новачків. Професійні програмісти часто використовують вікно консолі для тестування застосунків і компонентів.

Віконні застосунки на базі WinForms. На відміну від консольних, у них використовують графічний інтерфейс користувача, де команди користувача передають шляхом клацань мишки на екранних кнопках і піктограмах, а введення інформації здійснюють через різні текстові вікна та вікна списків.

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

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

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

Web-служби. **Web-служби** – це важлива частина нової технології, що змінює шляхи використання Internet, а з ними – уявлення про те, як розробляти та використовувати застосунки. Web-служби подають просто компоненти або застосунки, доставлені на комп'ютер з Internet. Із web-служб, розміщених на різних комп'ютерах, і пов'язаних з Internet, може бути сформовано нові web-служби.

Це відкриває для програмістів різні цікаві можливості. Зокрема, виникає можливість збирати компоненти й застосунки не тільки із класів .NET Framework і вихідного коду, написаного самим програмістом, але й з web-служб, визначених в Internet.

1.2. Етапи розроблення програмного забезпечення

Процес розроблення програмного забезпечення (ПЗ) – це послідовність дій, кінцевим продуктом якої є комп'ютерна програма. За минулі 20 років було виділено чітко визначені етапи процесу розроблення ПЗ. У цьому пункті розглядають лише найбільш важливі з них [4; 5]:

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

2. Проєктування програми. Розроблення концепції, що дозволяє втілити вимоги до специфікації у програмі, яка працює. Вона визначає шляхи реалізації функціональності, описаної у специфікації, засобами мови програмування високого рівня, наприклад C#.

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

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

Ці етапи слід виконувати в зазначеній тут послідовності. Однак іноді виникають ситуації, коли для продовження роботи над програмою потрібно повернення до попереднього етапу.

Розгляньмо кожний з етапів більш докладно.

Створення специфікації програми. Навіщо визначати вимоги до програми? По-перше, якщо чітко невідомо, що програма має 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 розв'язання).

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

Проєктування програми. Проєктування програми може містити множину різних дій залежно від розміру й характеру конкретного проєкту. Великий проєкт може охоплювати декілька стадій, описаних далі:

а) розподіл усього проєкту на різні функціональні підсистеми. Прикладами підсистем можуть бути графічні інтерфейси користувача, генератори звітів, інтерфейси до баз даних. Слід зазначити, що проєкти, розглянуті далі, є занадто малими, і їх не варто розподіляти на підсистеми. Отже, наведені далі програми можна потенційно розглядати як частину функціональної підсистеми великого проєкту;

б) розподіл кожної підсистеми на модулі. Термін "модуль" дуже гнучкий і набуває різних значень у 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 цієї підсистеми виконувати вимоги, адресовані до неї.

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

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

г) внутрішнє проєктування методів. Проєктуванням на цьому рівні звичайно займається програміст, який розробляє конкретний метод. На цьому етапі розробляють алгоритми виконання дуже вузьких, конкретних завдань і пишуть перші оператори мовою високого рівня, наприклад C#.

Написання програми. Ця частина процесу створення ПЗ є обов'язковою в проєктах будь-якого масштабу.

Тестування й налагодження програми. Програму піддають різним тестам (як під час написання, так і після нього). Тести є необхідними, щоб переконатися, що програма виконує саме те, що має виконувати. **Тестування**, зокрема, дозволяє виявити (і виправити) помилки, допущені у процесі проєктування та написання програми.

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

Невеликі, неформальні проєкти (наприклад лабораторний практикум із цієї навчальної дисципліни) часто містять лише рівні 3 і 4. Програміст може виконати їх, сидячи перед комп'ютером. Стадію проєктування програми реалізовано або в голові у програміста, або на аркуші паперу у формі декількох діаграм, або у пошуку декількох стандартних алгоритмів у підручнику.

1.3. Огляд середовища розроблення Visual Studio .Net

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

.NET – це загальний термін для багатьох служб, які надають і використовують під час створення та виконання програми на C# [5].

Особливості інфраструктури .NET-платформи. C# повністю залежить від .NET, і тому походження багатьох концепцій C# сягає далеко .NET.

Далі перелічено особливості інфраструктури .NET-платформи:

.NET надає засоби для виконання інструкцій, що містяться у програмі, написаній на C#. Цю частину .NET називають середовищем виконання **(execution engine)**;

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

.NET звільняє програміста від виснажливого процесу і такого, що нерідко веде до помилок під час управління 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 і ретельно перевірений професійними програмістами Microsoft.

У .NET спрощено підготовку програми до використання **(розгортання)**.

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

Нині на платформу .NET перенесено близько 20 мов. Оскільки для виконання коду, написаного будь-якою мовою, що підтримується платформою .NET, використовують те саме середовище виконання, його часто називають єдиним середовищем виконання **(Common Language Runtime, CLR)**.

Програму, під час створення якої було передбачено можливість повторного використання, називають компонентом (**програмним компонентом**).

Мови програмування та компілятори. Програмувати на перших комп'ютерах, виготовлених у 40-х рр. минулого століття, було дуже непросто. Для цього використовували **машинну мову**, що складалася з послідовностей бітів, які прямо управляють простими діями процесора. Програмування на рівні машинної мови – заняття надзвичайно трудомістке та виснажливе.

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

Першим результатом пошуків стали так звані **асемблери** – мови, у яких використовували більш зрозумілі людині команди, наприклад move, getint або putint. Але, незважаючи на те що асемблери були трохи простішими для читання й розуміння, їх поєднувала з машинним кодом одна важлива загальна риса: розроблювач під час написання програми мав мислити в термінах низькорівневих операцій процесора та пам'яті.

Однак еволюція комп'ютерів і щораз вищі потреби у все більш складних програмах привели до появи повністю **машинно незалежних мов** програмування. Першою з них стала FORTRAN, створена в середині 50-х рр. XX ст. Незабаром виникли інші мови високого рівня. Сьогодні їхня кількість за деякими оцінками перевищує дві тисячі.

Одним з останніх доповнень у родині мов високого рівня став C#. Але як би високо ми не відходили від базових інструкцій процесора, нам,

як і раніше, є потрібним машинний код, який апаратне забезпечення комп'ютера може розуміти, а отже, і виконувати.

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

На рис. 1.3 наведено ілюстрацію того, як вихідний код типової мови високого рівня перетворюється на програму, що виконують.

Написаний текст, який містить інструкції мови високого рівня, називають *вихідним кодом* [6; 7].

У разі C# цей вихідний код зберігають у файлі з розширенням **.cs**.

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

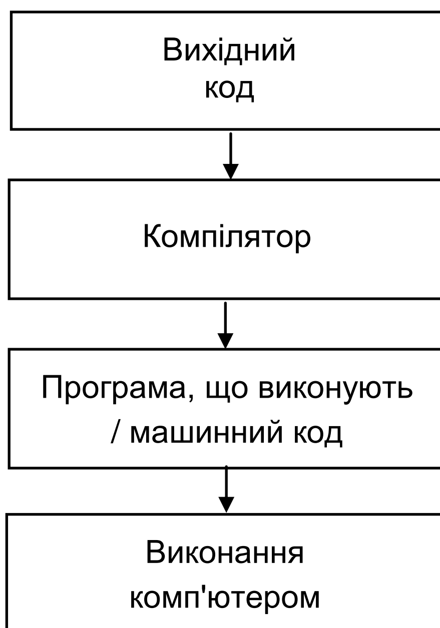


Рис. 1.3. **Традиційний процес компіляції**

Компіляція в .NET вихідного коду C#. Традиційний процес компіляції вихідного коду, написаного мовою програмування високого рівня, у програму, що виконують, мав кілька недоліків. Найбільш істотними з них є такі:

Недолік 1. Для конкретної апаратної платформи, що характеризується типом процесора тощо, потрібен свій компілятор, оскільки в кожної з них – своя машинна мова. Відповідно, якщо треба виконувати програму, написану на C#, на чотирьох комп'ютерах із процесорами різних виробників, буде потрібно чотири різні компілятори C#. До того ж щораз, коли

виробник апаратного забезпечення випускає нове покоління своїх процесорів, у компілятор доводиться вносити зміни та доповнення.

Недолік 2. У більшості програмістів є улюблена мова програмування, якій вони віддають перевагу над усіма іншими. Можливо, кращим рішенням було б дозволити кожному члену команди писати своєю улюбленою мовою, але це нелегко, якщо дотримуватися процесу компіляції, розглянутому на рис. 1.3.

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

Цю проблему було покликано розв'язати так звані **компонентні системи** (як-от **CORBA** і **COM**). Вони визначали стандарти взаємодії між різними частинами програм.

Програміст A писав компонент X, скажімо, мовою Visual Basic, і цей компонент міг взаємодіяти з компонентом Y, розробленим програмістом B на C++.

Компонентні системи мали комерційний успіх. Однак їхнє поширення спричинило інші проблеми, однією з яких стала неможливість забезпечити взаємодію "зовнішнього" компонента з іншими частинами програми на такому самому рівні, якби всі частини програми було написано однією мовою.

У C# і .NET реалізовано рішення описаних раніше двох проблем. Розгляньмо всі складові процесу компіляції програми в .NET (рис. 1.4) [8].

Насамперед, варто звернути увагу на появу на рис. 1.4 ще двох мов – C++ і Visual Basic. І незалежно від того, якою мовою (із підтримуваних .NET) написано програму, це ніяк не впливає на процес її компіляції в .NET.

Після того як написано вихідний код, його потрібно відкомпілювати в машинний код. Однак спочатку його компілюють в іншу мову, що називають **Microsoft Intermediate Language (MSIL)**. Крім того, усі компілятори, орієнтовані на .NET-платформу, мають генерувати на виході код цієї проміжної мови MSIL.

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

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

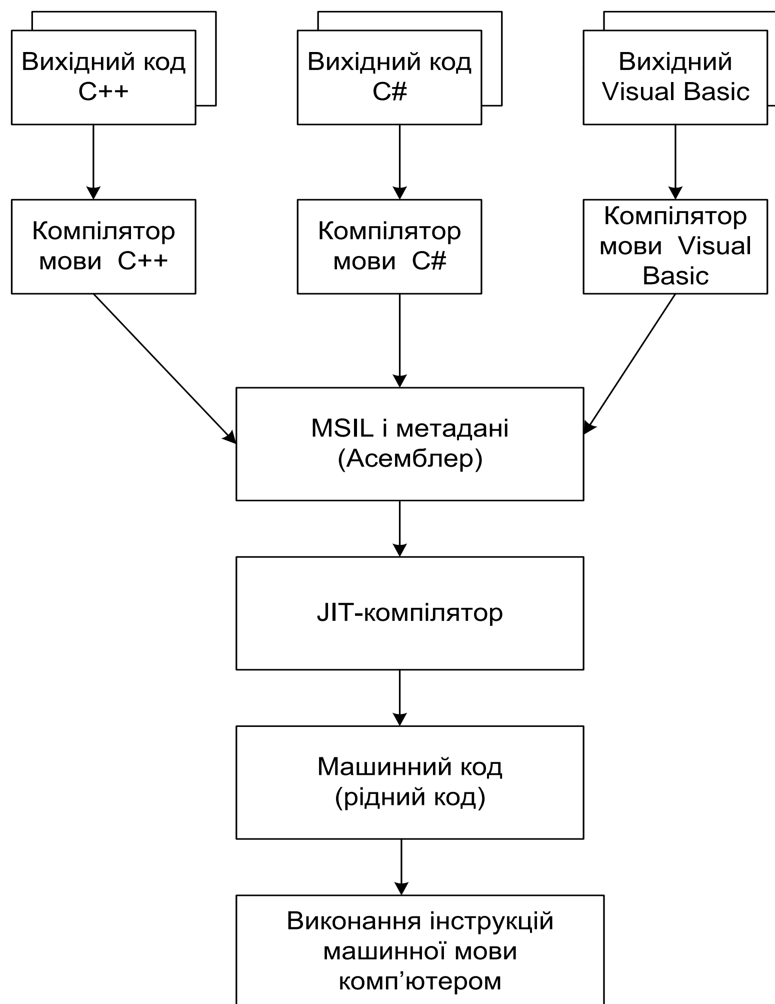


Рис. 1.4. **Процес компіляції в .NET**

Код, що генерують JIT-компілятором, нічим не відрізняється від машинного коду, що генерують звичайним компілятором, однак JIT-компілятор використовує трохи іншу стратегію. Замість того щоб, інтенсивно використовуючи пам'ять і витрачаючи значний час, перетворити на машинний код відразу весь код MSIL, він компілює в машинний код лише ті частини застосунку, які реально є потрібними нині. У результаті код компілюють "на ходу", безпосередньо перед виконанням, і JIT-компілятор не витрачає час на компіляцію MSIL-коду, що не використовують.

Переваги архітектури .NET.

1. Увівши MSIL (див. рис. 1.4) між мовою високого рівня та машинною мовою, ми фактично відокремили ці мови одну від одної.

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

На кожному комп'ютері застосовують свій JIT-компілятор, що перетворює код MSIL на машинний код, сумісний із цією конкретною конфігурацією. У результаті все, що потрібно, – це компілювати код, написаний мовою високого рівня, в універсальний код, мова якого залишається незмінною. Це і є розв'язанням згаданої раніше першої проблеми.

Розгляньмо блок "MSIL і метадані", показаний на рис. 1.4. Термін *"метадані"* можна перекласти як "дані про дані". Метадані, які генерують компілятором мови високого рівня, містять докладний опис елементів вихідного коду. Опис цей є настільки докладним, що вихідний код інших мов зможе використовувати цей код так, якби він був написаним тією самою мовою. Тепер програмісти, які пишуть на C++, C# и Basic, реально зможуть працювати в межах одного проєкту і, отже, у такий спосіб долають недолік 2.

Важливо знати про наявність MSIL, але в повсякденному програмуванні стикатися з ним прямо не доводиться. Звичайно застосовують дві команди: першу – для компіляції програми в MSIL-код і метадані, другу – для виконання програми (водночас буде викликано JIT-компілятор). Фактично виконання програми – це виконання кінцевого результату роботи компіляторів. MSIL у цьому процесі залишається "невидимим" для користувача.

Повторне використання програмного забезпечення. Повторне використання програмного коду покладено в основу програмування в .NET і C#. На ранній стадії написання кожну програму подано зовсім незалежним проєктом, розроблюваним із нуля. Однак це не зовсім раціональний спосіб створення програм, і сьогодні більшість із них створено за іншою методологією, основу якої становить повторне використання заздалегідь розроблених і налагоджених програм або їхніх частин.

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

Один із найбільш привабливих боків об'єктно орієнтованих мов (зокрема C#) – це ретельно продумане підтримування повторного використання коду. Наприклад, **class** виявився дуже зручним 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 потрібно, ресурсів, наприклад зображень.

Складанням є будь-яка програма, написана для .NET, чи то компонент для повторного використання, чи то самодостатня програма, що виконують.

Складання можна розглядати із двох поглядів. Із погляду розроблювачів складання, які розглядають його ізсередини, на рівні вихідного коду, і з погляду користувачів складання, які розглядають його іззовні, коли потрібно підібрати прийнятний компонент для повторного використання в тому або тому проєкті.

Бібліотека класів .NET Framework. Протягом багатьох років у різних типах програм постійно використовували ті самі функції та алгоритми. Прикладом може бути сортування списку, спеціалізовані інженерні розрахунки, математичні обчислення тощо. Цей факт усвідомило багато компаній і розроблювачів ПЗ, що почали створювати бібліотеки класів, у яких містилися подібні широко використовувані функції. Сьогодні важко знайти застосунок, у якому б не використовували частини з повторно використовуваних бібліотек класів.

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

Серед основних механізмів, використовуваних у BCL, принципи ООП, технологія складань і пов'язані з нею концепції. У результаті BCL є простою у використанні та забезпечує широкі можливості повторного використання коду.

У мові C# немає власної бібліотеки класів, бо повністю покладається на BCL і є тісно інтегрованим із нею. Відповідно, програму, написану на C#, неможливо запустити без BCL та середовища виконання .NET.

В основі всіх класів, написаних у C#, лежить один конкретний клас BCL, і багато конструкцій C# є лише поданням тих класів і їхніх функцій, що містяться у BCL. Наявність BCL значно спрощує доступ до служб операційної системи. Замість того щоб використовувати малозрозумілі команди і складні вирази, служби ОС стають доступними у формі, набагато більш дружній користувачеві. Прикладом може бути надаване BCL підтримання віконних графічних інтерфейсів користувача.

Контрольні запитання

1. Що означає .NET? У чому особливість .NET-платформи?
2. Перелічіть основні етапи розроблення програми, що виконують.
3. Навіщо необхідний етап компіляції?
4. У чому особливість компіляції в .NET вихідного коду C#?
5. У чому полягає сутність 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 та поводження.
8. Що таке "клас"?
9. Навіщо необхідно повторне використання програмного забезпечення? Наведіть приклади повторного використання.
10. Наведіть призначення бібліотеки класів .NET Framework.
11. Опишіть можливі типи C#-додатків і галузі їхнього застосування.

2. Поняття типу даних

2.1. Концепція типу даних

Усяка алгоритмічна мова містить три складові частини: **алфавіт** (кінцева множина відмінних між собою символів, використовуваних у цій мові); **синтаксис** (сукупність правил, що визначають припустимі (правильні) конструкції цієї мови). Синтаксис мови C# визначено у спеціальній граматиці); **семантика** (сукупність правил, що визначають значеннєвий зміст окремих конструкцій. Семантика забезпечує однозначність тлумачення всіх понять мови) [6 – 10].

Алфавіт – це символи, що використовують у мові C# під час написання програм. Кожний файл – це текст. Для запису програм використовують знаки у відповідному кодуванні.

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

Ідентифікатори. Послідовність символів із латинських букв, символів підкреслення й арабських цифр, що починається з букви та слугує для називання різних елементів програми.

Програма – це запис алгоритму однією з мов програмування. Програма містить розділ команд і розділ опису даних.

Дані – це формалізоване подання всіх тих об'єктів (предметів, фактів, ідей), із якими може оперувати ПК. Вони містять змінні та константи. Перш ніж задавати у програмі дії над даними, змінні та константи мають бути визначеними.

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

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

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

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

В описі даних має міститися (для компілятора) така інформація, задана типом даних:

ім'я змінної або константи;

обсяг пам'яті, необхідної для зберігання значень;

які дії можна виконувати зі змінною або константою;

вид і спосіб виділення пам'яті;

початкове значення змінної або значення константи.

C# є жорстко типізованою мовою. Під час його використання програміст має оприлюднювати тип кожного об'єкта, що він створює (наприклад, цілі числа, числа з рухомою точкою, рядки, вікна, кнопки тощо).

Класифікація типів даних. C# розподіляє типи на два види: убудовані типи (або прості типи), визначені в мові, і типи, визначені користувачем (типи, які вибирає програміст).

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

Основна відмінність між ними – це спосіб, яким їхні значення зберігають у пам'яті.

Змінна типу-значення містить значення, збережене безпосередньо в ній (тобто у відповідних комірках пам'яті комп'ютера). Прикладом може бути тип int. Механізм оприлюднення змінної myNumber типу int і надання їй літерала 345 можна зобразити, як показано на рис. 2.1.

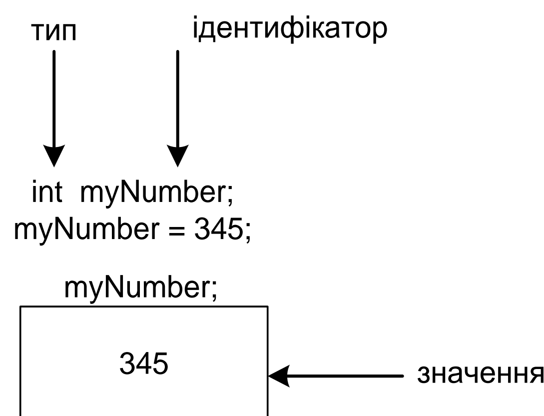


Рис. 2.1. **Тип int є типом-значенням**

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

Посилання становить позицію (адресу) об'єкта в пам'яті. Для ілюстрації звернімося до вже вивченого типу string (який, як з'ясуємо далі, є посилальним). Змінна типу string не містить рядок, а оприлюднюється для зберігання посилання на рядок. Сам рядок розміщено за визначеною адресою в пам'яті.

Розгляньмо такий фрагмент коду:

```
string   myText;  //  оприлюднення змінної myText типу string.
```

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

Після цього myText можна застосувати в такому операторі надання:

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

У результаті чого змінній myText надано адресу рядка "Lets go to the C to catch a #". Однак у комірках пам'яті, де міститься змінна myText, немає ніякого тексту, там тільки адреса пам'яті, названа також посиланням або покажчиком.

Зазначене ілюструє рис. 2.2, на якому текст розміщено за довільною адресою 4027, що міститься у змінній myText.

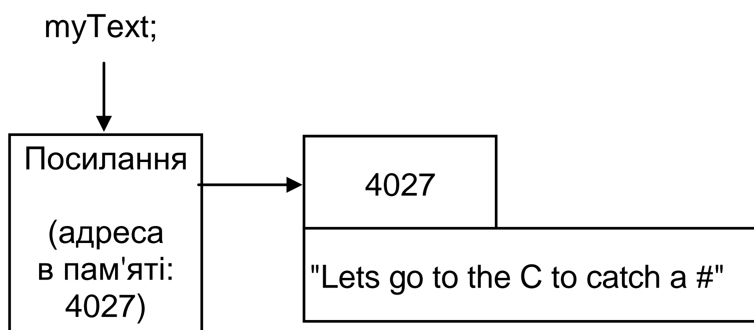


Рис. 2.2. **Тип string є посилальним**

Важливо зазначити, що фактичну адресу під час використання посилань у вихідному коді програми ніколи не застосовують.

Усі класи є посилальними типами.

Адресу місця в комп'ютерній пам'яті, де зберігають об'єкт, називають *посиланням*, або *покажчиком* на цей об'єкт.

Здебільшого відмінності в роботі з типами-значеннями та посилальними типами є незначними. Приклад цього – можливість застосування рядків у попередніх програмах без використання поняття "посилання".

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

2.2. Базова структура C#-програми

Лістинг, наведений далі, незважаючи на свою простоту, містить основні компоненти типової програми на C# [14].

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 наведено на лістингу 2.2 (рядки двох лістингів точно відповідають один одному).

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: Оприлюднення 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#. Розгляньмо кожну частину програми докладно. Номера рядків відповідають номерам рядків із лістингу 2.1.

Коментарі:

Рядок 1 містить коментар, уміст якого ігнорує компілятор. Його використовують для опису дій, які виконує програма. У цьому разі він просто повідомляє про те, що програму написано на C#:

01: // Проста програма на C#

Подвійний символ скісної 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 є некоректним:

// Проста програма на C# class Hello

тому що весь рядок, включно з class Hello, компілятор розглядає як коментар.

Визначення класу. Для пояснення рядка 2 потрібно звернутися до концепції ключового, або зарезервованого, слова. Ключове слово має спеціальне значення в мові C#, яке розпізнає компілятор.

У рядку 2 для визначення класу використовують ключове слово class:

02: class Hello

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

У лістингу 2.1 подано кілька ключових слів: class, public, static, void, string та if. Ключові слова мають для компілятора спеціальне значення. Їх не можна використовувати для інших цілей у C# (на що вказує термін "зарезервовані"). Слід зазначити, що ключове слово може бути частиною імені, тому назва classVariable є цілком коректною.

Ідентифікатори (імена). Імена у вихідному коді часто називають **ідентифікаторами**. Багато елементів – класи, об'єкти, методи, змінні екземпляра – мусять завжди мати ідентифікатори. На відміну від ключових слів C#, вибір усіх ідентифікаторів залишається за програмістом. Тут є декілька правил.

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

Приклади *припустимих ідентифікаторів*:

Elevator
_elevator
My2Elevators
My_Elevator
MyElevalor

Приклади *неприпустимих ідентифікаторів*:

Ele vator
6Elevators

У C# враховують регістр, тому великі та малі літери вважають 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 застосовують у парах. Коли в коді трапляється {, це значить, що десь далі обов'язково буде }, що їй відповідає. Дужка }, що відповідає рядку 3:

03: {

міститься в рядку 16. Ще одна пара фігурних дужок буде в рядках 6 та 15.

Оскільки { у 3 розташовано 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. 2.3) тепер можна розмістити методи та змінні екземпляра за умови, що всі оприлюднення містяться всередині блоку.

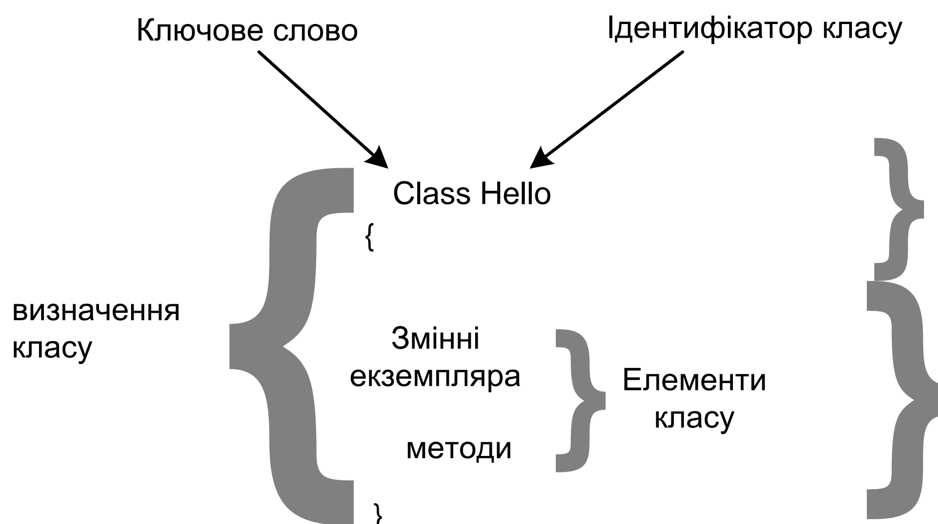


Рис. 2.3. **Визначення класу**

У рядку 4:

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

міститься вже знайомий символ коментаря //.

