ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

Дібрівний О.А., Гребенюк В.В.

Вступ до об'єктно орієнтованого програмування C#

Київ 2018

Дібрівний О.А., Гребенюк В.В. Вступ до об’єктно орієнтованого програмування С#: Навчальний посібник. – К.: Державний університет телекомунікацій, 2018, - 190сторінок.

Метою цієї книги є опис базових принципів функціонування платформи .NET, системи типів .NET для різних інструментальних засобів розробки, що використовуються при створенні додатків .NET (багато з таких інструментальних засобів є програмними продуктами з відкритим вихідним кодом). Тут же представлені базові можливості мови програмування С #.Також в книзі розглядаються базові конструкції мови програмування С #. Ви освоїте техніку побудови класів, з'ясуйте різницю між типами, що характеризуються значеннями, і посилальними типами, приведенням до об'єктного типу і відновленням з "об'єктного образу", а також роль улюбленого всіма базового класу System.Object. Також розглядається структурований підхід до обробки винятків, які дозволяють вирішувати відповідні проблеми. Особлива увага приділяється тому як C# поєднується з базовими принципами ООП – спадкуванням, інкапсуляцією та поліморфізмом.

Насамперед цей посібник буде корисний людям які мають бажання почати роботу в середовищі розробки Visual Studio на мові С#.

Рецензенти: к.т.н., доц. Корнага Я.І.

Проф. Вишнівський В.В.

Рекомендовано:

Зміст

Розділ 1. Основні поняття про C# та платформу .NET 6
1.1.Поняття про платформу .NET 6
1.2. Створення мови програмування C#. 7
1.3. Представлення даних. Необхідність типізації. Двійкова арифметика. 8
1.4. Основні поняття програмування. 10
Розділ 2. Поняття про інтегроване середовище розробки VisualStudio (на прикладі Visual Studio 2017). Структура C#-програми. 14
2.1. Основні можливості інтегрованого середовища розробки VisualStudio .NET. 14
2.2. Структура програми мовою C#. 16
Розділ 3. Основні елементи мови C#. 20
3.1. Основні вбудовані типи мови C# 20
3.2. Визначення та ініціалізація змінних, область їх видимості. 21
3. Приведення типів. 23
4. Літерали (константи) мови C#. 24
Розділ 5. Операції мови C#. 27
5.1. Арифметичні операції. 27
5.2. Операції інкременту та декременту. 27
5.3. Операції відношення (порівняння). 28
5.4. Логічні операції. 28
5.5. Порозрядні (бітові) операції. 29
5.6. Умовна (тернарна) операція. 31
5.7. Операції присвоєння. 31
5.8. Пріоритет операцій. 32
Завдання для самоконтролю 33
Розділ 6. Основні інструкції керування мови C# – розгалуження та цикли 34
6.1. Розгалуження у мові C# 34
6.2. Цикли у мові C# 39
6.3. Керування виходом із циклів C# 45
Завдання для самоконтролю 46
Розділ 7. Масиви в мові C# 49
7.1. Визначення та ініціалізація масиву. 49
7.2. Цикл foreach 52
7.3. Багатовимірні масиви. 53
7.4. Використання деяких методів класу System.Array. 55
7.5. Масиви масивів. Непрямокутні масиви. 56
Завдання для самоконтролю 57
Розділ 8. Структуровані типи даних (колекції) в мові C# 61
8.1. Основні структури даних та їх призначення 61
8.2. Використання списку ArrayList та узагальненого списку List 62
8.3. Використання асоційованого списку Hashtable та узагальненого словника Dictionary. 66
Завдання для самоконтролю 67
Розділ 9. Класи і методи в мові C#. 69
9.1. Визначення класу. 69
9.2. Методи класу. 71
9.3. Методи з параметрами. 73
9.4. Конструктор класу. 75
9.5. Передача об'єктів методам. 77
9.6. Використання модифікаторів для параметрів методів. 80
9.7. Методи, що повертають об'єкти. 83
9.8. Перевантаження методів. 85
9.9. Перевантаження конструкторів. 89
9.10. Використання ключового слова this. 90
9.11. Деструктор класу. 93

9.12. Метод Main (). 96
Завдання для самоконтролю 96
10.Статичні члени класу. 98
10.1. Статичні дані-члени класу. 98
10.2. Статичні методи-члени класу. 100
10.3. Статичний конструктор класу. 103
10.4. Статичні класи, локалізація та глобалізація 105
Завдання для самоконтролю 107
Розділ 11.Властивості та індексатори. 109
11.1. Властивості. 109
11.2. Індексатори. 113
Завдання для самоконтролю 118
Розділ 12. Спадкування в мові C#. 119
12.1. Поняття про спадкування та ієрархію класів. 119
2. Спадкування та правила доступу до членів класів. 121
12.3. Конструктори базового та похідних класів. 123
12.4. Посилання на екземпляри базового та похідних класів. 126
12.5. Поняття про поліморфізм. 129
12.6. Віртуальні функції – більш детальний погляд. 132
12.7. Абстрактні методи та класи. 138
Завдання для самоконтролю 141
Розділ 13. Структури та переліки в мові C#. 142
13.1. Структури. 142
13.2. Переліки. 148
Розділ 14. Класи колекцій і протоколи ітерації 151
14.1. Використання колекцій. 151
14.2. Generics. 152
14.3. Ітератори 155
14.4. LINQ 156
Завдання для самоконтролю 156
Розділ 15. Обробка винятків. Застосування конструкцій checked і unchecked 157
15.1. Ієрархія винятків 157
15.2.Основи обробки винятків 157
15.3.Застосування конструкцій checked і unchecked. 159
Завдання для самоконтролю 160
Розділ 16.Взаємодія з файловою системою 161
16.1. Модель потоків в C#. Простір System.IO. клас Stream 161
16.2. Аналіз байтових, символьних, бінарних класів потоків. 161
16.3. Використання класів FileStream; StreamWriter, StreamReader; BinaryWriter, BinaryReader; Directory, DirectoryInfo; File, FileInfo для файлових операцій. 161
16.4. Використання класу Path. 164
Завдання для самоконтролю 164
Розділ 17. Делегати. Лямбда-вирази 165
17.1. Делегати 165
17.2. Анонімні методи 166
17.3. Лямбда-вирази 166
Завдання для самоконтролю 167
Розділ 18.Події 168
18.1. Події 168
18.2. Застосування подій 168
18.3. Стандартна модель подій 168
Завдання для самоконтролю 169
Розділ 19. Багатопоточність 170
19.1. Багатопоточність. 170

19.2. Основи багатопоточної обробки. 170
19.3. Простір імен System.Threading. 173
19.4. Делегати ThreadStart, ParameterizedThreadStart. 173
19.5. Оператор lock. Monitor - клас. 173
19.6. Асинхронне програмування. Async, Await. 174
Завдання для самоконтролю 175
Розділ 20. Серіалізація 176
20.1. Поняття атрибутів. 176
20.2. Що таке серіалізація? 176
20.3. Відносини між об'єктами 176
20.4. Графи відносин об'єктів. 177
20.5. Атрибути для серіалізації [Serializable] і [NonSerialized]. 177
20.6. Формати серіалізації. 178
Завдання для самоконтролю 179
Розділ 21. Перевантаження операторів 180
21.1. Перевантаження операторів. 180
21.2. Перевантаження унарних операторів 181
21.3. Перевантаження бінарних операторів. 181
21.4. Перевантаження операторів порівнянь. 181
21.5. Перевантаження операторів true і false. 182
21.6. Перевантаження логічних операторів. 182
21.7. Перевантаження операторів перетворення. 182
Завдання для самоконтролю 182
Розділ 22. Збірка сміття 184
22.1. Життєвий цикл об'єктів 184
22.2. Збирач сміття 184
22.3. Деструктор і метод Finalize 184
22.4. Метод Dispose і інтерфейс IDisposable 185
22.5. Поняття поколінь при збиранні сміття. 185
Завдання для самоконтролю 186
Список використаної літератури. 187

Розділ 1. Основні поняття про C# та платформу .NET

1.1.Поняття про платформу .NET

Невпинний розвиток комп'ютерних технологій та проникнення Інтернету у всі галузі нашого життя диктує нові вимоги до комп'ютерних програм. Тепер програми існують у безкрайній мережі, що пов'язує мільйони комп'ютерів. Отже, вони повинні підтримувати можливість створення програмного забезпечення із включенням програм, написаних різними мовами програмування, які здатні функціонувати в різних операційних системах. На жаль, існуючі мови важко пристосовувати до гармонійної міжмовної взаємодії, оскільки вони як раз прив'язані до різних операційних систем або навіть різних версій однієї операційної системи.

Крім того, дуже рідко програми, що мають ринкову цінність, створюються однією людиною – набагато частіше над ними працюють цілі колективи програмістів, які можуть використовувати для розробки різні мови програмування та технології. Особливо це стосується програм, призначених для роботи з мережею. До створення технології Microsoft .NET Framework така розробка програм передбачала цілу низку необхідних узгоджень між різними частинами програми, оскільки мови та технології, що використовувалися, хоча й мали схожий синтаксис, проте спиралися на різні принципи роботи.

Microsoft .NET Framework — це програмна технологія, запропонована фірмою Microsoft для створення програм різних типів, в тому числі і веб-застосувань. Однією з центральних ідей цієї технології є сумісність різних служб, створених різними мовами програмування. Програми можуть звертатись до методів та класів, створених на цій платформі з використанням різних мов. Ця можливість реалізована завдяки тому, що середовище розробки .NET при компіляції кожної програми, створеної будь-якою мовою програмування, що існують на платформі .NET, створює двійковий код, спеціальною проміжною мовою, що одержала назву MSIL (Microsoft Intermediate Language або як спочатку MultiSystem Intermediate Language), або просто IL. Цей код подібний до байт-коду мови Java і містить так звану збірку (assembly). Крім власне інструкцій IL–мовою збірка включає ще так звані метадані – інформацію про всі використані в програмі типи та інформацію про саму збірку (версія збірки, обмеження по безпеці, тощо).

На наступному етапі, коли платформно-незалежний код мовою IL має бути пристосований до довільної конкретної платформи, на якій буде «жити» програма, цю задачу виконує спеціальний JIT (just-in-time compiler) компілятор («компіляція на льоту»). Роботою цього компілятора керує спеціальна служба CLR (Common Language Runtime) – стандартне середовище виконання для мов .NET. CLR керує всіма задачами низького рівня, пов'язаними з розгортанням застосувань, використанням пам'яті, тощо. Крім цього, частиною .NET є стандартна система типів CTS (Common Type System), та потужна бібліотека базових класів, які використовуються усіма мовами платформи .NET.

Отже, платформа .NET – це нова модель для створення програмних застосувань (applications). Серед багатьох інших переваг цієї моделі однією з основних є можливість повної мовної взаємодії при створенні програм, а також використання потужної бібліотеки базових об'єктів.

1.2. Створення мови програмування C#.

Хоча платформа .NET і передбачає можливість використання різних мов програмування (наприклад, C++ та Visual Basic), спеціально для неї група програмістів фірми Microsoft розробила нову мову програмування, яка одержала назву C#. Ця мова дозволяє у повному обсязі скористатися можливостями, що надає технологія .NET. Фактично, C# являє собою гібрид різних мов програмування, від кожної з яких вона взяла найкраще.

Будучи спадкоємицею мови C++, мова C# використовує подібний до неї синтаксис, проте позбавлена неоднозначностей, які допускали компілятори C++. Багато спільного має також із мовою Java, яка створювалась спеціально як мова, що дозволяє реалізовувати застосування, які можуть використовуватись в різних операційних системах. Її створення як раз і було намаганням розв'язати так звану проблему «перенесення» комп'ютерних програм в інше операційне середовище. Ця можливість була реалізована за рахунок компіляції у два етапи. На першому етапі Java-компілятор створює машиннонезалежний так званий «байт-код». Цей байт-код виконується віртуальною машиною Java (JVM – Java Virtual Machine). Вона являє собою спеціальну операційну систему. Отже, програма мовою Java може бути виконана на будь-якій платформі, де реалізовано JVM. А оскільки така реалізація була відносно не складною, то JVM були достатньо швидко та успішно реалізовані у досить різноманітних програмних середовищах. Проте мова Java не вирішила проблему міжмовної взаємодії в програмах.

На відміну від Java, C# генерує код, що може бути використаний лише у середовищі виконання .NET – так званий **керований код** (managed code). Зокрема, це передбачає автоматичне керування пам'яттю та відсутність потреби працювати напряму із вказівниками на адреси у пам'яті, що дуже зменшує кількість ймовірних помилок при розробці та використанні програми.

Ще одна приємна риса, яка відрізняє мову C# – можливість одночасно створювати і програму, і документацію до неї у форматі XML. Для цього треба лише використовувати спеціальний тип коментарів та після завершення роботи згенерувати окремий файл із документацією до програми (ця можливість вбудована у середовище розробки.)

Компонент .NET Microsoft Visual Studio – це інтегроване середовище розробки програмного забезпечення, яке забезпечує абсолютно комфортний інтерфейс для створення C#-програм.

1.3. Представлення даних. Необхідність типізації. Двійкова арифметика.

Одним із фундаментальних понять програмування (детальніше про більшість з них ми будемо вести розмову пізніше) є тип даних, який визначає, по-перше, множину значень, по-друге, множину допустимих операцій із цими значеннями і нарешті, спосіб збереження значень та виконання операцій. Тобто, будь-яка інформація, якою оперує програма, відноситься до певного типу даних. В чому полягає необхідність такого підходу?

Справа в тому, що навіть у мовах низького рівня, зокрема в мові Асемблер, для дійсних та цілих чисел відводяться різні об’єми пам’яті, а отже, і дії з ними виконуються по-різному. В мовах високого рівня цей факт також не міг не знайти свого відображення. Крім того, типи даних в програмуванні просто не можуть однозначно відповідати прийнятим у математиці типам чисел. Так, у математиці множина цілих чисел не обмежена ні знизу, ні згори. Реалізувати подібний тип у комп’ютері неможливо. Тому, як правило, у мовах програмування для різних видів обчислень використовується множина типів цілих чисел, які мають різні діапазони значень в залежності від виділеного для них об’єму пам’яті.

Для довідки, мінімальна одиниця інформації в комп’ютері складає 1 біт (1б), в якому якраз і може бути записані 0 або 1. 1 байт (1Б) об’єднує вісімку бітів, а далі все одноманітно – 1 КБ(кілобайт) = 1024 Б ($1024 = 2^{10}$), 1 МБ(мегабайт) = 1024 КБ, 1 ГБ(гігабайт) = 1024 МБ.

Розглянемо (коротко) кодування інформації за допомогою двійкової системи числення. Будь-яка позиційна система числення з основою (базою) q системи числення вимагає відповідно q символів для представлення будь-якого числа. Так, в десятковій системі числення використовується 10 цифр від 0 до 9, в двійковій всього 2 цифри – 0 та 1, в 16-ковій – 16 символів – це цифри від 0 до 9 та 6 латинських літер від A до F. Їх відповідність між собою показано у таблиці 1.1.

Таблиця 1.1.

Відповідність між різними системами числення

10-кова система	2-кова система	16-кова система	10-кова система	2-кова система	16-кова система
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Нехай тепер $A_{(q)}$ – довільне дійсне число, записане у позиційній системі числення з основою q. Тоді, якщо його запис в цій системі має вигляд $A_{(q)} = a_{n-1}a_{n-2}..a_1a_0,a_{-1}..a_{-m}$, то

$$A_{(q)} = a_{n-1}q^{n-q} + a_{n-2}q^{n-2} + ... + a_1q^1 + a_0q^0 + a_{-1}q^{-1} + ... + a_{-m}q^{-m} \quad (1.1)$$

Наприклад, $12{,}34_{(10)} = 1 \cdot 10^1 + 2 \cdot 10^0 + 3 \cdot 10^{-1} + 4 \cdot 10^{-2}$, або

$$101{,}011_{(2)} = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 5{,}375_{(10)}.$$

Формулу (1) можна записати у вигляді:

$$A_{(q)} = (...((a_{n-1}q + a_{n-2})q + a_{n-3})q + ... + a_1)q + a_0 + a_{-1} +$$
$$+ (...(a_{-m+1} + a_{-m}q^{-1})q^{-1} + ...)q^{-1}. \quad (1.2)$$

Тепер зрозуміло, що для переведення цього числа з десяткової системи числення у систему числення з основою q, для визначення цілої частини числа треба ділити його цілу частину на основу q,записуючи залишки у зворотному порядку, до тих пір, поки не одержимо результат, рівний 0 або 1. А для визначення дробової частини – треба взяти дробову частину у десятковій формі та множити її на основу q, записуючи кожного разу цифри, які отримуватимемо у цілій частині результату. Множення виконувати до тих пір, поки дробова частина результату не стане нульовою, або не заповнимо розрядну сітку.

Розберемо приклад. Нехай маємо перевести число $123_{(10)}$ до двійкової та шістнадцяткової систем числення. При діленні числа 123 на 2 одержимо 61 і залишок **1**. Залишок запам'ятаємо, а результат 61 ділимо на 2 знову. Одержуємо 30 і залишок **1**. Продовжуючи процес, одержимо залишки (перевірте!) **0, 1, 1**. Врешті решт отримаємо залишок **1** і результат **1**. Запишемо ці цифри у зворотному порядку: $1111011_{(2)} = 123_{(10)}$.Для одержання 16-кового представлення розділимо число у двійковій формі на четвірки цифр і знайдемо відповідні 16-кові цифри у таблиці: $1111011_{(2)} = 01111011_{(2)} = 7B_{(16)}$.

Нехай тепер маємо дробове число $0{,}4567_{(10)}$,яке переведемо у 2-ковий дріб. При цьому вважатимемо, що розрядна сітка рівна 8 символам. Отже, множимо 0,4567 на 2 і фіксуємо значення цілої частини результату. Маємо 0,9134, запам'ятаємо **0**. Знову множимо лише дробову частину результату, матимемо 1,8268. Запам'ятаємо **1** і множимо 0,8268. Продовжуємо, поки не отримаємо 8 цифр. Таким чином одержимо (перевірте!) $0{,}4567_{(10)} \approx 0{,}01110100_{(2)} \approx 0{,}74_{(16)}$.

1.4. Основні поняття програмування.

Будь-яка мова має певний мовний лексикон – словник мови. Кожна мова програмування також базується на своєму словнику. Проте всі мови програмування спираються на певний набір основних термінів так чи інакше відбитий у мові. З'ясуємо основні з них, починаючи з простіших.

Перш за все слід згадати константи та змінні. Значення цих термінів знайоме будь-кому, хто вивчав математику. ***Константа*** – це величина, яке не змінює свого значення. Отже, значення константи має бути визначене в той момент, коли ви *описуєте* константу в програмі, тобто знайомите з нею компілятор.

Змінна – це очевидно, величина, яка протягом роботи програми може набувати різних значень. Вище вже говорилось про необхідність типізації даних, тобто кожна змінна програми має відноситись до певного типу даних, допустимих в межах синтаксису (тобто правил використання слів) даної мови програмування. Змінні також мають бути «представлені» компілятору. Проте тут слід розрізняти два етапи: ***опис*** (декларацію) змінної, який для компілятора є повідомленням про характер змінної та дає змогу контролювати її використання в програмі, та ***визначення*** змінної, тобто власне створення змінної в пам'яті. Надання початкового значення змінній називається ***ініціалізацією***.

За своїми властивостями змінні також різні. Крім змінних, які можуть зберігати певні значення (числові, символьні, текстові, тощо), за що вони ще називаються змінними ***value-типу*** , або просто змінними-значеннями, існують змінні ***reference-типу***, або змінні-посилання (або адресні змінні). Їх різнить місце розташування та особливості звертання до них. Звичайні змінні розташовані у так званому ***стеку*** – спеціальній області пам'яті програми, елементи якої обслуговуються за принципом черги (англ. LIFO, last input – first output) – елемент, записаний останнім, вибирається першим. Змінні-посилання розташовуються у так званій «купі» (область Heap) і звертання до них, принаймні технічно це реалізовано саме так, відбувається опосередковано з допомогою її адреси.

Фізичні та математичні константи мають усталені імена, наприклад, e – стала Ейлера, G – гравітаційна стала, тощо. Змінні в математиці також позначають звичними літерами – x, y, z . Імена змінних та констант в програмі звуться ***ідентифікаторами***. Складати ідентифікатори програміст має право довільно, не обмежуючи свою фантазію, в межах, звісно власної культури та правил синтаксису мови. Хороший ***стиль програмування*** (більше про стиль програмування можна самостійно прочитати у посібниках [1, 2]) означає, як мінімум, що програма, яку ви створюєте, має властивість ***readability*** – зручність для читання та сприйняття, а отже, і внесення змін та супроводження її. Тому ідентифікатори слід вибирати так, щоб вони відбивали зміст змінної або константи. Наприклад, константи прийнято записувати, використовуючи лише великі літери – MAX, або MIN, для змінних зазвичай використовують малі літери. При цьому для позначення лічильних змінних вибирають літери i,j,k, робочі (тимчасові) змінні часто

позначають temp , work або просто val. Більшість мов програмування серед символів свого алфавіту містить знак підкреслення. Його зручно вживати, складаючи ідентифікатори з кількох частин, наприклад, array_size, vektor_len або file_num. Деякі спеціалісти вважають більш прийнятною формою для ідентифікаторів, що складаються з декількох частин, використання великих літер для початку кожної частини, наприклад, formName, тощо. При цьому використовують також ряд правил скорочення.

Змінні та константи використовують у ***виразах*** – вони є ***операндами***, які з'єднані допустимими ***операціями***, тобто діями, що мають бути виконані над операндами. Кожний тип даних має визначений набір операцій, які по кількості операндів діляться на ***унарні*** (коли операція стосується одного операнда), наприклад, знак мінус перед числом, та ***бінарні*** (ті, що стосуються двох операндів), наприклад арифметичні додавання, віднімання, ділення та множення. Існує також спеціальна операція вибору, яка є ***тернарною*** (тобто стосується трьох операндів.)

Взагалі кажучи, робота зі змінною складається з двох етапів: декларації (проголошення змінної) та ініціалізації (присвоєння змінній значення). У випадку використання задекларованої змінної, якій не було присвоєне значення, може виникнути помилка компіляції:

Potential use of unassigned variable

Для того, щоб краще запам'ятати це, згадаємо дитячу казку про Буратіно. У цій казці Мальвіна питає Буратіно: «*Вам дали 5 яблук, а Ви віддали комусь 3 яблука, скільки в Вас залишилось?*». З точки зору програмування завдання є неоднозначним, бо невідомо *скільки яблук у Буратіно було спочатку*. А програмістам потрібно пам'ятати, що ***змінні потрібно завжди ініціалізувати***. Проте, якщо компілятор мови C# пропустить неініціалізовану змінну, то він присвоїть їй значення за наступними правилами:

- якщо це змінна-посилання (reference-типу), то її значенням буде **null** (це адресний нуль)
- якщо змінна логічного типу, то її значенням буде **false**
- якщо змінна є екземпляром структури, то всі її поля набудуть значень за замовчуванням
- якщо змінна належить до цілого типу, або типу з рухомою крапкою, або до типу **Decimal**, то її значенням буде 0.

Будь-яка програма складається з ***операторів***, які описують певну послідовність дій. Як правило, оператори поділяються на групи, що певною мірою описують їх функціональність. **Базовими елементами програми** називаються такі основні логічні структури:

1. слідування — це послідовність операторів (груп операторів), які виконуються один за одним в порядку їх запису в тексті програми;

2. розгалуження — керівна структура, яка в залежності від виконання заданої умови визначає вибір для виконання одного з двох заданих у цій структурі операторів (груп операторів);
3. повторення — цикл, у якому група операторів може виконуватися знову, якщо справедлива задана умова.

Слід зауважити, що насправді до трьох основних структур слід додати допоміжні: неповне розгалуження, багатогілкове розгалуження (вибір з декількох варіантів), виклик процедури (особливо важливим це стає у випадку рекурсивної процедури) У будь-якій мові програмування існують групи операторів розгалуження та операторів циклу.

З'ясуємо зміст ще одного терміну, що вживається в програмуванні, – ***модуль***. Модулем називають окрему програмну одиницю, яка автономно (незалежно) компілюється. Термін «модуль» вживається у двох сенсах. Традиційно під модулем розуміли підпрограму (процедуру або функцію), тобто послідовність зв'язаних фрагментів програми, які виконують окрему задачу. Звертання до модуля відбувається по імені. Важливо підкреслити два основних моменти – окрема компіляція та реалізація однієї окремої задачі. (Детальніше про принцип модульності див. у [2]). З часом під терміном модуль стали розуміти також окремий набір (бібліотеку) програмних елементів, націлених на розв'язання однієї або кількох пов'язаних між собою складних задач. Цей термін добре знайомий тим, хто вивчав мови програмування Pascal або Modula.

Отже, спростимо зараз термін «модуль» до терміну «***функція***» та поговоримо про них. Звертання до функції відбувається за її іменем. Інші функції, що її викликають, обмінюються з нею інформацією через інтерфейс функції – список її параметрів. Ці параметри у визначенні функції носять назву ***формальних параметрів***. Формальних, тому що функція задає формальну схему дій з цими параметрами. Так само у визначеному інтегралі змінна інтегрування є формальною. Від заміни її імені з x на y або z значення інтегралу не зміниться. У момент виклику функції формальні параметри мають бути конкретизовані – на їх місце підставляються необхідні значення. Тому параметри у виклику функції звуть ***фактичними параметрами*** або просто ***аргументами***. Результат своєї діяльності функція повертає модулю, який її викликав. Як правило, це відбувається у вигляді присвоєння цього результату деякій змінній або використання цього результату у деякому виразі. Модуль, який не формує результат свого виклику, у деяких мовах програмування, зокрема у мові Pascal, зветься процедурою. В інших мовах програмування (C, C++, Java, C#) процедури як окремий модуль не реалізовані – їх заміняють функції з порожнім (void) результатом. Щодо назв функції рекомендовано вживати пари «дієслово-іменник», у якості прикладу можна використати **DoSomething**.

Слід зазначити, що в момент виклику модуля (функції) з технічної точки зору відбуваються дуже складні і важливі маніпуляції, які зазвичай приховані від уваги програміста. Оскільки модуль після компіляції зберігається окремо, то в момент його виклику і передачі йому керування система має створити

для нього всі необхідні йому параметри. Вони створюються у стеку і в них копіюються значення фактичних параметрів. Тобто модуль працює з комплектом копій переданих йому змінних, а після завершення роботи модуля стек звільняється від записаних у нього параметрів. Такий спосіб передачі параметрів є найпростішим і забезпечує модулю доступ лише до значень вхідних даних. Він називається ***передачею параметрів за значенням***. Якщо ж необхідно, щоб модуль повернув через свій інтерфейс деяку інформацію, необхідно задіяти інший, більш складний механізм передачі параметрів. Такі параметри мають бути спеціальним чином виділені. У мові Pascal, наприклад, для цього використовують спеціальне службове слово – специфікацію var. Це означає, що параметр буде передаватись, як адресна змінна. Тобто модуль одержить для роботи значення адреси місця розташування аргументу, таким чином матиме доступ не до копії змінної, а безпосередньо до неї самої. Цей спосіб передачі параметрів із зрозумілих причин називається ***передачею за адресою*** або за посиланням.

Розділ 2. Поняття про інтегроване середовище розробки VisualStudio (на прикладі Visual Studio 2017). Структура C#-програми.

2.1. Основні можливості інтегрованого середовища розробки VisualStudio .NET.

Інтегроване середовище розробки (Integrated Development Environment – IDE) Visual Studio 2005 – це універсальне середовище єдиного формату для всіх мов програмування .NET. Тобто при створенні проекту будь-якого типу будь-якою мовою програмування з набору ви матимете справу з одним і тим самим середовищем розробки. Більшість його можливостей ви засвоїте при самостійному вивченні, зараз же розглянемо лише самі основні з них.

Перш за все, Visual Studio має справу з проектами. В перекладі з латинської «проект» – той, що виступає попереду, висунутий наперед. Під проектом в даному разі розуміють всю сукупність програмних засобів для реалізації деякої задачі. Тобто проект може містити кілька програмних файлів, бази даних, класи тощо.

Для створення проекту використаємо послідовність команд меню : File -> New Project. Далі у діалоговому вікні необхідно вибрати один із запропонованих типів проектів. Зупинимо свій вибір на консольному застосуванні – Console Application. Маємо можливість обрати ім'я для свого застосування, наприклад, Console_First.

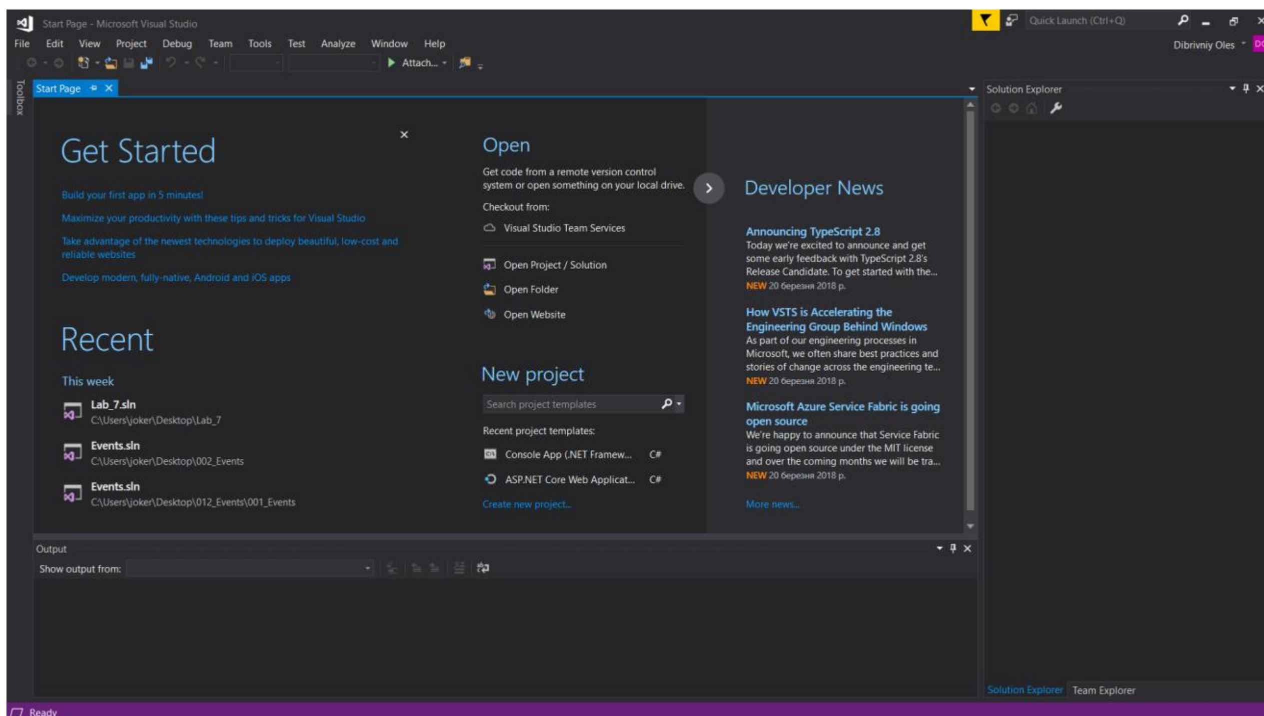

Рисунок 2.1. Головне вікно Visual Studio 2017

В результаті Visual Studio сформує заготовку проекту Console_First. Власне, він цілком дієздатний, в чому можна переконатись з допомогою команд Debug -> Start Without Debugging. Зверніть увагу, що код основної програми мовою C# міститься у файлі Program.cs. Розширення .cs є

обов'язковим для кодів програм, що обробляються компілятором мови C#. Компілятор є одним із елементів VisualStudio і запускається командою пункту меню Debug. Можна скористатись також комбінаціями відповідних «гарячих» клавіш.

Залишимо поки що програмний код і розглянемо деякі діалогові вікна. Вікно **Solution Explorer** (оглядач рішень) дозволяє керувати рішенням – реалізацією проекту. Можна включати в проект додаткові елементи, такі як бази даних, класи, бібліотеки тощо, або навпаки – виключати, можна перейменувати певні елементи. Рухаючись по розгалуженням рішення, можна лівою кнопкою миші згортати або розгортати директорії елементів проекту, змінюючи значки «+» та «-». Вікно **Properties** (властивості) відображує головні властивості вибраного елементу проекту. Зробимо, наприклад, активним Program.cs у вікні Solution Explorer. Тоді у вікні Properties ми побачимо характеристики саме цього програмного файлу. Зокрема, у полі File Name міститься ідентифікатор даного програмного файлу. Змінимо його, наприклад, на My_Program.cs. У діалоговому вікні, що виникне на екрані, побачимо зауваження про перейменування файлу і пропозицію перейменувати відповідним чином всі посилання у проекті, з якою варто погодитись.

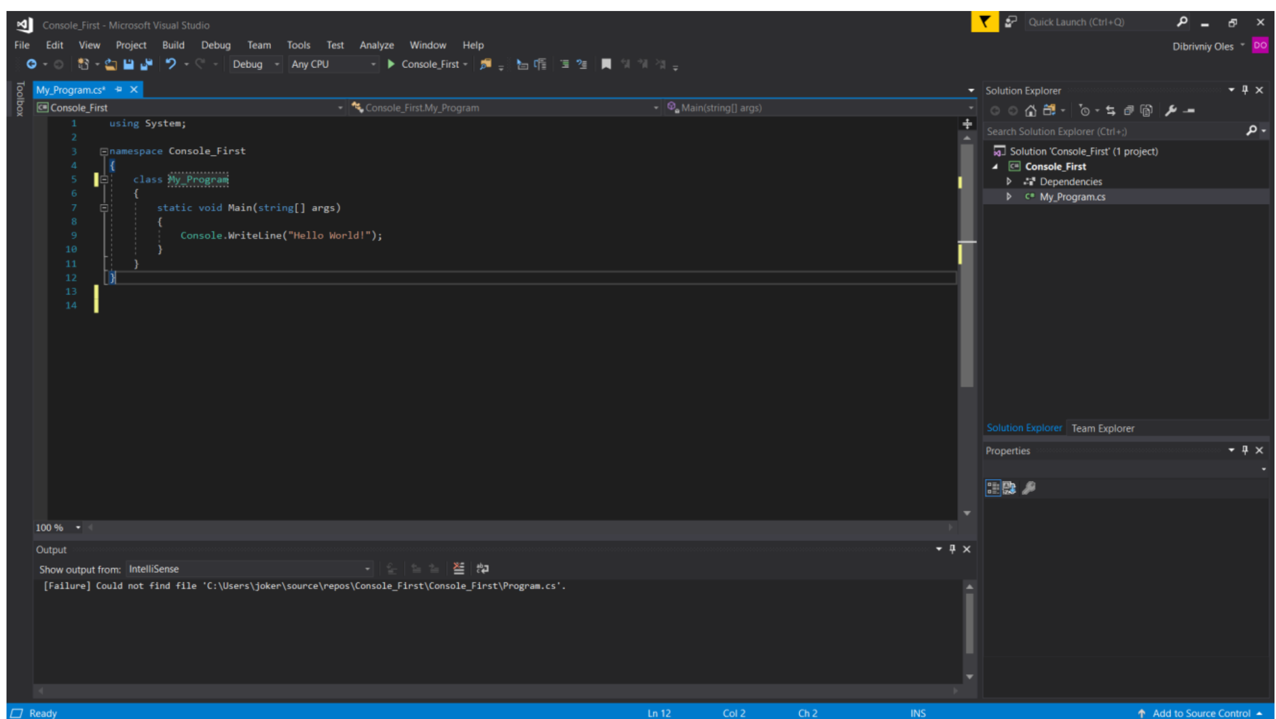

Рисунок 2.1. Створення консольного додатку у Visual Studio 2017

Зверніть увагу, що середовище VisualStudio дозволяє вам вибирати оптимальні форми відображення всіх вікон на екрані. Вони можуть бути заховані у панелях по боках екрану, якщо активізувати піктограму «Auto Hide». Якщо ця піктограма знаходиться у позиції, в якій нагадує канцелярську кнопку, – вікно закріплене на екрані, якщо ж «кнопка» перевернута горизонтально, вікно згортається у невеличку панель. Непотрібні вікна можна

видалити з екрану при натисканні мишою кнопки Close. Реанімувати після цього вікно на екрані, можна через команду View.

При роботі з кодом програми є можливість згортати та розгортати цілі фрагменти коду подібно до того, як це діяло у вікні Solution Explorer. І, безумовно, вражаючий ефект створює підтримка технології IntelliSense, яка при наборі тексту «підказує» вам завершення початого рядка.

Ну, і звичайно, звертайте увагу на повідомлення компілятора у вікні Error List. Всі пропущені дужки, крапки з комою будуть ним помічені просто під час набору тексту програми. Крім засобів поточного синтаксичного аналізу тексту до ваших послуг інтегрований налагоджувач, роботу з яким можна розпочати з допомогою панелі інструментів Debug. Ліва кнопка миші дозволить вмикати та вимикати точки переривань (breakpoints) на лівому сірому стопці поруч з програмним кодом. Команди Debug -> Windows дозволять вибрати потрібне вікно для контролю виконання програми.

2.2. Структура програми мовою C#.

Розглянемо наступний приклад:

```
/*  Це текст першої програми мовою C# */
//  Коментар може бути поміщений після двох слешів
using System;
class Program
{
   // Виклик будь-якої програми означає виклик функції Main()
   public static void Main()
   {             //  Вивели заголовок
      Console.WriteLine("Перша програма" + "!!!");
      //  Вивели запрошення
      Console.WriteLine("Введіть ціле число ");
      int i;  //  Визначили цілу змінну
            //  Прочитали цілу змінну
      i = int.Parse(Console.ReadLine());
      Console.WriteLine("i = " + i); //  Вивели її значення
                              //  Вивели запрошення
      Console.WriteLine("Введіть дійсне число ");
      double f;   //  Визначили дійсну змінну
              //  Прочитали дійсну змінну
      f = double.Parse(Console.ReadLine());
      Console.WriteLine("f = " + f); //  Вивели її значення
   }
}
```

На що необхідно звернути увагу? Перш за все, текст, поміщений між знаками **/*   */** є коментарем, тобто поясненнями до коду програми, які ігноруються компілятором. Проте, такі пояснення конче необхідні тим, хто працює з програмою і вважаються обов’язковими. Коментарем також вважається і все, що знаходиться праворуч від знаків **//** – так зручно оформлювати коментарі до окремих рядків. І одразу зауважимо, що мова C# є регістро-залежною, отже великі та маленькі літери не слід плутати.

Далі, у рядку **using System;** фіксується простір імен **System**, що містить елементи бібліотеки класів .NET. Це спрощує звертання до потрібних нам методів цих класів, зокрема до методів класу **Console**, необхідних при роботі консольних застосувань.

Наступний рядок програми містить текст **class Program** і означає, що вміст фігурних дужок **{ }** визначає члени класу з іменем **Program**. Ім'я (ідентифікатор) класу **Program** компілятор створив таким самим, як і ідентифікатор основного програмного файлу Program.cs. Проте, ім'я класу може бути довільно змінене.

Рядок **public static void Main()** – це заголовок функції **Main**. Як уже зазначалось всі підпрограми мови C# по суті є функціями. Більше того, ці функції повинні бути членами деякого класу, як у даному прикладі – класу **Program**. Функції-члени класів в мові C# звуться методами. Головним методом проекту повинен бути метод із зарезервованим іменем **Main** – цей метод використовується як точка входу при роботі нашого застосування. (Зверніть увагу на написання – на відміну від мов C та C++ назва цієї функції починається з великої літери). Наступна пара фігурних дужок містить тіло цієї функції. Службове слово **public** визначає специфікатор доступу до члену класу. В даному разі відкритість методу **Main** не є обов'язковою вимогою синтаксису, проте ми будемо дотримуватись саме такої нотації при визначенні методу **Main**. Службове слово **static** означає, що даний метод може бути використаний без створення власне об'єкту, що належить класу **Program**. В даному випадку саме таке визначення методу **Main** є принциповим, адже цей метод викликається при запуску програми. І нарешті, слово **void** означає, що метод **Main** не повертає ніякого результату. Порожні круглі дужки після імені методу обов'язкові і можуть містити список формальних параметрів методу, якщо вони потрібні для його роботи. В нашому прикладі метод **Main** працює без вхідних даних. Вся логіка класу **Program** зосереджена у методі **Main** – інших методів в даному класі просто немає. Цей метод звертається до класу **Console**, визначеного у просторі імен **System**, як вже згадувалось вище. Методи цього класу **WriteLine** та **ReadLine** використовуються відповідно для виводу рядка у стандартний потік виведення та для введення рядка із стандартного потоку введення. Оскільки стандартні потоки є символьними (стандартний потік введення спрямований з клавіатури, а стандартний потік виведення спрямований на консоль), необхідно перетворити послідовність символів у число, дійсне чи ціле, якщо ми збираємось вводити інформацію саме такого типу. Це перетворення (конвертацію) послідовності символів у число ціле або дійсне виконують різні методи з однаковими іменами **Parse**. Вони є членами структур з іменами **int** та **double**. Знак «+», який ми використовуємо при виклику методу **WriteLine**, означає з'єднання (конкатенацію) двох послідовностей символів –

текстової константи у подвійних лапках та зображення у вигляді символів відповідної змінної.

Якщо в програмі декілька функцій **Main**, тобто передбачається декілька точок входу у програму, то виникне помилка компіляції. Для вирішення конфлікту потрібно в середовищі (або у командному рядку при компіляції) явно вказати, метод Main якого класу потрібно використати.

Змінимо цей приклад наступним чином.

```
/*  Це текст другої програми мовою C# */
using System;
class Program
{
   public static void Main()
   {
      Console.WriteLine("Перша програма" + "!!!");
      Console.WriteLine("Введіть ціле число");
      int i;
      i = int.Parse(Console.ReadLine());
      Console.WriteLine("i = " + i);
      Console.WriteLine("Введіть дійсне число ");
      double f;
      f = double.Parse(Console.ReadLine());
      Console.WriteLine("f = " + f);
      // Складаємо вираз
      f = Math.Sqrt(i * i + f * f);
      Console.WriteLine("Результат = " + f);
      f = Math.Tan(f);
      Console.WriteLine("Результат = " + f);
   }
}
```

Розглянемо, що змінилось в цьому прикладі порівняно з попереднім варіантом. Тут додались обчислення та виведення значень двох виразів. В першому визначається $\sqrt{i^2 + f^2}$, а в другому – $\mathrm{tg}(\sqrt{i^2 + f^2})$. Обчислення виразів відбувається завдяки звертанню до методів **Sqrt** та **Tan** класу **System.Math** , який містить велику кількість усталених математичних функцій та констант. Клас **Math** є статичним, тобто не можливо створити екземпляр цього класу. Для використання математичних функцій використовується наступний синтаксис звертання: **Math.<ім'я функції>**

Найбільш поширені математичні функції та сталі наведені у таблиці 2.1. (параметр **x** функцій має тип **double**).

Таблиця 2.1.

Основні математичні функції в C#

Abs(x)	модуль числа	**Log10(x)**	логарифм за основою 10
Acos(x)	arccos x	**Max(x,y)**	максимум з двох чисел
Asin(x)	arcsin x	**Min(x,y)**	мінімум з двох чисел
Atan(x)	arctg x	**Pi**	число пі
Ceiling(x)	округлення до більшого (англ. ceiling - стеля)	**Pow(x,y)**	піднесення x до степеня y
Cos(x)	cos x	**Round(x)**	округлення за звичайним правилом
Cosh(x)	ch x	**Sign(x)**	знак числа x
E	число Ейлера	**Sin(x)**	sin x
Exp(x)	експонента x	**Sinh(x)**	sh x
Floor(x)	округлення до меншого (англ. floor - підлога)	**Sqrt(x)**	квадратний корінь з x
Log(x)	натуральний логарифм, ln x	**Tan(x)**	tg x

Розділ 3. Основні елементи мови C#.

3.1. Основні вбудовані типи мови C#

Мова C# – це мова жорстокої типізації. Це означає, що кожний об’єкт в програмі має відноситись до одного з визначених типів. Всі типи даних, що використовуються в C#, діляться на 2 категорії типи-значень (***value types***) та типи-посилання (***reference types***). Про відмінності між ними говорилось раніше, а зараз розглянемо основні вбудовані типи мови C#. Вони представлені у таблиці 1.

Таблиця 3.1

Основні типи данних що використовуються в мові C#

	Назва типу	**Системна назва**	**Опис типу**	**Розмір у бітах**	**Діапазон значень**
1	**bool**	**Boolean**	логічний	8	false, true
2	**byte**	**Byte**	8-розрядний цілий без знаку	8	0 255
3	**sbyte**	**SByte**	8-розрядний знаковий цілий	8	-128 +127
4	**short**	**Int16**	короткий знаковий цілий	16	-32 768 +32787
5	**ushort**	**UInt16**	короткий цілий без знаку	16	0 65535
6	**int**	**Int32**	знаковий цілий	32	-2 147 483 648 +2 147 483 647
7	**uint**	**UInt32**	цілий без знаку	32	0 +4 294 967 295
8	**long**	**Int64**	довгий знаковий цілий	64	-9 223 372 036 854 775 808 +9 223 372 036 854 775 807
9	**ulong**	**UInt64**	довгий цілий без знаку	64	0 +18 448 744 073 709 5 51 615
10	**float**	**Single**	дійсний	32	1,401298E-45 3,402823E+38
11	**doubl e**	**Double**	дійсний подвоєної точності	64	E-324 E+308
12	**decim al**	**Decimal**	числовий для фінансових розрахунків	96	29 значущих розрядів
13	**char**	**Char**	символьний	16	
14	**string**	**String**	Набір символів Unicode		

3.2. Визначення та ініціалізація змінних, область їх видимості.

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

```
using System;
namespace Declaration_of_variables
{
   class Program
   {
      static void Main()
      {                    // визначення змінної f з ініціалізацією
         float f = 1.5F;
         char c;     // просто визначення змінної c
         int i = 0;
         bool b = true;
         decimal d = 1.5555555555555555555555555M;
         // визначення змінної x з динамічною ініціалізацією
         double x = Math.Sin(Math.PI / 3);
         Console.WriteLine(
         "f = {0} i = {1} x = {2} d = {3}", f, i, x, d);
         Console.WriteLine(b.ToString());
      }
   }
}
```

На екрані побачимо:

f = 1,5 i = 0 x = 0,866025403784439 d = 1,5555555555555555555555555

True

Зауваження.

1. Зверніть увагу, що при визначенні дійсних констант у програмі, зокрема 1.5, використовується десяткова крапка, а при зображенні їх на екрані, або при зчитуванні десяткових значень з клавіатури використовується десяткова кома (оскільки цей знак визначається по замовчуванню в залежності від локалізації поточного користувача, тобто, при російсько- або україномовній локалізації подільником буде кома).
2. Константа 1.5 у програмі сприймається компілятором як така, що має тип **double** , тому присвоєння цього значення змінній **f** типу **float** неможливе безпосередньо, для цього необхідно вжити специфікатор формату **float** – символ **F** (або **f**, наприклад **float x = 1.0f;**). Те саме стосується значення, яке присвоюється змінній **d** – відповідна константа помічається специфікатором формату **decimal**-символом **M** (наприклад, **decimal x = 1.5M;** див. детальніше про це далі).
3. При динамічній ініціалізації змінної їй може бути присвоєне лише те значення, яке відоме в цій точці програми.