Метод Main() та його визначення. У рядку 5 починається визначення методу під ім'ям Main. У 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. Із цього методу починає виконання кожний затосунок на C# – його викликає середовище виконання під час запуску програми.

Наприклад, складний застосунок для робіт з електронними таблицями, написаний на C#, може містити тисячі методів із різними ідентифікаторами, але тільки метод Main викликає середовище виконання .NET під час запуску програми.

Точне значення всіх елементів рядка 5 поки що не будемо обговорювати, оскільки це потребує більш детального розуміння певних об'єктно орієнтованих принципів C#. Отже, клас складається з інтерфейсу, реалізованого за допомогою відкритих методів, і схованої частини, що складається із закритих методів і змінних екземпляра.

Ключове слово public у рядку 5 є специфікатором доступу, воно дозволяє управляти видимістю елементу класу. У цьому разі (перед методом Main) воно вказує, що Main є відкритим методом і, отже, частиною інтерфейсу класу Hello. У результаті метод Main можна викликати ззовні об'єкта Hello. Основні елементи визначення методу показано на рис. 2.4.

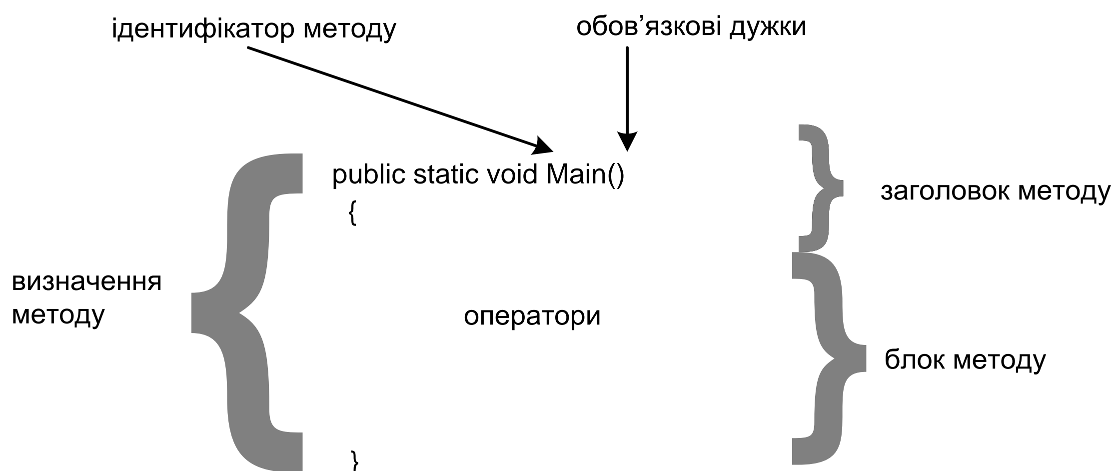


Рис. 2.4. **Визначення методу**

Кожна програма на C# має містити метод Main(). Під час її запуску середовище виконання .NET, насамперед, шукає цей метод. Якщо його визначено, із нього починається виконання, якщо ні – виводиться повідомлення про помилку.

Main() на лістингу 2.1 розташовано всередині класу Hello, а середовище виконання .NET – зовні. Під час спроби запуску Main() середовище

буде розглядати його як ще один об'єкт, що запитує доступ до методу класу. Тому його потрібно відкрити, зробивши частиною інтерфейсу класу. Щоб середовище .NET могло дістати доступ до Main(), його слід завжди оприлюднювати 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 звернімося знову до обговорення розбіжностей 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 виконувати будь-яких дій. Ключове слово static дозволяє відійти від цієї схеми та скористатися методами класу, не створюючи конкретного екземпляра об'єкта.

Коли static додано в заголовок методу, це повідомляє клас про те, що для використання методу не потрібно створювати екземплярів за межами класу. Отже, метод Main() можна використовувати до створення певного об'єкта класу Hello. У цьому разі це обов'язково, тому що Main() викликає середовище виконання .NET до того, як створюють які-небудь об'єкти.

Щоб зрозуміти значення ключового слова void у рядку 5, потрібно звернутися безпосередньо до того, як працюють методи. У цьому розділі буде наведено 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 становить збережений блок даних. Ключове слово string указує, що answer належить типу string:

```
07:     string  answer;
```

Ідентифікатор (answer) програміст вибирає на свій розсуд, а string є зарезервованим словом.

Розміщення answer після string у рядку 7 означає, що оприлюднена змінна answer типу string.

Кожна змінна, що використовують у програмі на C#, має бути оприлюднена.

Змінну answer застосовують у рядках 11 та 12.

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

У C# рядки тексту позначають " " (подвійні лапки). Складові частини визначення змінної показано на рис. 2.5.

З рис. 2.5 видно, що змінна складається із трьох елементів:

ідентифікатора (у цьому разі – answer);

типу, тобто виду інформації, що вона може зберігати (у цьому разі – string, тобто послідовність символів);

значення, тобто збереженої інформації. Поточне значення на рис. 2.5 дорівнює "Julian is a boy".

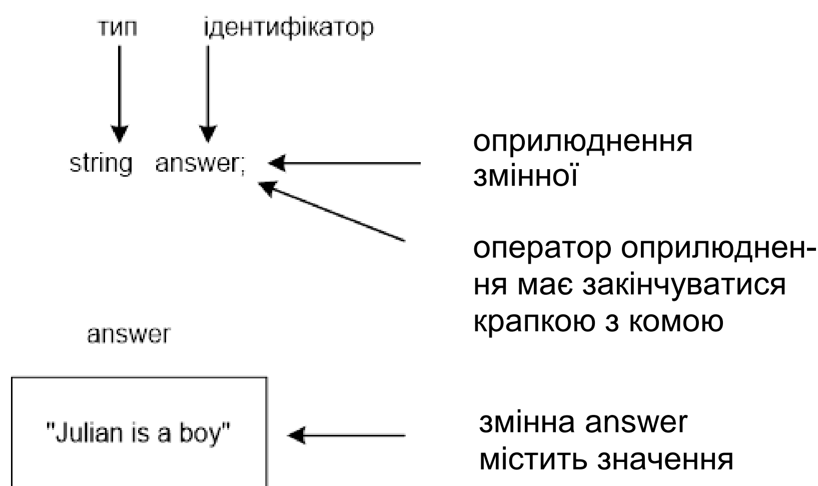


Рис. 2.5. **Тип, ідентифікатор та значення змінної**

Будь-яке завдання, що виконують програмою на C#, можна розподілити на послідовність інструкцій. Найпростішу інструкцію називають оператором. Усі оператори закінчують символом крапки з комою.

Рядок 7 містить оператор оприлюднення змінної, тому його, як і інші, закінчують крапкою з комою.

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

Запуск методів .NET-платформи. Оператор у рядку 9

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

змушує програму вивести на екран таке:

Do you want me to write the two words?

Нині 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 коротко відбувається в рядку 9.

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

Коли метод виконує певне завдання у програмі, це називають *викликом*. Елемент усередині круглих дужок (текст "Do you want me to write the famous words?" у прикладі) називають *аргументом*. Аргумент містить інформацію, необхідну викликуваному методу для виконання завдання. Аргумент передають методу WriteLine під час виклику. Після цього метод звертається до даних уже за допомогою своїх внутрішніх операторів. Рядок 9, як і 7, містить оператор і тому його закінчують крапкою з комою.

Розгляньмо, як саме в рядку 9 використовують метод класу System.Console. Як уже підкреслювали, класи є "схемами", а об'єкти – "виконавцями". Метод класу можна використовувати в тому разі, коли в його оприлюдненні наявне ключове слово static, яке згадували раніше. Отже, метод WriteLine є доступним без створення конкретного екземпляра об'єкта System.Console.

Загальний механізм виклику методу. Інструкції методу містяться всередині його визначення у формі операторів. Викликати метод – це означає виконати його інструкції. Виконання відбуваються послідовно в тому порядку, у якому їх написано у вихідному коді.

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

Виклик методу має такий синтаксис: ім'я об'єкта (або класу, якщо метод оприлюднено як static), точка, ім'я методу та завершальна пара

круглих дужок ( ), у яких можуть міститися аргументи. Останні становлять дані, передані методу.

Виклик нестатичного методу:
Ім'яОб'єкта.Ім'яМетоду(Необов'язкові_аргументи)

Виклик статичного методу:
Ім'яКласу.Ім'яМетоду (Необов'язкові_аргументи)

Замінивши загальні елементи реальними іменами, легко визначити оператор із рядка 14 лістингу 2.1:
Syslem.Console.WriteLine("Bye Bye!");

Після завершення методу потік управління повертається в точку, із якої відбувся виклик (рис. 2.6).

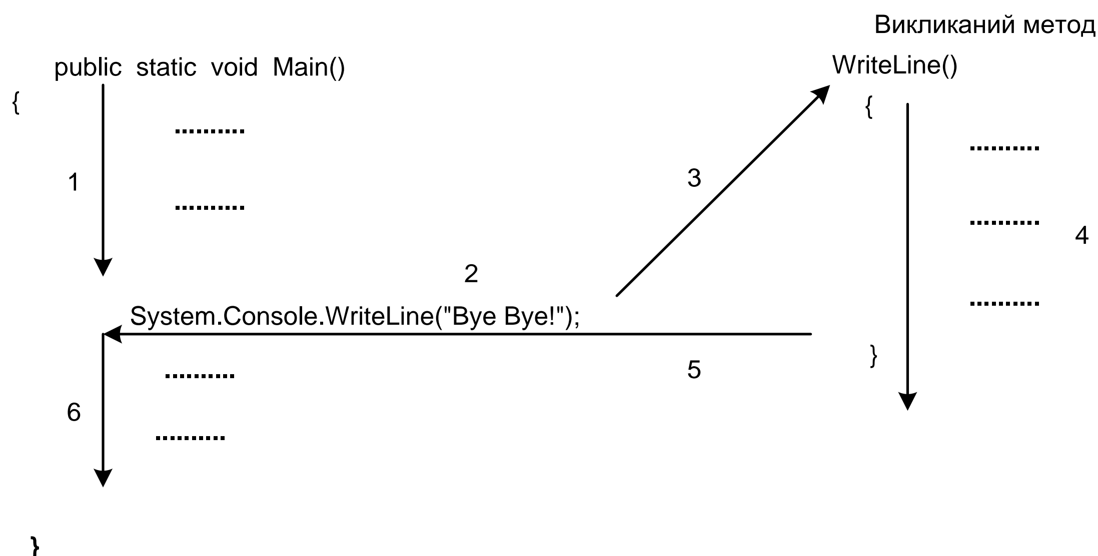


Рис. 2.6. **Рух потоку виконання програми під час виклику методу**

Можна виділити такі 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 всередині System.Console WriteLine (...).
5. Повернути управління 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 виводять рядок:

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

і курсор переміщають на один рядок нижче.

Повідомлення. Розгляньмо рядок 9 лістингу 2.1. У ньому міститься оператор, що викликає метод WriteLine. В об'єктно орієнтованому програмуванні для позначення такого виклику часто застосовують ще один термін – "повідомлення".

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

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

Загальна схема ООП передбачає, що об'єкти виконують дії, які запускають під час отримання повідомлень. У розглянутому прикладі дією є виведення на консоль:

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

Надання значення змінної. У рядку 11 знову повторно використовують клас System.Console. Цього разу застосовують інший із його статичних методів – ReadLine, що призупиняє виконання програми, очікуючи введення від користувача. Відповіддю може бути введений текст, який завершують натисканням клавіші Enter. Як випливає з назви, метод ReadLine читає введення:

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

Під час натискання Enter текст, уведений користувачем, зберігають у змінній answer. Очевидно, коли користувач уводить 'y', answer містить "y", коли 'n' , – "n" тощо. За це відповідає знак рівності (=), розташований після answer.

У C# знак рівності (=) використовують трохи інакше, ніж у стандартній арифметиці.

Механізм задавання нового значення змінній answer називають *наданням*. Говорять, що введений текст надано змінній answer. Загальний вираз у рядку називають *оператором надання*, а сам знак рівності (=) називають *операцією надання* (у цьому контексті). Якщо знак "рівність" використовують в інших контекстах, він має інші назви.

Розгалуження за допомогою оператора if. Одне із застосувань знака "рівність" показано в рядку 11. У рядку 12 його використовують в зовсім іншому контексті: цього разу – у стандартному арифметичному:

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

У C# два послідовні знаки рівності (==) позначають операцію рівності, використовувану для порівняння виразів ліворуч і праворуч від нього.

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

Вираз, що може набирати тільки одне із двох значень (true або false), називають *логічним*.

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

Якщо answer == "y" дорівнює false (неправда), потік виконання має перейти до рядка 14. На рис. 2.7 за допомогою стрілок показано хід потоку виконання в рядках 12 – 14.

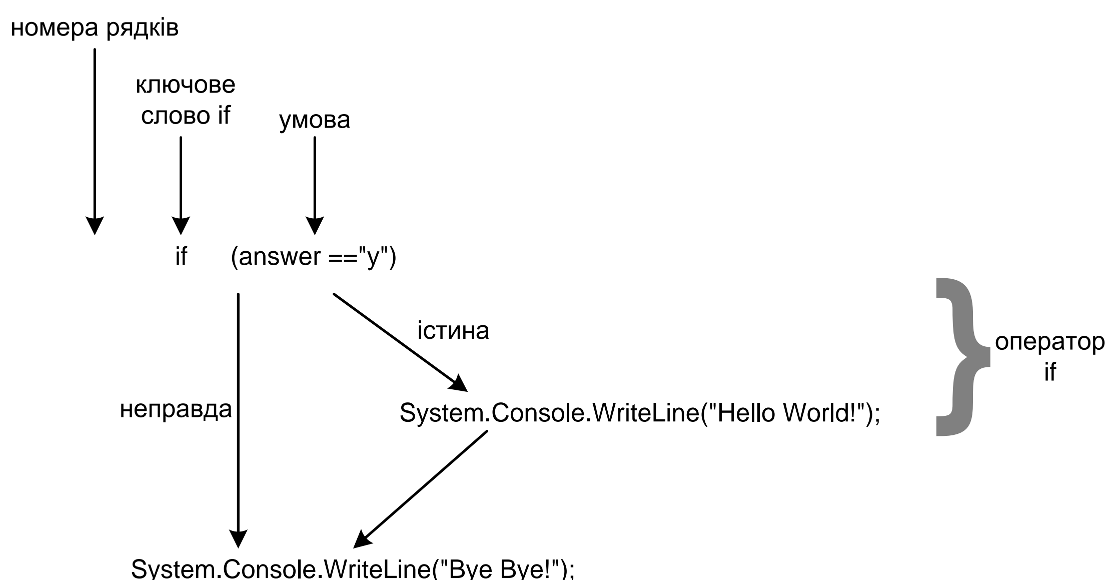


Рис. 2.7. **Послідовність виконання програми з оператором if**

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

Тільки answer, що дорівнює "y", змушує програму вивести Hello World!. Будь-яка інша відповідь користувача робить логічний вираз таким, що дорівнює false, у результаті чого рядок 13 не виконують.

У рядках 12 та 13 міститься оператор if. Він управляє потоком виконання, дозволяючи вибрати два різні напрямки. Такі оператори, як if, називають *операторами розгалуження*.

Завершення методу Main() і класу Hello:

Дужка } в рядку 15 завершує блок методу Main(), початий у 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 називають *лексемами*. Їх слід розділяти 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 використовують у логіці та лінгвістиці. Спроба розриву лексем призводить до некоректного коду.

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

Ієрархічну структуру основних типів C# наведено в табл. 2.1 [2; 5; 8].

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 із 13 простих типів C#. Прості типи використовують для зберігання числових значень і окремих символів.

Типи-перерахування. Ці типи забезпечують засоби для створення символічних констант, зокрема, їхніх наборів: дні тижня (понеділок, вівторок тощо), місяці (січень, лютий, березень тощо), кольори (зелений, червоний, синій тощо) 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, тоді як структури – це типи-значення. Фактично, усі 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 їхньої внутрішньої реалізації, наявної в неабстрактних методах класу. Класи можуть реалізовувати інтерфейси, конкретизуючи абстрактне поводження, установлене інтерфейсом. Інтерфейси дозволяють програмістові реалізувати найбільш складні концепції ООП.

Типи-делегати. Подібно інтерфейсам, делегати використовують 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) передають у програмі як звичайні посилання та виконують 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 підрозділі, і містить правила їхнього використання в застосунках, що виконують у середовищі .NET. Як випливає з назви, усі реалізовані на платформі .NET мови програмування, включно із C#, основані на типах, визначених CTS.

2.3. Характеристика й особливості застосування простих типів

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

Під час вибору змінної певного типу програміст фактично задає вид величини, що змінна може зберігати, і набір операцій, у яких вона може брати участь. Кожний простий тип характеризується такими властивостями:

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

Діапазон значень змінної. Наприклад, діапазон типу int:

від -2 147 483 648 до 2 147 483 647.

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

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

У C# визначено 13 простих типів [5 – 8; 14], перелічених у табл. 2.2.

Таблиця 2.2

Прості типи в мові C#

Ключові слова мови C#	Tun .NET CTS	Види значення	Використовувана пам'ять	Діапазон і точність
1	2	3	4	5
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 -32 768 до 32 767
ushort	System.UInt16	Ціле число	16 бітів	Від 0 до 65 535

Закінчення табл. 2.2

1	2	3	4	5
int	System.Int32	Ціле число	32 біти	Від -2 147 483 648 до 2 147 483 647
uint	System.UInt32	Ціле число	32 біти	Від 0 до 4 294 967 295
long	System.Int64	Ціле число	64 біти	Від -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
ulong	System.UInt64	Ціле число	64 біти	Від 0 до 18 446 744 073 709 551 615
char	System.Char	Ціле число (один символ)	16 бітів	Усі символи Unicode
float	System.Single	Число з рухомою точкою	32 біти	Від (+/-)1.5 * 10-45 до (+/-)3.4 * 1 038 Приблизно 7 значущих цифр
double	System.Double	Число з рухомою точкою	64 біти	Від (+/-) 5.0 * 10-324 до (+/-)3.4 * 1 030 15-16 значущих цифр
decimal	System.Decimal	Десяткове число (високої точності)	128 бітів	Від (+/-) 1.0 * 10-28 до (+/-)7.9 * 1 028
bool	System.Boolean	true або false	1 біт	Немає

Хоча тип bool (останній рядок табл. 2.2) розглядають тут як простий тип, він пов'язаний із управлінням потоком виконання програм. Ключове слово належить до символу, що використовують у вихідному коді C# під час оприлюднення змінної.

Простір імен System .NET Framework містить усі прості типи. Кожне ключове слово, показане в першому стовпці, – це псевдонім типу, визначеного у CTS. Наприклад, ключове слово int позначає System.Int32 у CTS. Отже, у вихідному коді можна використовувати як короткий псевдонім типу, так і його довге повне ім'я.

Два такі вирази є ідентичними:

```
int   myVariable;
System.Int32   myVariable;
```

У 3-му стовпці визначено чотири різні групи простих типів, що містяться в C# – ціле число, число з рухомою 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 розроблено для окремих символів, його розглядають 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 невід'ємні числа.

У табл. 2.2 містяться і три типи з рухомою точкою – float, double і decimal, використовувані для зберігання чисел, що містять дробову частину (наприклад, 6.87, 9.0 і 100.01). Основні відмінності – діапазон, використовувана пам'ять і точність.

Синтаксис оприлюднення змінних. Дотепер для зображення синтаксису C# використовували його опис (на основі розмовної мови) та приклади. Однак синтаксис потребує дуже точного запису (аж до крапки з комою), тому його опис розширено до точної форми.

Вибрана тут форма запису – це спрощена версія часто використовуваної для опису синтаксису комп'ютерної мови нотації, названої формою *Бекуса–Наура* (Backus–Naur), або BNF. Її було розроблено Дж. Бекусом і П. Науром для опису мови Алгол 60.

Форма запису синтаксису складається з таких елементів:

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

Метазмінні (розміщені в кутових дужках) у 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 риска **|**, що вказує на можливі альтернативи.

Як приклад розгляньмо оператор оприлюднення змінної:

int myNumber;

Оператор_оприлюднення_змінної :: =

<Тип> <Ідентифікатор_змінної>;

У цьому разі **:: =** вказує, що далі йде визначення змінної, оприлюдненої в операторі.

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

Другу синтаксичну змінну <Ідентифікатор_змінної> може бути замінено ім'ям змінної, наприклад myNumber.

Оператор оприлюднення слід закінчувати крапкою з комою.

На рис. 2.8 розглядають приклад, коли для точного визначення конструкції C# потрібно три крапки "**...**" і символ необов'язкового елемента [ ].

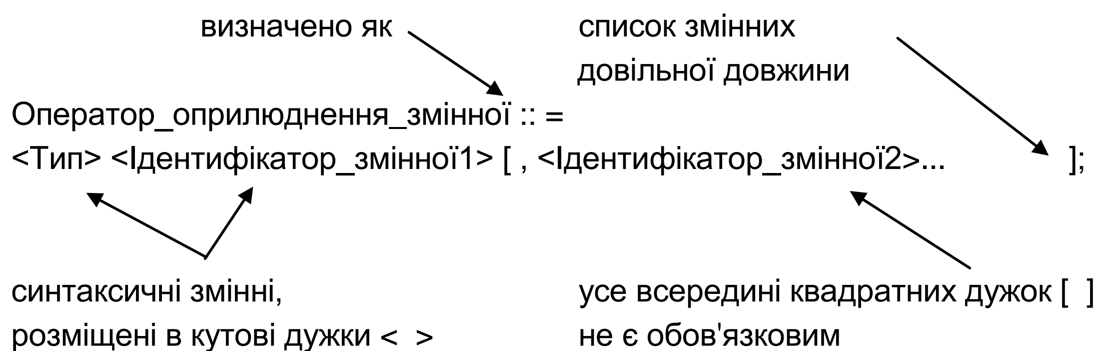


Рис. 2.8. **Приклад, коли для точного визначення конструкції C# потрібно три крапки "..." і символ необов'язкового елемента [ ]**

Нарешті, потрібна можливість виразити випадок, коли кілька альтернативних елементів можна використовувати в одній позиції.

Наприклад, під час оприлюднення методів і змінних екземпляра класу застосовують ключові слова public і private:

private int currentFloor;

Хоча це можна розцінювати як неправильну методику програмування, але змінну екземпляра можна оприлюднити і public:

public int currentFloor;

Можна взагалі опустити специфікатор доступу:

int currentFloor;

Водночас змінна currentFloor буде мати специфікатор доступу за замовчуванням – private.

Можливість використання або public, або private позначають вертикальною рискою (|): public | private. Оскільки тут специфікація є необов'язковою, вона має такий вигляд: [public | private]. Повне оприлюднення змінної тепер визначено в такий спосіб (рис. 2.9):

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

Рис. 2.9. **Повне оприлюднення змінної**

Цілочисельні типи. Як наведено в табл. 2.2, у C# визначено дев'ять цілочисельних типів. Цілі числа – це числа без дробової частини, наприклад 34, 0 або –7 653. Оскільки множина цілих чисел не є обмеженою, а комп'ютерна пам'ять має кінцевий обсяг, можна подати лише деяку її підмножину.

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

Деякі типи (як-от int), названі знаковими типами, можуть зберігати й від'ємні і додатні значення. Інші типи – беззнакові – можуть мати тільки додатні значення (включно з нулем). Є чотири знакових (sbyte, short, int i 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 мати символи unicode. Але тому що це специфічний тип, безпосередньо пов'язаний із типом string, спочатку тут подано лише вісім раніше згаданих цілочисельних типів.

Біти. Один біт має тільки два значення – 0 і 1. Відповідно, два біти можуть мати 2 × 2 = 4 різні значення, 3 біти – 2 × 2 × 2 = 8, а n бітів – 2 × 2 × … × 2 значень:

8 бітів – 256 значень;

16 бітів – 65 536 значень;

32 біти – 4 294 967 296 значень;

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

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

Таблиця 2.3

Знакові та беззнакові цілочисельні типи

Знакові	**Беззнакові**	**Використовувана пам'ять**
sbyte	byte	8 бітів
short	ushort	16 бітів
int	uint	32 біти
long	ulong	64 біти

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, 1 009 і –487 – приклади літералів.

Усі цілочисельні літерали мають певний тип точно так само, як і всі оприлюднені змінні. Компілятор C# під час визначення типу літерала дотримується точних правил: він розглядає розмір літерала і наступний за ним необов'язковий суфікс.

Якщо суфікса немає, компілятор вибирає перший із таких типів: int, uint, long і ulong. Якщо об'єднати цей факт із діапазонами, зазначеними в табл. 2.2, можна обчислити тип кожного літерала. Наприклад, такі літерали:

43 200 типу int;

2 507 493 742 типу uint;

-25 372 936 858 775 201 типу long;

270 072 036 654 375 827 типу long;

17 016 748 093 204 541 685 типу ulong.

Суфіксом літерала може бути U, L або UL.