При виводі на консоль можливо організувати форматування виводу, тобто одержати значення на екрані у зручному для сприйнятті вигляді. Для цього у

текстову константу – аргумент методу **Console.WriteLine** – треба помістити так звані плейсхолдери з номером потрібного елементу списку виводу. На місці кожного з таких плейсхолдерів на екрані з'явиться відповідне значення. Більше того, можливо використання форматів виводу. Повний синтаксис для визначення формату виводу: **{n, m : fk}**. Загальний вигляд плейсхолдеру такий Тут **n** означає порядковий номер елементу у списку виводу (нумерація починається з нуля), **m** – ширина поля виводу, **f** – символ специфікації формату, число **k** задає точність. Основні символи специфікацій формату наступні:

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

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

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

N або **n** – формат для виводу дійсних значень з відокремленням трійок

розрядів пробілами;

C або **c** – грошовий формат;

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

Повернемось до визначення змінних. До цього моменту всі змінні, які ми використовували, визначались у функції **Main**. Вони можуть використовуватись в будь-якій точці функції **Main**, починаючи з точки визначення – це і є їх область видимості. Проте можна визначати змінні всередині будь-якого блоку, тобто в області програми, обмеженої парою фігурних дужок. Такі змінні створюються, коли виконання програми доходить до даного блоку, і зникають, коли блок виконаний. Це забезпечує механізм інкапсуляції, тому що звернутись до такої змінної із зовнішніх по відношенню до даного блоку частин програми неможливо. Цей факт ілюструє наступний приклад.

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

```
            }
            // Помилка - змінна j тут вже не існує
            Console.WriteLine("j = " + j.ToString());
        }
    }
}
```

Зверніть увагу на назви прикладів. Ми намагаємось відбити зміст прикладу у його назві. Раніше вже згадувалось про культуру вибору ідентифікаторів для змінних. Те саме було застосоване тут.

3. Приведення типів.

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

```
using System;
namespace Convert_of_variables_1
{
    class Program
    {
        static void Main()
        {
            float f = 0;
            double x = f;       // таке присвоєння припустиме
            f = x;              // а таке – ні
            f = (float)x;       // явне приведення типу
            Console.WriteLine("f = " + f.ToString());
            Console.WriteLine("x = " + x.ToString());
        }
    }
```

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

(тип_до_якого_приводимо_вираз)вираз;

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

1. якщо один із операндів має тип **decimal** , то і другий буде приводитись до такого типу (але якщо другий операнд мав тип **float** або **double**, результат буде помилковим);
2. якщо один із операндів має тип **double**, то і другий буде приводитись до такого типу **double**;
3. якщо один із операндів має тип **float**, то і другий буде приводитись до типу **float**;
4. якщо один із операндів має тип **ulong**, то і другий буде приводитись до типу **ulong** (але якщо другий операнд мав цілий знаковий тип, результат буде помилковим);
5. якщо один із операндів має тип **long**, то і другий буде приводитись до типу **long**;
6. якщо один із операндів має тип **uint**, а другий має тип **sbyte, short** або **int** , то обидва операнди будуть приведені до типу **long**;
7. якщо один із операндів має тип **uint** , то і другий буде приводитись до типу **uint**;
8. **інакше обидва операнди перетворюються до типу int;**
9. Останнє правило пояснює, чому в наступному коді виникає помилка.

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

      }
   }
}
```

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

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

4. Літерали (константи) мови C#.

В мові C# ***літералом*** називається деяке фіксоване значення. Іншими словами це таке значення, яким можна ініціалізувати змінну, присвоїти константі тощо. Літерали можуть мати довільний допустимий ***тип значень***. Тип літерала визначається згідно певних правил. Цілий літерал не містить десяткової крапки чи знаку порядку. Автоматично відноситься до найменшого знакового цілого типу, починаючи із типу **int**, до множини значень якого входить літерал. Тобто літерали 25, -10 мають тип **int**, а

літерал 3 333 333 333 має тип **long**. Якщо потрібно віднести цілий літерал до іншого типу, треба це явно вказати, додавши один із суфіксів безпосередньо після літералу: символи U або u для літералу беззнакового типу, символи L або l для довгого цілого. Таким чином, 1L – це літерал типу **long**, 1U – типу **uint**, а 1UL – типу **ulong**.

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

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

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

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

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

Console.WriteLine("Це \tприклад \n\”ESC-послідовності\”);

на екрані ми побачимо:

Це   приклад

”ESC-послідовності”

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

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

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

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

Після запуску побачимо на екрані наступний текст:

Це

буквальний літерал, а це -

" " - подвійні лапки у ньому;

' '   - одинарні лапки у ньому

Розділ 5. Операції мови С#.

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

5.1. Арифметичні операції.

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

Зауваження.

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

5.2. Операції інкременту та декременту.

Унарні операції інкременту (++) та декременту (**--**) викликають відповідно збільшення або зменшення операнду дійсного або цілого типу на одиницю. Фактично, ці операції задають неявне присвоєння нового значення змінній. Цікавості та певної загадковості цим операторам надає можливість різного часу виконання. А саме, вони мають префіксну форму (коли знак операції ++ чи **--** записується **перед** змінною) та постфіксну форму (коли знак операції ++ чи **--** записується **після** змінної). В обох випадках змінна, до якої застосована операція інкреметну або декременту, буде збільшена або зменшена на одиницю, проте при префіксній формі ці зміни відбудуться перед використанням змінної у виразі, а при постфіксній формі – після використання змінної у виразі. Розглянемо приклад, що демонструє ці особливості.

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

В результаті на екрані побачимо :

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

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

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

5.3. Операції відношення (порівняння).

Бінарні операції відношення : (==) – перевірка на рівність, (!=) – перевірка на нерівність, (>) – перевірка чи більший перший операнд за другий, (>=) – перевірка чи не менший перший операнд за другий, (<) – перевірка чи менший перший операнд за другий, (<=) – перевірка чи не більший перший операнд за другий. Кожна з цих операцій формує значення **true** або **false** в залежності від результату порівняння. Мова C# дозволяє перевіряти на рівність або нерівність довільні об'єкти, тобто ці операції застосовні до всіх типів. Решту ж порівнянь можна виконувати лише над операндами тих типів, які підтримують операцію відношення порядку, тобто лише до числових типів. Розглянемо приклад:

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

Після запуску програми одержимо наступний результат:

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

5.4. Логічні операції.

Логічні операції виконуються над операндами логічного (булівського) типу та визначають результат логічного типу. Логічні операції мають дві форми: звичайну та скорочену. Розглянемо спочатку набір звичайних

логічних операцій: (**&**) – логічне множення або логічне «І», (**|**) – логічне додавання або логічне «АБО», (**^**) – логічне виключне «АБО», (**!**) – логічне заперечення «НІ». Значення логічних операцій наведені у таблиці 5.1. для різних значень операндів (**op1** та **op2**).

Таблиця 5.1.

Результати логічних операцій в залежності від операндів

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

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

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

5.5. Порозрядні (бітові) операції.

Порозрядні операції виконуються над відповідними бітами внутрішнього подання лише **цілих** операндів. Результатом виконання є ціле відповідного операндам типу. Таких операцій є 6 : (**&**) – порозрядне «І», тобто кон'юнкція

бітів, (|) – порозрядне «АБО»,тобто диз'юнкція бітів, (^) – порозрядне виключне «АБО», (>>) – порозрядний зсув праворуч, (<<) – порозрядний зсув ліворуч, (~) – порозрядне заперечення. Всі операції, крім останньої, є бінарними. Слід зауважити, що при зсуві ліворуч (<<) на вказану правим операндом кількість позицій внутрішнє подання лівого операнду просто зсувається ліворуч, а позиції, що вивільняються, заповнюються нулями. При зсуві праворуч (>>) позиції, що вивільняються ліворуч, заповнюються нулями для беззнакового операнду, якщо ж зсув виконується для знакового операнду, то на вільні ліві позиції розповсюджується знаковий розряд, тобто для від'ємного числа вільні позиції ліворуч заповняться одиницями. Це обов'язково слід враховувати. У таблиці 5.2. вказані результати бітових операцій для різних значень операндів (**op1** та **op2**).

Таблиця 5.1.

Результати побітових операцій в залежності від операндів

op1	**op2**	**op1 & op2**	**op1 | op2**	**op1 ^ op2**	**~op1**
1	**1**	**1**	**1**	**0**	**0**
1	**0**	**0**	**1**	**1**	**0**
0	**1**	**0**	**1**	**1**	**1**
0	**0**	**0**	**0**	**0**	**1**

Наступний приклад демонструє можливості операцій над бітами.

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

```
}
  }
```

Виконання цієї програми приведе до наступних результатів:

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

5.6. Умовна (тернарна) операція.

Умовна операція має наступну синтаксичну форму:

<вираз_1> ? < вираз_2> : < вираз_3>

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

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

Після виконання цього фрагменту на екрані побачите повідомлення :

максимум із 3 та 4 є 4

5.7. Операції присвоєння.

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

<змінна> = <змінна> <op> <вираз> ;

Наприклад, інструкція **x += 1;** еквівалентна наступній інструкції: **x = x + 1;**

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

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

5.8. Пріоритет операцій.

При складанні виразів слід враховувати, що порядок виконання операцій у виразі визначається пріоритетом операцій. Проте правилом хорошого тону слід вважати використання дужок, які підкреслюють порядок обчислень та роблять вираз більш зрозумілим. Наступна таблиця вказує основні операції мови C# в порядку спадання їх пріоритетів від найвищого до найнижчого.

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

Зауваження. Різницю у пріоритетах префіксних та постфіксних операцій інкременту-декременту можна побачити у наступному прикладі.

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

Тобто, вираз записаний у вигляді **z = x+++y;** був проінтерпретований згідно таблиці пріоритетів як

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

Завдання для самоконтролю

Написати програму у якій визначити необхідні змінні та записати у вигляді операції присвоєння наведену нижче формулу, обчисливши цей вира для різних вхідних значень.

1. Обчислити період коливання маятника заданої довжини.
2. Обчислити силу струму по відомим значенням напруги та опору електричного ланцюга.
3. Обчислити опір електричного ланцюга по відомим значенням напруги та сили струму.
4. Обчислити опір електричного ланцюга, що складається із трьох послідовно з'єднаних резисторів
5. Обчислити опір електричного ланцюга, що складається із двох паралельно з'єднаних резисторів.
6. Обчислити об'єм циліндру по відомим значенням висоти та радіуса основи.
7. Обчислити об'єм конуса по відомим значенням висоти та радіусу основи
8. Обчислити об'єм та площу поверхні кулі заданого радіуса.
9. Обчислити відстань, яку пробіг спринтер за певний час, якщо він рухається щ відомим прискоренням.
10. Обчислити час вільного падіння деякого тіла з відомої висоти та з заданою початковою швидкістю.
11. Перерахувати інтервалу часу, заданого у секундах у години, хвили на секунди(наприклад 3665 секунд = 1 год. 1хв. 5сек.)
12. Обчислення температури розчину води, одержаної при змішанні двох ємностей води при відомій температурі.
13. Обчислення периметру правильного n-кутника, описаного навколо кола заданого радіуса.
14. Обчислити час, через який зустрінуться два тіла, що рухаються рівноприскорено назустріч одному, якщо відомі їх початкові швидкості, прискорення та відстань між ними

Розділ 6. Основні інструкції керування мови C# – розгалуження та цикли

6.1. Розгалуження у мові C#

Для реалізації розгалужень (одного із трьох основних елементів програм) в мові C# використовуються конструкції **if** та **switch**.

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

if (<умова>)　　інструкція_1;
then　　　інструкція_2;

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

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

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

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

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

```
        }
    }
}
```

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

```
using System;
namespace Construct_if_else
{
    class Program
    {
        static void Main()
        {
            // Мінімальна сума для бонусу
            const decimal MINSUM = 10000;
            // Значення справжнього паролю
            const string RIGHT_PAROL = "myparol";
            string parol;
            decimal sum;
            Console.WriteLine("Введіть пароль");
            parol = Console.ReadLine();
            // Перевірка правильності паролю
            if (parol == RIGHT_PAROL)
            {   // Пароль правильний
                Console.WriteLine("Введіть суму внеску");
                sum = int.Parse(Console.ReadLine());
                if (sum > MINSUM)  // Перевіряємо суму внеску
                    Console.WriteLine(
                    "Вітаємо! Ви одержуєте бонус!");
                else
                    Console.WriteLine("Накопичуйте ще!");
            }
            else                // Пароль неправильний
                Console.WriteLine(
                "Ви ввели невірний пароль");
        }
    }
}
```

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

```
using System;
namespace Nested_if
```

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

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

Перевірка k == 10 дає false

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

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

```
            Console.WriteLine(
              "Неприпустимий знак операції");
        }
    }
}
```

Для більш зручної та компактної реалізації подібних розгалужень з багатьма альтернативами в мові C# використовується конструкція **switch**. Вона має наступний синтаксис.

switch (<вираз>)

{

case <константа_1> : <інструкції>

break;

case <константа_2> : <інструкції>

break;

…

case <константа_n> : <інструкції>

break;

default : <інструкції>

break;

}

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

Перепишемо попередню програму за допомогою конструкції **switch**.

```
using System;
namespace Construct_switch
{
   class Program
   {
      static void Main()
      {
         char op;
         int x = 10, y = 20;
         Console.WriteLine(
```

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

Зауваження.

1. В даному прикладі конструкція **switch** керувалась змінною символьного типу – це цілком припустимо, адже символьний тип є підмножиною цілого, оскільки містить коди символів.
2. Однією (або декількома) із інструкцій **switch** може бути вкладений **switch**.
3. Кожний з варіантів обов'язково повинен закінчуватися оператором **break,** який передає керування оператору, наступному за конструкцією **switch** (тобто означає вихід за межі поточної конструкції).
4. Якщо одну й ту саму групу інструкцій потрібно виконати для деякої множини значень констант, ці константи потрібно послідовно перелічити із службовими словами **case**. Наприклад, як у наступному прикладі програми:

```
using System;
namespace Construct_case_case
{
```

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

6.2. Цикли у мові C#

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

6.2.1. Цикл for

Синтаксис цього циклу наступний.

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

або

for (<вираз-ініціалізація>;<вираз-умова>;<вираз-ітерація>)

{

<група інструкцій>;

}

Виконання циклу **for** відбувається наступним чином.

1. Обчислюється **<вираз-ініціалізація>** .
2. Обчислюється **<вираз-умова>** . Якщо він має значення **false** , то дія циклу закінчується. Інакше виконується **<інструкція циклу>**.
3. Обчислюється **<вираз-ітерація>** .
4. Відбувається перехід до кроку 2.

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

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

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

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

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

У наступному прикладі розглядається ціле число типу **byte** (тип **byte** може бути замінений з відповідними виправленнями у програмі на довільний цілий ***беззнаковий*** тип – беззнаковий тому що використовується операція зсуву праворуч, про особливості якої ми говорили раніше). По результату порозрядного множення числа на маску **0X1**, яка у внутрішньому поданні є послідовністю нулів з одиницею лише в останньому розряді, встановлюємо

зміст останнього розряду числа. А оскільки треба «переглянути» всі розряди, то ми просто по черзі використовуємо порозрядний зсув числа праворуч, починаючи з найстаршого розряду, а отже, з максимального зсуву на **sizeof(byte)*8-1** позицій.

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

Зауваження.

1. Всі вирази у заголовку циклу не є обов'язковим і можуть бути пропущені. В разі відсутності **<виразу-умови>** його значенням вважається true. Таким чином можна записати «нескінчений» цикл:

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

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

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

6.2.2. Цикл while

Синтаксис цього циклу наступний.

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

Порядок виконання циклу наступний:

1. Обчислюється **<вираз-умова>**, значення якого має бути типу **bool**. Якщо значенням є **false** , то дія циклу закінчується. Інакше виконується **<інструкція циклу>**.
2. Після цього повторюється перший крок.

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

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

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

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

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

```
            {
                num /= 10;   // Зменшуємо число на один розряд
                count++;
            }
            Console.WriteLine("Кількість цифр = {0}", count);
        }
    }
}
```

6.2.3. Цикл do-while

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

do

{

<інструкції циклу>

} while (<вираз-умова >)

Порядок виконання циклу наступний:

1. Виконуються всі інструкції в тілі циклу.
2. Обчислюється **<вираз-умова>** , значення якого має бути типу **bool**. Якщо його значенням є **false** , то дія циклу закінчується. Інакше повторюється перший крок.
   Цикл **do-while** відрізняється від циклів **while** та **for** тим, що його інструкції виконуються ***завжди*** принаймні один раз.

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

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

```
        }
    }
}
```

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

```
using System;
namespace Construct_do_while
{
    class Program
    {        // Запит на повторний запуск програми
             // Погодження – натискання символу 1
        static void Main()
        {
            string answer;
            do
            {
                Console.WriteLine("Тут будуть інструкції програми ");
                Console.WriteLine("Продовжувати виконання? \n

                                 Для підтвердження натисніть 1");

                answer = Console.ReadLine();
            } while (answer == "1");
        }
    }
}
```

Зауваження.

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

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

Зазначимо ще, що враховуючи сказане вище, можна запропонувати певні правила вибору типу циклу для різних конкретних випадків, а саме:

1. якщо за логікою програми можлива ситуація, коли тіло циклу взагалі не виконується, краще використовувати цикл з передумовою (**while**);
2. якщо за логікою програми тіло циклу повинно бути виконане принаймні один раз, краще використовувати цикл з післяумовою (**do-while**);
3. якщо кількість повторень виконання тіла циклу відома заздалегідь, краще використовувати цикл **for**.

6.3. Керування виходом із циклів C#

Інколи виникає необхідність термінового переривання циклів. Таку логіку програми забезпечує інструкція **break;**, з якою ми познайомились у конструкції **switch**. Якщо всередині циклу знаходиться інструкція **break**;, то відбувається вихід із циклу, а управління передається інструкції, що безпосередньо слідує за даним циклом. Таким чином можна, наприклад перервати виконання «нескінченного» циклу.

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

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

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

Радикальну дію викликає інструкція безумовного переходу **goto**. Її вживання у програмах вкрай небажане (тому що воно порушує послідовну логіку виконання програми та ускладнює її налагодження) і може бути зумовлене лише крайньою необхідністю, наприклад, дострокового виходу із кількох вкладених циклів в разі виникнення помилки. Синтаксис цієї інструкції наступний:

```
goto <мітка>;
```

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

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

if (counter > 0) goto label;

Завдання для самоконтролю

1. Обчислити задану суму або добуток числового ряду для заданих початкових індексів nn та nk. Обов'язково перевіряти нерівностей *0≤nn≤nk.*

1. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{k^2-3}{(-1)^k k^2+5}$
2. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{k^2+(-1)^k k-1}{k^2+1}$
3. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{k^2-(-1)^{k+1} k^3}{k^2+k+1}$
4. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{(-1)^k k^2-1}{k^2+3}$
5. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{k^2-1}{(-1)^{k+1} k^2+7}$
6. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{(-1)^{k+1} k^2-2}{3k^2-2k}$
7. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{k^2+(-1)^{k-1} 2k-1}{k^2+8}$
8. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{k^2-3}{k^2-(-1)^k k+3}$
9. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{k^2-(-1)^{k+1} k}{k^2+2}$
10. $\prod_{k=nn}^{nk} a_k$, де $a_k = \frac{k^2-3-8k}{(-1)^k k^3+8}$
11. $\prod_{k=nn}^{nk} a_k$, де $a_k = \frac{(-1)^k k^3+k^2-3}{k^3+8}$
12. $\prod_{k=nn}^{nk} a_k$, де $a_k = \frac{k^3+k^2-k}{(-1)^k k^2-1}$
13. $\prod_{k=nn}^{nk} a_k$, де $a_k = \frac{(-1)^k k^3+(-1)^k k^2}{5k^3+5}$
14. $\prod_{k=nn}^{nk} a_k$, де $a_k = \frac{5k^3+5}{k^2+(-1)^{k-1} 2k-1}$
15. $\prod_{k=nn}^{nk} a_k$, де $a_k = \frac{(-1)^{k+1} k^2-2}{(-1)^k k^3+8}$

2. У всіх завданнях даного пункту потрібно вивести логічне значення True, якщо приведений вислів для запропонованих початкових даних є істинним, і значення False у супротивному випадку. Всі числа, для яких вказано кількість цифр (двозначне число, тризначне число і т.д.), вважаються цілими.

1. Перевірити істинність вислову: "Квадратне рівняння $A \cdot x^2 + B \cdot x + C = 0$ із даними коефіцієнтами A, B, C має дійсні корені".

2. Перевірити істинність вислову: "Дані числа x, y є координатами точки, що лежить у другому квадранті ".
3. Перевірити істинність вислову: "Дані числа x, y є координатами точки, що лежить у першому або третьому квадранті".
4. Перевірити істинність вислову: "Точка з координатами (x, y) лежить всередині прямокутника, ліва верхня вершина якого має координати (x1, y1), права нижня — (x2, y2), а сторони паралельні координатним осям".
5. Перевірити істинність вислову: "Дане ціле число є парним двозначним числом".
6. Перевірити істинність вислову: "Дане ціле число є непарним тризначним числом".
7. Перевірити істинність вислову: "Серед трьох даних цілих чисел є хоч би одна пара співпадаючих".
8. Перевірити істинність вислову: "Серед трьох даних цілих чисел є хоч би одна пара взаємно протилежних".
9. Перевірити істинність вислову: "Сума цифр даного тризначного числа є парним числом".
10. Перевірити істинність вислову: "Сума двох перших цифр даного чотиризначного числа рівна сумі двох його останніх цифр".
11. Перевірити істинність вислову: "Дане чотиризначне число читається однаково зліва направо і справа наліво".
12. Перевірити істинність вислову: "Всі цифри даного тризначного числа різні".
13. Перевірити істинність вислову: "Цифри даного тризначного числа утворюють зростаючу послідовність".
14. Перевірити істинність вислову: "Цифри даного тризначного числа утворюють зростаючу або спадаючу послідовність".
15. Перевірити істинність вислову: "Цифри даного тризначного числа утворюють арифметичну прогресію".
16. Перевірити істинність вислову: "Цифри даного тризначного числа утворюють геометричну прогресію".
17. Дані координати (як цілі від 1 до 8) двох різних полів шахівниці. Якщо тура за один хід може перейти з одного поля на інше, вивести логічне значення True, інакше вивести значення False.
18. Дані координати (як цілі від 1 до 8) двох різних полів шахівниці. Якщо король за один хід може перейти з одного поля на інше, вивести логічне значення True, інакше вивести значення False.
19. Дані координати (як цілі від 1 до 8) двох різних полів шахівниці. Якщо слон за один хід може перейти з одного поля на інше, вивести логічне значення True, інакше вивести значення False.
20. Дані координати (як цілі від 1 до 8) двох різних полів шахівниці. Якщо ферзь за один хід може перейти з одного поля на інше, вивести логічне значення True, інакше вивести значення False.
21. Дані координати (як цілі від 1 до 8) двох різних полів шахівниці. Якщо кінь за один хід може перейти з одного поля на інше, вивести логічне значення True, інакше вивести значення False.

22. Використовуючи техніку вкладених циклів намалювати за допомогою *, букви Ш, У, Х.
23. 4*. За допомогою циклів і розгалужень реалізувати простий калькулятор для дій(-,+,/, *), який буде враховувати пріорітет операцій.

Розділ 7. Масиви в мові C#.

7.1. Визначення та ініціалізація масиву.

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

Синтаксис визначення масиву наступний:

<тип_елементів> [] <ідентифікатор_масиву>;

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

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

<тип_елементів> [] <ідентифікатор_масиву> =

new <тип_елементів> [кількість_елементів];

Індексами елементів масиву будуть значення **0, 1, …, кількість_елементів-1**. (Зверніть особливу увагу на те, що індексація елементів починається з нуля!) Для ініціалізації масиву можна використати фігурні дужки, в яких перелічити значення його елементів. В цьому разі **кількість_елементів** при визначенні масиву можна не вказувати – компілятор визначить її по кількості ініціалізованих елементів. До елементів масиву звертаються, вказуючи ідентифікатор масиву та індекс потрібного елементу у квадратних дужках. Розглянемо приклади визначення масивів.

```
using System;
namespace Array_1
{
   class Program
   {
      static void Main()
      {      // Робота з масивом
         const int SIZE = 10;
         int[] iArray = new int[SIZE];     // Визначаємо масив
         for (int i = 0; i < SIZE; i++)
            // Елементи масиву ініціалізуються нулями!
            Console.WriteLine(
                     "iArray [{0}] = {1}", i, iArray[i]);
      }
   }
      }
```

Потрібно зазначити, що в мові C# на відміну від мови C або C++ у якості розміру масиву можна задавати звичайну змінну, проте, в даному випадку використання сталої є ознакою хорошого стилю програмуванням, це полегшує читання та подальше супроводження програми.

В даному прикладі масив був створений, але не проініціалізований. Тим не менше на екрані ми побачимо, що всі елементи масиву мають нульові значення – про це дбає компілятор, створюючи змінну ***reference***-типу. Проте, хорошим стилем у програмуванні вважаються явні ініціалізації. Зробимо це у наступному прикладі.

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

Зауваження.

1. В даному прикладі у рядку **int[] iArray = {1, 2, 3, 4, 5};** створюється масив цілих чисел **iArray**, який ініціалізується п’ятьма значеннями. Зверніть увагу, в цьому випадку службове слово **new** та вказання

кількості елементів не потрібно – компілятор автоматично визначає розмір масиву за кількістю ініціалізованих елементів.

2. У рядку **double[] dArray = new double[5] { 1.1, 2.2, 3.3, 4.4, 5.5 };** створюється та ініціалізується дійсний масив із 5 елементів, про що явно повідомляється компілятору. Ця вказівка в принципі є надмірною, хоча й синтаксично правильною. Але в разі явного визначення розміру масиву службове слово **new** є необхідним.
3. Інструкцією **dArray = new double[4] { 1.2, 2.3, 3.4, 4.5 };** для уже визначеного ідентифікатора **dArray** створюється та ініціалізується новий масив із чотирьох елементів.
4. Зручною формою для умови продовження циклу **for** є використання властивості **Length** , яка для будь-якого масиву визначає кількість елементі у ньому.
5. Зверніть увагу на інструкції :
   double[] dArray_new;

dArray_new = dArray;

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

Щоб підкреслити «об''єктну природу» масивів, розглянемо ще один приклад, що стосується зауваження 5. В ньому створюються, ініціалізуються та виводяться на екран 2 різних масиви. Потім ідентифікатору першого масиву присвоюється ідентифікатор другого. Фізично масиви знаходяться на своїх місцях в пам'яті, проте їх ідентифікатори посилаються тепер на одну й ту саму область, що ми і бачимо при виводі першого масиву. Посилання на перший масив тепер втрачене і в певний час він буде знищений з пам'яті спеціальною системою збору «сміття» **GC – Garbage Collector** (детальніше про це див. далі)

```
using System;
namespace Array_3
{
   class Program
   {
      static void Main(string[] args)
      {   // Визначаємо та ініціалізуємо перший масив
         int[] iArray1 = { 1, 2, 3, 4, 5 };
         Console.WriteLine("Перший масив");
         for (int i = 0; i < iArray1.Length; i++)
            Console.WriteLine(
                  "Елемент масиву [{0}] = {1}", i, iArray1[i]);
         Console.WriteLine("Другий масив");
         // Визначаємо та ініціалізуємо другий масив
         int[] iArray2 = { 6, 7, 8, 9, 10 };
         for (int i = 0; i < iArray2.Length; i++)
```

```
          Console.WriteLine(
              "Елемент масиву [{0}] = {1}", i, iArray2[i]);
        // Присвоєння ідентифікаторів масиву -
        // тепер перший ідентифікатор вказує на другий масив
        // посилання на перший масив - втрачене
        iArray1 = iArray2;
        Console.WriteLine("Ще раз другий масив");
        // Друкуємо другий масив ще раз
        for (int i = 0; i < iArray2.Length; i++)
          Console.WriteLine(
              "Елемент масиву [{0}] = {1}", i, iArray2[i]);
      }
    }
   }
```

Важливо наголосити також, що C# суворо контролює значення індексів елементів масиву. Якщо значення індексу виводить за межі виділеної для масиву області пам'яті – виникає помилка типу **«index out of range»**. Таким чином хорошим стилем програмування є використання конструкцій типу **try** ... **catch** для обмеження можливих випадків виходу за межі масиву.

7.2. Цикл foreach

Цикл **foreach** використовується для опитування елементів ***колекцій*** або масивів. Під колекцією в мові C# розуміють набір об'єктів, який задовольняє певним вимогам. Одним із видів колекцій є масив. Цикл **foreach** має наступний синтаксис.

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

В результаті виконання даного циклу **<змінна>** пробігає значення всіх елементів колекції, при цьому щоразу виконується **<інструкція>**. Очевидно, що в разі, коли цей цикл використовується для масивів, **<тип>** у циклі **foreach** має збігатись із типом елементів масиву. Ітераційна **<змінна>** циклу не може бути змінена, вона доступна лише для читання.

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

```
using System;
namespace Array_foreach
{
   class Program
   {
      static void Main(string[] args)
      {
         const int SIZE = 5;
         int[] iArray = new int[SIZE];   // Декларуємо  масив
                                 // Заповнюємо масив випадковими значеннями
         Random r = new Random(1);      // Декларація об'єкту
                                 // класу Random
         for (int i = 0; i < SIZE; i++)
         {      // Одержуємо випадкове число від 0 до 100
            iArray[i] = r.Next(100);
            Console.WriteLine(
                "Елемент масиву [{0}] = {1}", i, iArray[i]);
         }
         // Підсумовуємо елементи масиву та шукаємо мінімальний та
         // масксимальний елементи
         int sum = 0;
         int min = iArray[0];
         int max = iArray[0];
         // Переглядаються всі елементи масиву
         foreach (int elem in iArray)
         {
            sum += elem;
            if (elem < min) min = elem;
            if (elem > max) max = elem;
         }
         Console.WriteLine("Сума = {0}", sum);
         Console.WriteLine("Min = {0} Max = {1}", min, max);
      }
   }
}
```

7.3. Багатовимірні масиви.

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

<тип_елементів> [,] <ідентифікатор_масиву>;

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

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

```
using System;
namespace Array_two_dimensional_1
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            float[,] f_arr1;     // Декларація двовимірного масиву
                            // Ініціалізація двовимірного масиву розміру 2*3
            f_arr1 = new float[,] {
                        { 1, 2, 3 },
                        { 4, 5, 6 }
                        };
            foreach (float elem in f_arr1)  // Вивід першого масиву
            {
                Console.WriteLine(elem);
            }
            Random r = new Random();    // Створюємо об’єкт Random
                                // Створюємо інший двовимірний масив
            float[,] f_arr2 = new float[3, 4];
            for (int i = 0; i < f_arr2.GetLength(0); i++)
                for (int j = 0; j < f_arr2.GetLength(1); j++)
                    // Елементи-випадкові числа
                    f_arr2[i, j] = (float)r.NextDouble();
            // Вивід першого масиву у вигляді таблиці
            Console.WriteLine("Перший масив");
            for (int i = 0; i < f_arr1.GetLength(0); i++)
            {
                for (int j = 0; j < f_arr1.GetLength(1); j++)
                {
                    Console.Write(f_arr1[i, j]);
                    Console.Write("\t");
                }
                Console.WriteLine();
            }
            // Вивід другого масиву у вигляді таблиці
            Console.WriteLine("Другий масив");
            for (int i = 0; i < f_arr2.GetLength(0); i++)
            {   // Цикл по стовпчиках
                for (int j = 0; j < f_arr2.GetLength(1); j++)
                {
                    Console.Write(f_arr2[i, j]);
                    Console.Write("\t");    // Табуляція між елементами
                }
                Console.WriteLine();
            }
        }
    }
}
```

Зауваження.

1. Масив **f_arr1** ініціалізується двома наборами значень – для першого та другого рядків. Кожен з них розміщується у окремих фігурних дужках і відповідає окремому рядку масиву.
2. Зверніть увагу, як оформлюється вивід масивів у табличному вигляді – використовується метод **GetLength(dim)**, який повертає кількість

елементів масиву по вказаній вимірності **dim**. Тобто при **dim = 0** одержуємо кількість рядків, при **dim = 1** одержуємо кількість стовпчиків і т. д.

7.4. Використання деяких методів класу System.Array.

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

```
using System;
namespace Array_4
{
   class Program
   {
      static void Main(string[] args)
      {
         const int SIZE = 10;       // Розмір масивів
         int[] iArray = new int[SIZE];
         Console.WriteLine("Введіть {0} цілих чисел", SIZE);
         for (int i = 0; i < SIZE; i++)
         {
            Console.Write("[{0}] = ", i);
            iArray[i] = int.Parse(Console.ReadLine());
         }
         Console.WriteLine("Створюємо копію масиву:");
         // Копію необхідно привести до відповідного типу масиву
         int[] iCloneArray = (int[])iArray.Clone();
         Console.WriteLine("Сортуємо цю копію по зростанню");
         Array.Sort(iCloneArray);          // Сортування
         for (int i = 0; i < SIZE; i++)
            Console.WriteLine("iCloneArray [{0}] = {1}", i,
                        iCloneArray[i]);
         Console.WriteLine("Переставляємо елементи");
         Array.Reverse(iCloneArray);        // Перестановка
         for (int i = 0; i < SIZE; i++)
            Console.WriteLine("iCloneArray [{0}] = {1}", i,
                          iCloneArray[i]);
         Console.WriteLine("Зануляємо 5 елементів,

                        починаючи з індекса 3");

         Array.Clear(iCloneArray, 3, 5); // Занулення елементів
         for (int i = 0; i < SIZE; i++)
```

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

7.5. Масиви масивів. Непрямокутні масиви.

Можливо також створити масив, елементами якого є масиви різної довжини – так званий «ламаний» або «рваний» масив (хоча він і не є повністю сумісний з усіма стандартами технології .NET, але використання таких масивів все ж дозволяється). Для ілюстрації використання «ламаних» масивів розглянемо наступний приклад.

```
using System;
namespace Array_jagged
{
   class Program
   {
      static void Main()
      {  // Створюємо вказівник на масив із 3-х масивів
         int[][] jagArray = new int[3][];
         // Тепер створимо кожний із 3-х масивів
         for (int i = 0; i < jagArray.Length; i++)
         {
            jagArray[i] = new int[3 + i];
         }
         // Заповнюємо та виводимо масиви на екран трапецією
         for (int i = 0; i < jagArray.Length; i++)
         {
            Console.Write(
                  "Довжина рядку {0}: ", jagArray[i].Length);
            for (int j = 0; j < jagArray[i].Length; j++)
            {
               jagArray[i][j] = (i + 1) * (j + 1);
               Console.Write(" {0} ", jagArray[i][j]);
            }
            Console.WriteLine();
         }
      }
   }
}
```

У даному прикладі **jagArray** є ідентифікатором масиву із трьох елементів, кожний з яких є в свою чергу масивом відповідно із трьох, чотирьох та п’яти елементів. Вони створюються у циклі. Далі елементи цього масиву масивів заповнюються в залежності від значень індексів елементів та виводяться на екран у вигляді трапеції. Зверніть увагу, як відбувається звертання до елементів цього специфічного масиву. Ми використовуємо дві пари окремих квадратних дужок для кожного індексу **jagArray[i][j]**, а не одну, як у прикладі **Array_two_dimensional_1 – f_arr2[i, j]**.

Завдання для самоконтролю

Одновимірні масиви

1. Даний масив розміру N. Вивести його елементи в зворотному порядку.
2. Даний масив розміру N. Вивести спочатку його елементи з парними(непарними) індексами, а потім — з непарними (парними).
3. Даний масив цілих чисел A розміру N. Вивести номер першого (останнього) з тих його елементів A[i], які задовольняють подвійній нерівності: A[1]< A[i]< A[N]. Якщо таких елементів немає, то вивести 0.
4. Даний масив цілих чисел розміру N. Перетворити його, додавши до парних (непарних) чисел перший (останній) елемент. Перший і останній елементи масиву не змінювати.
5. Даний масив цілих чисел розміру N. Вивести спочатку всі його парні (непарні) елементи, а потім — непарні (парні).
6. Поміняти місцями мінімальний і максимальний елементи масиву розміру N.
7. Даний масив цілих чисел розміру N. Замінити всі додатні (від’ємні) елементи на значення мінімального (максимального) елементу.
8. Даний масив розміру N. Переставити в зворотному порядку елементи масиву, розташовані між його мінімальним і максимальним елементами.
9. Даний масив розміру N. Здійснити циклічний зсув елементів масиву ліворуч (праворуч) на одну позицію.
10. Даний масив розміру N і число k до (0 < k < N). Здійснити циклічне зсув елементів масиву ліворуч (праворуч) на k позицій.
11. Даний масив цілих чисел розміру N. Перевірити, чи утворюють його елементи арифметичну (|геометричну) прогресію. Якщо так, то вивести різницю (знаменник) прогресії, якщо ні — вивести 0.
12. Даний масив ненульових цілих чисел розміру N. Перевірити, чи чергуються в ньому 1)парні та непарні і 2) додатні і від ємні числа. Якщо чергуються, то вивести 0, якщо ні, то вивести номер першого елементу, що порушує закономірність.
13. Даний масив розміру N. Знайти кількість його локальних мінімумів (максимумів).
14. Даний масив розміру N. Знайти максимальний (мінімальний) з його локальних мінімумів (максимумів).
15. Даний масив розміру N. Визначити кількість інтервалів індексів, для яких його елементи монотонно зростають (спадають).
16. Даний масив розміру N. Визначити кількість його проміжків монотонності (інтервалів індексів, для яких його елементи монотонно зростають (спадають)).
17. Дано дійсне число R і масив розміру N. Знайти елемент масиву, який найближчий (найдальший) від даного числа.
18. Дано дійсне число R і масив розміру N. Знайти два елементи масиву, сума яких найближча (найдальша) від даного числа.
19. Даний масив розміру N. Знайти номери двох найближчих чисел з цього масиву.
20. Даний масив цілих чисел розміру N. Визначити максимальну кількість його однакових елементів.

21. Даний масив цілих чисел розміру N. Видалити з масиву всі елементи, що зустрічаються 1) менше двох разів; 2) більше двох разів; 3) рівно двічі; 4) рівно тричі.
22. Даний масив цілих чисел розміру N. Якщо він є перестановкою, тобто містить всі числа від 1 до N, то вивести 0, інакше вивести номер першого неприпустимого елементу.
23. Даний масив розміру N. Перетворити його, вставивши перед (після) кожного додатного (від'ємного) елементу нульовий елемент.
24. Даний масив цілих чисел розміру N. Назвемо серією групу однакових елементів, що розташовані підряд, а довжиною серії — кількість цих елементів (довжина серії може бути рівна 1). Вивести масив, що містить довжини всіх серій початкового масиву.
25. Даний масив цілих чисел розміру N. Перетворити масив, збільшивши (зменшивши) кожну його серію на один елемент.
26. Даний масив цілих чисел розміру N. Перетворити масив, збільшивши 1) першу; 2) останню 3) всі серії найбільшої довжини на один елемент.
27. Даний масив цілих чисел розміру N. Вставити 1) перед 2) після кожної серії нульовий елемент.
28. Дано ціле число k і масив цілих чисел розміру N. Видалити з масиву всі серії, довжина яких 1) менше; 2) рівна; 3) більша за k.
29. Дано ціле число k і масив цілих чисел розміру N. Замінити серію, довжина якої) менше; 2) рівна; 3) більша за k на один нульовий елемент.
30. Дані два масиви A і B розміру N, елементи яких впорядковані по зростанню (спаданню). Об'єднати ці масиви так, щоб результуючий масив залишився впорядкованим.
31. Впорядкувати масив розміру N по 1) зростанню; 2) спаданню .
32. Даний масив розміру N. Вивести індекси масиву в тому порядку, в якому відповідні їм елементи утворюють 1) зростаючу ; 2) спадаючу послідовність.
33. Дана точка A і множина B з N точок. Знайти номер точки з множини B, найбільш 1) близької; 2) віддаленої від точки A.
34. Дано множину A з N точок. Серед всіх точок цієї множини, що лежать у 1) першій; 2) другій; 3) третій; 4) четвертій чверті, знайти точку, найбільш наближену (віддалену) від початку координат. Якщо таких точок немає, то вивести точку з нульовими координатами.
35. Дано множину A з N точок. Знайти пару різних точок цієї множини з 1) мінімальною; 2) |максимальною відстанню між ними і саму цю відстань (точки виводяться в тому ж порядку, в якому вони перелічені при завданні множини A).
36. Дано множину A з N точок. Знайти таку точку з даної множини, сума відстаней від якої до решти точок 1) мінімальна; 2) максимальна і саму цю суму.
37. Дано множини A і B, що складаються відповідно з N1 і N2 точок. Знайти 1) мінімальну; 2) |максимальну відстань між точками цих множин і самі точки, розташовані на цій відстані.
38. Дано множину A з N точок. Знайти 1) найменший1; 2) найбільший периметр трикутника, вершини якого належать різним точкам множини A,

і самі ці точки (точки виводяться в тому ж порядку, в якому вони перелічені при завданні множини А).

39. Дано множину А з N точок з цілими координатами. Порядок на координатній площині визначимо таким чином: (x1,y1) < (x2,y2), якщо або x1 < x2, або x1 = x2 і y1 < y2. Розташувати точки даної множини по 1) спаданню; 2) зростанню відповідно згідно указаного порядку.

Двовимірні масиви

1. Дано число k (0 < k < max(m, n)) і матриця розміру m * n. Знайти суму і добуток елементів к-го стовпчика даної матриці.
2. Дана матриця розміру m * n. Знайти суми елементів всіх її 1) парних; 2) непарних строк (стовпчиків).
3. Дана матриця розміру m * n. Знайти 1) мінімальне; 2) максимальне значення в кожному рядку (стовпчику).
4. Дана матриця розміру m * n. У кожному рядку (стовпчику) знайти кількість елементів, 1) більших 2) менших середнього арифметичного всіх елементів цього рядка (стовпчика).
5. Дана матриця розміру m * n. Перетворити матрицю, помінявши місцями мінімальний і максимальний елемент в кожному рядку (стовпчику).
6. Дана матриця розміру m * n. Знайти 1) мінімальне; 2) максимальне значення серед сум елементів всіх її рядків (стовпчиків) і номер рядка (стовпчика) з цим мінімальним або |максимальним значенням.
7. Дана матриця розміру m * n. Знайти 1) мінімальний; 2) максимальний серед 1) максимальних; 2) мінімальних елементів кожного рядка (стовпчика).
8. Дана матриця цілих чисел розміру m * n. Вивести номер її 1) останнього; 2) першого рядка (стовпчика), що містить рівну кількість додатних і від'ємних елементів (нульові елементи не враховуються). Якщо таких рядків (стовпчиків) немає, то вивести 0.
9. Дана матриця розміру m * n. Вивести номер її 1) першого; 2) останнього рядка (стовпчика), що містить тільки додатні елементи. Якщо таких рядків (стовпчиків) немає, то вивести 0.
10. Дана матриця цілих чисел розміру M x N. Різні рядки (стовпчики) матриці назвемо схожими, якщо співпадає множина чисел, що зустрічаються в цих рядках (стовпчиках). Знайти кількість рядків (стовпчиків), схожих на 1) перший; 2) останній рядок (стовпчик).
11. Дана матриця цілих чисел розміру M x N. Знайти кількість її рядків (стовпчиків), всі елементи яких різні.
12. Дана матриця цілих чисел розміру M x N. Вивести номер її 1) першого; 2) останнього рядка (стовпчика), що містить максимальну кількість однакових елементів.
13. Дана квадратна матриця порядку M. Знайти суму елементів її 1) головної; 2) побічної діагоналей.

14. Дана квадратна матриця порядку M. Знайти суми елементів її діагоналей, паралельних 1) головній; 2) |побічній (починаючи з одноелементної діагоналі 1) A[1,M]; 2) A[1,1]).
15. Дана квадратна матриця порядку M. Вивести 1) мінімальні; 2) максимальні з елементів кожної її діагоналі, паралельних 3) головній; 3) побічній (починаючи з одноелементної діагоналі 1) A[1,M]; 2) A[1,1]).
16. Дана квадратна матриця порядку M. Замінити нулями елементи матриці, що лежать 1) нижче 2) вище 3) головної; 4) побічної діагоналі.
17. Дана квадратна матриця порядку M. Дзеркально відобразити її елементи відносно 1) горизонтальної осі симетрії; 2) вертикальної осі симетрії; 3) головної діагоналі; 4) побічної діагоналі матриці.
18. Дана квадратна матриця порядку M. Повернути її елементи на 1) 90 2) 180; 3) 270 градусів в додатному напрямку.
19. Дана матриця розміру m * n. Вивести кількість 1) рядків; 2) стовпчиків, елементи яких монотонно зростають (спадають).
20. Дана матриця розміру m * n. Знайти 1) мінімальний; 2) максимальний серед елементів тих рядків (стовпчиків) матриці, які впорядковані або за зростанням, або за спаданням. Якщо такі рядки (стовпчики) відсутні, то вивести 0.
21. Дані два числа k_1 і k_2 і матриця розміру 4 х 10. Поміняти місцями рядки (стовпчики) матриці з номерами k_1 і k_2.
22. Дана матриця розміру m * n. Поміняти місцями рядки (стовпчики), матриці, що містять мінімальний і максимальний елементи матриці.
23. Дана матриця розміру m * n. Поміняти місцями стовпчик з номером 1 та 1) перший; 2) останній із стовпчиків, що містять тільки додатні елементи.
24. Дано число k і матриця розміру m * n . Видалити рядок (стовпчик) матриці з номером k.
25. Дана матриця розміру m * n. Видалити рядок (стовпчик), що містить 1) мінімальний; 2) максимальний елемент матриці.
26. Дана матриця розміру m * n. Видалити 1) перший; 2) останній; 3) всі стовпчики, що містять тільки додатні елементи.
27. Дано число k і матриця розміру m * n. 1) Перед; 2) після рядком (стовпчиком) матриці з номером k вставити рядок (стовпчик) з нулів.
28. Дана матриця розміру m * n. Продублювати рядок (стовпчик) матриці, що містить її 1) мінімальний; 2) максимальний елемент.
29. Дана матриця розміру m * n. 1) Перед; 2) після першим (останнім) стовпчиком, що містить тільки додатні елементи, додати стовпчик, що складається з одиниць.
30. Дана матриця цілих чисел розміру M x N. Знайти елемент, що є максимальним в своєму рядку і мінімальним в своєму стовпчику. Якщо такий елемент відсутній, то вивести 0.
31. Дана матриця розміру M x N. Елемент називається локальним мінімумом (максимумом), якщо він менший (більший) за всі сусідні з ним елементи. Замінити всі локальні 1) мінімуми ; 2) максимуми даної матриці на 0.
32. Дана матриця розміру M x N. Поміняти місцями її 1) рядки; 2) стовпчики так, щоб їх 3) мінімальні; 4) максимальні елементи утворювали зростаючу (спадаючу) послідовність.

Розділ 8. Структуровані типи даних (колекції) в мові C#

8.1. Основні структури даних та їх призначення

Часто виникає програмна необхідність у певній структуризації інформації та організації методів доступу до таких структур даних. Програміст, наприклад, може мати справу із переліком назв деяких товарів, або з більш складними наборами даних. Використання масивів в цьому випадку може бути зручним підходом, проте, по-перше, звертання до елементу за індексом не завжди зручне (наприклад, товари треба вибирати за ціною), по-друге масиви мають один дуже суттєвий недолік – їх розмір фіксований і у випадках, коли кількість елементів наперед невідома, необхідно створювати масив іншого розміру перед зміною кількості елементів. Таку операцію можна виконувати наступним чином:

```
using System;
namespace LabDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         const int NEWSIZE = 10;   // кількість елементів
         int[] arr1 = { 1, 2, 3, 4, 5 };
         // створюємо та ініціалізуємо новий масив
         int[] arr2 = new int[NEWSIZE];
         for (int i = 0; i < arr1.Length; i++)
         { arr2[i] = arr1[i]; }
      }
   }
}
```

За допомогою вбудованого статичного методу **Copy()** класу **System.Array** зробити все набагато простіше та правильніше, як показано у наступному прикладі. Параметри методу **Copy()**наступні: вихідний масив; індекс вихідного масиву, з якого починається копіювання; масив, у який відбувається копіювання; індекс масиву-призначення у який відбудеться копіювання та кількість елементів для копіювання.