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

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

```
int  aNumber;
aNumber = 99;                    // Присвоювання 99 за основою 10
aNumber = 0x99;          // Присвоювання 99 за основою 16
```

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

Оператори присвоювання. Використовуючи форму запису синтаксису, наведену раніше, можна записати оператор присвоювання для простого типу в загальній формі:

<Ідентифікатор_змінної> :: = <Вираз> ;
де
<Вираз>
:: = <Літерал>
:: = <Ідентифікатор_змінної>
:: = <Числовий_вираз>

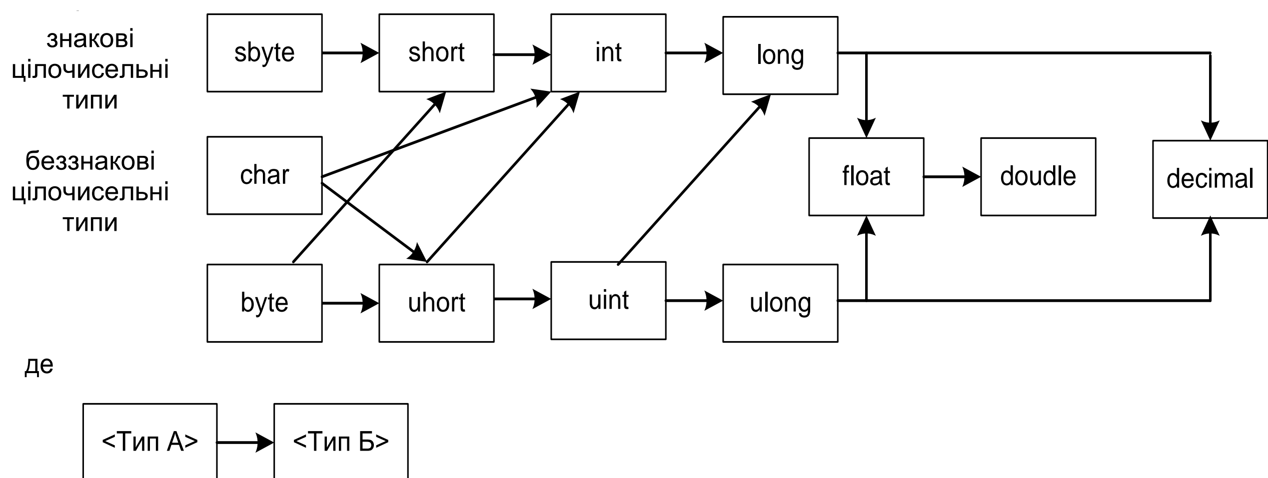
Слід зазначити, що тут <Числовий_вираз> – це припустима комбінація числових значень і операцій: наприклад, (count * 4) + (distance – 100).

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

Перетворення вбудованих типів. Об'єкти одного типу можуть бути перетвореними на об'єкти іншого типу неявно або явно.

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

Явні перетворення здійснюють, коли програміст "приводить" значення до іншого типу.



означає, що неявне перетворення Типу А на Тип Б можливо

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

Неявні перетворення гарантують також, що дані не буде загублено. Наприклад, ви можете неявно приводити від short (2 байти) до int (4 байти). Незалежно від того, яке значення буде в short, його не буде втрачено під час перетворення до int:

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

Якщо ви виконуєте зворотне перетворення, то, зазвичай, можете втратити інформацію. Якщо значення в int є більшим, ніж 32 767, воно буде усіченим під час перетворення. Компілятор не виконуватиме неявне перетворення від int до short:

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

Програміст має виконати явне перетворення, використовуючи оператор приведення:

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

Типи з рухомою точкою. Змінні з рухомою точкою дозволяють зберігати числа із дробовою частиною, як, наприклад, число 2.99 або 3.141 592 653 589 7931.

Числа з рухомою точкою дозволяють подати значно ширший діапазон величин, ніж найбільш об'ємний цілий тип long. Під час написання чисел такої величини зручно використовувати нотацію, названу науковою нотацією, або е-нотацією (експонентною нотацією) і нотацією з рухомою точкою.

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

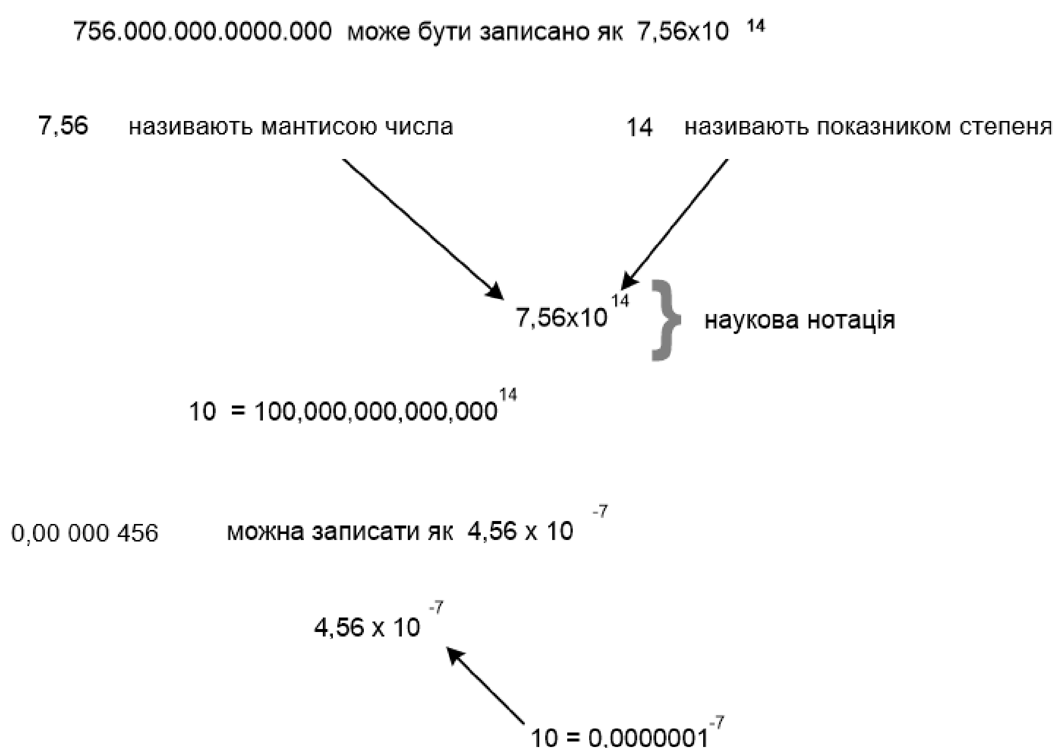


Рис. 2.11. **Наукова нотація чисел із рухомою точкою**

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

Можна використовувати форму, яка є знайомою нам із повсякденного життя – зі знаком десяткового дробу, супроводжуваним цифрами (наприклад, 134.87 і 0.000 034 5), або використовувати показаний тут експонентний формат. Однак знак множення й 10 вилучено із запису і замінено буквою E (або e) із показником, написаним після нього.

Наприклад, число 0.000 000 456 (дорівнює 4.56* 10-7) записують у C# як +4.56E-7.

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

Далі наведено декілька чисел з рухомою точкою, заданих у C# із застосуванням звичайної форми запису:

```
56.78        // звичний запис із рухомою точкою;
0.645        // число з рухомою точкою;
7.0          // також число з рухомою точкою;
456E-7       // те саме, що і 4.56e-7;
8e3          // те саме, що і +8.0e+3;
-8.45e8      // від'ємна величина.
```

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

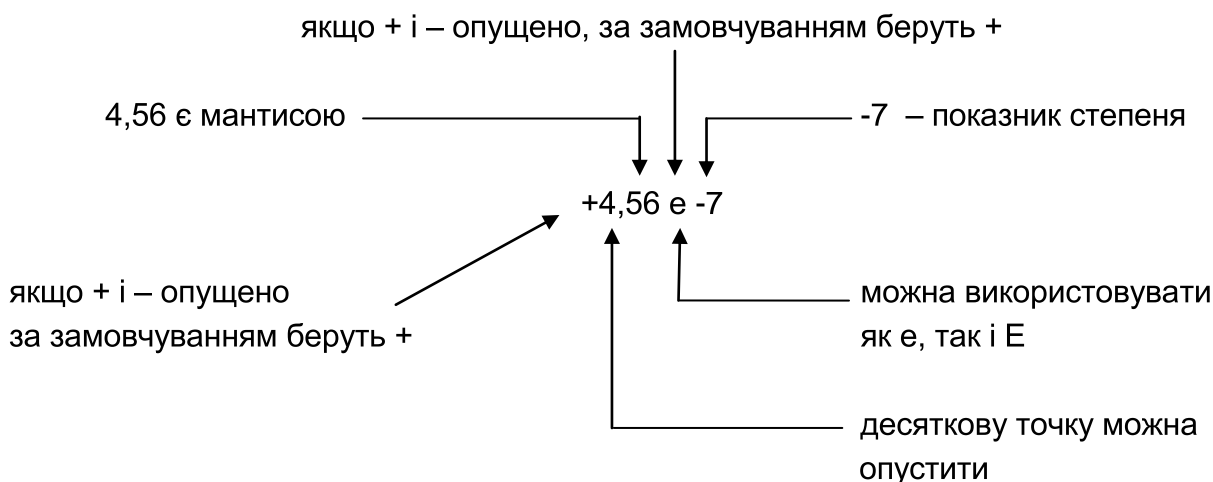


Рис. 2.12. **Деталізація наукової нотації чисел із рухомою точкою**

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

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

У C# є два типи з рухомою точкою – float та double. Їх відрізняють два основні атрибути – кількість значущих цифр, які вони можуть подавати (це пов'язано з поняттям мантиси, що вже обговорювали), і діапазон зміни величини показника степеня (пов'язане з поняттям коефіцієнта масштабування).

Змінні типу float можуть містити величини від -3.4E38 до 3.4E38. Вони можуть бути близькими до нуля (не дорівнюючи йому), як 1.5e-45 або -1,5e-45. Величини подають за допомогою приблизно семи значущих цифр. Тип float займає 32 біти (4 байти) пам'яті.

Змінні типу double можуть містити величини від -1.7E308 до 1.7E308. Вони можуть бути близькими до нуля (не дорівнюючи йому), як 5.0E-324 або -5.0E-324. Величини подають за допомогою 15 – 16 значущих цифр. Тип double займає 64 біти (8 байтів) пам'яті.

Особливості операцій над числами з рухомою точкою. Операції за участю величин із рухомою точкою ніколи не видають переривання в аварійних ситуаціях. Замість цього, залежно від виконуваної операції, виходить один із таких результатів:

додатний (+0) або від'ємний (-0) нуль,

додатна (+∞) або від'ємна (-∞) нескінченність,

NaN (Not a Number – не число).

Якщо роблять спробу надати величину x змінній типу float, де x належить інтервалу -1.5E-45 > x > 1.5E-45 і x != 0 (інакше кажучи, x є дуже близьким до нуля, але ним не є), результатом операції буде додатний (якщо x додатний) або від'ємний (якщо x від'ємний) нуль.

Якщо роблять спробу надати величину x змінній типу double, де x належить інтервалу -50E-324 > x > 5.0E-324 і x != 0 (інакше кажучи, x є дуже близьким до нуля, але ним не є), результатом операції буде додатний (якщо x додатний) або від'ємний (якщо x від'ємний) нуль.

Якщо роблять спробу надати величину x змінній типу float, де x належить діапазону -3.4E38 > x або x > 3.4E38 (інакше кажучи, x занадто велика за модулем величина), результат операції буде таким: від'ємною (позначуваною у виведенні C# як -Infinity) або додатною (позначуваною у виведенні Infinity) нескінченністю.

Якщо роблять спробу надати 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 операції буде від'ємною (позначуваною у виведенні C# як -Infinity) або додатною (позначуваною у виведенні Infinity) нескінченністю.

Якщо роблять спробу виконати некоректну операцію над типами з рухомою точкою, результатом буде NaN. Приклад неправильної операції: 0.0/0.0.

Літерали з рухомою точкою. Числа з рухомою точкою на зразок 5.87 або 8.24E8 у програмі на C# за замовчуванням належать до типу double. Щоб визначити величину як float, потрібно додати суфікс f (або F). Отже, 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 виразів, розглянутих у математиці як такі, що дорівнюють, не завжди дорівнюють під час обчислень з рухомою точкою. Це викликано тим фактом, що величина з рухомою точкою, обчислена одним способом, часто відрізняється від цієї самої величини, обчисленої іншим способом.

Розгляньмо приклад програми. Що можна чекати під час виведення?

```
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 1.000 000 014 901 161 2

У рядку 9 лістингу число 0.2f додано п'ять разів, і результат надано змінній типу double mySum. У математиці 5 × 0.2 точно дорівнює 1. Проте під час порівняння mySum із 1.0 операцією рівності == в рядку 10 ця умова дорівнює false, і тому рядок 11 не виконується.

Змінна mySum дорівнює 1.000 000 014 901 161 2, як показано у прикладі виведення.

Бажано уникати використання операцій порівняння для величин із рухомою точкою. Якщо необхідно виконати таке порівняння, потрібно враховувати діапазон можливої похибки.

Дотримуючись попередньої поради, можна вказати припустиму точність, скажімо +/–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 є подібним до типів із рухомою точкою, оскільки він дозволяє подати числа із дробовою частиною і використовує концепцію величини, значущих цифр та коефіцієнта масштабування. Однак цей тип є значно більш точним, має менший діапазон і займає більше пам'яті. Отже, тип decimal є корисним для обчислень, де потрібна висока точність.

Хоча значення типу decimal використовують для подання чисел із дробовою частиною на зразок типів із рухомою точкою, вони не належать до останніх.

Тип decimal не підтримує додатні/від'ємні нулі, додатні/від'ємні нескінченності або NaN як типи з рухомою точкою. Замість цього величини в інтервалі -1.0E-28 і 1.0E-28 уважають нулем, а величини поза припустимим діапазоном генерують переривання переповнення згори/знизу.

На відміну від цілочисельних типів, числа з рухомою точкою і типу decimal дозволяють подавати числа із 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 належить за замовчуванням до типу

double. Отже, щоб літерал мав тип decimal, потрібно додати до числа суфікс m (або M).

Величини цілочисельних типів (за потреби) неявно перетворюють на decimal. Отже, літералам цих типів, наприклад 10, 756 і –963, суфікс m не потрібен.

2.4. Константні величини

Часто у вихідній програмі використовують фіксовані числа, наприклад: 3.141 592, або інші величини, які не змінюються протягом виконання програми. Мова C# дозволяє оприлюднювати імена для літералів або інших виразів і використовувати їх замість запису фактичного значення.

Наприклад, замість

distance = secondsTraveled * 186 000;

можна надати значенню 186 000 ім'я SpeedOfLight і привести вираз до такого вигляду:

distance = secondsTraveled * SpeedOfLight;

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

int SpeedOfLight = 186 000;

Це дозволяє застосовувати SpeedOfLight під час обчислення відстані. Але не слід забувати, що значення SpeedOfLight може бути випадково зміненим у програмі. Тому, щоб залишити величину SpeedOfLight незмінною, під час її визначення потрібно вказати ключове слово const:

const int SpeedOfLight = 186 000;

Ім'я, що становить постійну величину, у C# називають **константою**.

Класифікація константних величин. Константи розподіляють на такі групи:

- Числові
  - Цілі
  - Речовинні
- Перерахувальні
- Символьні (літерні)
    - Клавіатурні

Кодові (керівні або розділові символи)
Кодові числові
Рядкові (рядки або літерні рядки)
Іменовані (символічні)

Синтаксичний блок оприлюднення константи.

Оператор_оприлюднення_константи ::=

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

Примітка: тут <Константний_вираз> може складатися лише з одного літерала, як у такому рядку коду:

```
public const double Pi = 3.141 59;
```

Цей вираз може також складатися з комбінації літералів, інших констант і операцій за умови, що його може бути обчислено під час компіляції:

```
public  const  double  Pi = 3.141 59;
public const double TwicePi = 2 * Pi;      // Коректно, тому що значення
2 і Pi  є відомими під час компіляції.
```

Оприлюднення константи може міститися у визначенні класу – водночас константа стає його елементом. Константи завжди мають специфікацію static і тому є доступними іншим об'єктам за межами класу тільки з використанням ім'я класу, за яким іде операція точки та ім'я константи, а не конкретного екземпляра об'єкта. Сказане демонструємо в наведеному далі 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 супроводжено операцією уточнення (.) та ім'ям константи MassOfElectron.

Спроба звернення 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 такі символьні константи:

Клавіатурні: '1', 't', 'y' – клавіатурний символ задано в апострофах;

Наприклад, char t = 'd'; // d

Кодові – для задавання деяких керівних (табл. 2.4) і розділових символів, наприклад, '\n', '\t';

Таблиця 2.4

Керівні послідовності

Керівні послідовність	Призначення	Подання Unicode
\'	Одинарні лапки	\u0027
\"	Подвійні лапки	\u0022
\\	Зворотна скісна 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

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

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

char t = '\x1A' ; // код символу "←"

Контрольні запитання

1. Перелічіть 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 оприлюднення змінних.
6. Як здійснюють перетворення вбудованих типів даних?
7. Коли доцільно використовувати типи з рухомою точкою?
8. Опишіть класифікацію константних величин.
9. Наведіть синтаксичний блок оприлюднення константи.

3. Програмування обчислювальних процесів

3.1. Програмування лінійних обчислювальних процесів

Будь-яку програму може бути складено за допомогою чотирьох основних структур:

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

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

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

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

Структуру проходження вбудовано в C#. Поки не зазначено інше, комп'ютер виконує інструкції C# одну за іншою в тій послідовності, у якій їх записано.

Огляд основних операцій C#. У C# є доступною велика кількість операцій. За винятком тих, які призначено для спеціальних цілей, що залишилися, їх можна розподілити на чотири основні категорії: арифметичні, відношення, логічні та побітові.

Арифметичні операції застосовують до значень простих числових типів. Вони становлять основу для обчислень у C#.

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

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

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

Побітові операції дозволяють працювати з окремими бітами операнда.

Арифметичні операції (додавання (+), віднімання (–), множення (*) і ділення (/)) у поєднанні із числами (названими операндами) формують арифметичні вирази.

Арифметичні вирази на C# досить легко інтерпретувати, тому що вони (більш-менш) відповідають усім відомим арифметичним правилам. Усім чотирьом базовим операціям, згаданим у попередньому підрозділі, потрібен один операнд ліворуч та один праворуч. Арифметичну операцію, що поєднує у виразі два операнди, наприклад **distance1 + + distance2,** називають *бінарною*. Операнд у такому контексті може набирати різних форм. Це може бути просто число (наприклад, 345.23), змінна, константа, що подає числове значення, або сам числовий вираз, наприклад: **distance1 * 100 + distance2 * 300.**

Операндом може бути й виклик методу. У цьому разі, природно, метод має повертати значення. Як ілюстрацію наведено виклик методу **Average** (рис. 3.1).

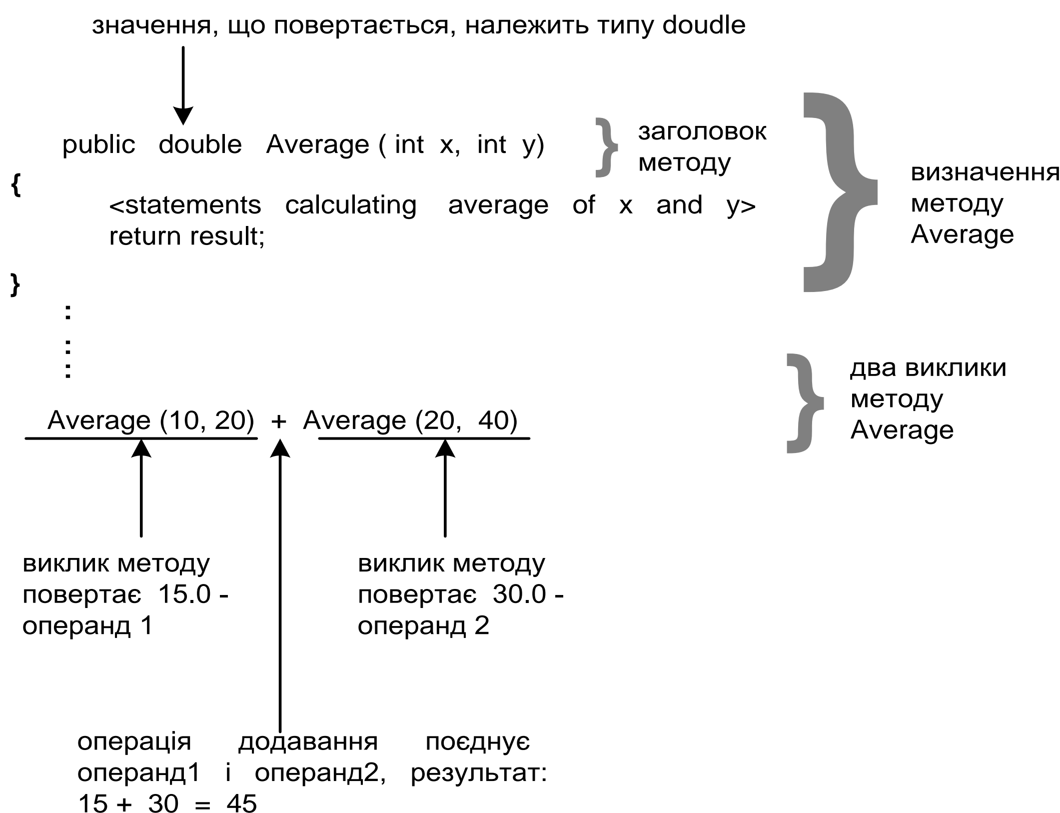


Рис. 3.1. **Виклик методу Average**

Обидва виклики відбуваються до того, як виконано операцію додавання. Після їхнього завершення відповідні конструкції можна розглядати

як числа (15.0 і 30.0) типу **double**, який указано в заголовку методу **Average**).

Синтаксичні блоки. Усі п'ять форм операнда можна лаконічно виразити, використовуючи такий синтаксичний блок.

Синтаксичний блок "Операнд". Операнд можна подати одним із п'яти різних способів:

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

Поняття числового виразу можна також формалізувати, як показано в синтаксичному блоці "Числовий вираз":

Числовий вираз
::= <Операнд> <Бінарна_операція> <Операнд>
::= <Числовий_вираз> <Бінарна_операція>
<Числовий_вираз>

Логічні операції. Мова C# містить три логічні операції, семантично еквівалентні словам **"І"**, **"АБО"** та **"НЕ"** звичайної мови (табл. 3.1).

Таблиця 3.1

Часто використовувані логічні операції

Логічні операції	Назви	Позначення	Кількість операндів	Приклади
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# позначають як **&&,** логічне АБО – **||**, логічне НЕ – **!** (знак оклику). Таблицю істинності (табл. 3.2) для типових логічних операцій наведено далі.

Таблиця 3.2

Таблиця істинності для типових логічних операцій

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

Коли програма опрацьовує логічний вираз з операцією **&&,** наприклад **(distance == 1 000) && (mass == 3 000)**, спочатку визначають значення **(distance == 1 000)**.

Якщо воно дорівнює **false**, компілятор уже знає, що й весь вираз буде дорівнювати **false**, незалежно від другої частини **((mass == 3 000)).** Тому **(mass == 3 000)** не опрацьовують. Такий механізм називають *коротким замиканням.*

Аналогічно, якщо програма містить вираз з операцією ||, наприклад **(distance == 1 000) || (mass == 3 000)**, і перша частина **((distance == 1 000))** дорівнює **true**, другу частину ігнорують, тому що, незалежно від її значення, весь вираз дорівнює **true**.

Мова C# містить ще три логічні операції, названі 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" під час опрацювання виразів, тому їх виконують трохи повільніше.

Здебільшого застосовують **&&** та **||**, але в рідких випадках необхідно, щоб програма завершила опрацювання всіх логічних підвиразів, навіть якщо це й потрібно для визначення значення загального виразу.

Пріоритети операцій. Будь-який вираз, де використовують більше ніж два операнди, завжди можна розподілити на підвирази, кожний із яких

складається тільки із двох операндів та однієї бінарної операції. Після обчислення результатів підвиразів їх використовують на наступному рівні.

Розгляньмо вираз:

4 * 5 + 40 * 10 – 20 * 40 / 10 + 70

Легко побачити, що він складається з декількох підвиразів. На рис. 3.2 показано, як, користуючись визначенням числового виразу, можна виконати обчислення.

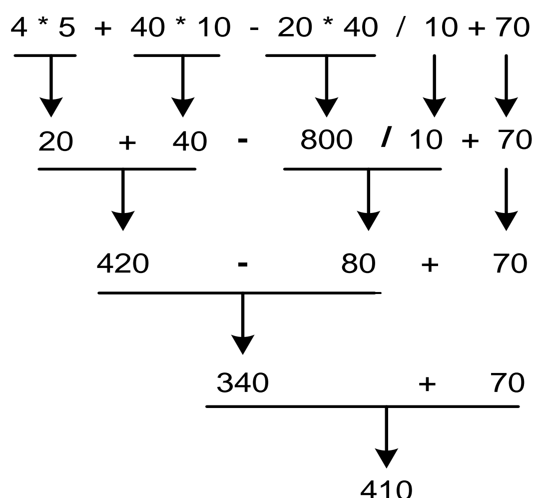


Рис. 3.2. **Черговість виконання операцій під час обчислення виразів**

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

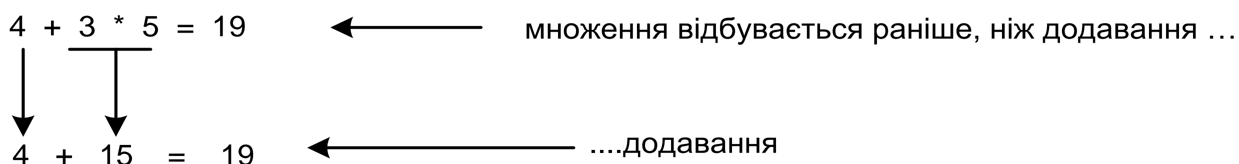


Рис. 3.3. **Операція множення відбувається до додавання**

Отже, коли операнд (наприклад, число 3 на рис. 3.3) може бути опрацьований більше ніж однією операцією (+ або * у цьому разі), мова C# діє за правилами старшинства (пріоритетів) операцій.

Для простоти в числових виразах на рисунках використовували числа у явній формі. Очевидно, що замість них можна було б застосувати комбінацію будь-яких визначень операндів, наведених у синтаксичному блоці "Операнд".

Крім чотирьох бінарних операцій, C# містить і інші арифметичні операції. Порядок виконання кожної з них щодо інших чітко визначено в таблиці пріоритетів [7; 9 – 13].

Огляд операцій та їхніх пріоритетів наведено в табл. 3.3.

Таблиця 3.3

Пріоритети операцій

Категорії	Операції	Асоціативність	Значення
1	2	3	4
Угрупування	(<Вирази>)	Зліва направо	Круглі дужки для угрупування
Первинні	<Ім'я_об'єкта>.<Ім'я_елемента>	Зліва направо	Доступ до елемента
	<Ім'я_методу>(<Аргументи>)	Зліва направо	Виклик методу
	<Ім'я_масиву>[Індекс]	Зліва направо	Доступ до масиву
	<Змінна>++	Справа наліво	Постфіксна операція інкремента
	<Змінна> --	Справа наліво	Постфіксна операція декремента
	new <Ім'я_класу> (<Аргументи>)		Створення екземпляра класу
	typeof(<Тип>)		Визначення типу
	sizeof(<Тип>)		Визначення розміру структури
Унарні	+<Вираз>	Справа наліво	Унарний плюс
	-<Вираз>	Справа наліво	Унарний мінус
	!<Логічний_вираз>	Справа наліво	Логічне заперечення
	++<Змінна>	Справа наліво	Префіксна операція інкремента
	--<Змінна>	Справа наліво	Префіксна операція декремента
	(<Тип>) <Вираз>	Справа наліво	Приведення типу

Закінчення табл. 3.3

<table>
<tr><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><td rowspan="3">Мульти-
плікативні</td><td><Вираз1> * <Вираз2></td><td>Зліва направо</td><td>Множення</td></tr>
<tr><td><Вираз1> / <Вираз2></td><td>Зліва направо</td><td>Ділення</td></tr>
<tr><td><Вираз1> % <Вираз2></td><td>Зліва направо</td><td>Ділення за модулем</td></tr>
<tr><td>Адитивні</td><td><Вираження1> + <Вираження2>
<Вираз1> - <Вираз2></td><td>Зліва направо
Зліва направо</td><td>Додавання
Віднімання</td></tr>
<tr><td rowspan="5">Відно-
шення</td><td><Вираз1> < <Вираз2></td><td>Зліва направо</td><td>Менше</td></tr>
<tr><td><Вираз1> <= <Вираз2></td><td>Зліва направо</td><td>Менше або дорівнює</td></tr>
<tr><td><Вираз1> > <Вираз2></td><td>Зліва направо</td><td>Більше</td></tr>
<tr><td><Вираз1> >= <Вираз2></td><td>Зліва направо</td><td>Більше або дорівнює</td></tr>
<tr><td><Об'єкт> is <Тип></td><td>Зліва направо</td><td>Належність типу</td></tr>
<tr><td>Рівність</td><td><Вираз1> == <Вираз2>
<Вираз1> != <Вираз2></td><td>Зліва направо
Зліва направо</td><td>Дорівнює
Не дорівнює</td></tr>
<tr><td rowspan="3">Логічні
побітові</td><td><Логічний_вираз1> &
<Логічний_вираз2></td><td>Зліва направо</td><td>Побітове І</td></tr>
<tr><td><Логічний_вираз1> ^
<Логічний_вираз></td><td>Зліва направо</td><td>Побітове АБО,
що виключає</td></tr>
<tr><td><Логічний_вираз1> |
<Логічний_вираз2></td><td>Зліва направо</td><td>Побітове АБО</td></tr>
<tr><td rowspan="3">Логічні</td><td><Логічний_вираз1> &&
<Логічний_вираз2></td><td>Зліва направо</td><td>Логічне І</td></tr>
<tr><td><Логічний_вираз1> ||
<Логічний_вираз2></td><td>Зліва направо</td><td>Логічне АБО</td></tr>
<tr><td><Логічний_вираз1> ?
<Вираз1> : <Вираз2></td><td>Зліва направо</td><td>Умовна операція</td></tr>
<tr><td rowspan="6">При-
свою-
вання</td><td><Змінна> = <Вираз></td><td>Справа налі-
во</td><td>Просте
присвоювання</td></tr>
<tr><td><Змінна> *= < Вираз ></td><td>Справа налі-
во</td><td>Множення
і присвоювання</td></tr>
<tr><td><Змінна> /= < Вираз ></td><td>Справа налі-
во</td><td>Ділення
і присвоювання</td></tr>
<tr><td><Змінна> % = < Вираз ></td><td>Справа налі-
во</td><td>Ділення за модулем
і присвоювання</td></tr>
<tr><td><Змінна> += < Вираз ></td><td>Справа налі-
во</td><td>Додавання
і присвоювання</td></tr>
<tr><td><Змінна> - = < Вираз ></td><td>Справа налі-
во</td><td>Вирахування
і присвоювання</td></tr>
</table>

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

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

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

Можна скористатися ключовим словом **namespace** для того, щоб явно задати простір імен для будь-якого блоку коду, розміщеного у фігурних дужках. Імена, які містяться в такому просторі імен, якщо до них звертаються не із цього простору імен, слід кваліфікувати.

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

Наприклад:

```
namespace LevelOne
{
// програма, що міститься у просторі імен LevelOne
// у ній описано ім'я NameOne
}
// програма, що міститься у глобальному просторі імен
```

У цій програмі описано єдиний простір імен – **LevelOne**. (У цьому разі у програмі не міститься ніякого коду, що виконують).

Код, що міститься всередині простору імен **LevelOne**, може просто посилатися на ім'я **LevelOne**, і ніякої класифікації не потрібно. Однак у коді, що міститься у 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 саме ключове слово **namespace**. Під час

звернення до вкладених просторів імен потрібно вказувати всю їхню ієрархію, відокремлюючи один рівень ієрархії від іншого за допомогою точки. Розгляньмо такі простори імен:

```
namespace LevelOne
{
// програма, що міститься у просторі імен LevelOne
namespace LevelTwo
{
// програма, що міститься у просторі імен LevelOne.LevelTwo
// у ній описано ім'я NameTwo
}
}
// програма, що міститься у глобальному просторі імен
```

У цьому разі звернення до імені **NameTwo** із глобального простору імен мусить мати вигляд **LevelOne.LevelTwo.NameTwo,** із простору імен **LevelOne** – **LevelTwo.NameTwo**, а із простору імен **LevelOne.LevelTwo** – **NameTwo**.

Після того, як простір імен визначено, виникає можливість використовувати оператор **using** для спрощення доступу до імен, що в ньому містяться.

У наступній програмі передбачено, що код, який міститься у просторі імен **LevelOne**, мусить мати доступ до імен простору імен **LevelOne.LevelTwo**, без якого б не було класифікації:

```
namespace LevelOne
{
using LevelTwo;

namespace LevelTwo
{
// тут описано ім'я NameTwo
}
}
```

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

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

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

Оператор **using** усього лише спрощує звернення до цих імен і дозволяє скоротити код, котрий дуже подовжено, роблячи його більш зрозумілим.

Ділянка видимості змінних. У більшості програм, поданих раніше, для позначення контексту класу або методу, зокрема методу **Main()**, використовували блокові конструкції з фігурними дужками.

Надалі важливо подати нову концепцію, названу *ділянкою видимості*. Вона визначає, які сегменти (а відтак, і змінні) вихідного коду є видними (і, отже, доступними) з інших його частин.

Ділянка видимості становить сегмент вихідного коду, де можна використовувати певний ідентифікатор. Змінні можна використовувати лише всередині їхньої ділянки видимості. Фактично ділянкою видимості змінної є той блок вихідного коду, де її оприлюднено.

Дві основні ділянки видимості в C# – блок класу та блок методу – показано на рис. 3.4.

Хоча відмінність між ними, радше є штучною, ділянка визначення класу має кілька характеристик, яких немає в ділянці визначення методу. Наприклад, оператори **if-else** можна розмістити тільки в ділянки визначення методу.

Подальше обговорення присвячене ділянці визначення методу і блокам усередині неї.

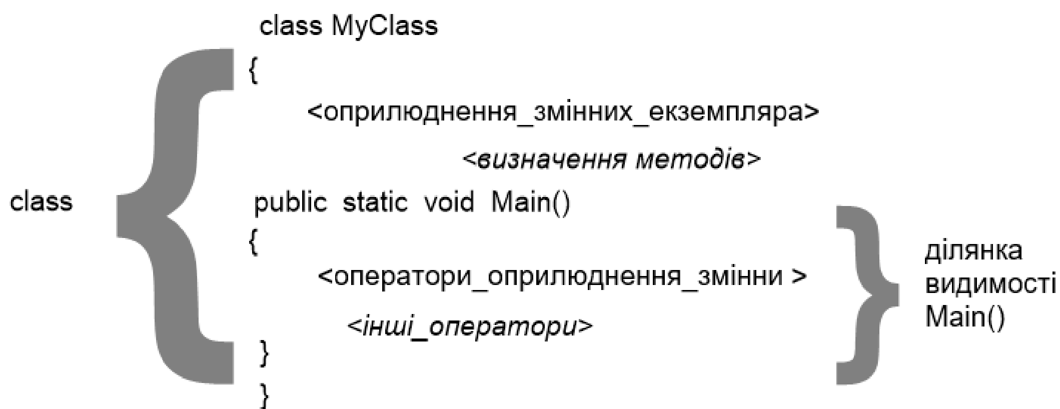


Рис. 3.4. **Блок класу і блок методу**

Новий блок (а отже, і нову ділянку 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. **Приклад вигляду блоку з ділянками видимості методу**

Коли блок **B** міститься всередині другого блоку **A**, ділянку видимості першого називають *внутрішньою*, а другого – *зовнішньою*.

Ці терміни мають відносний сенс, оскільки ділянка блоку **A** є внутрішньою щодо ділянки блоку **Main()**.

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

Змінну, оприлюднену всередині методу, називають *локальною*. Такі змінні є недоступними за межами блоків, у яких їх оприлюднено.

Змінні, оприлюднені всередині блоку, є доступними тільки в ньому (це стосується й змінних інших внутрішніх блоків, укладених у нього) і тільки після оприлюднення змінної. Змінну можна оголосити в будь-якому місці блоку. Дію цих правил показано на рис. 3.6.

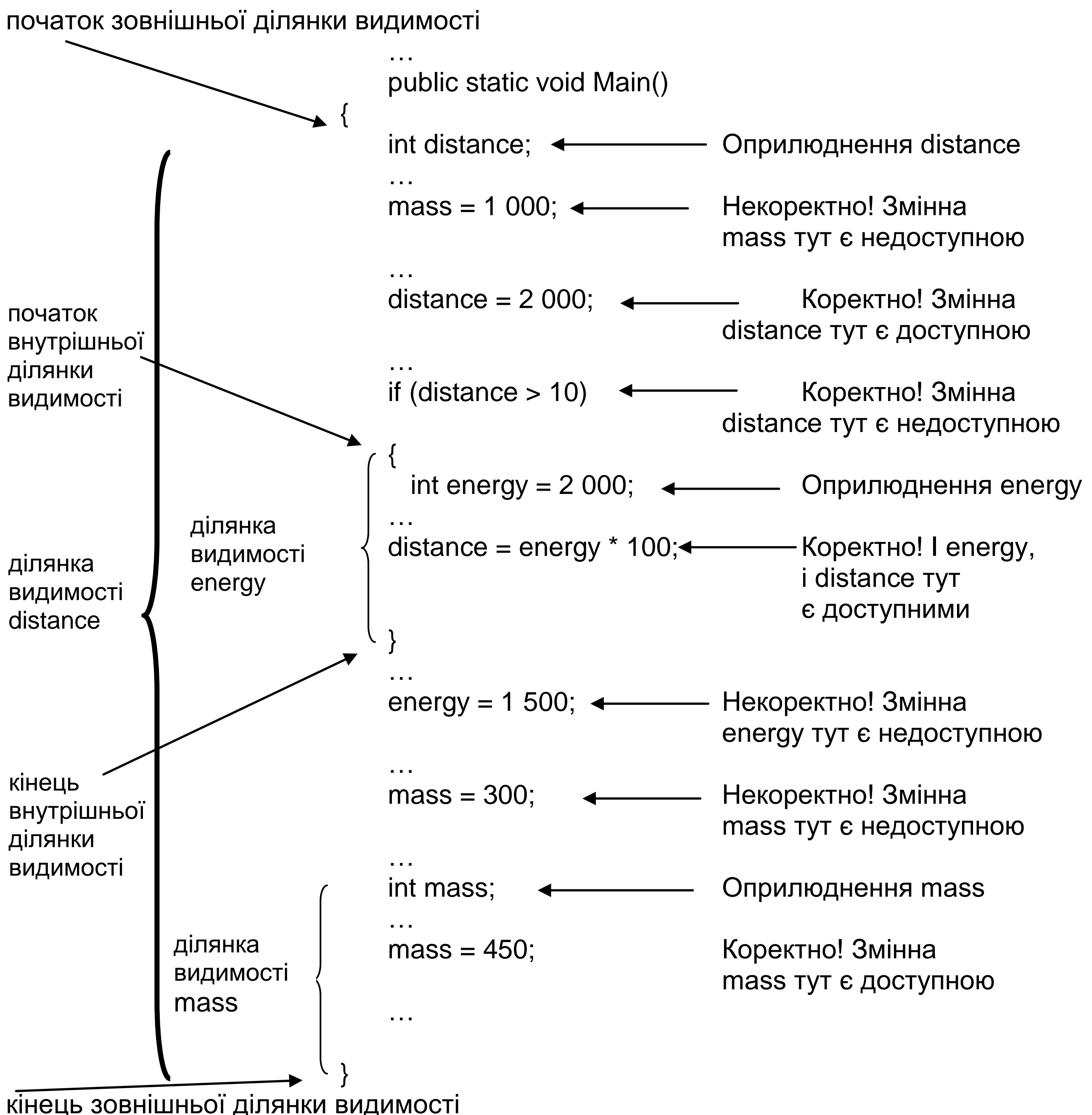


Рис. 3.6. **Блоки, ділянки видимості й доступ до змінних**

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

Оскільки змінна **distance** э оприлюдненою на початку зовнішньої ділянки видимості, вона є доступною у всьому блоці **Main()**, включно з кодом внутрішнього блоку оператора **if**.

Змінна **mass** є оприлюдненною ближче до завершення методу **Main(),** тому вона є доступною у відносно невеликому сегменті коду, між її оприлюдненням і закінченням ділянки видимості **Main()**.

Змінна **energy** є оприлюдненною всередині блоку **if** і доступною лише всередині нього.

Оприлюднення змінної у внутрішній ділянці з ім'ям, ідентичним імені змінної в зовнішній, є помилкою. Причина полягає в тому, що це надавало б іншого значення імені змінної із зовнішньої ділянки.

Ділянка видимості та час існування змінних. В обговоренні імен змінних (ідентифікаторів) ішлося, що це зручний спосіб посилатися на певні ділянки пам'яті.

Під час розгляду концепції ділянки видимості важливо розрізняти можливість використовувати задане ім'я для посилання на дані в пам'яті (ділянка видимості) і час, протягом якого ці дані в пам'яті існують. Останнє називають *часом існування* змінної.

Різницю між ділянкою видимості і часом існування показано на рис. 3.7.

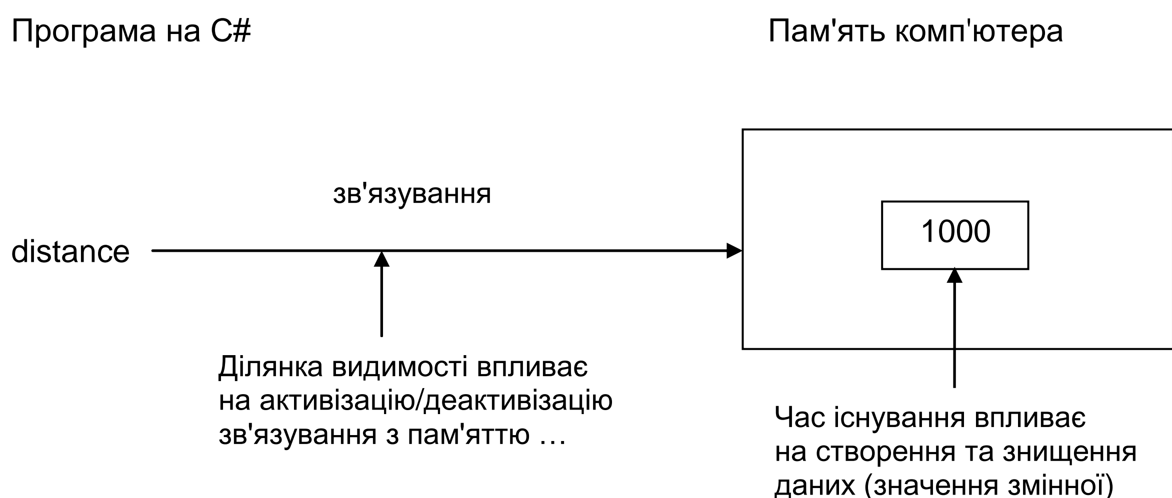


Рис. 3.7. **Ділянка видимості та час існування**

Час існування змінної – це час між її створенням і знищенням. Із моменту створення змінної й до знищення її значення зберігають у пам'яті комп'ютера.

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

Відповідно до загального правила, локальні змінні в C# створюють тоді, коли потік управління програми входить у ділянку їхньої видимості, і знищують, коли потік залишає її. У результаті час існування змінної обмежено ділянкою її видимості.

Хоча наведене правило здається очевидним, його виконують не для всіх мов програмування. Деякі з них дозволяють потоку програм входити й залишати ділянку видимості змінної декілька разів, зберігаючи водночас значення останньої. Під час кожного наступного входження змінна має те саме значення, що вона мала, коли потік управління залишав ділянку видимості.

Форматування числових значень. Дотепер числа виводили з використанням простого вбудованого формату, наприклад:

```
Console.WriteLine("Distance  traveled:" + 10 000 000.432);
```

Водночас на консоль відображено таке:

```
Distance traveled: 10 000 000.432
```

Однак, змінюючи зовнішній вигляд числа за допомогою ком (у закордонній нотації комою прийнято відокремлювати розряди, кратні тисячам), наукової нотації та обмеженої кількості десяткових цифр, можна поліпшити його читаність і зробити більш компактним під час виведення на екран. У табл. 3.4 наведено відповідні приклади.

Таблиця 3.4

Приклади форматованих чисел

Імена змінних	Числа	Відформатовані числа
distance	7000 000 000 000 000	7.00E+015
Mass	3.878 390 298 378 987 7362	3.8784
Length	20 000 000	20,000,000

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

На рис. 3.8 показано використання методу ToString для перетворення числа 20 000 000.459 6598 1m типу decimal на рядок типу string з комами (для відокремлення тисяч) і лише двома десятковими розрядами.

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

Аргумент складається із символу, названого *символом формату* (у цьому разі N), що задає формат, і необов'язкового числа специфікатора точності (у цьому разі 2), яке має різний сенс для різних символів формату.

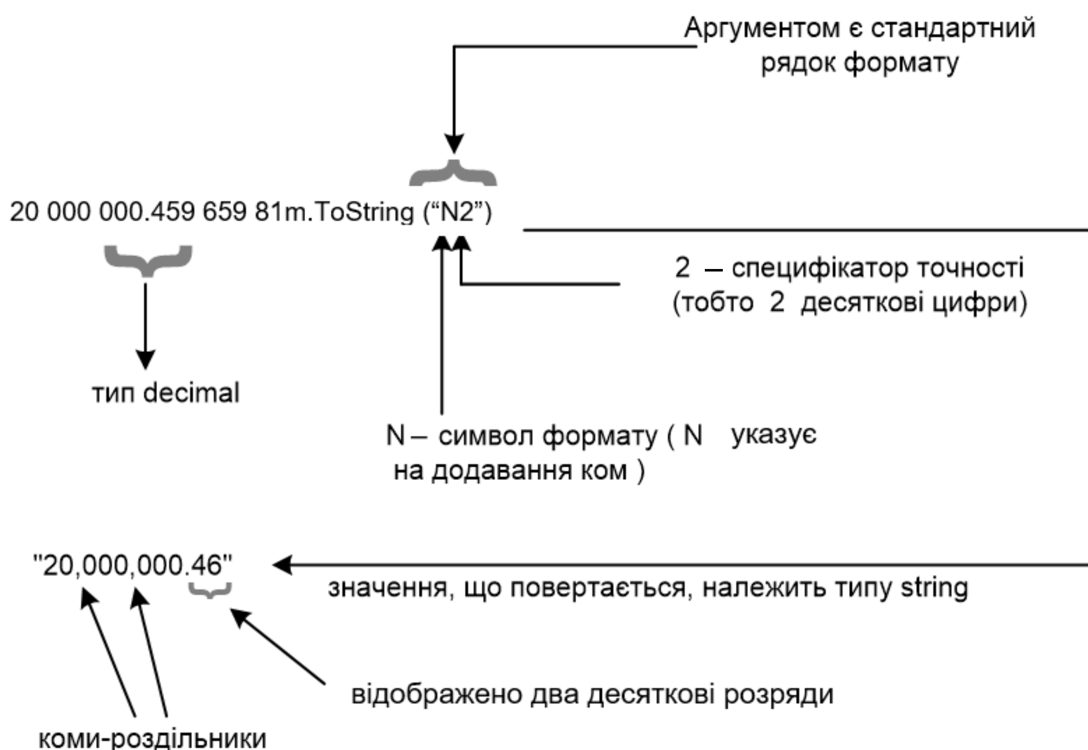


Рис. 3.8. **Форматування літерала типу decimal**

Використовувані символи та відповідні їм формати наведено в табл. 3.5.

Таблиця 3.5

Використовувані в C# символи формату

Символи	Описи	Приклади
C, c	Валюта. Форматування, специфічне для настроювань локалізації. Вони містять інформацію про тип грошової одиниці та інших параметрів, які можна змінювати, залежно від країни	2 000 000.456m.ToString("C") Повертає "$2,000,000,46" (Якщо операційна система є настроєною відповідно до американських стандартів)
D, d	Ціле число. Специфікатор точності встановлює мінімальну кількість цифр. Виведення доповнено провідними нулями, якщо кількість цифр фактичного числа є меншою, ніж специфікатор точності. *Примітка:* цей символ формату застосовують тільки для цілочисельних типів	45 687.ToString("D8") Повертає: "00 045 678"
E, e	Експонентна (наукова) нотація. Специфікатор точності визначає кількість десяткових цифр, за замовчуванням дорівнює 6	345 678 900 000.ToString("E3") Повертає "3.457E+011" 345 678 912 000.ToString("e") Повертає "3.456 789e011"
F, f	Фіксована точка. Специфікатор точності вказує кількість десяткових цифр	3.766 789 2.ToString("F3") Повертає "3.767"
G, g	Загальний. Найбільш компактний формат під час вибору E або F. Специфікатор точності встановлює максимальну кількість цифр у поданні числа	65 432.987 65.ToString("G") Повертає "65 432.98765" 65 432.987 65.ToString("G7") Повертає "65 432.99" 65 432.987 65.ToString("G4") Повертає "6.543E4"
N, n	Число. Число з комами-роздільниками. Специфікатор точності встановлює кількість десяткових цифр	1 000 000.123m.ToString("N2") Повертає "1,000,000.12"
X, x	Шістнадцяткове число. Специфікатор точності встановлює мінімальну кількість цифр, що подають у рядку. Для досягнення певної ширини додають провідні 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 є незручним, якщо в рядок типу string уставлено декілька по-різному відформатованих чисел.

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

Console.WriteLine("The length is: " + 10 000 000.432 4ToString("N2") + "The widlh is: " + 65 476 356 278.098 746.ToString("N2") + "The height is: " + 4 532 554 432.456 84.ToString("N2"));

На екран виводять таке:

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

Вираз був би більш чітким, якби можна було розподілити статичний текст і числа та вказати (за допомогою невеликої кількості специфікаторів) позицію і формат кожного числа.

Мова C# надає чітке розв'язання цієї проблеми.

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

Console.WriteLine("The length is: {0} The width is: {1} The height is: {2}", 10 000 000.432 4, 65 476 356 278.098 746, 4 532 554 432.456 84);

де {0} належить до першої величини (10 000 000.432 4) у списку чисел після рядка тексту, {1} – до другої (65 476 356 278.098 746), а {2} – до третьої.

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

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

де <Формат> подає статичний текст зі специфікаторами формату (див. табл. 3.5);

<Значення0>, <Значення1>... подають значення, що вставляють у рядок <Формат> у позиціях, заданих специфікаторами.

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

Наприклад, виклик методу Console.WriteLine (рис. 3.9), приводить до такого виведення на консоль: Mass: 100 Distance 50 Energy 35.

Специфікатор формату дозволяє також указати ширину простору, що відведено під виведення <Значення>. Для цього після індексу через кому задано ширину в символах.

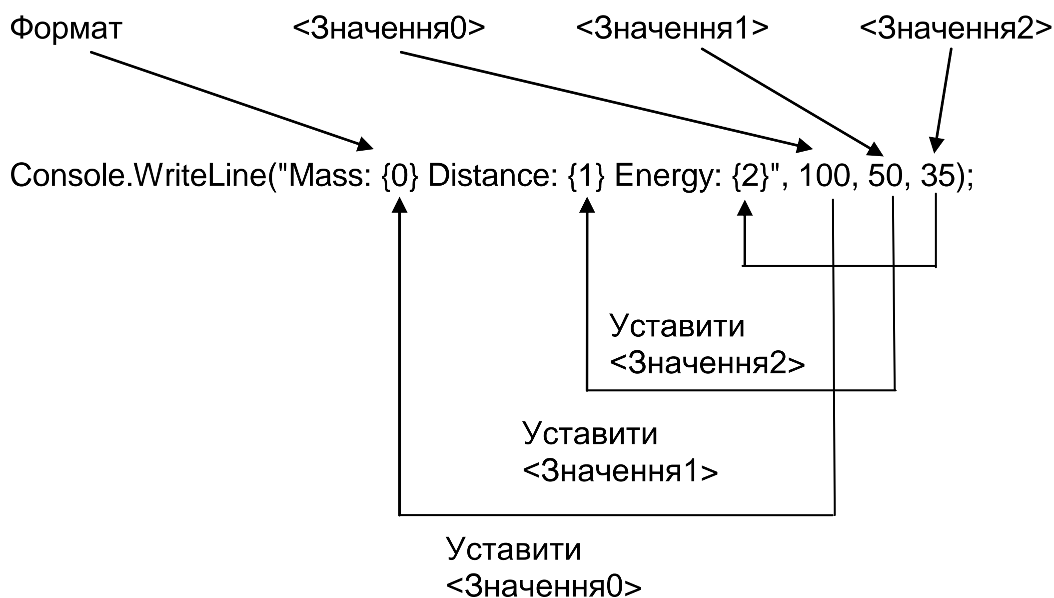


Рис. 3.9. **Приклад виклику методу Console.WriteLine**

Додатне число означає вирівнювання <Значення> праворуч, а від'ємне – ліворуч.

Наприклад:

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

виводить

Distance: 100 miles

ширина дорівнює 10 символам
і число вирівняно праворуч,

тоді, як код

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

↖

від'ємне число вказує
на вирівнювання ліворуч.

Виводить:

```
Distance: 100        miles
```

↑

ширина дорівнює 10 символам
і число вирівняно ліворуч.

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

Символ формату йде після крапки з комою. Він має таке саме значення, як під час використання з методом ToString.

Наприклад, вираз

```
Console.WriteLine("Distance:{0,-12E2}  Mass:  {1,-12N}",  100 000  000, 5 000 000);
```

спричиняє таке виведення:

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

Для друкування на консолі рядка з фігурними дужками досить просто опустити <Значення0>, <Значення1> тощо у виклику Console.WriteLine. Компілятор C# вибере іншу версію (завдяки перевантаженню) метода Console.WriteLine, де дужки { } ігнорують і друкують як є.

Наприклад,

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

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

Приклади лінійних програм. 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("Уведіть ціле число A:");
    int myInt = Convert.ToInt32 (Console.ReadLine()); // 7
    Console.WriteLine("Число A менше ніж 10? {0}", myInt < 10); // true
    Console.WriteLine("Число міститься 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 із яких ілюструє виконання однієї з операцій, і поточний виконуваний код, що демонструє операції відношень.

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

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

У вихідному коді подано програму **SimpleCalculator.cs**, що обчислює суму, добуток, максимум і мінімум двох чисел, уведених користувачем. Відповідь виводять на консоль.