```
using System;
namespace LabDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         const int NEWSIZE = 10; // кількість елементів
         int[] arr1 = { 1, 2, 3, 4, 5 };
         // створюємо та ініціалізуємо новий масив
         int[] arr2 = new int[NEWSIZE];
         // копіюємо arr1.Length елементів масиву arr1 у масив
         // arr2, починаючи з індексу 0
         Array.Copy(arr1, 0, arr2, 0, arr1.Length);
      }
   }
    }
```

Отже масив є найпростішою уже знайомою нам структурою даних. Крім масивів для організації різноманітних структур інформації в мові C# існують спеціальні класи об’єктів, що спрощують роботу із структурами даних зі змінною кількістю елементів. Для таких структур даних використовується термін «колекція» - це набір об’єктів, всі елементів якого можуть бути перебрані по черзі. Для ідентифікації кожного елементу використовується унікальне значення деякого ключа – у найпростішому випадку це може бути просто ціле число, власне, аналог індексу у масиві. Існують два основних підходи до організації структур даних: коли всі дані є довільними об’єктами (мають тип **object** – такі класи з’явились в .NET версії 1.1) та коли дані є типізованими (узагальнені колекції (Generic), що з’явились їм на зміну в версії .NET 2.0).

Нижче будуть розглянуті 4 типи структур даних:

1. Набір даних із цілими ключами
   1) **ArrayList** – масив-список, елементи такої структури даних належать до типу **object**.
   2) **List<T>** – типізований масив-список, всі елементи такої структури даних належать до типу **T**.
2. Набір даних у вигляді пар значення-ключ із нецілими ключами
   1) **Hashtable** – дані та ключі даних є довільними (належать до типу **object**)
   2) **Dictionary <T1,T2>** – дані та ключі є типізованими і належать до типів **T1** та **T2**.

8.2. Використання списку ArrayList та узагальненого списку List

Узагальнений список типу **ArrayList** (цей клас знаходиться в просторі імен **System.Collections**) можна використовувати для організації даних довільних типів. Всі дані, що містяться в цій колекції, мають тип **System.Object**, від якого успадковані всі типи даних. Розглянемо приклад роботи з цим класом

```
using System;
using System.Collections;
namespace LabDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         // ініціалізуємо перелік
         ArrayList al = new ArrayList();
         // поки що колекція порожня, але до неї можна додати
         // довільні об'єкти
         al.Add("Slon"); // додали рядок
                     // додамо елементи масиву
         al.AddRange(new int[] { 1, 2, 3, 4 });
         // додамо цілий масив як один елемент
         al.Add(new int[] { 5, 6, 7, 8 });
         // дані можна вставляти після довільної позиції,
```

```
        // наприклад, після третьої (адже першою є позиція 0!)
        al.Insert(2, "Giraf");
        // отримати дані можна з допомогою циклу foreach
        foreach (object o in al)
        { Console.WriteLine(o); }
        Console.WriteLine("============");
        // елементи можна видаляти за значенням
        al.Remove(3);  // видалили елемент із значенням 3
                    // елементи можна видаляти також за індексом
        al.RemoveAt(0); // видалили початковий елемент
                    //елементи можна перебрати з допомогою звичайного циклу for
        for (int i = 0; i < al.Count; i++)
        {
          // до елементів можна звертатись за номером індексу
          Console.WriteLine(al[i]);
        }
        // елементи завжди можна перетворити на звичайний масив
        object[] ob = al.ToArray();    // маємо масив об'єктів
      }
    }
}
```

В результаті виконання цієї програми на екрані побачимо: **Slon 1**

Зауваження:

1. Поточна кількість елементів в колекціях типу **ArrayList** та **List<T>** визначається властивістю **Count**. Кількість елементів, для збереження яких виділена пам'ять, визначається властивістю **Capacity**.
2. До колекцій типу **ArrayList** та **List<T>** можна додавати елементи по одному в кінець (метод **Add**) та всередину (метод **Insert**).
3. До колекцій типу **ArrayList** та **List<T>** можна додавати групу елементів разом (масивом) в кінець колекції (метод **AddRange**) та всередину (метод **InsertRange**).
4. З колекцій типу **ArrayList** та **List<T>** можна видаляти елементи за індексом (метод **RemoveAt**) або за значенням (метод **Remove**). Потрібно зазначити, що видалення елементу, який відноситься до reference-типу, відбувається за порівнянням посилань, тобто так, як працює оператор ==.
5. Колекції типу **ArrayList** та **List<T>** можна перетворити на масив за допомогою виклику методу **ToArray** (у випадку нетипізованої колекції метод поверне масив **object []**, інакше повертається масив того типу об'єктів, з типу якого складена колекція).

Але для отримання значення з переліку **ArrayList** потрібно виконати явне перетворення типів вигляду:

int i = (int) al[2];

або

int i = Convert.ToInt32(al[3]);

Такі операції суттєво уповільнюють виконання програми, і тому на зміну колекції **ArrayList** прийшов клас **List<T>**. Він знаходиться у просторі імен

System.Collections.Generic і реалізує так звані типізовані колекції, тобто колекції, в яких тип елементів задається на етапі компіляції. Всі дані, що включаються до цієї колекції, перевіряються на сумісність ще на етапі компіляції. Це зменшує кількість помилок при розробці програми та пришвидшує виконання самої програми. Розглянемо приклад роботи із колекцією **List<T>**.

```
using System;
using System.Collections.Generic;
namespace LabDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         // працюємо з типізованим списком
         List<int> l = new List<int>();
         l.Add(10);
         l.Add('c');
         l.Add((int)4L);
         foreach (int i in l)
         {
            Console.WriteLine(i);
         }
      }
   }
}
```

Ще раз підкреслимо, що при отриманні значень з нетипізованої колекції, швидкість значно уповільнюється, і тому варто при можливості завжди використовувати типізовані колекції. Для перевірки швидкості роботи із різними видами структур даних, розглянемо наступний приклад.

```
using System;
using System.Collections;
namespace UseColletionSpeed
{
   // оцінка швидкодії сортування колекції ArrayList
   class Program
   {
      static void Main(string[] args)
      {
         // декларуємо колекцію для збереження даних
         ArrayList a = new ArrayList();
         Random r = new Random();    // генератор випадкових чисел
         DateTime t1 = DateTime.Now; // початковий час
                              // наповнюємо ArrayList 4 мільйонами елементів
         for (int i = 0; i < 4000000; i++)
         {
            a.Add(r.Next());
         }
         a.Sort();   // виконуємо сортування елементів колекції
         DateTime t2 = DateTime.Now; // кінцевий час
                              // інтервал часу між початком та завершенням роботи
         TimeSpan s = t2.Subtract(t1);
```

```
            // виводимо результат
            Console.WriteLine(s.Seconds * 1000 + s.Milliseconds);
        }
    }
}
```

Тест показав 7593 мілісекунди. Змінемо цю програму для використання типізованої колекції наступним чином:

```
using System;
using System.Collections.Generic;
namespace UseColletionSpeed
{
    // оцінка швидкодії сортування колекції List< >
    class Program
    {
        static void Main(string[] args)
        {
            // декларуємо колекцію для збереження даних
            List<int> a = new List<int>();
            Random r = new Random();    // генератор випадкових чисел
            DateTime t1 = DateTime.Now; // початковий час
                            // наповнюємо List< > 4 мільйонами елементів
            for (int i = 0; i < 4000000; i++)
            {
                a.Add(r.Next());
            }
            a.Sort();   // виконуємо сортування елементів колекції
            DateTime t2 = DateTime.Now; // кінцевий час
                            // інтервал часу між початком та завершенням роботи
            TimeSpan s = t2.Subtract(t1);
            // виводимо результат
            Console.WriteLine(s.Seconds * 1000 + s.Milliseconds);
        }
    }
}
```

Тепер інтервал часу зменшиться до 1091 мілісекунди на тому ж самому обладнанні. Оцінимо, як зміниться інтервал часу роботи при переході від типізованої колекції до звичайного масиву у наступному прикладі:

```
using System;
namespace UseColletionSpeed
{
    // оцінка швидкодії сортування масиву
    class Program
    {
        static void Main(string[] args)
        {
            // декларуємо масив для збереження
            int[] a = new int[4000000];
            Random r = new Random();    // генератор випадкових чисел
            DateTime t1 = DateTime.Now; // початковий час
                            // наповнюємо масив 4 мільйонами елементів
            for (int i = 0; i < 4000000; i++)
```

```
            {
                a[i] = r.Next();
            }
            Array.Sort(a);  // виконуємо сортування елементів масиву
            DateTime t2 = DateTime.Now; // кінцевий час
                                 // інтервал часу між початком та завершенням роботи
            TimeSpan s = t2.Subtract(t1);
            // виводимо результат
            Console.WriteLine(s.Seconds * 1000 + s.Milliseconds);
        }
    }
}
```

Тут швидкість програми майже не змінилась (890 мілісекунд).

Таким чином, можна зробити висновок, що при передачі даних між фрагментами програми для збільшення швидкодії корисно перетворювати колекції на масиви з допомогою методу **ToArray**. Зазначимо, що насправді колекції містять саме масиви (це можна перевірити за допомогою .NET рефлектору).

8.3. Використання асоційованого списку Hashtable та узагальненого словника Dictionary

Іншою групою методів є списки, ключами в яких не є цілі числа. Ці колекції призначені для збереження інформаційного значення, асоційованого з певним ключом (у вигляді пар ключ-значення). Унікальність гарантується за ключом.

Подібні структури даних часто використовуються в професійних програмах для створення пар «об'єкт-дія» або «об'єкт-атрибут». Перший приклад показує використання класу Hashtable:

```
using System;
// простір імен для використання Hashtable
using System.Collections;
namespace UsingHashTable
{
    class Program
    {
        static void Main(string[] args)
        {
            // створюємо нетипізований асоційований список
            Hashtable ht = new Hashtable();
            // ключ та значення можуть бути довільними
            ht[new int[] { 1, 2, 3 }] = 25;
            ht[DateTime.Now] = "arbuz";
            ht[1.2f] = 10;
            // єдиним шляхом отримати дані є використання
            // колекції ключів або значень
            foreach (object o in ht.Keys)
            {
                Console.WriteLine("{0} ===> {1} ", o, ht[o]);
            }
```

```
        }
    }
}
```
Другий приклад ілюструє використання класу **Dictionary:**
```
using System;
// простір імен для використання Dictionary
using System.Collections.Generic;
namespace UsingHashTable
{
    class Program
    {
        static void Main(string[] args)
        {
            // створюємо нетипізований асоційований список
            Dictionary<string, decimal> d =
                    new Dictionary<string, decimal>();
            // компілятор перевіряє відповідність типів ключа та значення
            d["Ivanov"] = 185;
            d["Petrov"] = 180;
            // єдиним шляхом отримати дані є використання
            // колекції ключів або значень
            foreach (string s in d.Keys)
            {
                Console.WriteLine("{0} ===> {1} ", s, d[s]);
            }
        }
    }
}
```
<u>Зауваження до обох прикладів</u>:

1. І для типізованого і для нетипізованого асоційованих списків існує колекція ключів **Keys**.
2. І для типізованого і для нетипізованого асоційованих списків існує колекція значень **Values**.
3. Отримання значення в обох типах списків за певним ключем відбувається за допомогою оператора **[ ]**.
4. допомогою методу **ContainsKey**.
5. Перевірка наявності значення в обох типах списків відбувається за допомогою методу **ContainsValue**.
6. При додаванні нових значень елементів до колекції типу Hashtable перевірка відповідності типу значення та типу ключа відбуваються на етапі компіляції. Перевірка наявності ключа в обох типах списків відбувається за

Завдання для самоконтролю

1.Створити ліст об’єктів, дозволити можливість заповнення з клавіатури і вивести кількість об’єктів кожного типу(Char, String, Int, Doble, Bool).

2.Створити ліст стрінгових змінних, дозволити можливість заповнення з калвіатури. Вивести індекси змінних рівних перевірочній(теж ввести з клавіатури). Скопіювати ліст в масив.

3.Створити ліст чарових змінних, дозволити можливість заповнення з калвіатури. Вивести індекси змінних рівних перевірочній(теж ввести з клавіатури). Скопіювати ліст в масив.

4.Створити ліст інтових змінних, дозволити можливість заповнення з калвіатури. Вивести індекси змінних рівних перевірочній(теж ввести з клавіатури). Скопіювати ліст в масив.

5.Створити ліст даблових змінних, дозволити можливість заповнення з калвіатури. Вивести індекси змінних рівних перевірочній(теж ввести з клавіатури). Скопіювати ліст в масив.

6.Створити ліст інтових змінних, який може вміщати тільки 1 і 0, заповнити його з клавіатури. Порахувати кількість 1 і 0. Скопіювати отриманий ліст в масив.

7.Створити ліст інтових змінних, який може містити тільки прості числа. Порахувати кількість повторень кожного простого числа. Скопіювати отриманий ліст в масив.

8.Створити ліст стрінгових значень, дозволити можливість заповнення з клавіатури. Вивести кількість елементів рівної двожини.

9.Створити ліст інтових значень, який може вміщати тільки 2 та 1, вивести серії повторень, для кожного чисел. Видалити елементи з простими індексами.

10.Створити ліст інтових значень 0-9, дозволити введення з клавіатури. Порахувати кількість входження кожного числа в ліст. Створити масив, в якому індекс буде відповідати числу, а кількість входження цього числа – змінній на цьому індексі.

11.Зчитати число з клавіатури та записати його в ліст, розбивши по порядку на на цифри. В отриманому лісті порахувати кількість повторень кожної цифри. Перевірити чи серед цих чисел є задане.

12.Створити ліст інтових значень, дозволити заповнення з клавіатури. Знайти суму всіх простих елементів і різницю парних. Порахувати середнє арифметичне всіх елементів.

Розділ 9. Класи і методи в мові C#.

Концепція ООП реалізована в мові C# у формі класів. Клас – це тип даних, що містить власне дані та методи для маніпулювання ними. Специфікація класу дозволяє створювати об'єкти, які реалізують даний клас. Їх називають екземплярами класу або просто об'єктами. Похідні класи можуть спадкувати свої властивості від базових класів, підтримуючи їх інтерфейс, завдяки поліморфізму.

Насправді, всі елементи програми мовою C#, про які вже йшла мова: константи, змінні, блоки операторів, методи, тощо, – можуть існувати лише в рамках якогось класу. До цього часу ми використовували лише той клас, що створювався для нового проекту автоматично.

9.1. Визначення класу.

Визначення класу описує його члени, в першу чергу дані та методи – функції, які обробляють та керують цими даними. Клас може містити деякі спеціальні члени – статичні поля та методи, властивості, події, тощо. Для початку познайомимось із деякими основними членами класу. Синтаксис визначення класу вимагає використання службового слова **class**, за яким слідує власне блок визначення членів класу.

Отже, синтаксис визначення класу наступний (взагалі порядок визначення членів класу може бути довільним):

```
class <ідентифікатор_класу> {
// декларації даних-членів класу
<специфікатор_доступу> <тип> <ідентифікатор_змінної_1>;
<специфікатор_доступу> <тип> <ідентифікатор_змінної_2>;
//...
// декларації методів-членів класу
<специфікатор_доступу> <тип_результату>
   <ідентифікатор_методу_1> (<параметри>)
{
// код методу
}
//...
<специфікатор_доступу> <тип_результату>
   <ідентифікатор_методу_N> (<параметри>)
{
```

```
// код методу
}
}
```

Зауваження. <Специфікатор_доступу> визначає право використання даного члену класу поза його межами і може мати наступні значення: **public** – для відкритих членів класу, **protected** та **private** – для закритих членів класу (різницю між ними буде з’ясовано дещо пізніше). Відкриті члени класу можуть використовуватись в усіх фрагментах програмного коду, яким доступне визначення класу. Закриті члени класу доступні лише членам самого класу. Якщо специфікатор доступу не вказаний, він матиме значення **private** за замовчуванням.

Отже, визначення класу – це визначення нового типу даних (***reference-type***), а об’єктом є конкретний екземпляр даного класу, подібно до того, як тип **int** є одним із стандартних типів даних, а змінна **val** типу **int** – це конкретний екземпляр (об’єкт) цілого типу. Для створення екземпляру даного класу необхідно використати оператор **new** . Розглянемо деякі приклади.

```
using System;
namespace Class_Student
{
   class Student
   {                  // Відкриті дані-члени класу
      public string name;
      public string surname;
      public byte course;
      public byte group;
      public double module_mark;
   }
   class Program
   {
      static void Main()
      {
         Student s1, s2;  // декларація двох екземплярів Student
         s1 = new Student();     // створюємо першого студента
         s2 = new Student();     // створюємо ще одного студента
         s1.name = "Oleg";   // заповнюємо дані першого студента
         s1.surname = "Petrenko";
         s1.course = 1;
         s1.group = 1;
         s1.module_mark = 30;
         s2.name = "Maria";  // заповнюємо дані другого студента
         s2.surname = "Pushkina";
         s2.course = 2;
         s2.group = 1;
         s2.module_mark = 60;
         Console.WriteLine("Перший студент:");
         Console.WriteLine(
         "{0} (1} {2} курс {3} група {4} балів", s1.name,
```

```
        s1.surname, s1.course, s1.group, s1.module_mark);
        Console.WriteLine("Другий студент:");
        Console.WriteLine(
        "{0} (1} {2} курс {3} група {4} балів", s2.name,
        s2.surname, s2.course, s2.group, s2.module_mark);
      }
   }
}
```

Проанализуємо уважно цей приклад. Програмний код включає 2 класи – **Student** та **Program**. Перший з них містить 5 змінних – **name, surname, course, group** та **module_mark**. Це дані-члени класу **Student** і оскільки всі вони визначені із специфікатором **public**, доступ до них можливий із інших класів програми. У класі **Program**, точніше у його єдиній основній функціїї **Main**, спочатку декларуються, а потім і створюються 2 екземпляри класу **Student** – **s1** та **s2** з допомогою оператору **new**. Кожний з цих об’єктів має свій власний набір даних **name, surname, course, group** та **module_mark**. Їх називають даними екземпляру. Доступ до даних конкретного екземпляру відбувається із вказанням обох ідентифікаторів: і екземпляру, і члену класу, відокремлених крапкою. Власне, такі «складені імена» ми вже використовували, наприклад, при звертанні до методів **ReadLine** або **WriteLine** класу **Console**. В операторі «крапка» ліворуч завжди вказується ідентифікатор об’єкту, а праворуч – ідентифікатор члену цього об’єкту. Таким чином, далі всі члени об’єктів **s1** та **s2** одержують певні значення, які і виводяться на екран. Проте в цьому першому прикладі наш класс **Student** є «пасивним», адже члени даного класу використовуються відкрито «без участі та нагляду» класу.

9.2. Методи класу.

Додамо в клас методи у наступному прикладі. В ньому визначений клас, що містить інформацію про точку на площині. Точка задається своїми полярними координатами, проте завдяки методам класу можна одержати значення її координат у декартовій системі.

```
using System;
namespace Polar_Point
{
   class Polar_Point          // клас - полярна точка
   {      // дані-члени класу - полярні радіус та кут
      public double r, phi;
      // методи-члени класу
      public double xCoord()  // абсциса полярної точки
      {
         return r * Math.Cos(phi);
      }
      public double yCoord()  // ордината полярної точки
      {
         return r * Math.Sin(phi);
      }
   }
```

```
    class Program
    {
        static void Main()
        {      // створення екземпляру полярної точки
            Polar_Point p1 = new Polar_Point();
            p1.r = 10;      // задання полярного радіусу
            p1.phi = Math.PI * 0.25;  // задання полярного кута
                            // Друкуємо абсцису полярної точки
            Console.WriteLine("абсциса = {0}", p1.xCoord());
            // Друкуємо ординату полярної точки
            Console.WriteLine("ордината = {0}", p1.yCoord());
        }
    }
}
```

Клас **Polar_Point** містить 2 змінні дійсного типу: r та **phi** – це відкриті (**public**) дані-члени класу, а також 2 методи-члени класу **xCoord()** і **yCoord()** – це функції, що повертають дійсні значення декартової абсциси та ординати полярної точки. Повернення результату ці методи здійснюють завдяки інструкції

return <вираз>;

Тип виразу має збігатись із типом, вказаним у визначенні даного методу в класі. Якщо деякий метод не повинен повертати ніякого результату при його визначенні вказується службове слово **void**, яке означає «порожній». В тілі такого методу інструкція **return** відсутня. Проте її можна додати для переривання виконання методу за якоюсь умовою.

У класі **Program** створюється екземпляр **p1** класу **Polar_Point** та здійснюється доступ до його даних **r** та **phi**, а потім відбувається звертання до його методів **xCoord()** і **yCoord()**.

Зверніть увагу на програмний рядок:

Polar_Point p1 = new Polar_Point();

У ньому визначається змінна **p1** reference-типу, яка відразу ініціалізується адресою області пам'яті, виділеної під об'єкт (екземпляр) **p1** класу **Polar_Point**. Той самий результат можна було одержати за два кроки подібно до попереднього прикладу:

// декларація полярної точки

p1 = new Polar_Point();// створення екземпляру полярної точки

Може виникнути слушне питання: чому при створенні змінної **p1** операцією **new** поруч з іменем класу **Polar_Point** використовуються круглі дужки так, ніби відбувається звертання до методу? Справа у тому, що насправді для створення екземпляру класу автоматично викликається спеціальний метод, одноіменний з класом – так званий конструктор класу, про який говоритимемо далі. Зверніть увагу також, що звертання до методів класу

Polar_Point відбувається з допомогою операції доступу «крапка» – спочатку ідентифікатор екземпляру класу, потім ідентифікатор члену класу: **p1.xCoord()** або **p1.yCoord().**

Зауваження. Зовсім не обов'язково визначення класу **Polar_Point** розміщувати у одному файлі із класом **Program** – він може знаходитись в окремому файлі проекту. Одним із способів створення окремого файлу для декларації класу є наступний. Перемістіть курсор на назву проекту **Polar_Point** попереднього прикладу у вікні Solution Explorer (див малюнок екрану). Натиснувши на праву кнопку миші, виберіть команду Add -> New Item. У вікні «Add New Item» виберіть піктограму «Class». Вам залишилось лише заповнити поле Name для назви файлу класу та натиснути на кнопку Add. Відповідний файл у проекті буде для вас створений. Помістіть у нього визначення класу **Polar_Point** та використовуйте цей клас для створення об'єктів та роботи з ними у файлі **Program.cs**.

9.3. Методи з параметрами.

У прикладах класів створені нами методи мали порожній список параметрів, хоча за синтаксисом будь-який метод може використовувати параметри, які дозволяють йому спілкуватись з іншими частинами програми, зокрема, одержуючи необхідну для роботи вхідну інформацію або навпаки – повертаючи оброблену інформацію. Детальну розмову про використання параметрів у функціях матимемо пізніше, а зараз зауважимо, що у більшості звертань до методів C# ми використовували параметри: наприклад, звертаючись до методів **Console.WriteLine** або **Math.Cos** чи **Math.Sin**, ми записували у дужках аргументи для їх виклику. Спробуємо наділити наші методи класів параметрами.

Перш за все, нагадаємо (про це згадувалось вище), що передача аргументів у функції може відбуватись двома способами: ***за значенням*** (call-by-value) або ***за посиланням*** (call-by-reference). У першому випадку для аргументу, що передається у функцію, у стеку створюється копія, в яку записується значення аргументу. Таким чином, функція може використовувати значення аргументу, проте не має можливості вплинути на оригінал. У другому випадку функція одержує посилання на місце розташування аргументу у пам'яті, тому зміни у функції цього аргументу призводять до реального впливу на нього.

Ситуація дещо ускладнюється, коли у функцію передається змінна-посилання, яка вже сама містить адресу певного об'єкту, розміщеного у динамічній пам'яті (наприклад, масиву). Якщо змінна-посилання сама передається за посиланням, то ми можемо як змінити об'єкт, на який вказує ця змінна, так і змінити саму змінну-посилання (наприклад, «переставити» її на інший об'єкт або перетворити її значення на **null**), як цього і слід було чекати. Проте, якщо змінна-посилання передається у функцію за значенням,

то функція має справу з копією посилання – тобто не може вплинути на оригінал самого посилання, але має прямий доступ до об’єкта або значення, на яке вказує посилання, а отже, може змінювати цей об’єкт або значення. Якщо у функцію передається аргумент, що за своєю природою має reference-тип, зрозуміло, що це передача за посиланням, адже функція одержує адресну змінну, отже, має доступ до самого аргументу. Якщо ж у функцію передається аргумент value-типу, то він потрапляє у неї за значенням (насправді існує можливість передати його і в інший спосіб). У списку параметрів декларації функції вони перелічуються через кому із вказанням типу кожного з параметрів.

У наступному прикладі метод класу має три параметри дійсного, цілого та символьного типів, які використовуються для ініціалізації даних членів цього класу.