```
01: using System;
02:
03: /*
04:  * Цей клас визначає суму, добуток,
05: * мінімум і 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("Уведіть перше число: ");
15:            x = Convert.ToInt32(Console.ReadLine());
16:            Console.Write("Уведіть 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:
Уведіть перше число: 3
Уведіть 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 за назвою **System**. У ньому містяться класи, фундаментальні для будь-якої програми на C#. **System** містить 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 в разі будь-якого звернення до класів простору імен **System** ідентифікатор останнього потрібно 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 цьому разі будь-який клас простору імен **System** можна викликати без явної вказівки префікса **System**. Попередній приклад коду можна, отже, урізати до такого:

Console.WriteLine("Bye Bye!");

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

Перетворення значення типу string у тип int. Уведення користувача завершується натисканням **Enter**. Визначений методом **Console.ReadLine( )** результат

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

належить типу **string**. Будь-яке число, що вводять, наприклад **432**, розглядають як набір символів – таких само, як **ABC** або **#@$.**

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

Щоб перетворити введення користувача на число, у рядку 15 застосовують метод **ToInt32** класу **Convert**. Результат перетворення (праворуч від знака рівності в рядку 15) є числом, його надано змінній **x**. Природно, значення, уведене користувачем, має збігатися із цілим числом. Уведення **57,53,** або **three hundred**, викликає помилку, а значення **109** або **64 732** 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 у програмах використовували готові методи на зразок **Console.ReadLine()** і **Console.WriteLine()**. Незважаючи на величезну кількість убудованих методів .**NET Framework**, а також доступність інших комерційних бібліотек класів, під час створення власного коду будуть потрібні й власні, визначені користувачем методи.

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

Під час виклику метод **Sum()** має два числа. Він додає їх і повертає результат у точку виклику.

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

У загальному випадку класи подають об'єкти, а методи – дії. Саме це варто відображати у їхніх іменах: 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 є багато в чому схожим на заголовок методу **Main** у рядку 9 (нагадаємо, що специфікатор доступу **public** додає метод в інтерфейс класу, якому той належить).

Далі ключове слово **void** замінено на **int**, а в дужках після **Sum** містяться конструкції, схожі на оприлюднення змінних. Це потребує деякого пояснення.

Рядок 25 становить інтерфейс методу **Sum**. Він відповідає за взаємодію між методом, що викликає **Sum** (у цьому разі **Main**), і тілом **Sum**.

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

На рис. 3.10 показано процес виклику методу **Sum(int a, int b)**, оприлюдненого в рядку 25, із рядка 18 методу **Main()** як **Sum(x, y)**.

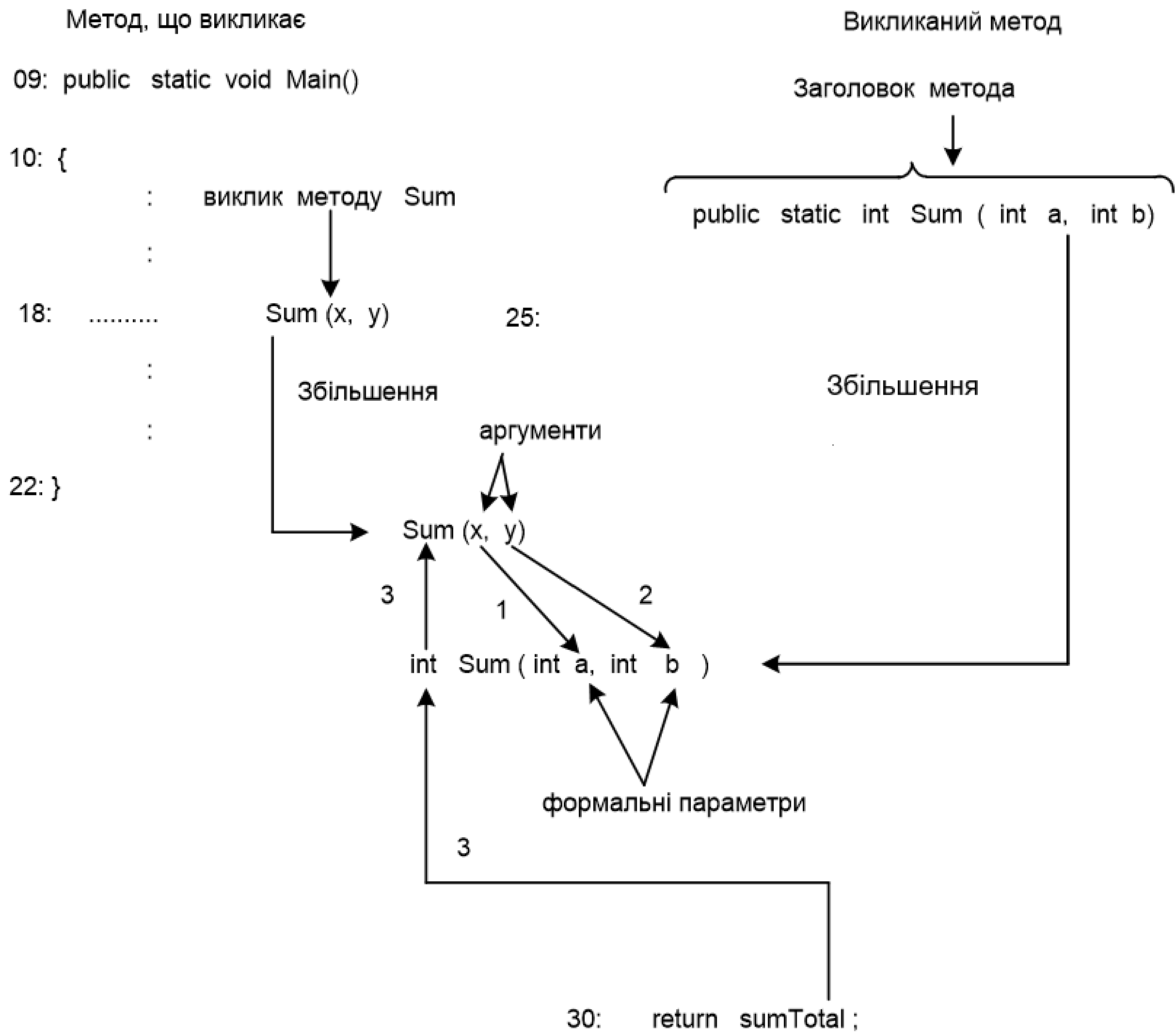


Рис. 3.10. **Виклик методу, визначеного користувачем**

Дві стрілки, позначені як "Збільшення", указують, які частини показано детальніше.

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

Формальний параметр застосовують у тілі методу для звернення до того значення, що було збережено в ньому в момент виклику методу. У цьому разі аргументи **x** та **y** у рядку 18 присвоюють параметри **a** і **b** (це показано стрілками 1 і 2) так само, якби були виконаними оператори присвоювання **a = x**; і **b = y**;.

Змінні **a** і **b** використовують у тілі методу (у рядку 29) як звичайні оприлюднені змінні. Їхні значення дорівнюють **x** та **y**.

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

```
30:        return sumTotal;
```

Ключове слово **return** належить до операторів повернення управління. Коли запускають такий оператор, виконання методу зупиняється. Потік виконання повертається в точку виклику й підставляє вираз, зазначений після **return** (у цьому разі **sumTotal**, див. стрілку 3 на рис. 3.10). Тому тип **sumTotal** має бути сумісним із типом, оприлюдненим у заголовку методу (тут **int**).

Коли перед ім'ям методу стоїть слово **void**, це значить, що після його завершення в точку виклику не повертаються ніякі дані.

Об'єднання рядків і викликів методів. Розгляньмо в рядку 18 блок **Sum(x, y),** який, як уже показано, викликає метод **Sum**, визначений у 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 управління, можна фактично замінити **Sum(x, y)** значенням **sumTotal** із рядка 30.

Хоча конструкція **Sum(x, y)** міститься всередині виклику іншого методу – **Console.WriteLine(),** — C# визначає, у якій послідовності мають відбуватися виклики.

Тут виклик здійснюють тоді, коли потрібно значення **Sum(x. y)**. Після того як виконано код із рядків 25 – 31 і потік виконання повертається в рядок 18, його можна подати в такому вигляді:

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

Оскільки 11 (передбачають, що раніше було введено числа a = 3, b = 8) міститься всередині виклику методу **WriteLine**, це значення автоматично перетвориться на рядок. Отже, маємо такий вигляд оператора:

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

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

Об'єднання двох рядків разом називають *конкатенацією*, а символ + у цьому разі – *операцією конкатенації*.

Після завершення рядок 18 набуває такого вигляду:

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

Рядки 34 – 40 є дуже схожими на рядки 25 – 31, єдина розбіжність 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 є ідентичними.

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

Порядок, у якому визначено методи класу, є довільним і не пов'язаним із тим, як їх викликають із вихідного коду. Наприклад, можна змінити порядок визначень методів **Sum** і **Product** у класі **SimpleCalculator**.

Клас Math. Рядок 20 є схожим із 18 та 19, однак у ньому для визначення найбільшого із двох чисел застосовують 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** наведено далі:

```
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 є очевидним: його визначено абревіатурою й тому тут не наведено.

3.2. Оператори управління програмою

Поняття потоку управління програмою. У розглянутих лінійних програмах зазвичай усі оператори в методі виконували послідовно. Програма, основана лише на послідовному виконанні, завжди виконує ті самі дії. Вона не здатна реагувати на поточні умови.

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

Потік управління становить порядок, у якому виконують оператори програми. Крім того, часто використовуються терміни "порядок виконання" і "керівний потік".

Приклад програми з лістингу 2.2 (див. підрозділ 2) дозволяв користувачеві впливати (уведенням **yes** або **no**) на потік управління та здійснювати виведення повідомлення "Hello World!". Така логіка потребує

можливості робити вибір між двома або декількома гілками на основі умови. Оператор **if**, поданий у програмі лістингу 2.2, разом з оператором **switch**, про який буде сказано далі, формують групу виразів, призначених саме для цієї мети.

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

Структури вибору альтернатив.

C# забезпечує три типи структур вибору альтернатив [2; 7; 10]:

єдиний вибір – структура **if** (ЯКЩО);

подвійний вибір – структура **if / else** (ЯКЩО / ІНАКШЕ);

множинний вибір – структура **switch**.

Структура вибору if. Синтаксис оператора **if** подано в такому синтаксичному блоці:

Оператор_if ::=

if(<Умова>)

<Оператор>;

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

<Оператор>, що слідує за логічним виразом, виконують лише в тому разі, якщо останній є істинним.

Графічну схему оператора показано на рис. 3.11.

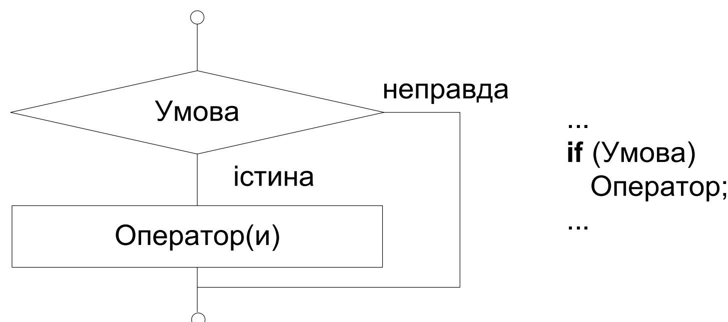


Рис. 3.11. **Графічна схема оператора if**

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

```
using System;

        class Class1
        {
                static void Main( )
                {
                        int valor;
                        Console.WriteLine("Уведіть число місяця");
                        valor = Convert.ToInt32(Console.ReadLine());
                        if (valor < 1  ||  valor >31)
                                Console.WriteLine("Помилка введення!");
                        Console.WriteLine("Ви ввели число, що дорівнює
{0}", valor);
                }
        }
```

Результат роботи програми:

```
Уведіть число місяця
17
Ви ввели число, що дорівнює 17
Press any key to continue
Уведіть число місяця
32
Помилка введення!
Ви ввели число, що дорівнює 32
Press any key to continue
```

Як оператори не можна використовувати оприлюднення і визначення.

Однак тут можуть бути складені оператори та блоки:

Складений_оператор ::=
{
<Оператори>
}

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> виконують лише в тому разі, коли логічний вираз **(<Умова>)** дорівнює **true**.

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

Перед **else** обов'язково ставлять крапку з комою.

Графічну схему оператора показано на рис. 3.12.

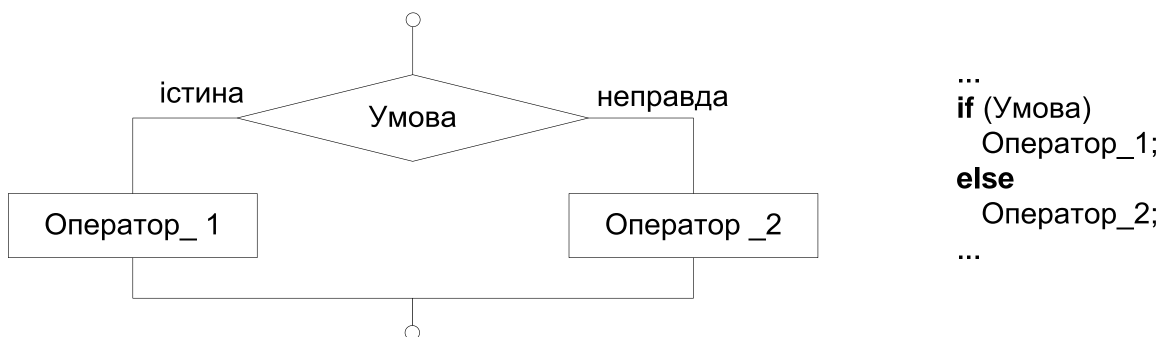


Рис. 3.12. **Графічна схема оператора if / else**

Приклад. Визначення мінімуму із двох чисел.

```
using System;
    class Class1
    {
        static void Main()
        {
            int x, y, min;
```

```
            Console.WriteLine("Послідовно введіть два цілі 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 вкладеними один в одного.

Приклад. Визначення максимального числа із трьох чисел a, b, c.

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

            int a, b, c, max;
            Console.WriteLine("Послідовно введіть три цілі 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.

Множинний вибір – структура switch. Оператор **switch** дозволяє програмі вибрати одну з кількох дій на основі значення заданого виразу. Логіка, реалізована **switch**, є подібною до логіки оператора **if / else**.

Синтаксичний блок оператора **switch** такий:

```
switch (<вираз_switch >)
{
    case <Константний_вираз>:
        [ <Оператор>;
        <Оператор>; <Оператор>;
        <Оператор_break>; | <Оператор_goto> ]

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

    <Будь-яка кількість блоків case>

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

Графічну схему оператора **switch** показано на рис. 3.13.

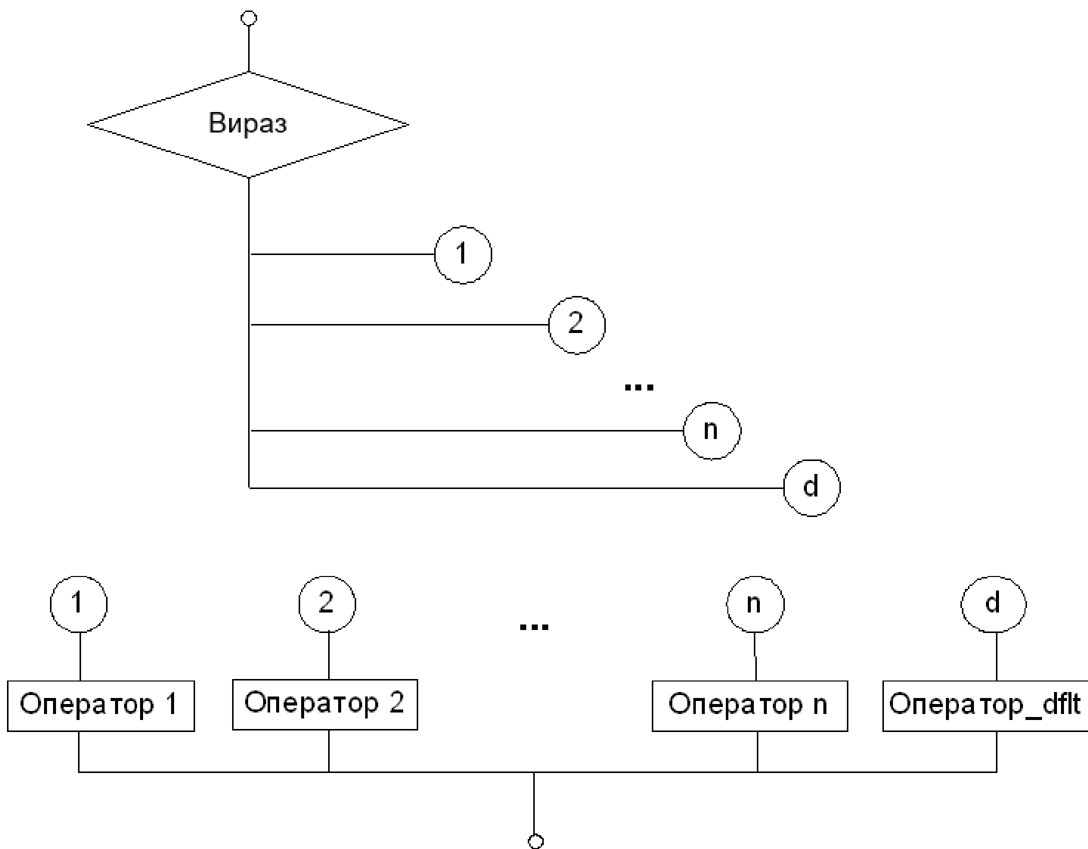


Рис. 3.13. **Графічна схема оператора switch**

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

Термін **(<вираз_switch>)**, використовуваний разом із ключовим словом **switch**, – це загальноприйнята назва керівного виразу.

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

<Константний_вираз>, що йде вслід за ключовим словом **case**, називають значенням або **case**-позначкою. Кожна з них має бути унікальною.

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

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

Коли потік управління переходить від одного розділу **switch** до іншого, таке виконання називають *провалом*. Для запобігання йому застосовують оператор **break** або **goto**.

Приклад. Аналіз 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 маєте ви-
брати 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("Вибрано Add\n");
					break;
				case 'D':
					Console.Write("Вибрано Delete\n");
					break;
				case 'S':
					Console.Write("Вибрано Sort\n");
					break;
				case 'Q':
					Console.Write("Вибрано Quit\n");
					break;
				default:
					Console.Write("Уведено помилковий символ\n");
					break;
```

```
            }
        Console.Write("\nДля завершення програми натисніть <Enter>");
        Console.ReadLine();  // для паузи
        Console.ReadLine();  // для паузи
        }
}
```

Один із можливих результатів роботи програми:

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

Для завершення програми натисніть <Enter>.
```

Умовний вираз. У C# є ще одна скорочена форма умовного оператора – так званий умовний вираз. Його синтаксис такий:

<Логічний_вираз_1>   ?   <Вираз_2> : <Вираз_3>;

Сенс цієї конструкції полягає в такому:

обчислюють **<Логічний_вираз_1>**; якщо він є істинним (**true**), то результатом буде **<Вираз_2>**, інакше результатом буде **<Вираз_3>**.

Насправді, умовний вираз є еквівалентним такому умовному операторові:

```
if  ( Логічний_вираз_1 )
      Вираз_2;
else
      Вираз_3;
```

Приклад. Визначення максимуму із двох чисел.

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

Структури повторення. Термінологія. Оператори циклу використовують для організації багаторазово повторюваних обчислень.

Будь-який цикл складається з:

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

початкових установлень;

модифікації параметра циклу;

перевірки умови продовження виконання циклу.

Один прохід циклу називають *ітерацією*. Перевірку умови виконують 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) показано на рис. 3.14.

Різниця між ними полягає в тому, що тіло циклу з постумовою завжди виконують хоча б один раз, після чого перевіряють, чи треба його виконувати ще раз. Перевірку необхідності виконання циклу з передумовою виконують до тіла циклу, тому можливо, що її не виконають жодного разу.

Змінні, які змінюють у тілі циклу та використовують під час перевірки умови продовження, називають *параметрами циклу*.

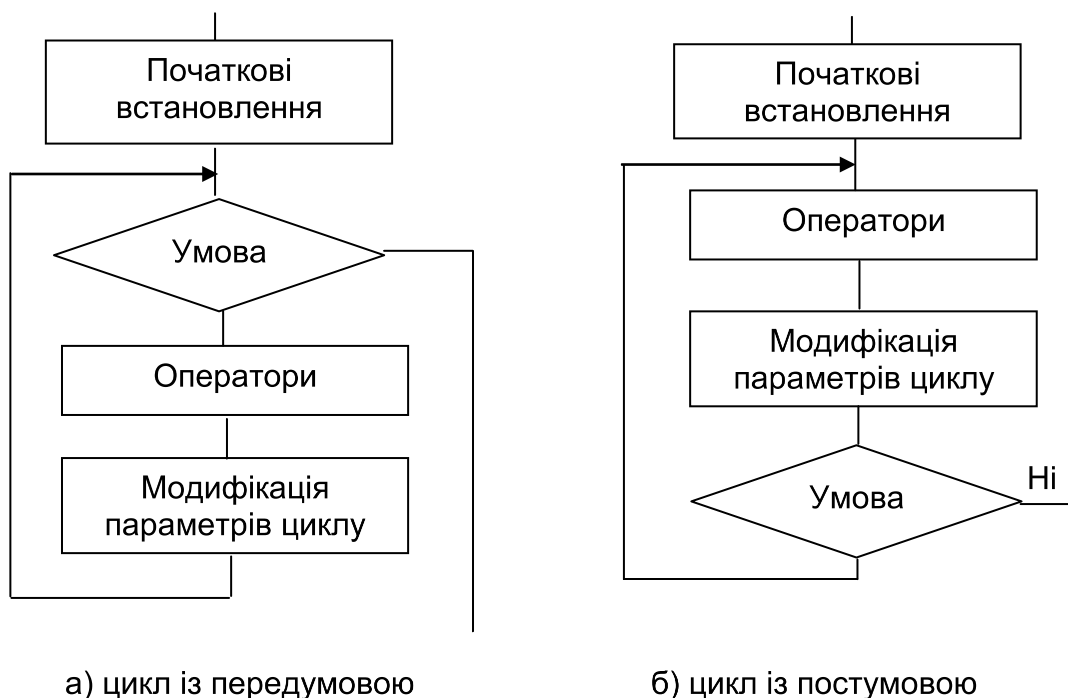


Рис. 3.14. **Графічні схеми, що демонструють роботу циклів**

Цілочисельні параметри циклу, що змінюють із постійним кроком на кожній ітерації, називають *лічильниками циклу*.

Початкових установлень може явно не бути у програмі, їх сенс полягає в тому, щоб до входу в цикл задати значення змінним, які в ньому використовують.

Цикл завершується, якщо умову його продовження не виконано.

Можливе примусове завершення як поточної ітерації, так і циклу загалом. Для цього слугують оператори **break**, **continue, return, goto**. Передавати управління іззовні всередину циклу не рекомендовано.

У C# є чотири різні оператори ціклу – **while, do while, for, foreach**.

Цикл із передумовою (while). Цикл із передумовою реалізує графічну схему (без блоку початкових установлень), показану на рис. 3.14 (а), і має такий синтаксис:

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

де:
<Умова_циклу> ::= <Логічний_вираз>
<Тіло_цикпу> ::= <Оператор>;
::= <Складений_оператор>

Тіло циклу повторюють доти, доки **<Умова_циклу>** є істинною **(true)**.

Виконання оператора починають з обчислення логічного виразу **(<Умова_циклу>)**. Якщо він є істинним (не дорівнює **false**), виконують **<Тіло_циклу>**.

Якщо під час першої перевірки вираз дорівнює **false**, цикл не виконують жодного разу. Логічний вираз обчислюють 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

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

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

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

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

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

```
                double f;
                Console.WriteLine("Уведіть 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:
Уведіть 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. Угадування числа.

Користувачеві надано 10 спроб для вгадування заданого у програмі числа 25.

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

		int	i = 1,  respuesta = 1;
		Console.WriteLine("Уведіть 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("Вітаємо, Ви вгадали число!");
	}
}
```

Результат:
Уведіть 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
Вітаємо, Ви вгадали число!
Press any key to continue

Цикл із постумовою (do while). Цикл із постумовою реалізує графічну схему (без блоку початкових установлень), показану на рис. 3.14 (б), і має такий вигляд:

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

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

<Тіло_циклу> повторюються доти, доки **<Умова_циклу>** дорівнює **true**.

Спочатку виконують простий або складений оператор, що становлять тіло циклу, а потім обчислюють логічний вираз **(<Умова_циклу>)**. Якщо він істинний (не дорівнює **false**), тіло циклу виконують ще раз.

Цикл завершується, коли **(<Умова_циклу>)** буде дорівнювати **false** або в тілі циклу буде виконано який-небудь оператор передавання управління.

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. Угадування букви.

```
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 ніж задумана 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 ніж задумана мною або ВИЩЕ (перед моєю) за абеткою

```
m
  шукай ВИЩЕ


g
  шукай ВИЩЕ


d
```

шукай НИЖЧЕ

е
УГАДАВ!!!

Кінець гри
Press any key to continue

Оператор циклу for. Цей оператор використовують у тих випадках, коли кількість повторень тіла циклу заздалегідь є відомою або може бути визначеною 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> використовують для оприлюднення та присвоєння початкових значень величинам, що використовують у циклі. У цій частині можна записати декілька операторів, розділених комами.

Ділянкою дії змінних, оприлюднених у частині ініціалізації циклу, є цикл.

Ініціалізацію виконують один раз на початку виконання циклу.

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

<Оператори_оновлення> виконують після кожної ітерації циклу. Вони зазвичай слугують для зміни параметрів циклу. У частині модифікацій можна записати декілька операторів через кому.

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

Цикл із параметром реалізовано як цикл із передумовою (див. рис. 3.14 (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
```

3.3. Керівні оператори в циклах

Крім умовних операторів і операторів циклу, є ще три оператори, призначені для цих самих цілей. Їх застосовують рідше, тому що часте використання погіршує наочність програми та підвищує ймовірність помилок.

Оператор break. Може використовуватися в циклах усіх трьох типів.

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

Якщо оператор **break** перебуває всередині вкладених циклів, то його дію поширено тільки на той цикл, безпосередньо у якому він міститься.

Приклад. Угадування числа.

Потрібно визначити задумане (визначене у програмі) число із трьох спроб.

У цьому прикладі під час вгадування 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("Уведіть 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("Вітаємо, Ви вгадали число!");
                Console.WriteLine("Гру закінчено. Удачі!");
        }
}
```

Один із можливих результатів:

```
Уведіть 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.
Гру закінчено. Удачі!
Press any key to continue
```

Оператор continue. Оператор **continue** можна використовувати тільки серед операторів тіла циклу. Він викликає пропуск частини ітерації всередині циклу, що залишилася, і перехід до наступної ітерації.

Приклад. Контроль за введенням значень дня місяця.

Уводять числа місяця для опрацювання. Потрібно здійснити перевірку правильності введення. Число 31 означає кінець опрацювання.

```
using System;
class Class1
{
```

```
        static void Main()
        {
                int dia = 1;
                while( dia !=31 )
                        {
                                Console.WriteLine("Уведіть день місяця");
                                dia = Convert.ToInt32(Console.ReadLine());
                                if( dia < 1  ||  dia >=31 )
                                        continue;
                                Console.WriteLine("Опрацювання введеної дати
{0}",dia);
                        }
                Console.WriteLine("Кінець опрацювання");
        }
}
```

Результат:
Уведіть день місяця
3
Опрацювання введеної дати 3
Уведіть день місяця
35
Уведіть день місяця
2
Опрацювання введеної дати 2
Уведіть день місяця
78
Уведіть день місяця
9
Опрацювання введеної дати 9
Уведіть день місяця
31
Кінець опрацювання
Press any key to continue

У цьому прикладі неправильне введення значення призводить до пропуску частини ітерації, призначеної для опрацювання цього значення.

Помітьмо, що така зміна умови

if (dia < 1 || dia > 31)

на зворотне

if (dia > 0 && dia < 32)

дозволяє вилучити використання оператора **continue**.

Із допомогою оператора **continue** іноді можна скоротити деякі програми, особливо якщо вони містять укладені оператори **if else.**

Якщо **continue** використовують у циклі **for**, то треба мати на увазі таке: перед початком нової ітерації спочатку обчислюють вираз модифікації (визначають нове значення параметра циклу) і лише потім управління передають на заголовок.

Приклад. Застосування операторів **continue** і **break** у 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

Оператор goto (перехід на задану позначку). Оператор goto є неякісним засобом. Його використання призводить до значного ускладнення логіки програми.

Є лише один випадок, коли програмісти-професіонали допускають використання **goto**, – це вихід із вкладеного набору циклів під час виявлення помилок (**break** дає можливість виходу лише з одного циклу).

Укладені цикли. Укладеним циклом називають конструкцію, у якій один цикл виконують усередині іншого.

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

Приклад. Заповнення заданого прямокутника 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 усіх типів: **for, do/while, while**, якщо цього потребує логіка побудови програми.

Приклад. Уведення 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("Уведіть день місяця");
```

```
                    dia = Convert.ToInt32(Console.ReadLine());
                }
                while  ( dia < 1  ||  dia > 31 );
                    Console.WriteLine("Опрацювання введеної дати
{0}",dia);
            }
            Console.WriteLine("Кінець опрацювання");
        }
}
```

Результат:

```
Уведіть день місяця
3
Опрацювання введеної дати  3
Уведіть день місяця
17
Опрацювання введеної дати  17
Уведіть день місяця
32
Уведіть день місяця
1
Опрацювання введеної дати  1
Уведіть день місяця
0
Уведіть день місяця
7
Опрацювання введеної дати  7
Уведіть день місяця
9
Опрацювання введеної дати  9
Кінець опрацювання
Press any key to continue
```

Зовнішній цикл (for) виконують п'ять разів, а внутрішній (do/while) будуть виконувати доти, доки не буде введено правильне значення дня місяця.

Рекомендації з вибору циклів. Оператори циклу є взаємозамінними. Далі наведено деякі рекомендації з вибору найкращого оператора циклу в кожному конкретному випадку:

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

оператор **for** переважає в більшості інших випадків (однозначно – для організації циклів із 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.

Оператора **foreach** буде розглянуто рід час опрацювання масивів.

Контрольні запитання

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 їх використовують?
6. Розкрийте сутність таких понять: "простір імен", "ділянка видимості змінних", "ділянка видимості" та "час існування змінних".
7. Напишіть програму обчислення суми, добутку, максимуму й мінімуму трьох чисел.
8. Як перетворити значення типу string на тип int?
9. Опишіть загальну схему створення і виклику призначених для користувача методів.
10. Охарактеризуйте клас Math. Наведіть приклад програми, де використовують убудовані математичні методи.
11. Дайте поняття потоку управління програмою.
12. Що становить структура вибору if і коли її треба використовувати?
13. Опишіть структуру вибору if/else.
14. У чому полягає специфіка множинного вибору і як цей вибір доцільно реалізувати за допомогою структури switch?
15. Наведіть синтаксис умовного виразу і відповідний приклад його використання.

16. Дайте загальний огляд структур повторення.

17. Опишіть особливості використання циклу з передумовою (while). Наведіть приклади.

18. Коли доцільно використовувати цикл із постумовою (do while)? Наведіть приклади.

19. Дайте синтаксис оператора циклу for.

20. Які керівні оператори можуть використовувати в циклах?

21. Опишіть оператори break і continue.

22. У чому полягають особливості застосування оператора continue в циклі for?

23. Охарактеризуйте оператор goto.

24. Що таке "вкладені цикли" та коли їх доцільно використовувати?

25. Наведіть загальні рекомендації щодо вибору циклів.

26. Розробіть алгоритм (графічну схему) і напишіть програму для обчислення такого виразу:

$Y = 1 + x / 1! + x_2 / 2! + \ldots + x_n / n!$ із заданою точністю eps = 0.1 e-3.

Розділ 2
Організація й опрацювання складених типів даних

4. Масиви

4.1. Загальні відомості про масиви

Оголошення й визначення масиву. Тип масиву є дуже схожим на тип **string** – саме тому тип **string** часто називають масивом символів.

Змінна типу **string** може містити набір чітко розташованих і проіндексованих символів. Доступ до кожного з них здійснюють таким чином (рис. 4.1).

У цьому фрагменті коду підраховують кількість елементів **b** у рядку **myText** і виводять 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** становить основу для масиву.

```
...
  string   myText  =  " This   is   a   short   text"  ;
char   ch;
ch  =  myText [ 2 ] ;
...
```

доступ до третього символу (i) за унікальним індексом [2]

для подання всієї колекції символів використовують одне ім'я

або:

```
string myText = "To b or not to be said the bee"
int bCounter = 0;
for (int i=0; i<myText.Length; i++)
{
     If (myText[i].ToString().ToUpper() == "B"
     bCounter++;
}
     Console.WriteLine("Number of b ' s in text: " + bCounter);
```

ініціалізуючи змінну і значення 0, інкрементуючи її в кожному циклі:

програма звертається до кожного конкретного символу:

і підраховує кількість символів b

Рис. 4.1. **Інтерпретація типу string як масиву символів**

Отже, масив – це об'єкт, що, подібно об'єкту **string**, має багато вбудованих методів, використовуваних для доступу до елементів колекцій даних та їхнього опрацювання [8].

Між рядками **string** і масивами є чимало подібності, але тоді як тип **string** обмежено лише поданням колекцій символів, *масив може містити колекцію величин будь-якого типу*, включно з яким-небудь із простих типів **int**, **long**, **decimal** тощо, а також будь-яку групу об'єктів.

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

Усі елементи масиву мають належати тому самому типу. Тип елементів масиву називають типом масиву, або **базовим типом масиву**.

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

Змінну типу **array** оприлюднюють у вихідній програмі за допомогою зазначення масиву, що супроводжують порожньою парою квадратних дужок та ім'ям масиву. У такому рядку: **decimal [ ] accountBalances;** оприлюднюють змінну **accountBalances**, яка є здатною зберігати посилання на об'єкт масиву, що містить колекцію чисел типу **decimal**.

Попереднім оприлюдненням створюють порожній контейнер, здатний зберігати посилання на об'єкт масиву. Сам об'єкт масиву з колекцією величин типу **decimal** ще не створено, і пам'ять під нього не виділено.

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

У рядку коду на рис. 4.2 створюється об'єкт масиву (класу **System.Array**), що містить колекцію із п'яти величин типу **decimal**. Посилання на нього надано змінній **accountBalances**.

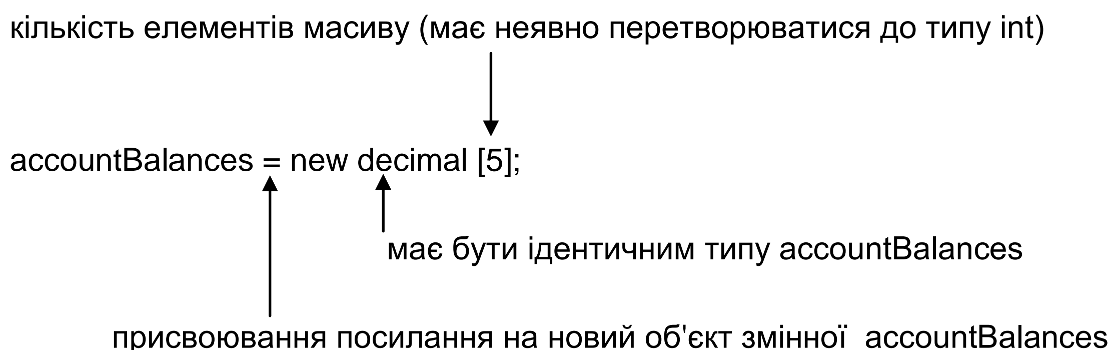


Рис. 4.2. **Створення об'єкта масиву із п'ятьома елементами**

Оператори оприлюднення й присвоювання можна об'єднати в одному рядку, як показано далі:

decimal [ ] accountBalances = new decimal [5];

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

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

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

де оприлюднено змінну **rainfall**, якій присвоєно посилання на новий об'єкт базового типу **decimal**.

Після того як оприлюднено змінну масиву **accountBalances** і їй присвоєно новий об'єкт масиву, маємо поточний стан, показаний на рис. 4.3.

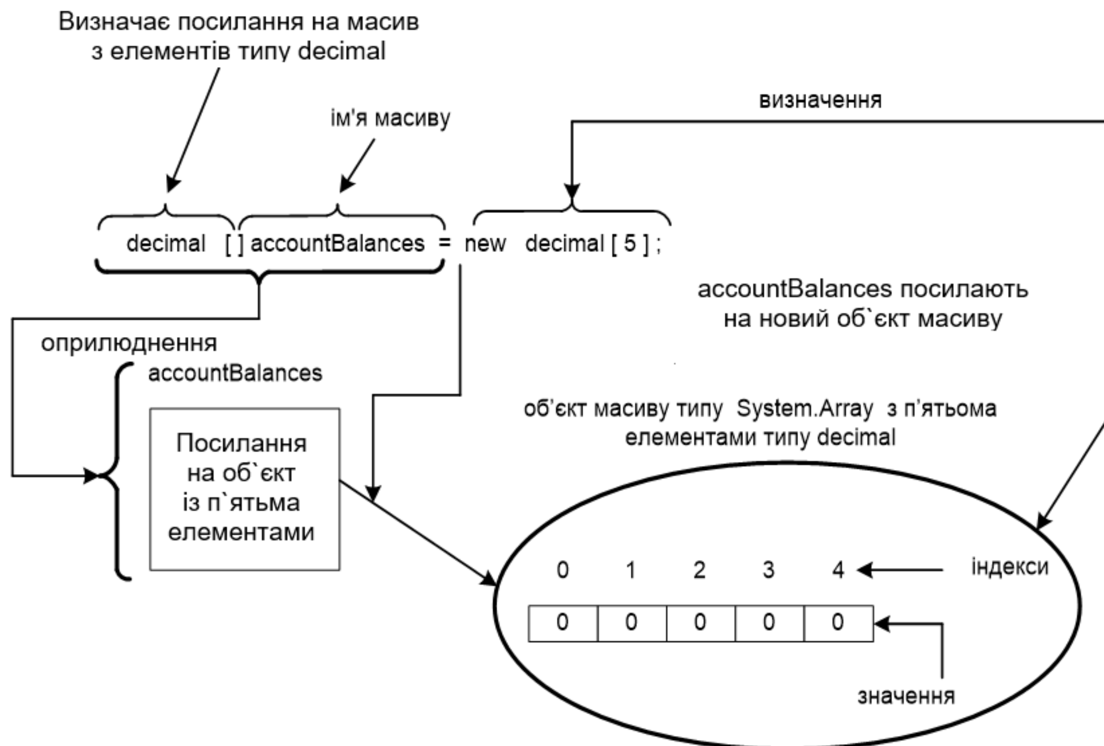


Рис. 4.3. **Посилання на об'єкт масиву**

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

Розмір масиву (або довжина масиву) – це загальна кількість елементів, які масив може містити.

Під час створено масиву можна визначати елементи й присвоювати їм початкові значення.

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

числовим величинам типів **short**, **int**, **float**, **decimal** тощо присвоюють значення **нуль**;

величинам типу **char** присвоюють символ **\u0000**;

величини типу **bool** ініціалізують значенням **false**;

величини посилальних типів ініціалізують значенням **null**.

У наведеному прикладі тип елемента масиву – **decimal**, тому всі величини ініціалізують нулем, як показано на рис. 4.3.

Якісна методика програмування передбачає явну ініціалізацію у програмі, а не на етапі компіляції.

Значення null. Коли посилання не вказує на конкретний об'єкт, воно має значення **null**. Воно є сумісним з усіма посилальними типами, подано 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 оприлюднення та створення масиву наведено в такому синтаксичному блоці.

Оприлюднення масиву та присвоювання йому значення

Оприлюднення_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 ::= <Оприлюднення _масиву>
new <Базовий_тип> [<Довжина_масиву>];

<Базовий_тип> в оприлюдненні має бути ідентичним **<Базовий_тип>** в операторі породження об'єкта.

<Довжина_масиву> має бути позитивною й належати типу, що неявно перетворюють на **int**. Це може бути літерал, константа або змінна.

Квадратні дужки **[ ]** в цьому разі є частиною синтаксису; вони не вказують на необов'язковість узятих у них елементів синтаксису.

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

Доступ до окремих елементів масиву. Доступ до індивідуальних елементів масиву здійснюють так само, як і до окремих символів змінної типу **string** – за ім'ям змінної з наступною парою квадратних дужок, що містять індекс. У наступному рядку коду: **accountBalances[0] = 1000m;** величину **1000** типу **decimal** присвоюють першому елементові масиву, на який посилається **accountBalances**. Індекс може бути будь-яким числовим виразом, що дає в результаті невід'ємну величину типу, який неявно перетворюють на **int**.

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

Рядки (типу **string**) є *незмінними*: будь-який символ у 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 виразі.

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

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

Приклад. Нарахування відсотків.

У вихідному коді, поданому далі, оприлюднено масив із п'яти балансів рахунків. Суми присвоєно двом першим елементам, потім по них нараховують десять відсотків і результат виводять на екран. Інші три елементи масиву **accountBalances[2]**, **accountBalances[3]** і **accountBalances[4]** у цій програмі не використовують.

```
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 оприлюднено змінну **accountBalances**, що містить посилання на масив з елементами типу **decimal**.

У рядку 10 створено об'єкт класу **System.Array**, виділено пам'ять під нього, а посилання на нього присвоєно змінній **accountBalances**.

У рядках 14 і 16 елементам масиву з індексами 0 і 1 присвоєно задані користувачем значення. Якщо користувач вводить суми 1000 і 2000, об'єкт масиву має такий вигляд, як на рис. 4.4, де подано синтаксис доступу до кожного елемента масиву.

У рядках 18 і 19 на два рахунки нараховано відсоток, визначений **const interestRate** у рядку 7.

Нові значення виведено в рядках 22 і 23.

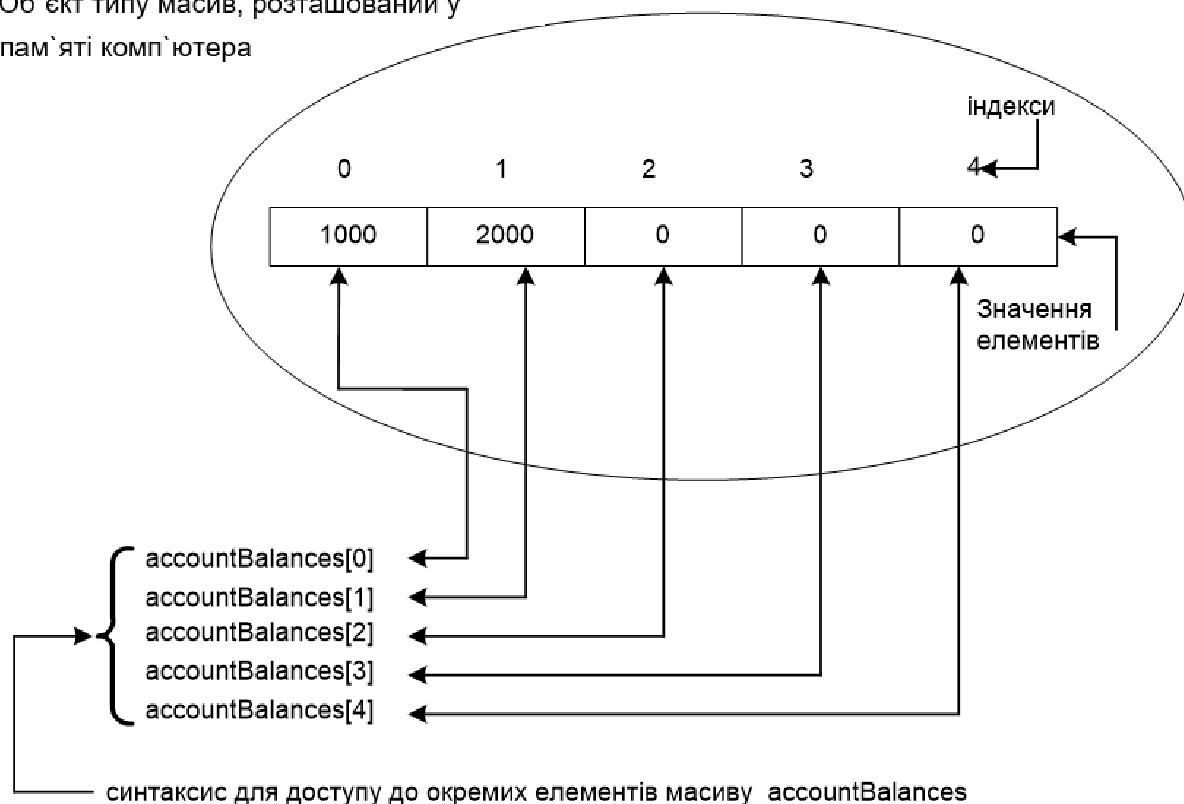


Рис. 4.4. **Два значення, що присвоюють accountBalances**

Синтаксис для доступу до окремих елементів масиву формалізовано в такому синтаксичному блоці.

Синтаксичний блок доступу до елементів масиву

Доступ_до_елемента_масиву ::=

<Ідентифікатор_масиву> [<Числовий_вираз>]

Квадратні дужки [ ] в цьому разі не вказують на необов'язковість узятих у них елементів синтаксису, фактично в них розміщено індекс у формі **<Числовий_вираз>.**

<Числовий_вираз> може бути будь-яким числовим виразом із типом, що неявно перетворюють на **int**. Тут можна використовувати вирази типу **sbyte**, **byte**, **short**, **ushort** і **char**. Будь-який інший тип або мусить мати визначений користувачем шлях явного перетворення, або мусить бути тим, що приводить до потрібного типу неявно.

Важливо розрізняти три випадки, у яких квадратні дужки використовують із масивами:

1) для оприлюднення змінної типу масив:

decimal [ ] accountBalances;

2) для зазначення довжини масиву під час створення об'єкта типу масив:

new decimal [5];

3) для доступу до індивідуальних елементів масиву:

accountBalances[0].

Приклад. Нарахування відсотків (ускладнений варіант).

Розгляньмо застосування масиву в поєднанні з оператором циклу. Функціональні властивості цієї програми є аналогічними програмі з попереднього прикладу, однак їхні реалізації розрізняють. Об'єднання у програмі циклу **for** із масивом **accountBalances** дозволяє користувачеві:

увести 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** посилають на об'єкт масиву, що містить п'ять величин типу **decimal** (рядки 9 і 11).

Цикл **for**, розташований у рядках 14 – 18, повторюють п'ять разів. Значення змінної **i** починається з 0 і збільшується на 1 за кожного повторення тіла циклу.

Виконання зупиняють, коли умова циклу дорівнює **false**.

Вираз **accountBalances.Length** є еквівалентним властивості **Length** класу **string** і є одним із багатьох корисних убудованих властивостей та методів об'єкта **System.Array**.

Властивість **Length** повертає довжину масиву, у цьому разі 5, і робить вираз **i < accountBalances.Length** таким, що дорівнює **false**, коли **i** дорівнює 5. Отже, коли тіло циклу виконано востаннє, **i** дорівнює 4.

Це добре погоджено з функціональністю, що реалізують у тілі циклу в рядках 16 і 17:

отримати нову суму від користувача і присвоїти її елементу масиву з індексом, що починається з 0 та збільшується на 1 для кожної нової суми.

Кожному з усіх п'яти елементів масиву користувач присвоює нову суму.

Цикл **for** у рядках 21 – 27 містить ті самі оператори ініціалізації, умови й оновлення циклу, як і попередній оператор **for**.

Нарахування відсотків здійснює оператор у рядках 23 і 24. Виведення нового балансу відбувається в рядках 25 і 26.

Слід зазначити, що ділянку видимості змінної **i**, оприлюдненої в рядку 14, обмежено рядками 14 – 18. Ця змінна відрізняється від **i**, оприлюдненої в рядку 21, яка має ділянку видимості в межах рядків 21 – 27.

У визначенні довжини масиву у програмі варто використовувати властивість **Length**, а не літерали або константи. Це дозволяє програмі самостійно контролювати зміну розміру масиву й потребує зміни коду лише у єдиному місці.

Ініціалізація масивів. Елементи масиву можна ініціалізувати будь-якими значеннями під час його створення [2; 7; 12; 13].

Спочатку потрібно оприлюднити змінну масиву. Але замість створення об'єкта із ключовим словом **new**, як у такому прикладі: **new int [6];** потрібно явно визначити список ініціалізованих величин, указуючи їх у дужках після оператора присвоювання (рис. 4.5).