```
using System;
namespace Param_Metod
{
   class OurClass
   {
      double x;  // Всі дані-члени цього класу - закриті!
      int i;
      char c;
      // Відкритий метод для ініціалізції закритих даних класу
      // x_, i_, c_ - параметри методу OurMetod
      public void OurMetod(double x_, int i_, char c_)
      {
         x = x_;
         i = i_;
         c = c_;
      }
      public double Get_x()
      { return x; }
      public int Get_i()
      { return i; }
      public char Get_c()
      { return c; }
   }
   class Program
   {
      static void Main()
      {
         OurClass cl = new OurClass();
         cl.x = 1;        // Помилка: x – закритий член класу
         cl.i = 1;        // Помилка: i - закритий член класу
         cl.c = 'A';      // Помилка: c - закритий член класу
                    // Ініціалізуємо закриті члени класу:
                    // Аргументи 1, 1, 'A' копіюються у x_, i_, c_
         cl.OurMetod(1, 1, 'A');
         Console.WriteLine("x = {0} i = {1} c = {2}",
                 cl.Get_x(), cl.Get_i(), cl.Get_c());
      }
```

```
    }
}
```

В цьому прикладі класс **OurClass** містить 3 члени класу, які не специфіковані, а отже, мають специфікацію **private** за замовчуванням. Тому спроба звернутись до цих змінних зовні спричинить помилку (закритий коментарем рядок **cl.x = 1**; і наступні за ним). Доступ до таких членів класу можливий лише через відкритий метод цього ж класу. У прикладі в цій ролі виступає функція **OurMetod**. Її список включає 3 параметри **x_** типу **double**, **i_** типу **int**, та **c_** типу **char**. У виклику цього методу **cl.OurMetod(1, 1, 'A')** вказані відповідно аргументи **1, 1, 'A'**. Кожен з них копіюється у нову змінну, створену в момент виклику методу у стеку, у строгій відповідності до порядку формальних параметрів у списку функції **OurMetod**. Решта методів класу **OurClass** необхідна, оскільки вони повертають значення закритих членів класу, тобто дають змогу зовнішнім функціям користуватись цими значеннями, проте не дають змоги змінити самі члени класу.

Те, що ми зробили в даному прикладі, є першим наближенням до реалізації механізму інкапсуляції даних у класах – дані захищені, а їх значення можуть використовуватись завдяки відкритим методам класу. В ідеалі всі дані-члени класу мають бути організовані із специфікацією **private**, щоб запобігти їх пошкодженню зовні і повністю контролювати доступ до них.

9.4. Конструктор класу.

Те, що було зроблено у прикладах з класами **Student** та **Polar_Point**, коли дані екземпляру задавались «вручну» припустимо лише для прикладу, адже з одного боку можливо «забути» проініціалізувати деяку змінну екземпляру, а далі використати її (за замовчуванням проініціалізовану компілятором!), з іншого боку хотілося б взагалі мінімізувати «ручне» маніпулювання при створенні екземплярів класів. Іншими словами роботу по створенню та ініціалізації екземплярів треба перекласти на компілятор. Тому метод, подібний до **OurMetod**, має викликатись автоматично при створенні кожного екземпляру класу **OurClass**. Такий метод класу називається ***конструктором класу***. Синтаксис його декларації наступний:

<специфікатор_доступу> <ідентифікатор_класу> (<параметри конструктора>)

{

// код конструктора

}

Наприклад, конструктор для класу з останнього прикладу міг би бути наступним:

public OurClass(double x_, int i_, char c_)

```
{
   x = x_;
   i = i_;
   c = c_;
}
```

Зазвичай конструктору встановлюють специфікатор **public**, оскільки екземпляри створюються поза межами класу, отже, конструктор має бути відкритим. Зверніть увагу, ідентифікатор конструктора збігається з ідентифікатором класу, а тип результату у нього – відсутній, не вказується навіть службове слово **void**. Оскільки в нашому прикладі конструктор ініціалізує 3 даних-члени класу **OurClass**, список його параметрів складається з трьох. При створенні екземпляру класу **OurClass** тепер необхідно вказати в дужках список із трьох аргументів, якими конструктор проініціалізує дані-члени свого об’єкту. Пригадаймо, що раніше, коли ми ще не обговорювали конструктори, при створенні екземпляру після імені класу ми писали круглі дужки. Тепер стає зрозумілим зміст такого синтаксису. Адже конструктор для класу викликається при створенні екземпляру незалежно від того, визначений у класі конструктор, чи ні. Просто в останньому випадку спрацьовує так званий конструктор за замовчуванням, який не має параметрів та присвоює відповідні нульові значення всім членам класу value-типу та значення null (нульовий вказівник) членам класу reference-типу. Проте, як тільки у класі створений явний конструктор, конструктор за замовчуванням стає недоступним. Перетворимо попередній приклад та проаналізуємо результат роботи програми.

```
using System;
namespace Param_Metod
{
   class OurClass
   {
      double x;    // Всі дані-члени цього класу - закриті!
      int i;
      char c;
      // Конструктор ініціалізує дані-члени класу значеннями
      // аргументів; x_, i_, c_ - параметри конструктора OurClass
      public OurClass(double x_, int i_, char c_)
      {
         x = x_;
         i = i_;
         c = c_;
      }
      public double Get_x()
      { return x; }
      public int Get_i()
      { return i; }
      public char Get_c()
```

```
        { return c; }
    }
    class Program
    {
        static void Main()
        {   // Екземпляр створює конструктор з аргументами
            OurClass cl = new OurClass(1, 1, 'A');
            Console.WriteLine("x = {0}\ti = {1}\tc = {2}",
                cl.Get_x(), cl.Get_i(), (char)cl.Get_c());
            // Інший екземпляр з іншими аргументами
            OurClass another_cl = new OurClass(1.5, 10, 'Z');
            Console.WriteLine(
              "x = {0}\ti = {1}\tc = {2}", another_cl.Get_x(),
              another_cl.Get_i(), another_cl.Get_c());
            // Тепер створити екземпляр без аргументів неможливо!
            OurClass bad_cl = new OurClass();// Помилка!
        }
    }
}
```

Підведемо підсумки.

1. Кожен клас має конструктор – метод одноіменний з класом, для якого не вказується тип результату. Конструктор автоматично викликається в момент створення екземпляру (об'єкту) класу. Аргументи для конструктора вказуються у круглих дужках.
2. Якщо клас не містить явно визначеного конструктора, то викликається конструктор за замовчуванням (by default) з порожнім списком аргументів. Він зануляє дані-члени класу.
3. Якщо в класі явно визначений конструктор, то конструктор за замовчуванням компілятором не використовується.
4. Безпосередньо викликати конструктор неможливо – це прерогатива компілятора, який звертається до конструктора при створенні об'єкту класу.

9.5. Передача об'єктів методам.

Раніше вже було наголошено, що звичайні змінні value-типу передаються у метод за значенням, а змінні reference-типу – за посиланням. Також обговорювались відмінності передачі через список параметрів методу змінних-значень та змінних-посилань. Для детальнішого розуміння цієї різниці розглянемо наступний простий приклад. В ньому визначається класс **MyClass**, який містить цілу змінну **i** та конструктор, що її ініціалізує значенням свого параметру. У класі **Program** визначені 2 функції – **MyFunc1** та **MyFunc2**, перша з яких збільшує на одиницю свій цілочислений параметр, а друга – інкрементує член класу **MyClass** . У функції **Main** змінна **i** ініціалізується значенням 10, а також створюється екземпляр класу **MyClass**, змінна i в якому має те саме значення 10.

```
using System;
namespace Object_Param
{
   class MyClass
   {
      public int i;          // Просто ціла змінна
      public MyClass(int i_)  // Конструктор класу
      { i = i_; }
   }
   class Program
   { // Функція одержує параметр by-value
      static void MyFunc1(int val)
      { val++; }
      // Функція одержує об'єкт by-referebce
      static void MyFunc2(MyClass m)
      { m.i++; }
      static void Main()
      {
         int i = 10;      // Ціла змінна із значенням 10
         MyClass m = new MyClass(10);
         Console.WriteLine("Перед MyFunc1: ");
         Console.WriteLine("Змінна i = " + i);  // Друкуємо i
         MyFunc1(i);     // У функції MyFunc1 i збільшилась
         Console.WriteLine("Після MyFunc1: ");
         // Але тут i не змінилась!
         Console.WriteLine("Змінна i = " + i);
         Console.WriteLine("Перед MyFunc2: ");
         Console.WriteLine("Змінна i = " + m.i);// Друкуємо i
         MyFunc2(m);    // У функції MyFunc2 i збільшилась
                     // Тут i дійсно збільшилась
         Console.WriteLine("Після MyFunc2: ");
         Console.WriteLine("Змінна i = " + m.i);
      }
   }
}
```

Після запуску програми одержуємо наступний результат.

Перед MyFunc1:

Змінна i = 10

Після MyFunc1:

Змінна i = 10

Перед MyFunc2:

Змінна i = 10

Після MyFunc2

Змінна i = 11

Цей результат показує, що функція **MyFunc1**, яка одержала параметр i за значенням, не мала доступу до самого аргументу **i**, використовуючи лише

його значення 10, на відміну від функції **MyFunc2**, яка одержала об'єкт за посиланням, а отже всі зміни, виконані функцією над своїм параметром, реально відбулись з об'єктом. Причина цього полягає у тому, що параметр **m** функції **MyFunc2** є посиланням на об'єкт **MyClass**. І хоча ***сам аргумент m передається у функцію за значенням***, проте його значення – це адреса розташування екземпляра **MyClass** у пам'яті, а отже, функція має доступ до членів цього об'єкту.

Щоб підкреслити важливість виділеної у попередньому абзаці фразі, розберемо ще один приклад. В цьому прикладі функція **MyFun1** на перший погляд міняє місцями 2 об'єкти. Точніше кажучи, вказівники на два об'єкти. У функції ж **MyFun2** міняються місцями значення членів двох об'єктів класу **MyClass**. З'ясуємо, як зміняться значення об'єктів **autumn** та **winter** – екземплярів класу **MyClass** після викликів цих функцій.

```
using System;
namespace Object_Param_1
{
   class MyClass
   {
      public string text;
      public MyClass(string s)      // Конструктор класу
      { text = s; }
   }
   class Program
   {
      static void MyFun1(MyClass m1, MyClass m2)
      {      // Переставляємо об'єкти
         MyClass m = m1;
         m1 = m2;
         m2 = m;
      }
      static void MyFun2(MyClass m1, MyClass m2)
      {      // Переставляємо змінні об'єктів
         string s = m1.text;
         m1.text = m2.text;
         m2.text = s;
      }
      static void Main()
      {
         MyClass autumn = new MyClass("осінь");
         MyClass winter = new MyClass("зима");
         MyFun1(autumn, winter);      // Чи поміняялись об'єкти?
         Console.WriteLine("Після MyFun1:");
         Console.WriteLine(autumn.text);  // Це зима? - Ні!
         Console.WriteLine(winter.text);  // Це осінь? - Ні!
                              // Чи помінявся зміст об'єктів?
         MyFun2(autumn, winter);
         Console.WriteLine("Після MyFun2:");
         Console.WriteLine(autumn.text);  // Це зима? - Так!
         Console.WriteLine(winter.text);  // Це осінь? - Так!
      }
```

```
    }
}
```
Після запуску програми на екрані побачимо:

Після MyFun1:

осінь

зима

Після MyFun2:

зима

осінь

Цей приклад ще раз демонструє, що одержавши за значенням вказівники на об’єкти, функції мають змогу змінювати їх зміст, проте не мають змоги змінити самі вказівники.

9.6. Використання модифікаторів для параметрів методів.

Для того, щоб надати методам можливість змінювати аргументи, які передаються їм за значенням, або повертати значення через список параметрів, досить використати спеціальні модифікатори. Вони вказуються перед параметром, що має бути зміненим, у визначенні методу та перед відповідним аргументом у виклику методу. Таку роль відіграють модифікатори **ref** та **out**. Наступна таблиця описує модифікатори параметрів.

Таблиця 9.1.

Роль модифікаторів ref та out

Модифікатор	***Призначення***
Відсутній	Параметр вважається ***вхідним***, тобто передається методу ***за значенням***
out	Параметр вважається ***вихідним***, тобто визначається всередині методу; параметр передається ***за посиланням***
ref	Параметр вважається ***вхідним-вихідним***, тобто його значення, що передається методу, може бути змінене; параметр передається ***за посиланням***

Нижче розглянемо два приклади, що ілюструють використання модифікаторів параметрів методів. У першому прикладі функція **NumOfDigit** (її результат помічений як **void**) використовує два параметри: перший без модифікатора – вхідний параметр **n** , другий з модифікатором **out** – вихідний параметр **num**, після завершення роботи функції містить кількість цифр заданого числа **n**. (Схожий приклад, проте з іншою специфікацією функції

уже розглядався). Зверніть увагу, що і у виклику функції модифікатор **out** має передувати відповідному аргументу.

```
using System;
namespace Param_out
{
   class Program
   {  // параметр n - вхідний, параметр num – вихідний (out)
      static void NumOfDigit(ulong n, out byte num)
      {
         num = 0;
         while (n != 0)
         {
            n /= 10;
            num++;
         }
      }
      static void Main()
      {
         Console.WriteLine("Введіть невід'ємне ціле число");
         ulong n = ulong.Parse(Console.ReadLine());
         byte num;
         // у виклику повторюється модифікатор out
         NumOfDigit(n, out num);
         Console.WriteLine(
           "Кількість цифр у числі {0} - {1}", n, num);
      }
   }
}
```

У другому прикладі функція **Cube** одержує дійсний параметр **x**, обчислює значення його кубу та повертає це значення у тому самому параметрі **x**. Цей параметр помічений модифікатором **ref**, причому модифікатор має з'явитись і у виклику функції, інакше виникне синтаксична помилка.

```
using System;
namespace Param_ref
{
   class Program
   {   // параметр x - вхідний/вихідний (ref)
      static void Cube(ref double x)
      { x = x * x * x; }
      static void Main()
      {
         Console.WriteLine("Введіть дійсне число");
         double x = double.Parse(Console.ReadLine());
         Cube(ref x); // у виклику повторюється модифікатор ref
         Console.WriteLine("Куб вашого числа : {0}", x);
      }
   }
}
```

Зауваження. Важливо зазначити принципову різницю між параметрами з модифікаторами **out** та **ref**. Аргумент, що передається методу на місці

вихідного параметра із модифікатором **out** не повинен ініціалізуватись перед викликом методу, адже його значення визначається у методі. Аргумент, що передається в ролі параметру з модифікатором **ref**, ***обов'язково повинен бути проініціалізованим*** перед викликом методу.

Повернемось до прикладу **Object_Param_1**. Тепер зрозуміло, як треба змінити функцію **MyFun1**, щоб вона реально міняла місцями об'єкти, що їй передаються. Якщо використати в цьому прикладі наступне визначення **MyFun1**,

```
static void MyFun1(ref MyClass m1, ref MyClass m2)
{       // Переставляємо об'єкти
  MyClass m = m1;
  m1 = m2;
  m2 = m;
}
```

результат виконання прикладу **Object_Param_1** буде наступним:

Після MyFun1:

зима

осінь

Тобто, завдяки передачі за посиланням (модифікатор **ref**), функція **MyFun1** має змогу змінювати самі об'єкти!

Розмова про модифікатори параметрів методів була б неповною, без згадки про ще один з них – модифікатор **params** вказує, що далі слідує список однотипних параметрів невизначеної довжини (можливо, нульової!). Для ілюстрації використання цього модифікатору, розберемо наступний приклад.

```
using System;
namespace Param_params
{
   class Program
   {       // Метод має змінну кількість параметрів
      static float Summator(params float[] array)
      {
         Console.WriteLine("Маємо {0} аргументів у списку",
                        array.Length);
         float sum = 0F;
         foreach (float elem in array)
            sum += elem;
         return sum;
      }
      static void Main()
      {
```

```
            Console.WriteLine(
                "Результат: {0}", Summator(100, 200, 300));
        }
    }
}
```

В цьому прикладі функція **Summator** одержує довільну кількість дійсних параметрів, про що свідчить модифікатор **params** перед масивом параметрів. Метод **Summator** обчислює суму переданих йому у виклику чисел і повертає її як свій результат. Наслідком виклику функції **Summator**, записаного у прикладі, буде повідомлення:

Маємо 3 аргументів у списку сумування

Результат: 600

Якщо рядок виклику змінити на наступний:

Console.WriteLine("Результат: {0}", Summator ());

то матимемо такий результат:

Маємо 0 аргументів у списку сумування

Результат: 0

Тобто дана функція може працювати з будь-якою кількістю дійсних параметрів, крім того, їй можна передати в ролі аргументу масив дійсних значень.

Зауваження.

1. На відміну від модифікаторів **out** та **ref** модифікатор **params** ***у виклику*** функції ***відсутній***.
2. Параметр з модифікатором **params** у декларації методу виглядає як масив – тобто всі елементи списку **params** ***повинні мати один тип*** (проте будь-який!).
3. Параметр з модифікатором **params** у декларації методу повинен бути ***останнім*** серед параметрів методу – отже, два таких параметри для одного методу неможливі.

9.7. Методи, що повертають об'єкти.

Методи класів мови C# можуть повертати результати практично довільних типів. Зокрема результатом методу може бути і екземпляр класу, і масив (вірніше, посилання на відповідні об'єкти), що неможливо для функцій багатьох інших мов програмування. Для наступного прикладу вдосконалимо клас **Polar_Point** з попереднього розділу, додавши у нього конструктор з двома параметрами для ініціалізації членів класу. Отже, створимо окремий файл проекту **Polar_Point.cs**, що містить визначення класу :

```
using System;
```

```
namespace Object_result
{
   class Polar_Point          // клас - полярна точка
   {
      //дані-члени класу - полярні радіус та кут
      public double r, phi;
      public Polar_Point(double r_, double phi_) // конструктор
      {
         r = r_;
         phi = phi_;
      }
      public double xCoord()  // абсциса полярної точки
      { return r * Math.Cos(phi); }
      public double yCoord()  // ордината полярної точки
      { return r * Math.Sin(phi); }
   }
}
```

У файл **Program.cs** помістимо наступний код, в якому визначається метод **Symetry**. Він створює точку, симетричну відносно полярної осі для полярної точки **p**, заданої як його параметр. Цю точку метод **Symetry** повертає як свій результат. Далі у функції **Main** для точки з полярними координатами 10 та 2*PI/3 створюється симетрична точка **p_new** як результат методу **Symetry**.

```
using System;
namespace Object_result
{
   class Program
   {   // Цей метод має результатом клас - він повертає об'єкт
      static Polar_Point Symetry(Polar_Point p)
      {
         Polar_Point temp = new Polar_Point(p.r, -p.phi);
         return temp;
      }
      static void Main()
      {
         Polar_Point p = new Polar_Point(10, Math.PI * 2 / 3);
         Console.WriteLine("Координати старої точки: {0} {1} ",
                         p.xCoord(), p.yCoord());
         // Об'єкт p_new створюється та повертається методом Symetry
         Polar_Point p_new = Symetry(p);
         Console.WriteLine("Координати нової точки: {0} {1} ",
                         p_new.xCoord(), p_new.yCoord());
      }
   }
}
```

На екрані побачимо наступний результат

Координати старої точки:

-5 8,66025403784439

Координати нової точки:

-5 -8,66025403784439

І на завершення розглянемо приклад методу, результатом якого є посилання на масив. Цей метод для довільного цілого параметра повертає масив його десяткових цифр.

```
using System;
namespace Array_result
{
   class Program
   {
      static int[] ArrayDigits(long n)
      {
         long temp = n;
         byte count = 0;
         while (temp != 0)  // Визначаємо кількість цифр у числі
         {
            temp /= 10;
            count++;
         }
         // Створюємо масив для цифр числа
         int[] digits = new int[count];
         if (n < 0) n = -n;// Для від’ємного числа змінюємо знак
         for (int i = 0; i < count; i++)
         {        // Визначаємо масив цифр числа
            digits[count - 1 - i] = (int)(n % 10);
            n /= 10;
         }
         return digits; // Повертаємо масив цифр числа
      }
      static void Main()
      {
         Console.WriteLine("Введіть ціле число");
         long n = long.Parse(Console.ReadLine());
         // Масив dig створюється методом ArrayDigits
         int[] dig = ArrayDigits(n);
         Console.WriteLine("Цифри вашого числа:");
         if (dig.Length == 0) Console.WriteLine(0);
         else
            for (int i = 0; i < dig.Length; i++)
               Console.Write("{0} ", dig[i]);
         Console.WriteLine();
      }
   }
}
```

9.8. Перевантаження методів.

Мова C# дозволяє реалізувати цікаву можливість при створенні методів – різні за змістом методи можуть мати однакові ідентифікатори. Ця технологія називається ***перевантаженням*** (***overloading***) методів. В програмуванні вживається термін «***сигнатура методу***» – сигнатура включає ідентифікатор методу та список його параметрів. Перевантаження дозволене для методів з різними сигнатурами. Тобто два або більше методів можуть мати однакові ідентифікатори при умові, що їх списки параметрів різняться або кількістю, або типами, або і тим, і іншим. Зверніть увагу, що результат, який

повертається методом, до сигнатури не включається, отже, при перевантаженні методам недостатньо мати відмінності лише в типі результату. Якщо виникає питання про корисність такої можливості, то зауважимо, що однією із поширених студентських помилок при використання стандартної функції мови С++ для обчислення модуля числа є виклик функції **abs(x)**, що повертає ціле значення модуля цілого числа x, в той час, коли необхідне звертання до функції **fabs(x)** з дійсними аргументом та результатом. А якщо згадати, що є ще функція **labs(x)** з довгим цілим результатом… Зрозуміло, що значно корисніше було б мати методи однакового призначення з однаковими ідентифікаторами для різних типів параметрів.

Розглянемо приклад, в якому перевантажені 4 методи **MyMethod** – з параметрами типів **int, float, double** та з двома цілими параметрами.