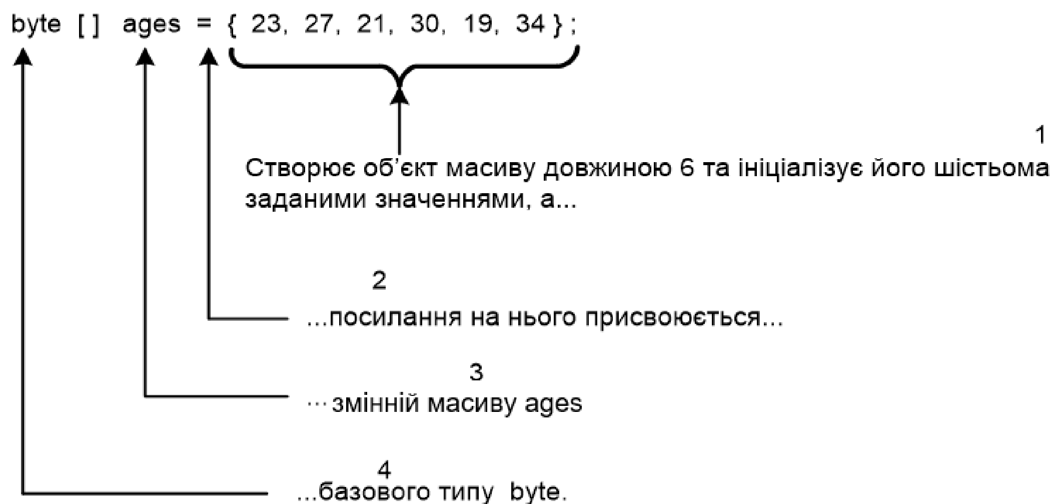


Рис. 4.5. **Приклад явної ініціалізації елементів масиву**

Об'єкт масиву автоматично створюється з кількістю елементів (у цьому прикладі 6), необхідних для зберігання значень у фігурних дужках. У цьому разі немає потреби визначати довжину масиву явно. У дужках задають значення тільки того типу, що може бути явно перетвореним на базовий тип, визначений в оприлюдненні масиву (у цьому випадку **byte**).

Довжину масиву можна визначити і явно (хоча за замовчуванням її задано кількістю елементів у фігурних дужках). Такий рядок є еквівалентним попередньому (див. рис. 4.5):

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:

довжину масиву видно відразу під час читання коду – це зручно, коли список значень у дужках є досить довгим;

компілятор порівнює довжину масиву, зазначену у квадратних дужках, із кількістю елементів у фігурних дужках. Будь-які невідповідності приводять до повідомлення про помилку (це дозволяє автоматично визначити пропущені або зайві значення).

Під час використання фігурних дужок для ініціалізації масиву необхідно задавати значення всіх елементів. Якщо деякі елементи потрібно пропустити, можна скористатися одним із таких прийомів:

задати ініціалізувальні значення в дужках, указавши водночас для деяких елементів значення за замовчуванням (наприклад, 0). Якщо, ска-

жімо, у прикладі, наведеному раніше, значення трьох останніх елементів є невідомими, можна використовувати такий оператор:

byte [ ] ages = { 23, 27, 21, 0, 0, 0};

Синтаксичний блок оприлюднення і явної ініціалізації масиву

Оприлюднення_і_явна_ініціалізація масиву ::=

<Базовий_тип> [ ] <Ідентифікатор_масиву> =
new <Базовий_тип> [<Довжина_масиву>]
{ <Значення1>, <Значення2>, ... <ЗначенняN> } ;

<Значення1>, **<Значення2>** тощо мають явно зводитися до **<Базовий_тип>**.

Кількість значень, що присвоюють масиву (позначена тут **N**), дорівнює довжині масиву.

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,** оприлюдненого та визначеного як

uint [ ] childbirths = {1340, 3240, 1003, 4987, 3877};

можна використовувати оператор **foreach**:

```
foreach ( uint  temp   in   childbirths )
{
        Console.WriteLine(temp);
}
```

який дає виведення:

1340 3240 1003 4987 3877

Розгляньмо докладно цей спосіб.

Оператор **foreach** складається із заголовка й тіла (рис. 4.6). Тіло циклу може бути одиночним або складеним оператором.

Перші два слова у круглих дужках заголовка є, відповідно, типом і ідентифікатором. Разом вони оприлюднюють ітераційну змінну оператора

foreach. У цьому разі її називають **temp** (від слова **temporary** – тимчасовий).

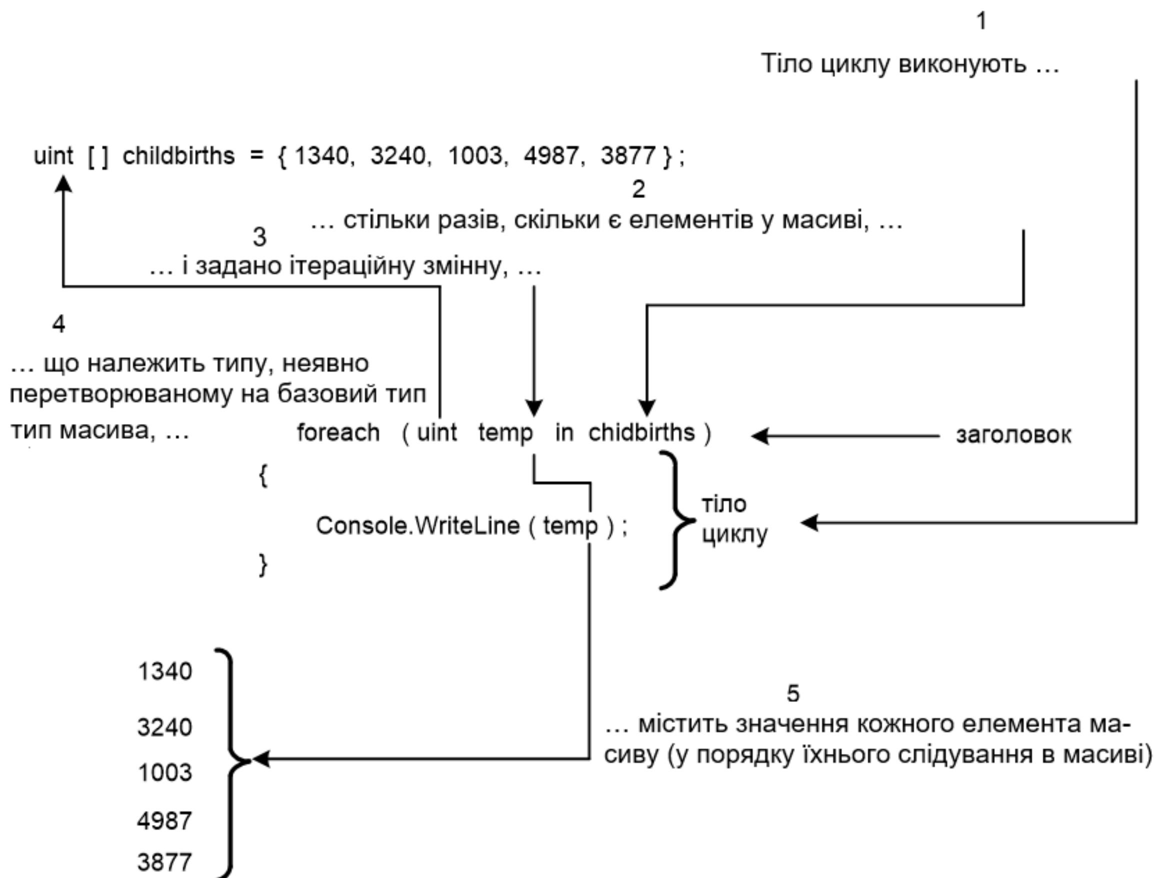


Рис. 4.6. **Оператор foreach**

Праворуч від ітераційної змінної міститься ключове слово **in**, за яким іде масив (у цьому разі **childbirths**). Важливо, щоб тип ітераційної змінної міг бути неявно перетвореним на базовий тип масиву.

Тіло циклу виконують один раз для кожного елемента. Ітераційну змінну можна використовувати й у тілі циклу. Під час першого проходу (виконання) вона дорівнює першому елементові масиву тощо.

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. Водночас кожний елемент масиву присвоюють ітераційній змінній без потреби явної індексації.

Лічильник, умова й оновлення циклу (потрібні у стандартному циклі **for**), у цьому разі є непотрібними, що забезпечує простоту та ясність коду.

4.2. Приклади опрацювання одновимірних масивів

*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;

            // Уведення масиву
            Console.WriteLine("Уведіть 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("За значення x={0}, y={1:N4}",mas[i], y);
            }
        }
    }
```

Результат роботи програми:

```
Уведіть 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
За значення x=-1, y=11,3415
За значення x=-0,93, y=9,8831
За значення x=-0,49, y=2,9917
За значення x=0, y=0,0000
За значення x=1,13, y=12,5030
За значення x=0,96, y=8,8576
За значення x=1,75, y=31,1723
Press any key to continue
```

Бульбашкове 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;

        // Уведення масиву
        Console.WriteLine("Уведіть 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
        }
}
```

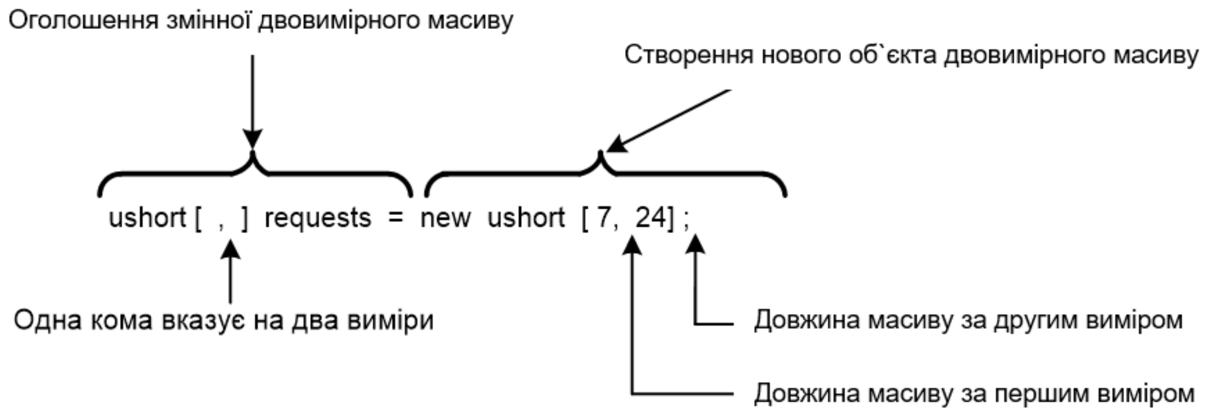
```
Результат роботи програми:
Уведіть значення елементів масиву
mas[0] = 1,3
mas[1] = -34,12
mas[2] = 5,23
mas[3] = 6
mas[4] = 0,23
mas[5] = -0,56
mas[6] = 65,3
Вихідний масив:
1,3  -34,12  5,23  6  0,23  -0,56  65,3
Масив після сортування:
65,3  6  5,23  1,3  0,23  -0,56  -34,12
Press any key to continue
```

4.3. Багатовимірні масиви

Оприлюднення та визначення двовимірного масиву. Мова C# дозволяє визначати двовимірні масиви, де для ідентифікації елемента потрібні два індекси. (Фактично в C# можна визначати масиви будь-якої розмірності).

На рис. 4.7 наведено приклад оприлюднення та визначення двовимірного масиву.

Оприлюднення змінної масиву, створення нового об'єкта й присвоювання змінній посилання на об'єкт (перший рядок на рис. 4.6) можна (як і в разі одновимірного масиву) поділити на два оператори. Результат показано в нижній частині рис. 4.7.



Попередній оператор, як в разі одновимірних масивів, можна розподілити на два рядки:

```
ushort [ , ] requests ;
requests = new ushort [ 7 , 24 ] ;
```

Рис. 4.7. **Приклад оприлюднення та визначення двовимірного масиву**

Індекс першого виміру змінюється від 0 до 6, а другого – від 0 до 23.

Доступ до елементів двовимірного масиву. Після оприлюднення змінної масиву і присвоювання їй посилання на двовимірний об'єкт можна звертатися до його окремих елементів.

За винятком того, що для звертання до елементів двовимірного масиву потрібен додатковий індекс, їх використовують аналогічно елементам одновимірного масиву. Отже, будь-який із них можна використовувати так само, як і окрему змінну базового типу.

Наприклад:

requests[0,0] = (ushort) 89;

Тут застосовано операцію зведення до типу (ushort), оскільки він є базовим типом requests. Воно потрібно, тому що літерал 89 належить до типу int, що не може бути неявно перетвореним на ushort.

Подання двовимірного масиву як масиву масивів. Два виміри масиву requests можна розглядати як масив масивів. Якщо звернутися до першого виміру requests (що подає, наприклад, дні тижня), сім днів можна вважати одновимірним масивом (рис. 4.8). Якщо тепер додати до розгляду години, то кожний елемент "день" можна вважати складеним з одновимірного масиву "години".

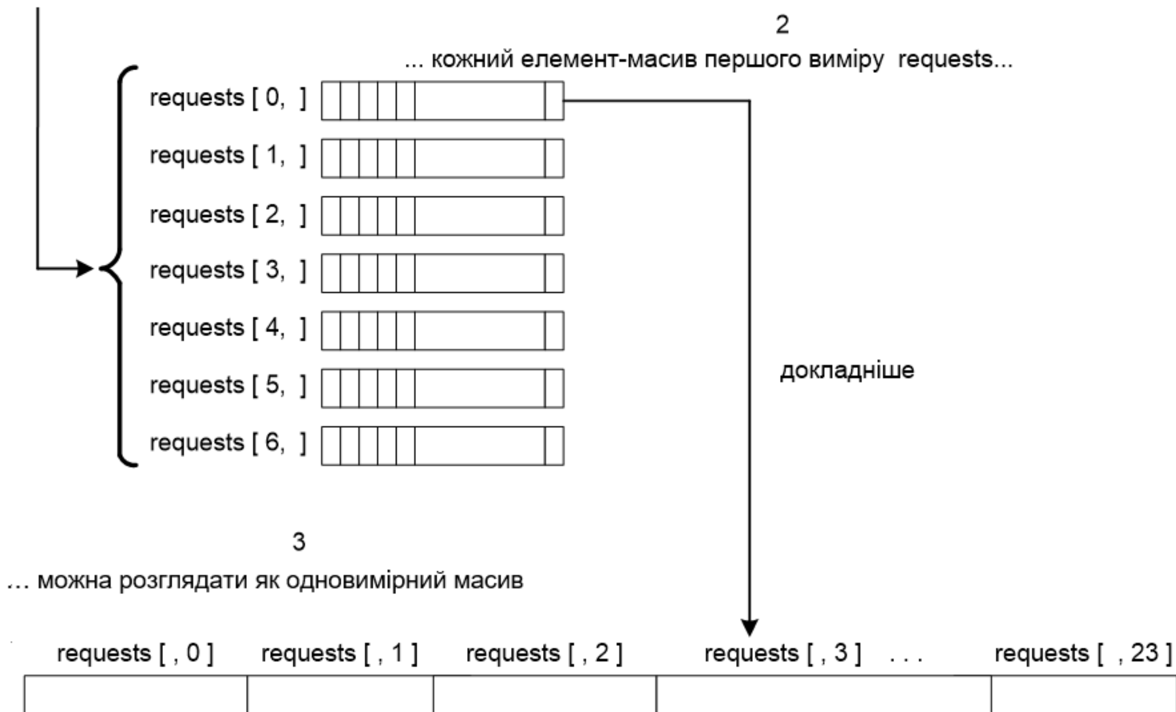


Рис. 4.8. **Подання двовимірного масиву як двох одновимірних масивів**

Приклад. Опрацювання матриці зарплати.

```
// Опрацювання матриці зарплати
using System;
class Class1
{
	static void Main()
	{
		int kol_rab;		// кількість робітників у бригаді
		int kol_brigad;	// кількість бригад
		char vibor;		// для організації діалогу
		do
		{
			//	Уведення матриці із клавіатури
			Console.WriteLine("Уведіть кількість робітників у бригаді...");
			kol_rab=Convert.ToInt32(Console.ReadLine());
```

```
                Console.WriteLine("Уведіть 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("Уведіть  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
                }
                // Опрацювання 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:

```
Уведіть кількість робітників у бригаді...
4
Уведіть кількість бригад...
2
Уведіть зарплату 1 робітника з 1 бригади
Matr[0,0]=?
345,7
Уведіть зарплату 1 робітника з 2 бригади
Matr[0,1]=?
654,1
Уведіть зарплату 2 робітника з 1 бригади
Matr[1,0]=?
123,9
Уведіть зарплату 2 робітника з 2 бригади
Matr[1,1]=?
765,1
```

Уведіть зарплату 3 робітника з 1 бригади
Matr[2,0]=?
342,1
Уведіть зарплату 3 робітника з 2 бригади
Matr[2,1]=?
765,3
Уведіть зарплату 4 робітника з 1 бригади
Matr[3,0]=?
943,3
Уведіть 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

Контрольні запитання

1. У чому полягають особливості призначення, оприлюднення та визначення масиву?
2. Як відтворюють 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 сутність алгоритму сортування елементів масиву методом "бульбашки"?

9. Як реалізують доступ до елементів двовимірного масиву?

10. У чому сутність подання двовимірного масиву як масиву масивів?

11. Наведіть приклади опрацювання матриць.

5. Структури

5.1. Загальні відомості про структури

Структури – це складені типи даних, побудовані з використанням інших типів. Вони становлять об'єднаний загальним ім'ям набір даних різних типів.

Окремі дані структури називають *елементами* або *полями*. Елементи однієї й тієї самої структури мусять мати унікальні імена, але дві різні структури можуть містити неконфліктні 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 вибране користувачем ім'я типу. Елементи, що входять до структури, розміщують у фігурних дужках, після яких ставлять крапку 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 можна використовувати для визначення змінних. У структурі обов'язково має бути вказано хоча б один компонент.

Припустімо, що потрібно створити тип для опису характеристики викладача університету. Цей тип має містити ім'я викладача, його кваліфікацію (добра, задовільна тощо), стаж роботи й поточну якість викладання (за 100-бальною оцінкою). Далі наведено опис структури, що задовольняє ці вимоги:

```
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. Отже, тепер можна створювати змінні типу Profesor так само, як змінні будь-якого базового типу, наприклад int або char.

Між фігурними дужками міститься список полів структури. Кожний елемент списку – це оператор визначення. Тут можна використовувати будь-який із типів даних C#, включно з масивами та іншими структурами. У цьому прикладі використовують 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 структуру оприлюднено, її можна використовувати. Для цього спочатку потрібно створити (визначити) екземпляр структури.

Екземпляр структури створюють за допомогою ключового слова new:

Profesor P_Econom_Inform = new Profesor( );

але, на відміну від класу, екземпляр структури можна створити й без new. Це має такий вигляд:

Profesor P_Econom_Inform;

Під час створення структури без ключового слова new її конструктори не викликають. Водночас значення всім її елементам слід присвоїти явно, звернувшись до них через ім'я структури, як показано далі:

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 викликаним, поки не будуть ініціалізованими елементи-дані. Тому останні потрібно оприлюднювати як public.

5.2. Приклади елементарного опрацювання структур

Приклад. Оприлюднення, 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; // Оприлюднення екземпляра струк-
тури
                //       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. Оприлюднення, визначення (із **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
```

5.3. Масиви структур

Приклад. Опрацювання 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()
        {
                //      Уведення вихідних даних
                Console.WriteLine("Уведіть кількість рядків у 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:

Уведіть кількість рядків у документі

3

Автор книги?

Рубіна

Вартість книги?

34,45

Кількість книг?

3

Автор книги?

Улицька

Вартість книги?

45,78

Кількість книг?

10

Автор книги?

Шевченко

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      Шевченко            75,23            3                225,69 |
|---------------------------------------------------------------------------|
| Разом:                        155,46          16              786,84 |
|---------------------------------------------------------------------------|
Press any key to continue
```

Контрольні запитання

1. Коли доцільно використовувати структури?

2. Як реалізують призначення, оприлюднення та визначення структур?

3. Наведіть приклади оприлюднення, визначення (без операції **new** та з операцією **new**), ініціалізації та виведення на екран елементів заданої структури.

4. Які особливості опрацювання елементів структур?

6. Функції

6.1. Загальні відомості про функції

Досить часто виникають ситуації, коли виконання певних завдань – наприклад, пошук максимального елемента масиву – потрібно здійснювати в різних місцях програми.

Розв'язанням такого роду проблем є застосування *функцій*.

Функції в C# – це засіб, що дозволяє виконувати деякі ділянки коду в довільному місці застосунку.

Функції володіють тією перевагою, що вони дозволяють робити програму більш зручночитаною, і ми маємо можливість групувати разом логічно пов'язані між собою частини коду. Діючи таким способом, можна зробити тіло самого застосунку невеликим, оскільки вирішення внутрішніх завдань застосунку будуть здійснювати окремо.

Функції можна також використовувати для створення *багатоцільових* програм, які виконують ті самі операції над різними даними.

Маємо можливість передавати функціям інформацію, із якою вони будуть працювати, у вигляді параметрів і здобувати результати роботи функції у вигляді *значень, що повертаються.* Наприклад, можна передати функції як параметр масив, у якому здійснюють пошук, і визначити елемент масиву з максимальним значенням як значенням, що повертається. Звідси випливає, що можна щораз використовувати ту саму функцію для роботи з різними масивами.

Параметри функції та значення, що повертається, разом називають *сигнатурою* функції.

У межах цієї теми розгляньмо способи опису та використання простих функцій, які не визначають і не повертають ніяких значень. Після цього перейдімо до способів передавання та здобування інформації від функцій.

Потім ми звернімося до ділянок дії змінних. Яким чином ці застосунки на C# локалізують у різних ділянках коду – це питання є особливо важливим, коли програму розподілено на множину окремих функцій.

Нарешті зосередьмося 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 передбачає непряме використання функції. Той самий делегат може бути використаним для виклику будь-якої функції, що має зазначену сигнатуру, а це дозволяє здійснювати вибір із декількох різних функцій у процесі виконання.

Опис і використання функцій. У цьому пункті зазначено, яким чином можна додавати функції до складу застосунків, а потім використовувати (викликати) їх із коду.

Розгляньмо такий 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 цій функції, виводить деякий текст у консольному вікні.

Опис функції складається з:

двох ключових слів: static і void;

ім'я функції, за яким розташовують параметри Write();

ділянки виконуваного коду, розміщеного у фігурних дужках.

Код, що використовують для опису функції Write(), має такий самий вигляд, як і код самого застосунку:

```
static  void  Main(string[ ]  args)
{
. . .
}
```

Це пояснено тим, що весь код, що створювали дотепер (не враховуючи опису типів), становить частину деякої функції.

Функція Main( ) забезпечує, як згадували в попередніх підрозділах, точку входу в консольний застосунок. Коли запускають застосунок, написаний C#, то відбувається виклик функції точки входу, що міститься в ньому, а коли ця функція закінчує свою роботу, виконання застосунку переривається. Будь-який виконуваний код, написаний C#, мусить мати точку входу.

Єдина відмінність між функцією Main( ) і функцією Write( ), не враховуючи тих рядків коду, які в них є, полягає в тому, що у круглих дужках, розташованих за ім'ям функції Main, міститься деякий код. Цей код слугує для задавання параметрів.

Як уже згадували раніше, обидві функції – і Main( ), і Write( ) – описують із використанням ключових слів static (статичний) і void (відсутній).

Ключове слово static стосується понять об'єктно орієнтованого програмування (до його розгляду перейдемо далі). На цьому етапі потрібно запам'ятати тільки те, що всі функції, які будуть задіяні в застосунках цього підрозділу, обов'язково мають використовувати це ключове слово.

Ключове слово void указує, що функція не повертає ніякого значення. Далі буде розказано, як потрібно писати в тих випадках, коли у функції є значення, котре повертається.

Продовжуючи розглядати цей застосунок, визначаймо код, що здійснює виклик функції Write( ).

Він складається з ім'я функції, за яким містяться порожні круглі дужки. Коли виконання програми досягне цієї точки, почнуть виконувати код, який міститься у функції Write( ).

Зверніть увагу, що використання круглих дужок, як під час опису функції, так і під час її виклику, є обов'язковим.

Значення, що повертаються. Найпростіший спосіб обміну даними з функціями – використання значення, що повертається. Функції, у яких застосовують значення, що повертаються, точно так само володіють чисельним значенням, як і будь-які змінні, використовувані під час обчислення виразів. 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 виразі: double myVal;

```
double multiplier = 5.3;
myVal = getVal( ) * multiplier;
```

Якщо функція мусить мати значення, що повертається, то потрібно внести дві зміни:

в описі функції замість ключового слова void указати тип значення, що повертається;

після завершення всіх обчислень у функції використовувати ключове слово return і передавати значення, що повертається, коду, що викликає.

Синтаксис коду для розглянутого типу функцій консольного застосунку буде мати такий вигляд:

```
static   <Тип, що повертається>   <ім'яФункції>(  )
{
. . .
        return
}
```

Єдиним обмеженням у цьому разі є вимога щодо **<Значення, що повертається>**, яке мусить мати тип **<Тип, що повертається>**, або ж має бути можливість його неявного перетворення на цей тип.

Тип **<Тип, що повертається>** може бути будь-яким, включно з найбільш складними типами серед розглянутих раніше.

Значення, що повертаються, звичайно є результатом деяких обчислень. Коли під час виконання функції досягнуто оператор return, управління негайно передають назад у викликаний код. Ніякі рядки коду після цього оператора виконувати не будуть. Звідси, однак, зовсім не випливає, що в тілі функції оператор return обов'язково має бути останнім. Він може бути використаним раніше, наприклад, за розгалуження за якою-небудь умови.