```
using System;
namespace Overloading
{
   class Program
   {// В цьому класі перезавантажуються методи
      public void MyMethod(int x)
      {
         Console.WriteLine("Метод з цілим параметром x = {0}", x);
      }
      public void MyMethod(float x)
      {
         Console.WriteLine("Метод з дійсним параметром x ={0}", x);
      }
      public void MyMethod(double x)
      {
         Console.WriteLine(
           "Метод з параметром подвоєної точності x = {0}", x);
      }
      public void MyMethod(int x, int y)
      {
         Console.WriteLine("Метод з двома цілими параметрами

                       x = { 0}
         y = { 1}
         ", x, y);
      }
      static void Main()
      {
         Program pr = new Program();
         pr.MyMethod(1);
         pr.MyMethod(1.5);
         pr.MyMethod(2.5F);
         pr.MyMethod(1, 2);
      }
   }
}
```

Виконання цієї програми показує, що кожен раз викликається метод із сигнатурою відповідною типу формальних аргументів. На екрані побачимо:

Метод з цілим параметром x = 1

Метод з параметром подвоєної точності x = 1,5

Метод з дійсним параметром x = 2,5

Метод з двома цілими параметрами x = 1 y = 2

Виникає слушне питання, а що коли методу передається параметр, наприклад, цілий, але не **int**? Спробуємо виконати виклик функції **MyMethod** наступним чином:ї

```
byte b = 8;
pr.MyMethod(b);
```

Одержимо наступний результат:

Метод з цілим параметром x = 8

Тобто викликається метод **MyMethod (int x)**. Компілятор намагається підібрати серед методів, які перевантажуються, той, що найкращим чином відповідає фактичному аргументу (в даному разі типу **byte**) згідно правил приведення типів. Тому наступні звертання до методу **MyMethod**

```
pr.MyMethod(100L);
pr.MyMethod('A');
```

призведуть до цілком очікуваних наступних повідомлень:

Метод з дійсним параметром x = 100

Метод з цілим параметром x = 65

Тобто при першому виклику відбувається приведення фактичного аргументу до найближчого типу **float**, а у другому випадку тип **char** приводиться до типу **int**, і ми бачимо на екрані код символу **'A'**.

Проте спроба виклику **pr.MyMethod(1.1, 2.2);** призведе до синтаксичної помилки, оскільки найбільш відповідним методом для цього виклику компілятор вважатиме **MyMethod(int x, int y)**, але не матиме можливості виконати неявне приведення фактичних аргументів типу **double** до типу **int**.

Додамо тепер ще один метод **MyMethod** у клас **Program**:

```
public void MyMethod(ref int x)
  {
   x++;
```

```
    Console.WriteLine("Метод з цілим параметром
             ref x = {0}", x);
   }
```

Він теж має один параметр типу **int** , проте цей параметр використовується із модифікатором **ref**. Перевіримо, чи відрізнятиме компілятор сигнатури двох перевантажених методів **MyMethod (int x)** та **MyMethod(ref int x)**. Внаслідок звертання

```
          int i = 1;
          pr.MyMethod(ref i);
```

одержимо результат **Метод з цілим параметром ref x = 2**. Тобто модифікатори **ref** та **out** також впливають на сигнатури методів у випадку їх перевантаження. Але дуже важливо зауважити, що методи, які різняться ***лише*** модифікаторами **ref** та **out**, компілятор не зможе перевантажити. Для ілюстрації спробуємо додати ще один метод **MyMethod** у клас **Program**:

```
public void MyMethod(out int x)
  {
   x = 100;
   Console.WriteLine("Метод з цілим параметром
                 out x = {0}", x);
  }
```

Результатом компіляції буде повідомлення про помилку:

«MyMethod» cannot define overloaded methods that differs only on ref and out –

неможливо перевантажити методи, які розрізняються лише **ref** та **out**.

В той же час визначення та використання методу

```
public void MyMethod(out float x)
  {
   x = 100;
   Console.WriteLine("Метод з дійсним параметром
                out x = {0}", x);
  }
```

буде цілком успішним, адже два методи **MyMethod(float x)** та **MyMethod(out float x)** компілятор вважатиме «достатньо різними» і матиме можливість їх перевантаження.

9.9. Перевантаження конструкторів.

Оскільки конструктор – це метод класу, то конструктори також можуть перевантажуватись, отже, у класі можна визначити кілька конструкторів із різними списками параметрів. Це дозволяє створювати об'єкти класу по-різному в залежності від обставин. У наступному прикладі у класі **Circle** визначено 3 перевантажених конструктори: один ініціалізує своїми параметрами всі 3 дані-члени класу – координати **x0, y0** центру кола та радіус **r**; другий має один параметр – значення радіусу, а центром кола вважається початок координат; нарешті, третій конструктор без параметрів взагалі – визначає коло одиничного радіусу із центром у початку координат. Визначення цього класу міститься в окремому файлі **Circle.cs** проекту.

```
using System;
namespace Overloading_Constructors
{
   class Circle
   {
      double x0, y0;   // координати центру кола
      double r;        // радіус кола
      public Circle(double x0_, double y0_, double r_)
      {      // Конструктор з трьома параметрами
         x0 = x0_; y0 = y0_; r = r_;
         Console.WriteLine("Створили коло - центр ({0};{1}),

                      радіус { 2}
         ", x0, y0, r);
      }
      public Circle(double r_)
      {      // Конструктор з одним параметром
         x0 = 0; y0 = 0; r = r_;
         Console.WriteLine("Створили коло - центр ({0};{1}),

                      радіус { 2}
         ", x0, y0, r);
      }
      public Circle()
      {      // Конструктор без параметрів
         x0 = 0; y0 = 0; r = 1;
         Console.WriteLine("Створили коло - центр ({0};{1}),

                       радіус { 2}
         ", x0, y0, r);
      }
   }
}
```

У файл **Program.cs** помістимо функцію **Main** , яка створює 3 екземпляри класу **Circle** з допомогою кожного з конструкторів. Зверніть увагу, що інструкцією **Circle c1 = new Circle();** викликається не конструктор за замовчуванням, оскільки він взагалі не діє у цьому випадку – адже у класі існують власні конструктори – а саме написаний нами конструктор без параметрів.

```
using System;
namespace Overloading_Constructors
{
   class Program
   {
      static void Main()
      {
         Circle c1 = new Circle();
         Circle c2 = new Circle(10);
         Circle c3 = new Circle(1, 2, 5);
      }
   }
}
```

На екрані одержимо наступні повідомлення:

Створили коло - центр (0;0), радіус 1

Створили коло - центр (0;0), радіус 10

Створили коло - центр (1;2), радіус 5

Таким чином, всі три об’єкти класу були створені різними конструкторами. Можливість перевантаження конструкторів надає гнучкості при створенні екземплярів. Ще більшого ефекту можна досягти, використовуючи можливість виклику одного конструктору іншим.

9.10. Використання ключового слова this.

Ключове слово **this** у методі класу означає посилання на поточний екземпляр, тобто той, для якого був викликаний даний метод. Пригадаємо визначення класу **Polar_Point** . Він містив методи, що визначають декартові координати даної полярної точки. Наведемо ще раз тут окремо визначення цих методів:

```
public double xCoord()  // абсциса полярної точки
{
   return r * Math.Cos(phi);
}
public double yCoord()  // ордината полярної точки
```

```
{
   return r * Math.Sin(phi);
}
```

Обидва методи безпосередньо звертаються до членів класу **r** та **phi** без вживання складених імен, що включають ідентифікатори класу або екземпляру. Проте у тих самих методах можна використати посилання this , щоб підкреслити що використовуються дані-члени даного поточного екземпляру класу:

```
public double xCoord()  // абсциса полярної точки
{
   return this.r * Math.Cos(this.phi);
}
public double yCoord()  // ордината полярної точки
{
   return this.r * Math.Sin(this.phi);
}
```

Тоді, якщо у функції Main був створений екземпляр **p1** класу **Polar_Point**

Polar_Point p1 = new Polar_Point();

то виклик методу **p1.xCoord()** або **p1.yCoord()** фактично означає, що значенням **this** є посиланням на екземпляр **p1** , тобто **this.r** це **p1.r** , а **this.phi** це **p1.phi**. Зазвичай посилання **this** не використовується таким чином – у цьому просто немає необхідності.

Конструктор класу **Polar_Point** міг би також бути написаним із використанням **this**. Це дозволило б використати для формальних параметрів ті самі ідентифікатори, що й для членів класу. І хоча така практика не вважається ідеальним стилем програмування, наведемо тут код такого конструктору задля демонстрації використання посилання **this**.

```
public  Polar_Point(double r, double phi)
{
   // Конструктор – формальні параметри мають ті самі
   // ідентифікатори, що й члени класу
  this.r = r; this.phi = phi;
}
```

Якби тут не було використано посилання **this** , то формальні параметри конструктора **double r, double phi** перекривали б члени класу **Polar_Point** з такими самими ідентифікаторами, і не було б можливості їх проініціалізувати вказаними значеннями.

І нарешті, саме використання посилання **this** забезпечує можливість викликати конструкторами класу один одного. Це позбавляє від необхідності писати фрагменти коду, що дублюються. Конструктор, що звертається до іншого конструктору класу, має особливий синтаксис:

```
<специфікатор_доступу> <ідентифікатор_класу>
(<список_параметрів_конструктора_1>) :
    this (<список_параметрів_конструктора_2>)
    {
    // код конструктора
    }
```

Якщо об'єкт класу створюється таким конструктором, то спочатку викликається той конструктор класу, який має список параметрів, відповідний до **<списку_параметрів_конструктора_2>** з урахуванням приведення типів, а ***лише потім виконується код початкового конструктора***, який може бути і порожнім взагалі. В цій синтаксичній конструкції **this (<список_параметрів_конструктора_2>)** називається ***ініціалізатором*** конструктору.

Повернемось до прикладу **Overloading_Constructors**, в якому визначається клас **Circle** . Файл **Circle.cs** змінимо наступним чином:

```
using System;
namespace Overloading_Constr_this
{
  class Circle
  {
    public double x0, y0;   // координати центру кола
    public double r;        // радіус кола
    public Circle(double x0_, double y0_, double r_)
    {      // Конструктор з трьома параметрами
      x0 = x0_; y0 = y0_; r = r_;
      Console.WriteLine("Створили коло - центр ({0};{1}),
```

```
                    радіус {2}", x0, y0, r);
    }
    public Circle(double r_) : this (0, 0, r_)
    {      // Конструктор з одним параметром   }
    public Circle() : this(0, 0, 1)
    {      // Конструктор без параметрів      }
  }
}
```

Зверніть увагу, що тут єдиним «діючим» конструктором є конструктор **Circle(double x0_, double y0_, double r_)**, адже два інших конструктори не роблять нічого, просто звертаються через **this** саме до нього, вказуючи потрібні набори параметрів для ініціалізації.

Якщо у в основному файлі проекту записати наступний код

```
using System;
namespace Overloading_Constr_this
{
   class Program
   {
      static void Main()
      {
         Circle c1 = new Circle();
         Circle c2 = new Circle(10);
         Circle c3 = new Circle(5, 4, 3);
      }
   }
}
```

то після запуску прикладу на виконання на екрані побачимо наступний результат:

Створили коло - центр (0;0), радіус 1

Створили коло - центр (0;0), радіус 10

Створили коло - центр (5;4), радіус 3

При створенні всіх цих об’єктів викликались 3 різних конструктори, проте повідомлення про створення кола, насправді надсилав лише один з них, до якого два інших звертались завдяки посиланню **this**.

9.11. Деструктор класу.

Деструктор (фіналізатор – finalyzer) класу – це метод класу, що викликається автоматично в момент знищення екземпляру. Його призначення – вивільнити

певні ресурси, які, можливо, використав конструктор при створенні об'єкту. На відміну від класичного варіанту мови C++, де втрачені посилання постають великою проблемою, у .NET існує система GC – Garbage Collector автоматичного знищення об'єктів, посилань на які не існує в даний момент. Якщо у класі визначений деструктор (а це, як ми бачили, зовсім не обов'язково), то він напевне буде викликаний, проте не можна точно визначити момент, коли об'єкт буде фізично знищеним. Це накладає певні застереження на використання деструкторів, якщо вони мають виконувати якісь важливі дії на певний момент часу.

Деструктор має ідентифікатор, що починається із знака операції (**~**), після якого слідує ідентифікатор класу. Так само як конструктор, деструктор не має типу результату, крім того, він не має специфікатору доступу та параметрів – отже, не перевантажується.

У наступному прикладі клас **Destructor** містить цілочисельний член класу **kod** та конструктор, що його ініціалізує та інформує про створення об'єкту. Деструктор класу містить одну інструкцію – повідомлення про знищення об'єкту. Мета цього прикладу – дослідити порядок викликів деструкторів. З цією метою у програму включимо ще один метод **void creator(int k)**, у якому створюється локальний об'єкт **Destructor d**. Цей об'єкт існує лише протягом роботи функції **creator**. Посилання на нього втрачає сенс в момент завершення функції **creator**. Але в який момент він буде знищений фізично? Щоб відповісти на це питання, у функції **Main** створимо у циклі дуже багато звертань до функції **creator**. При кожному звертанні створюватиметься новий локальний об'єкт класу **Destructor**, а кожний попередній помічатиметься як посилання, не пов'язане з жодним об'єктом. В залежності від швидкодії вашого комп'ютера кількість ітерацій циклу можна зменшити або збільшити. Через кожні 10000 кроків циклу змоделюємо затримку, щоб краще відслідкувати результат. Ви побачите, що об'єкти знищуються не зовсім у хронологічному порядку.

```
using System;
namespace Destructor
{
   class Program
   {
      class Destructor
      {
         public int kod;
         public Destructor(int kod_)
         {
            kod = kod_;
            Console.WriteLine("Створюється екземпляр {0}", kod);
         }
         ~Destructor()
         {
            Console.WriteLine("Знищується екземпляр {0}", kod);
```

```
            }
        }
        static void creator(int k)
        {
            Destructor d = new Destructor(k);
        }
        static void Main()
        {
            for (int i = 1; i < 40000; i++)
            {
                creator(i);
                if ((i % 10000) == 0) Console.ReadLine();
            }
        }
    }
}
```

Отже, на відміну від мови С++, в якій об’єкти створюються командою **new** , а знищуються командою **delete** , яка гарантує в цей момент виклик деструктора, в мові С# знищенням об’єктів керує система **GC**, а тому наявність деструктора не є обов’язковою для звичайних класів.

Система **GC** працює автоматично, вона гарантує, що непотрібні об’єкти знищуються та причому дана операція виконується лише один раз, а також, що знищуються лише ті об’єкти, на які немає жодного посилання. Тим самим виключаються стандартні проблеми при роботі з динамічною пам’яттю – наприклад, проблема втрачених посилань або витік пам’яті (коли на об’єкт немає посилань, тобто до нього неможливо отримати доступ, але він не знищений і займає місце у пам’яті) або проблема завислих вказівників (коли об’єкта вже немає, а посилання на відповідне місце у пам’яті існує).

Система **GC** використовує два варіанти роботи очистки динамічної пам’яті – окремо для порівняно малих об’єктів та порівняно великих. Збирач сміття запускається через певні проміжки часу і для малих об’єктів знищує вже непотрібні і одразу дефрагментує динамічну пам’ять, переписуючи дані так, що об’єкти стають розташовані поруч (тобто створює неперервну вільну область динамічної пам’яті), при цьому відповідним чином автоматично змінює посилання на ті об’єкти, які залишаються у динамічній пам’яті. Для порівняно великих об’єктів алгоритм роботи збирача сміття ускладнюється та використовується поняття покоління, до якого належить об’єкт. Всього поколінь три: 0, 1 та 2. В момент створення усі об’єкти належать поколінню 0. Через деякий час збирач сміття видаляє непотрібні об’єкти, а ті, що залишилися, переводяться у покоління 1. Покоління 1 «чиститься» рідше за покоління 0, але за тим самим принципом. Об’єкти покоління 1, які залишаються у пам’яті після видалення непотрібних, переходять у покоління 2 (воно є останнім можливим).

9.12. Метод Main ().

На завершення вивчення перевантаження методів, розглянемо, які існують різні синтаксичні форми методу **Main()** . Можливих варіантів всього чотири – по-перше, метод **Main()** може повертати результат типу **int** або не мати результату – **void**; і по-друге, він може мати параметр **args** типу **string[]** або не мати параметрів. Повернення результату необхідне, якщо програма або система, що викликає ваш C#--проект, аналізує результат завершення – нульовий результат зазвичай означає успішне завершення, всі інші сигналізують про наявність помилки того чи іншого роду.

Аргументами методу **Main()** у разі, коли цей метод має параметр, слугуватимуть аргументи командного рядка, які вказуються при запуску програми після її імені.

У наступному прикладі метод **Main()** друкує свої аргументи. Зверніть увагу, що наявність або відсутність аргументів у командному рядку неявно аналізується значенням **args.Length** – в разі, коли аргументи відсутні, на екрані побачимо повідомлення **«У командному рядку маємо 0 аргументів»**, а цикл **for** не виконається жодного разу.

```
using System;
namespace Main_args
{
   class Program
   {
      static int Main(string[] args)
      {   // Аналізуємо аргументи методу Main ()
         Console.WriteLine(
        "У командному рядку маємо {0} аргументів", args.Length);
         for (int i = 0; i < args.Length; i++)
         {   // Друкуємо аргументи командного рядка
            System.Console.WriteLine("Arg[{0}] = {1}", i, args[i]);
         }
         Console.ReadLine();
         return 0;   // Повертаємо результат
      }
   }
     }
```

Перевірити роботу цієї програми можна, запустивши її з командного рядка.

Завдання для самоконтролю

1. Обчислити задану суму або добуток числового ряду для заданих початкових індексів nn та nk. Обов’язково перевіряти нерівностей $0 \leq nn \leq nk$. Виконати завдання реалізувавши відповідний метод і перегрузити його.

1. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{k^2-3}{(-1)^k k^2+5}$

2. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{k^2+(-1)^k k-1}{k^2+1}$
3. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{k^2-(-1)^{k+1} k^3}{k^2+k+1}$
4. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{(-1)^k k^2-1}{k^2+3}$
5. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{k^2-1}{(-1)^{k+1} k^2+7}$
6. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{(-1)^{k+1} k^2-2}{3k^2-2k}$
7. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{k^2+(-1)^{k-1} 2k-1}{k^2+8}$
8. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{k^2-3}{k^2-(-1)^k k+3}$
9. $\sum_{k=nn}^{nk} a_k$, де $a_k = \frac{k^2-(-1)^{k+1} k}{k^2+2}$
10. $\prod_{k=nn}^{nk} a_k$, де $a_k = \frac{k^2-3-8k}{(-1)^k k^3+8}$
11. $\prod_{k=nn}^{nk} a_k$, де $a_k = \frac{(-1)^k k^3+k^2-3}{k^3+8}$
12. $\prod_{k=nn}^{nk} a_k$, де $a_k = \frac{k^3+k^2-k}{(-1)^k k^2-1}$
13. $\prod_{k=nn}^{nk} a_k$, де $a_k = \frac{(-1)^k k^3+(-1)^k k^2}{5k^3+5}$
14. $\prod_{k=nn}^{nk} a_k$, де $a_k = \frac{5k^3+5}{k^2+(-1)^{k-1} 2k-1}$
15. $\prod_{k=nn}^{nk} a_k$, де $a_k = \frac{(-1)^{k+1} k^2-2}{(-1)^k k^3+8}$

2. Створити клас Cars, в якому міститься набір з 5 машин німецького виробництва, для кожної з машин визначені 4характеристики: колекція кольорів в якому вона представлена, рік випуску і ціна. Четверту характеристику оберіть самі. Реалізувати можливість для покупця обрати машину з набору за її параметрами. Повідомити якщо не існує машини з потрібними параметрами, якщо існує – вивести на екран повний опис машини.
3. Написати програму. Через консоль ви вводите два числа і знак(-,+,/, *) В залежності від введеного знаку у вас виконується відповідний метод.
4. Створити перегрузку метода shake(), який робить коктелі. В залежності від вказаних параметрів у вас виконується той чи інший метод shake() і на консоль виводиться назва коктеля.
5. Створіть 3 класа СуперГероїв з відповідними методами. Всі вони мають ім’я, і набір суперможливостей, кожна з яких виводить на екран відповідним повідомленням.

10.Статичні члени класу.

10.1. Статичні дані-члени класу.

З’ясуємо нарешті зміст службового слова **static**, яке неодноразово використовувалось раніше. Перш за все зауважимо, що модифікатор **static** може бути приписаний як до даних-членів так і до методів-членів класу. Якщо у класі декларується змінна із модифікатором **static**, то така змінна спільно використовується всіма екземплярами класу – фактично вона є глобальною для класу, а для доступу до неї вказується не ідентифікатор екземпляру, а ідентифікатор класу. Розглянемо перший приклад. В ньому визначений клас **MyClass** , в якому є член класу **num** та статичний член класу **count**. Звертання до **num** можливе лише через ідентифікатор екземпляру – у прикладі це **m1** або **m2** (тобто **m1.num** або **m2.num**), адже цей член класу **num** існує окремо та незалежно для кожного екземпляру класу. Звертання ж до **count** має відбуватись через ідентифікатор класу **MyClass** (тобто **MyClass.count**), оскільки **count** існує в єдиному примірнику та спільно використовується обома екземплярами **m1** та **m2**. Тобто, фактично статичні дані-члени класу можна сприймати як глобальні змінні у рамках класу.

```
using System;

namespace Static_value

{

/// <summary>

/// Коментар для автодокументації - тут можуть бути XML-

/// дескриптори

/// Для створення пакету автодокументації при компвляції

/// потрібна опція /doc:My.xml. Тут My.xml - ім'я файлу для

/// документації

/// </summary>

class MyClass
{
   public static int count = 10;
   public int num;
   public MyClass(int num_)
   { num = num_; }
}
class Program
{
   static void Main()
   { // До статичного члена класу звертаємось з іменем класу
      Console.WriteLine(
         "Статичний член класу = {0}", MyClass.count);
      MyClass m1 = new MyClass(100);
      // До звичайного члена класу звертаємось з іменем екземляру
      Console.WriteLine("Номер = {0}", m1.num);
```

```
        MyClass m2 = new MyClass(200);
        // До звичайного члена класу звертаємось з іменем екземпляру
        Console.WriteLine("Номер = {0}", m2.num);
        Console.WriteLine(
            "Статичний член класу = {0}", MyClass.count);
    }
}
}
```

Після запуску прикладу на екрані побачимо наступні повідомлення:

Статичний член класу = 10
Номер = 100
Номер = 200
Статичний член класу = 10

При спробі звертання **m1.count** або **m2.count** виникна синтаксична помилка, так само, як і при звертанні **MyClass.num**.

Зауваження.

1. Зверніть увагу на коментар, що починається трьома знаками слеш. Це спеціальна форма коментарів, які при компіляції автоматично форматуються у **XML**-файл з автодокументацією до проекту. Про деталі дізнайтесь самостійно.
2. Перевірте самостійно, що статична змінна ініціалізується нульовим значенням, якщо вона не проініціалізована у класі – якщо видалити присвоєння **count = 10**, то на екрані ви побачите повідомлення: **Статичний член класу = 0** . Практично це означає, що статична змінна виникає та ініціалізується раніше, ніж буде створений хоч один екземпляр класу.
3. Всередині класу до статичної змінної не можна звернутись через посилання **this** – адже статична змінна не належить конкретному екземпляру класу.

Розглянемо ще один приклад. В ньому клас **Counter** також містить статичний член класу **count** , початково проініціалізований нулем (до речі, нульове значення компілятор приписав би будь-якому статичному члену класу value-типу) та звичайний член класу **numID** цілого типу для ідентифікації екземпляру. У конструкторі статичний член **count** збільшується на одиницю, у деструкторі – зменшується. Таким чином, поточне значення **count** зберігає кількість існуючих у даний момент екземплярів класу **Counter**, оскільки при створенні чергового об’єкту конструктор збільшує на одиницю статичну змінну **count**, а при знищенні об’єкту – деструктор її зменшує на одиницю. В методі **Main** у циклі створюється достатньо велика кількість екземплярів **Counter**, а через кожні 1000 кроків на екран виводяться значення **count** та номер **numID** поточного екземпляру. Завдяки втручанню системи GC матимемо цікаві результати.