Додавання оператора return до циклу for, блоку if або якої-небудь іншої структури призведе до негайного закінчення виконання як цієї структури, так і всієї функції загалом. Наприклад:

```
static double getVal( )
{
double checkVal;
// присвоювання змінній checkVal деякого значення,
// отриманого в результаті деяких обчислень.
if (checkVal < 5)
return 4.7;
return 3.2;
}
```

У цьому разі буде повернуто одне із двох значень, залежно від значення змінної checkVal.

Є єдине обмеження: оператор return мають виконувати до того як буде досягнуто фігурну дужку }, що закриває цю функцію. Такий код не є припустимим:

```
static double getVal( )
{
double checkVal;
// присвоювання змінній checkVal значення,
// визначеного в результаті деяких обчислень.
if (checkVal < 5)
return 4.7;
}
```

Якщо checkval >= 5, то не буде жодного оператора return, а це заборонено. Усі гілки мають закінчувати цим оператором.

Оператор return також можна застосовувати у функціях, оприлюднених із використанням ключового слова void (у них відсутнє яке-небудь значення, що повертається). У таких випадках функція просто припиняє роботу. Тому під час використання оператора return буде помилкою розміщати значення, що повертається, між ключовим словом return і наступною за ним крапкою з комою.

Параметри. Якщо функція мусить мати параметри, то потрібно задати:

список приймaних функцією параметрів у її описі, а також типи цих параметрів;

збіжний список параметрів за кожного виклику функції.

Це передбачає використання такого коду:

```
static <Тип, що повертається> <ім'яФункції> {<типПараметра> <ім'яПараметра>, . . . )
{
return   <Значення, що повертається>;
}
```

Тут може бути довільна кількість параметрів, для кожного з яких указують тип та ім'я. Як роздільник між параметрами ставлять коми.

Кожний із параметрів є доступним усередині цієї функції як змінна.

Наприклад, така функція приймає два параметри типу double і повертає їхній добуток:

```
static double product (double  param1, double  param2)
{
return param1 * param2;
}
```

Розгляньмо більш складний приклад, у якому визначають максимальний елемент заданого у 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 такий 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 нескладне завдання. Наведімо словесний опис алгоритму пошуку.

Локальній цілій змінній з ім'ям maxVal як початкове значення присвоюють перший елемент масиву, а потім здійснюють порівняння цього значення послідовно з усіма іншими елементами.

Якщо поточний елемент більше, ніж значення змінної maxVal, то поточне значення maxVal замінюють на це значення. Коли виконання циклу завершено, змінна maxVal містить найбільше значення цього масиву, що й повертається оператором return.

Код, розташований у Main (), оприлюднює та ініціалізує простий цілий масив, що будуть використовувати разом із функцією MaxValue( ):

```
int[ ] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
```

Під час виклику функції функцією MaxValue( ) значення присвоюють змінній maxVal типу int:

```
int maxVal = MaxValue(myArray);
```

Потім це значення виводять на екран за допомогою оператора

```
Console.WriteLine("The maximum value in myArray is {0}", maxVal);
```

Відповідність параметрів. Під час виклику функції її параметри мають точно відповідати її опису. Потрібно мати збіг типів параметрів, їхньої кількості та порядку їхнього слідування. Це означає, що, наприклад, така функція:

```
static void myFunction (string myString, double myDouble)
{
. . .
}
```

не може бути викликаною з використанням рядка: myFunction(2.6, " Hello ");

У цьому разі намагаймося передати значення типу double як перший параметр, а значення типу string – як другий, що не відповідає порядку, у 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 під час компіляції, тому що компілятор накладає вимогу точної відповідності сигнатурам викликуваних функцій.

Повертаючись до попереднього прикладу, бачимо, що така вимога означає: функцію MaxValue( ) можна використовувати тільки для визначення максимального цілого з масиву цілих чисел. Якщо змінимо код у Main( ) таким 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 неправильний тип параметра.

Далі – у пункті, присвяченому перевантаженню операторів, – буде розглянуто дуже корисний спосіб, що дозволяє розв'язати цю проблему.

Масиви параметрів. У C# передбачено можливість задавання одного (і тільки одного) спеціального параметра функції. Цей параметр, що обов'язково мають розташовувати останнім, відомий за назвою *масиву параметрів*. Він дозволяє під час звертання до функцій використовувати змінну кількість параметрів.

Масив параметрів описують за допомогою ключового слова params. Він слугує одним зі способів спрощення коду, оскільки, завдяки їм, не доводиться передавати масиви з викликаного коду. Навпаки, можна передати кілька параметрів того самого типу, які розміщують у масив для подальшого використання всередині функції.

Під час опису функції з масивом параметрів застосовують такий код:

```
static <Тип, що повертається> <ім'яФункції> ( <params>
                                        <тип>[ ] <ім'я> )
{
return <Значення, що повертається>;
}
```

Для того щоб викликати цю функцію, потрібен такий код:

```
<ім'яФункції> ( <значення1>, <значення2>, . . .) ;
```

У цьому разі **<значення1>**, **<значення2>** тощо – це значення типу **<тип>**.

Їх використовують для ініціалізації масиву з ім'ям **<ім'я>**. Ніяких обмежень на кількість параметрів, які може бути тут задано, немає.

Єдина пропонована до них вимога – вони всі мають 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 функцію sumVals( ) описано 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 у циклі всі значення масиву vals і підсумовує їх, повертаючи визначений результат.

Викликаймо цю функцію з Main( ) із п'ятьома цілими параметрами: int sum = sumVals (1, 5, 2, 9, 8);

Однак можна з тим самим успіхом викликати цю функцію, передаючи їй нуль, один, два або сто параметрів цілого типу: ніяких обмежень на кількість параметрів, що задають, не має.

6.2. Обмін інформацією з функцією

Передавання параметрів за посиланням і за значенням. В усіх попередніх функціях застосовували параметри, передані за *значенням*. Інакше кажучи, передавали значення у змінну, яку потім використовували всередині функції. Будь-які зміни, яких ця змінна зазнавала всередині функції, *не робили ніякого впливу* на параметр, використаний під час виклику функції. Як приклад розгляньмо 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 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 робить ніякого впливу на змінну myNumber, описану в Main( ), незважаючи на те що параметр, якому присвоюють її значення, подвоюється.

Але часто потрібно, щоб значення змінної myNumber було змінено, бо виникне проблема. Зазвичай 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 у 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, яку використовували під час її виклику, а не зі змінною, що має те саме значення. Отже, будь-які зміни змінної, що використовувалася функцією, відобразяться на значенні змінної, що використовували як параметр. Для цього під час опису параметра необхідно скористатися ключовим словом ref:

```
static void showDouble(ref int val)
{
val *= 2;
Console.WriteLine("val doubled = {0}", val);
}
```

Удруге ключове слово ref потрібно використовувати під час 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 разі на консоль буде виведено такий текст:

```
myNumber = 5
```

```
val doubled = 10
myNumber =10
```

Для змінних, використовуваних як параметри типу ref, є два обмеження.

По-перше, оскільки є ймовірність, що в результаті виклику функції значення викликуваного за посиланням параметра буде змінено, то під час виклику функції *заборонено використовувати змінні типу const.*

Звідси випливає, що такий виклик є неприпустимим:

```
const int myNumber = 5;
Console.WriteLine("mvNumber = {0}", myNumber);
showDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

По-друге, потрібно використовувати заздалегідь ініціалізовану змінну.

C# не гарантує, що параметр типу ref буде ініціалізованим тією функцією, у якій його використовують. Такий код також є неприпустимим:

```
int myNumber;
showDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

Вихідні параметри. На додаток до можливості передавати параметри за посиланням можемо вказати, що цей параметр є вихідним. Для цього у його опис додано ключове слово out, використовуване так само, як і ключове слово ref (як модифікатор параметра в описі функції й під час виклику функції).

Фактично цей спосіб надає майже такі самі можливості, що й передавання параметрів за посиланням, у тому сенсі, що значення параметра після виконання функції потрапляє у змінну, яку використовували під час виклику цієї функції.

Є, однак, і істотні відмінності.

Хоча використовувати змінну, якій не присвоєне початкове значення, як параметр типу ref неприпустимо, її можна застосовувати як параметр типу out.

Параметр типу out будуть розглядати функцією, у якій його використовують як такий, що не має початкового значення. Це означає, що хоча передавання змінної, якій присвоєно деяке значення як параметр типу out є припустимим, однак у процесі виконання функції значення, що зберігають у цій змінній, буде втрачено.

Як приклад розгляньмо розширення функції maxValue( ), що повертає елемент масиву з максимальним значенням. Модифікуймо цю функцію так, щоб визначити індекс елемента масиву, який містить найбільше значення (у разі, коли максимальне значення міститься в декількох елементах, будемо мати індекс першого максимального елемента). Для цього додаймо вихідний параметр:

```
static int MaxValue(int[ ] intArray, out int maxIndex)
{
int maxVal = intArray[0];
maxIndex = 0;
for (int i = 1; i < int Array.Length; i++)
{
if (intArray[i] > maxVal)
{
maxVal = intArray[i];
maxIndex = i;
}
}
return maxVal;
}
```

Цю функцію може бути використано в такий спосіб:

```
int[ ] myArray = {I, 8, 3, 6, 2, 5, 9, 3, 0, 2};
int maxIndex;
Console.WriteLine ("The maximum value in myArray is {0}",
maxValue(myArray, out maxIndex));
Console.WriteLine("The first occurrence of this value is at element {0}",
maxIndex + 1);
```

Результатом її виконання буде таке:

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 визначено в елементі 7)

Важливим моментом, на який варто звернути увагу, є потреба використовувати ключове слово out під час виклику функції (так само, як і ключове слово ref).

Ділянка дії змінних. Навіщо потрібно обмінюватися даними з функціями? Відповідь полягає в тому, що в C# доступ до змінних можна здійснювати тільки з певних ділянок коду. Прийнято говорити, що змінна має певну *ділянку дії*, у межах якої вона є доступною.

Повернімося до розгляду програми з пункту 6.1 (простір імен namespace Function_1).

Ця програма додатково виконує такі дії:

у функції Main ( ) описують та ініціалізують змінну з ім'ям myString;

функція Main( ) передає управління функції Write( );

у функції Write( ) описують та ініціалізують змінну з ім'ям myString, відмінну від змінної з ім'ям myString, описаної у функції Main( );

функція Write ( ) виводить на консоль рядок, що містить те значення змінної mystring, що їй присвоєно у функції Write ( );

функція Write( ) повертає управління функції Main( );

функція Main ( ) виводить на консоль рядок, що містить те значення змінної myString, що їй присвоєно у функції Main ( ).

Змінні, ділянку дії яких поширено тільки на одну функцію, називають *локальними*.

Є можливість опису глобальних змінних, чия ділянка дії охоплює відразу кілька функцій.

У розглянутій програмі цю змінну описано в такий спосіб:

static string myString;

Зверніть увагу, що нам знову треба було використовувати ключове слово static. Не будемо вдаватися в деталі: на цьому етапі досить знати, що для консольних застосунків цього типу в описі глобальних змінних *обов'язкове використання* або ключового слова static, або ключового

слова const. Якщо 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 із тим самим ім'ям у функціях Main( ) і Write( ), нам потрібно класифікувати ім'я глобальної змінної, використовуючи повністю кваліфіковане ім'я.

У цьому разі звертаймося до глобальної змінної за ім'ям class1.myString. Зверніть увагу, що це потрібно робити тільки в тому разі, якщо є глобальна й локальна змінні з однаковим ім'ям. Якби локальної змінної з ім'ям myString не було, то ми зовсім вільно могли б використовувати для звертання до глобальної змінної ім'я myString замість Class1.myString.

У тому разі, коли використовують локальну змінну, ім'я якої збігається з ім'ям глобальної змінної, говорять, що глобальна змінна є *схованою*.

Значення глобальної змінної задають у функції Main ( ):

```
Class1.myString = "Global string";
```

і використовують у 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 тому, що вони, за великим рахунком, виявляються непридатними для функцій загального призначення, здатних працювати з будь-якими переданими їм даними та не обмежуються даними, заданими конкретною глобальною змінною.

6.3. Функції та структури

Тип структур призначено для зберігання в одному місці декількох елементів інформації. Насправді можливостей у структур набагато більше, й однією з таких можливостей є здатність містити не тільки дані, але й функції.

Як простий приклад розгляньмо таку структуру:

```
struct customerName
{
public string firstName, lastName;
}
```

Якщо в нас є змінні типу customerName і нам потрібно вивести на консоль повне ім'я, будемо змушені конструювати це ім'я з його складових частин. Наприклад, для змінної типу customerName з ім'ям customer можна використовувати такий синтаксис:

```
customerName  myCustomer;
myCustomer.firstName = "Ivan";
myCustomer.lastName = "Jana";
Console.WriteLine("{0} {1}", customer.firstName, customer.lastName);
```

Маючи можливість додавати у структури функції, можемо спростити цю процедуру шляхом централізованого виконання типових завдань.

У цьому разі це можна зробити в такий спосіб:

```
struct customerName
{
public string firstName, lastName;
public string Name ( )
{
return firstName + "  " + lastName;
}
}
```

Ця функція є дуже схожою на інші функції, розглянуті раніше, за винятком того, що в ній немає модифікатора static. Причини цієї відсутності стануть зрозумілими пізніше, а поки досить просто запам'ятати, що це ключове слово для функцій структур не потрібне.

Можна скористатися цією функцією в такий спосіб:

```
customerName myCustomer;
myCustomer.firstName = "Ivan";
myCustomer.lastName = "Jana";
Console.WriteLine(customer.Name());
```

}

Цей синтаксис набагато простіший і зрозуміліший, ніж попередній.

Потрібно зазначити, що функція Name( ) має безпосередній доступ до полів структури firstName і lastName. У межах структури customerName їх можна вважати глобальними.

6.4. Перевантаження функцій

Оператор перевантаження надає можливість створювати одночасно декілька функцій, що мають однакові імена, але працюють із параметрами різних типів.

Наприклад, раніше використовували таку програму, у 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);
    }

}
```

Цю функцію можна використовувати тільки для масивів значень типу int.

Можна було б створити функції з іншими іменами, призначені для роботи з параметрами інших типів, перейменувавши наведену раніше функцію як-небудь на кшталт IntArrayMaxValue ( ) і додавши функції на зразок DoubleArrayMaxValue ( ) для роботи з іншими типами.

Як альтернативу можна просто додати в цю програму таку 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 тому, що ця функція працює зі значеннями типу double. Ім'я функції – MaxValue( ) – буде тим самим, однак сигнатура (це принципово) відрізняється. Було б помилкою описати дві функції з однаковим ім'ям і однаковою сигнатурою, однак оскільки в цьому разі сигнатури є різними, то все нормально.

Тепер є дві версії функції MaxValue( ), які приймають масиви типу int і масиви типу double та повертають максимальне значення типу int або типу double, відповідно. Отже, не потрібно явно вказувати, яку із цих двох функцій збираємося використовувати. Просто задаємо масив-параметр, і це приводить до виконання того варіанта, що відповідає типу використовуваного параметра.

На цьому етапі варто зазначити ще одну рису Visual Studio. Якщо в застосунку є дві однойменні функції, описані раніше, і набрати це ім'я, наприклад у Main( ), то Visual Studio виведе у формі довідки доступні варіанти перевантаження цієї функції.

Під час перевантаження функцій ураховують усі аспекти, що стосуються їхніх сигнатур.

Є 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 здійснюють винятково на підставі того, чи є у зверненні до функції ключове слово ref. За наступного виклику буде використано варіант, у якому параметр передано за посиланням:

```
showDouble(ref val);
```

А такий виклик дозволить передати параметр за значенням:

```
showDouble(val);
```

Аналогічним чином можна описувати функції, що відрізняються кількістю параметрів, яких вони потребують, і т. ін.

6.5. Делегати

Делегатом називають тип, що дозволяє зберігати посилання на функції.

Основне призначення делегатів навряд чи вдасться зрозуміти доти, доки не ознайомитеся з подіями та з їхнім опрацюванням, однак розгляд самого поняття делегатів дасть величезну користь. Коли пізніше прийде час їх використовувати, вони будуть вам уже знайомими, що зробить деякі складні теми набагато більш легкими для розуміння.

Оприлюднення делегатів багато в чому нагадує оприлюднення функцій; водночас відсутнє само тіло функції, але додано ключове слово delegate.

Оприлюднення делегата визначає сигнатуру функції, яка складається з типу, що повертається, і списку параметрів. Після оприлюднення делегата маємо можливість оприлюднити змінну типу цього делегата й потім ініціалізувати цю змінну, надавши їй посилання на довільну функцію, що володіє сигнатурою, яка збігається із сигнатурою делегата.

У результаті маємо можливість викликати цю саму функцію за допомогою такої змінної-делегата так, ніби остання сама була цією функцією.

Тепер, коли є змінна, що посилається на функцію, також маємо можливість виконувати й деякі інші операції, які не можуть бути виконаними

з використанням інших засобів. Наприклад, виникає можливість передавати змінну-делегат іншої функції як параметр, що дозволить цій функції використовувати цього делегата для виклику функції, на яку він посилається, без потреби 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 таке визначення:

```
delegate double processDelegate(double param1, double param2);
```

Ключове слово delegate указує, що це визначення є визначенням делегата, а не функції (це визначення розташовано в тому самому місці програми, де могло б перебувати й визначення функції).

Далі наведено сигнатуру, у якій задають значення типу double, що повертається, і два параметри типу double. Використовувані в сигнатурі імена є довільними, тому тип делегата та імена параметрів можна вибирати на свій розсуд.

У цьому разі вибрали ім'я делегата processDelegate, а параметри назвали param1 і param2.

Код функції Main( ) починається з того, що оприлюднюють 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 до програми не додано ніяких перевірок допустимості інформації, що вводить користувач. У реальній програмі нам довелося б витратити велику кількість часу, перевіряючи допустимість тих значень, які маємо для локальних змінних param1 і param2.

Потім запитуймо 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, ініціалізуймо змінну-делегата:

```
if (input == "M")
    process = new processDelegate(Multiply);
else
    process = new processDelegate(Divide);
```

Для присвоєння змінній-делегату посилання на функцію доводиться використовувати синтаксис, що має дивний вигляд. Так само, як і в разі

присвоювання значень масиву, для створення нового делегата використовують ключове слово new. Після цього слова вказують тип створюваного делегата й задають параметр, що визначає ту функцію, яку 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 виключно для здійснення присвоювання делегатові. Як параметр використовують просто ім'я тієї функції, яку потрібно використовувати, без дужок.

Нарешті, викликаймо вибрану функцію за допомогою делегата. Тут застосовують той самий синтаксис, незалежно від того, на яку функцію посилається делегат:

```
Console.WriteLine("Result: {0}", process(param1, param2));
```

У цьому разі поводжуймося зі змінною-делегатом так, якби вона становила ім'я функції. Однак, на відміну від функцій, над цією змінною можна виконувати деякі додаткові операції, наприклад, передавання її іншій функції як параметр. Простий приклад такого використання може мати приблизно такий вигляд:

```
static void executeFunction(processDelegate process)
{
process(2,2 3,3);
}
```

Це означає, що маємо можливість управляти поводженням функцій, передаючи їм делегати функцій (як параметри).

Наприклад, може бути функція, використовувана для сортування рядкового масиву за алфавітом. Є різні алгоритми сортування списків; вони мають різну швидкість, котра залежить від характеристик списку, який сортують. За допомогою делегатів маємо можливість визначити метод, який потрібно використовувати, передаючи функції, що виконує сортування, делегата тієї функції, у якій реалізовано відповідний алгоритм сортування.

Варіантів застосування делегатів є дуже багато, але, як зазначали раніше, найбільш активно їх використовують під час опрацювання подій.

Приклад використання функцій. Приклад опрацювання одновимірного масиву, що наведено далі, не має потреби в коментарях. Функціональність програми є очевидною з назви відповідних змінних і функцій.

Приклад. Опрацювання одновимірного 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

Підіб'ємо підсумки тем, які обговорювали в цьому підрозділі:

Визначення та використання функцій у консольних застосунках.

Обмін даними з функціями за допомогою параметрів і значень, що повертаються.

Масиви параметрів.

Передавання параметрів за посиланням і за значенням.

Використання вихідних параметрів для додаткових значень, що повертаються.

Поняття ділянки дії змінної.

Використання функцій у типах структур.

Перевантаження функцій.

Делегати.

Контрольні запитання

1. Наведіть призначення функції. Чому функція може бути прикладом коду, який повторно виконують?

2. Перелічіть основні етапи виконання функції.

3. Який синтаксис запису значень, що повертаються з функції?

4. Охарактеризуйте параметри функцій. У чому полягає відповідність параметрів?

5. Як опрацьовувати масиви параметрів?

6. У чому полягають основні ідеї реалізації процесу обміну інформацією з функцією?

7. Що таке "ділянка дії" змінних? Охарактеризуйте параметри та значення, що повертаються, за порівнянням із глобальними даними.

8. У чому полягають особливості передавання параметрів за посиланням і за значенням?

9. Які особливості використання функції та структури?

10. Які особливості використання функції та масиву?

11. Що таке "перевантаження функцій"? Коли його доцільно використовувати?

12. Що розуміють під делегатом (посиланням на функції)?

Використана та рекомендована література

Основна

1. Albahari J. C# 10 Pocket Reference: Instant Help for C# 10 Programmers / J. Albahari, B. Albahari. – California, USA : O'REILLY, 2022. – 270 p.

2. C# 11 and .NET 7 – Modern Cross-Platform Development Fundamentals: Start building websites and services with ASP.NET Core 7, Blazor, and EF Core 7. – 7th Edition. – Birmingham : Packt Publishing, 2022. – 818 p.

3. Griffiths I. Programming C# 8.0: Build Cloud, Web, and Desktop Applications / I. Griffiths. – California, USA : O'REILLY, 2020. – 800 p.

Додаткова

4. Baptista G. Software Architecture with C# 9 and .NET 5 / G. Baptista, F. Abbruzzese. – Second Edition. – California, USA : O'REILLY, 2020. – 700 p.

5. Vaskaran S. Simple and Efficient Programming with C#: Skills to Build Applications with Visual Studio and .NET. / S. Vaskaran. – New York : Apress, 2023. – 300 p.

Інформаційні ресурси

6. Вступ до C# [Електронний ресурс]. – Режим доступу : https://programm.top/uk/c-sharp/tutorial/introduction/.

7. Сайт персональних навчальних систем: Програмування засобів мультимедіа (2 курс, бакалавр, сем. 1, спец. 186) [Електронний ресурс]. – Режим доступу : https://pns.hneu.edu.ua/course/view.php?id=3896.

8. C# documentation [Електронний ресурс]. – Режим доступу : https://learn.microsoft.com/en-us/dotnet/csharp/.

9. C# Sharp Programming Exercises, Practice, Solution [Електронний ресурс]. – Режим доступу : https://www.w3resource.com/csharp-exercises/.

10. C# Tutorial [Електронний ресурс]. – Режим доступу : https://www.w3schools.com/cs/index.php.

11. GitHub Copilot and Visual Studio 2022 [Електронний ресурс]. – Режим доступу : https://visualstudio.microsoft.com/.

12. Learn C# Programming [Електронний ресурс]. – Режим доступу : https://www.programiz.com/csharp-programming.

13. Learn C# Programming: C# Fundamentals [Електронний ресурс]. – Режим доступу : https://www.tutorialsteacher.com/csharp.

14. OneCompiler's [Електронний ресурс]. – Режим доступу : https://onecompiler.com/csharp.

Зміст

Вступ 3
Розділ 1. Організація процедурно орієнтованих програм 5
1. Теоретичні та методологічні засади організації програм і даних 5
1.1. Основні концепції й термінологія.. 5
1.2. Етапи розроблення програмного забезпечення.............. 13
1.3. Огляд середовища розроблення Visual Studio .Net......... 15
Контрольні запитання 22
2. Поняття типу даних... 22
2.1. Концепція типу даних 22
2.2. Базова структура C#-програми.. 25
2.3. Характеристика й особливості застосування простих типів 39
2.4. Константні величини 52
Контрольні запитання 55
3. Програмування обчислювальних процесів................................. 56
3.1. Програмування лінійних обчислювальних процесів........ 56
3.2. Оператори управління програмою 85
3.3. Керівні оператори в циклах.. 105
Контрольні запитання 112
Розділ 2. Організація й опрацювання складених типів даних............... 113
4. Масиви.. 113
4.1. Загальні відомості про масиви 113
4.2. Приклади опрацювання одновимірних масивів 127
4.3. Багатовимірні масиви... 130
Контрольні запитання 135
5. Структури.. 136
5.1. Загальні відомості про структури 136
5.2. Приклади елементарного опрацювання структур 138
5.3. Масиви структур 140
Контрольні запитання 143
6. Функції... 143
6.1. Загальні відомості про функції 143
6.2. Обмін інформацією з функцією 155
6.3. Функції та структури 160

6.4. Перевантаження функцій 162
6.5. Делегати 164
Контрольні запитання 171
Використана та рекомендована література 172
Основна 172
Додаткова 172
Інформаційні ресурси 172

НАВЧАЛЬНЕ ВИДАННЯ

Браткевич Вячеслав Вячеславович
Хорошевська Ірина Олександрівна

ПРОГРАМУВАННЯ ЗАСОБІВ МУЛЬТИМЕДІА

Конспект лекцій
для студентів спеціальності
186 "Видавництво та поліграфія"
першого (бакалаврського) рівня

Самостійне електронне текстове мережеве видання

Відповідальний за видання *О. І. Пушкар*

Відповідальний редактор *О. С. Вяткіна*

Редактор *О. Г. Доценко*

Коректор *Н. Г. Войчук*

План 2023 р. Поз. № 8-ЕК. Обсяг 176 с.

Видавець і виготовлювач – ХНЕУ ім. С. Кузнеця, 61166, м. Харків, просп. Науки, 9-А

Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру
ДК № 4853 від 20.02.2015 р.