```
using System;
namespace Static_value_2
{
  class Counter
  {     // лічильник для існуючих екземплярів
```

```
        public static int count = 0;
        public int numID;   // це власний номер екземпляру
        public Counter(int n)   // конструктор
        {
            numID = n;   // встановлюємо свій номер
            count++;     // збільшуємо кількість екземплярів
        }
        ~Counter()     // деструктор
        {
            count--;     // зменшуємо кількість екземплярів
        }
    }
    class Program
    {
        static void Main()
        {
            for (int i = 0; i < 50000; i++)
            {
                Counter c = new Counter(i);
                if ((i + 1) % 1000 == 0)
                {
                    Console.WriteLine(
                        "Маємо {0} Counter\'ов ", Counter.count);
                    Console.WriteLine(
                        "Останній Counter з номером {0}", c.numID);
                }
            }
        }
    }
}
```

Зауваження. Зверніть увагу, що статичний член класу змінюють звичайні методи класу – у даному прикладі конструктор **Counter(int n)** та деструктор **~Counter()**, хоча це могли б бути і рядові методи класу **Counter**.

10.2. Статичні методи-члени класу.

Метод класу, визначений із модифікатором **static**, також є доступним на рівні самого класу, а не його екземплярів. Тобто для виклику такого методу непотрібний жодний екземпляр класу. Прикладом статичного методу є метод **Main()**, який викликається операційною системою. Зрозуміло, що в момент цього виклику жодного екземпляру жодного класу просто не може існувати. Іншим прикладом статичних методів, які ми неодноразово використовували у прикладах, є методи класу **Math** або **Console** – для звертання до них нам не було необхідності створювати відповідний об’єкт. Використання деяких стандартних статичних методів проілюструємо наступним прикладом.

```
using System;
namespace Static_Method_1
{
    class Program
    {
        static void Main()
        {
```

```
            double d = Math.Exp(-2); // Статичний метод - Exp()
            Console.WriteLine(d);  // Статичний метод - WriteLine()
                              // Статичний метод - ToString()
            string s = Convert.ToString(129, 2);
            Console.WriteLine(s); //s - зображення двійкового числа
            Console.WriteLine(Environment.OSVersion);
            Console.WriteLine(Environment.UserName);
            Console.WriteLine(Environment.ProcessorCount);
            Console.WriteLine(Environment.SystemDirectory);
            Console.WriteLine(Environment.UserDomainName);
            Console.WriteLine(Environment.GetFolderPath(
                     Environment.SpecialFolder.Desktop));
        }
    }
}
```

Зауваження. Зверніть увагу на корисні відомості, які містяться у статичних методах та властивостях класу **Environment.**

При використанні статичних методів слід пам'ятати про ряд обмежень, а саме:

1. Статичний метод може використовувати ***лише*** статичні дані-члени класу, адже статичний метод діє на рівні класу, не маючи доступу до екземплярів , а отже і до змінних екземплярів класу.
2. Статичний метод не може використовувати посилання **this**.
3. Статичний метод може викликати лише інші статичні методи класу. Щоб звернутись до нестатичного методу, потрібне посилання на екземпляр.

Проілюструємо роботу статичних методів наступними прикладами. В першому з них клас **StudyingStatics** містить звичайну цілу змінну **val** та статичну цілу змінну **stval** із модифікатором **private**, отже, потрібний метод класу **public static int get_()**, що повертає її значення. Крім того, конструктор ініціалізує змінну **val**, а ще один метод **public void add1()** додає до значення статичної змінної **stval** значення **val**.

```
using System;
namespace Static_Method_2
{
    class Program
    {
        class StudyingStatics
        {
            public int val;
            private static int stval = 10; // статичний член класу
            public StudyingStatics(int val_)
            { val = val_; }
            // статичний метод повертає значення статичного члену класу
            public static int get_()
            { return stval; }
            // нестатичний метод використовує як статичні так і
            // нестатичні члени класу
            public void add1()
```

```
        { stval += val; }
      }
      static void Main()
      {
        StudyingStatics s = new StudyingStatics(100);
        Console.WriteLine("Статична змінна = {0}",
              StudyingStatics.get_());
        s.add1(); // метод звертається до статичного члену класу
        Console.WriteLine("Статична змінна = {0}",
              StudyingStatics.get_());
      }
    }
}
```

Приклад успішно спрацьовує, на екрані з'являються повідомлення:

Статична змінна = 10

Статична змінна = 110

Проте спроба включити в клас наступний метод (він відрізняється від **add1()** лише модифікатором **static**)

```
// статичний метод не може звернутись до нестатичного члену
// класу
   public static void add2 ()
    { stval += val; }
```

приводить до синтаксичної помилки – статичний метод звертається до нестатичного члену класу.

У наступному прикладі використаємо той самий клас **StudyingStatics**, проте включимо у нього два статичних методи – перший інкрементує **stval**, а другий збільшує **stval** на значення нестатичного члену класу **val** екземпляру класу **StudyingStatics s**, що передається цьому методу. В цьому прикладі жодних проблем не виникає.

```
using System;
namespace Static_Method_3
{
  class StudyingStatics
  {
    public int val;
    private static int stval = 10; // статичний член класу
    public StudyingStatics(int val_)
    { val = val_; }
    // статичний метод повертає значення статичного члену класу
    public static int get_()
    { return stval; }
    // статичний метод змінює статичний член класу
    public static void incr()
    { stval++; }
```

```
        // статичний метод звертається до нестатичного члену класу
        // через екземпляр
        public static void change(StudyingStatics s)
        { stval += s.val; }
    }
    class Program
    {
        static void Main()
        {
            Console.WriteLine(
                "stval = {0}", StudyingStatics.get_());
            StudyingStatics.incr();
            Console.WriteLine(
                "stval після incr = {0}", StudyingStatics.get_());
            StudyingStatics s = new StudyingStatics(111);
            StudyingStatics.change(s);
            Console.WriteLine(
              "stval після change = {0}", StudyingStatics.get_());
        }
    }
}
```

Результатом цього прикладу будуть наступні повідомлення на екрані:

stval = 10

stval після incr = 11

stval після change = 122

10.3. Статичний конструктор класу.

Якщо клас потребує ініціалізації певних статичних змінних класу раніше, ніж буде створений перший екземпляр класу – йому потрібний статичний конструктор. Статичний конструктор визначається без жодного модифікатору доступу та жодних параметрів. Він викликається системою ***єдиний раз*** , проте момент, коли це станеться, не визначений, тому він не повинен містити код, виконання якого потрібне на певний момент часу. Зрозуміло також, що статичний конструктор може використовувати лише статичні члени свого класу. Цілком зрозуміло також, що статичний конструктор у класі може бути лише один. Зауважимо також, що допустимо при цьому мати в класі звичайний конструктор без параметрів – синтаксичного конфлікту не виникне.

У наступному прикладі в класі **MyClass** визначені 3 конструктори – один із них статичний, два інших мають відповідно один параметр та жодного. Причому останній з них просто викликає конструктор з одним параметром, передаючи йому аргумент, рівний 1. Ще два методи класу (статичний та звичайний) дозволяють слідкувати за значенням закритих членів класу **stval** та **val**. Метод **change(MyClass m)** змінює статичну змінну **stval**.

```
using System;
```

```
namespace Static_Constructor
{
   class Program
   {
      class MyClass
      {
         private static int stval;
         private int val;
         public MyClass(int val_) // Конструктор з 1 параметром
         {
            val = val_;
            Console.WriteLine("Працює конструктор з параметром");
         }
         public MyClass()
            : this(1)      // Конструктор без параметру
         {
            Console.WriteLine(
                 "Працює конструктор без параметрів");
         }
         // Статичний конструктор - викликається лише 1 раз!
         static MyClass()
         {
            stval = 999;
            Console.WriteLine("Працює статичний конструктор");
         }
         public int get_val()
         { return val; }
         public static int get_stval()
         { return stval; }
         public static void change(MyClass m)
         { stval += m.val; }
      }
      static void Main()
      {
         Console.WriteLine("До створення екземплярів: stval =
                     { 0}
         ", MyClass.get_stval());
        MyClass m = new MyClass();
         Console.WriteLine("Після створення екземпляру: stval =
          { 0}
         val = { 1}
         ", MyClass.get_stval(), m.get_val());
        MyClass.change(m);
         Console.WriteLine("Після change: stval = {0}",
               MyClass.get_stval());
         MyClass m_ = new MyClass(100);
         Console.WriteLine("Після створення екземпляру: stval =
           { 0}
         val = { 1}
         ", MyClass.get_stval(), m_.get_val());
        MyClass.change(m_);
         Console.WriteLine("Після change: stval = {0}",
                     MyClass.get_stval());
      }
```

```
    }
}
```

Зверніть увагу, що результатом цього прикладу буде 3 повідомлення про роботу конструкторів по створенню екземплярів – створювалось 2 об'єкти **m** та **m_**, проте один із конструкторів ініціює інший. Повідомлення про роботу статичного конструктора буде ***лише одне*** і з'явиться воно раніше, ніж будуть створені об'єкти **m** та **m_**:

Працює статичний конструктор

До створення екземплярів: stval = 999

Працює конструктор з параметром

Працює конструктор без параметрів

Після створення екземпляру: stval = 999 val = 1

Після change: stval = 1000

Працює конструктор з параметром

Після створення екземпляру: stval = 1000 val = 100

Після change: stval = 1100

Зауваження. Якщо клас містить ***лише*** статичні дані та методи, то необхідно заборонити використовувати екземпляри такого класу. Тоді є сенс зробити конструктор класу без параметрів закритим (**private**), щоб не спрацьовував конструктор за замовчуванням. Якщо в такому класі необхідна попередня ініціалізація певних даних, визначте статичний конструктор. Іншим вирішенням може бути проголошення всього класу статичним.

10.4. Статичні класи, локалізація та глобалізація

Існують спеціальні службові класи, завданням яких є обслуговування певних сторонніх операцій і екземпляри цих класів створювати не лише небажано, але й потенційно небезпечно. Такі класи, як уже зазначалось вище, помічають службовим словом **static**, яке гарантує неможливість створення екземплярів. До таких класів відносяться, наприклад, класи **Environment** (методи якого розглядались вище), **Convert** та **Math**.

Якщо в попередньому прикладі **Static_Method_1** проглянути визначення класу **Environment:**

```
namespace System
{
// Summary:
// Provides information about, and means to manipulate, the
// current environment and platform.
// This class cannot be inherited.
    [ComVisible(true)]
    public static class Environment
```

```
  {
  ...
  }
}
```

то можна побачити, що він не містить жодного екземплярного (нестатичного) методу, властивості або поля. Отже, статичні класи не можуть містити екземплярних даних та методів.

Розглянемо статичний клас **Convert**. З його назви зрозуміло, що він призначений для конвертації даних з одного формату до іншого. Найбільш вживаним способом його використання є звертання до перевантажених методів **Convert.ToInt32** та **Convert.ToString**. На базі них досить легко створити перетворення десяткових чисел до інших числових позиційних систем: двійкової, вісімкової та шістнадцяткової. В наступному прикладі показано, як можна це здійснити:

```
using System;
namespace Demo
{
   class Program
   {
      static void Main(string[] args)
      {
         // статичний метод Math
         double d1 = Math.Sin(10);
         double d2 = Math.Cosh(10);
         double d3 = Math.PI;
         // перетворення з 10 до 16-кової системи
         string s1 = Convert.ToString(500, 16);
         // перетворення з 8-кової до десяткової
         int i2 = Convert.ToInt32("234", 8);
         double df2 = Convert.ToDouble("10,2");
         Console.WriteLine(s1);
         Console.WriteLine(df2);
      }
   }
}
```

Ця програма буде прекрасно працювати, якщо в налаштуваннях вашої операційної системи подільником між цілою та дробовою частинами дійсного числа буде кома. Інакше виникне помилка. Це пов’язано з тим, що в різних культурах існують різні стандарти на представлення нецілих чисел (та взагалі різні культури вживають різні метричні системи, різні календарі, мають різні алфавіти, різні позначення національних валют, тощо). Оскільки платформа .NET налаштована на універсальність створюваних на її базі програм, в ній можливе використання багатьох різних національних культур. Цей підхід має назву ***глобалізації***. З іншого боку, конкретний користувач завжди має справу з усталеними для його культури форматами позначень дат, вимірів, тощо. Налаштування системи до вимог конкретної культури називається ***локалізацією***. Класи, які підтримують глобалізацію та локалізацію, містяться в просторі імен **System.Globalization**. Одним з найбільш вживаних класів є **CultureInfo**, який визначає форматування чисел та дат, а також встановлює

порядок сортування рядків, тощо. Щоб визначити, які культури встановлені у вашій операційній системі, можна скористатись наступною програмою.

```
using System;
using System.Globalization;
class Program
{
   static void Main(string[] args)
   {
      // всі культури в цій системі
      CultureInfo[] cultures =
        CultureInfo.GetCultures(CultureTypes.AllCultures);
      // перелік культур
      foreach (CultureInfo ci in cultures)
      {
         Console.WriteLine(ci.EnglishName);
         Console.WriteLine(ci.Name);
      }
   }
}
```

Як можна бачити, в типовій операційній системі встановлено велику кількість культур. Зазвичай ім'я культури складається або з двосимвольної назви країни (нейтральні культури) або з двосимвольної назви культури та країни. Наприклад, нейтральна українська мова описується як "**ua**", а повна назва "**uk-UA**". Англійських культур існує декілька (для різних регіонів).

Щоб вивести дату російською або українською мовою потрібно явно вказати культуру, як це показано у наступному прикладі:

```
using System;
using System.Globalization;
class Program
{
   static void Main(string[] args)
   {
      // D - повний формат дати
      // ru-RU російська локалізація
      Console.WriteLine(DateTime.Now.ToString(
              "D", new CultureInfo("ru-RU")));
      // D - повний формат дати
      // uk-UA російська локалізація
      Console.WriteLine(DateTime.Now.ToString("D",
               new CultureInfo("uk-UA")));
   }
}
```

Завдання для самоконтролю

1.Створити розширяючий метод для масиву цілих чисел, який сортує елементи масиву по зростанню.

2. Створити розширяючий метод для масиву цілих чисел, який сортує елементи масиву по спаданню.

3. Створити розширяючий метод для масиву цілих чисел, який шукає мінімум і максимум в масиві.

4. Створити розширяючий метод для масиву цілих чисел, який знаходить середнє арифметичне і середнє геометричне в масиві.

5. Створити розширяючий метод для масиву цілих чисел, який виводить його елементи в зворотному порядку.

6. Створити розширяючий метод для масиву цілих чисел, який спочатку виводить елементи з парними індексами а потім з непарними.

7. Створити розширяючий метод для масиву цілих чисел, який спочатку виводить парні елементи, а потім непарні.

8. Створити розширяючий метод для масиву цілих чисел, який міняє місцями максимальний і мінімальний елементи масиву.

9. Створити розширяючий метод для масиву цілих чисел, який додає в кінець масиву елемент.

10. Створити розширяючий метод для масиву цілих чисел, який копіює масив в новий масив дублюючи кожен елемент. Вихідний масив буде вдвічі більший.

11. Створити розширяючий метод для масиву цілих чисел, який замінює всі від’ємні елементи значенням максимального елементу.

12. Створити розширяючий метод для масиву цілих чисел, який замінює всі додатні елементи, значенням максимального.

Розділ 11.Властивості та індексатори.

11.1. Властивості.

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

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

Практично реалізація цієї ідеї означає, що кожне закрите поле повинне бути забезпечене парою відкритих методами для роботи з ними. Але мова C# має більш зручний інструмент – це саме властивості, які були анонсовані вище. Властивість являє собою пару аксесорів (accessor) – **get** та **set**, які

забезпечують доступ до певного поля класу. Властивість визначається за наступним синтаксисом:

```
<модифікатор доступу властивості> <тип властивості> <ідентифікатор властивості>
{
   get           // аксесор get (отримати)
   {
      // код аксесору get
   }
   set           // аксесор set (встановити)
   {
     // код аксесору set
   }
}
```

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

Виконаємо таке саме завдання, як і у попередньому прикладі, проте використаємо замість методів доступу до закритого поля **field** властивість. Тут клас **MyPropertyClass** має закрите поле **field** та властивість, що ним керує – **Field**. Аксесор цієї властивості **get** просто повертає значення поля

field, а аксесор **set** перевіряє значення змінної **value**, яка автоматично потрапляє у нього, – якщо це значення додатне, то воно присвоюється полю **field**, в іншому разі у **field** встановлюється прийняте за замовчуванням значення 1.

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

Зверніть увагу, коли ідентифікатор властивості фігурує у виразі

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

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

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

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

інкременту (декременту): **++mp.Field чи mp.Field++**. Порівняйте, як це відбувалось у прикладі **Use_Methods**.

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

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

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

1. Використовуються поля та властивості с точки зору синтаксису однаково.
2. Поле розміщується у пам'яті, а властивість – ні.
3. Властивість є логічним полем, доступ до якого здійснюється через аксесори **get** та **set**.
4. Властивості, як і поля, мають модифікатор доступу.
5. Властивості, як і поля, можуть бути статичними.
6. Ознаки схожості ***властивості*** проявляють і з ***методами*** класу, тому порівняємо і їх.
7. Властивість, як і метод, містить код.
8. Властивість, як і метод, приховує деталі свого функціонування.
9. Властивість, як і метод, може бути статичною.
10. Властивість, на відміну від методу, не може мати тип **void**.
11. Властивість, на відміну від методу, не має дужок із списком параметрів.

12.Властивість, на відміну від методу, не може перевантажуватись.

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

11.2. Індексатори.

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

```
<модифікатор доступу > <тип індексатору> this [<тип індексу> < ідентифікатор індексу>]
{
   get            // аксесор get (отримати)
   {
      // код аксесору get
   }
   set            // аксесор set (встановити)
   {
     // код аксесору set
   }
}
```

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

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

аксесор **get** або встановлює через аксесор **set** значення відповідного елементу масиву.

```
using System;
namespace Use_Indexer_0
{
   class MyIndexClass
   {
   private int size;              // розмір масиву
   private int[] arr;             // декларація масиву
   public MyIndexClass(int size_)  // конструктор
   {
      size = size_;
      arr = new int[size_];       // тут масив створюється
   }
   public int this[int ind]       // це і є індексатор
   {
      get    // повертаємо елемент масиву як результат
             // звертання до індексатору
      {
         if ((ind >= 0) && (ind < size)) { return arr[ind]; }
         return 0;
      }
      set     // присвоюємо елемент масиву як результат
              // звертання до індексатору
      {
         if ((ind >= 0) && (ind < size)) { arr[ind] = value; }
      }
   }
}
class Program
{
   static void Main()
   {
      MyIndexClass mic = new MyIndexClass(5);
      for (int i = 0; i < 5; i++)
      {
         mic[i] = i;          // тут працює set індексатору
                              // тут працює get
         Console.WriteLine("mic [{0}] = {1}", i, mic[i]);
      }
   }
}
}
```

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

mic [0] = 0

mic [1] = 1

mic [2] = 2

mic [3] = 3

mic [4] = 4

Розглянемо ще один приклад. У ньому описаний клас **Roots**, призначений для визначення коренів квадратного рівняння. Саме рівняння задається своїми коефіцієнтами **a, b, c** – вони є закритими членами класу разом із масивом **r** коренів, який може містити 0, 1 або два корені. У випадку нескінченої кількості коренів (рівняння вироджується) цей масив також порожній. Крім того, закритими членами цього класу є **discr** – дискримінант квадратного рівняння та **num** – кількість коренів рівняння. Останнім полем керує властивість **Num** призначена лише для читання. Конструктор класу ініціалізує поля **a, b, c** , визначає **discr** та викликає закритий метод класу **Calc_Roots()** , в якому зосереджена логіка розв'язування квадратного рівняння та створений (якщо рівняння має корені) масив коренів **r**. Індексатор, визначений у цьому класі, дозволяє після створення екземпляру **roots** класу **Roots** використовувати корені рівняння як елементи масиву **roots[i]**, де **i** пробігає індекси від 0 до **roots.Num** – кількість коренів рівняння. Нагадаємо, що у випадку нескінченої кількості коренів властивість **Num** містить максимальне ціле число, а масив коренів не визначений.

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

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

```
            {     // Тут працює індексатор
               Console.WriteLine(roots[i]);
            }
         }
      }
   }
}
```

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

1. Доступ до властивості здійснюється за її ідентифікатором, індексатор не має власного ідентифікатору, доступ здійснюється за індексом елементу.
2. Аксесор **get** властивості не має параметрів, в той час як **get** індексатора має один (або більше) параметрів – індекс.
3. Аксесор **set** властивості має неявний параметр **value**, а **set** індексатора крім **value** має ті самі індекси, що й його **get**.
4. Властивість може бути статичним членом класу, індексатор – ніколи, оскільки він визначається через посилання **this**
5. Властивості не перевантажуються в той час, як індексатор може бути перевантажений за рахунок використання індексу іншого типу, або іншої кількості індексів.

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

```
using System;
namespace Use_Indexer_2
{
   using System;
   class Grid
   {
      // кількість символів у латинському алфавіті
      const int NumRows = 26;
      const int NumCols = 10;   // кількість десяткових цифр
      int[,] cells = new int[NumRows, NumCols];
      // тут визначається двовимірний індексатор
      public int this[char symb, char dig]
      {
         get
         {  // символ - завжди буде великою літерою
            symb = Char.ToUpper(symb);
            if (((symb >= 'A') && (symb <= 'Z')) &&
               ((dig >= '0') && (dig <= '9')))
               return cells[symb - 'A', dig - '0'];
            else return 0;
         }
         set
         {
            symb = Char.ToUpper(symb);
            if (((symb >= 'A') && (symb <= 'Z')) &&
                ((dig >= '0') && (dig <= '9')))
```

```
                cells[symb - 'A', dig - '0'] = value;
            }
        }
    }
    class Program
    {
        static void Main()
        {
            Grid gr = new Grid();
            for (char c = 'A'; c <= 'Z'; c++)
            {
                for (char d = '0'; d <= '9'; d++)
                {       // Тут працює set індексатору
                    gr[c, d] = (int)(c - 'A') * (int)(d - '0');
                    // Тут працює get індексатору
                    Console.Write(gr[c, d] + " \t");
                }
            }
        }
    }
}
```

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

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

Завдання для самоконтролю

1. Створи клас Tovar, в якому існують наступні закриті поля:
   - назва товару
   - тип товару
   - опис товару
   - вартість товару
2. Створити клас Store, в якому існує закритий масив елементів типу Tovar, Забезпечити наступні можливості:
   - Вивід інформації про товар по номеру за допомогою індексу
   - Вивід на екран інформації про товар, наззву якрнр введено з коавіаиупи, якщо таких товарів нема, вививсти відповідне повідомлення.
3. Створити англійсько-український перекладач, за допомогою технології індексаторів.

Розділ 12. Спадкування в мові C#.

12.1. Поняття про спадкування та ієрархію класів.

Одним із трьох основних принципів об’єктно-орієнтованого програмування є спадкування, яке забезпечує можливість повторного використання існуючих класів. Мова C# надає можливості створювати нові класи, що є різновидами вже існуючих класів. Такий клас містить у собі всі члени так званого ***базового*** або батьківського класу, додаючи у нього особливості власного функціонування. Клас, який створюється на основі базового, називається ***похідним*** або дочірнім класом (інколи говорять: класом-нащадком). Це так зване класичне спадкування, яке встановлює між похідним та базовим класом відношення «***Є***»: похідний клас ***є*** різновидом базового. Зрозуміло, що і похідний клас може стати базовим при створенні наступного похідного класу від нього. Все гроно класів, які мають спільний корінь, разом із зв’язками між ними, називається **ієрархією класів**. Співвідношення між базовим та похідним класом позначається діаграмою, наведеною на малюнку.

Визначення похідного класу має наступний синтаксис:

class <ідентифікатор базового класу>

{

// код базового класу

}

class <ідентифікатор похідного класу> : <ідентифікатор базового класу>

{

// код похідного класу

}

Наведемо приклад створення похідного класу. Базовий клас **Building** буде визначати деяку будівлю. Оскільки офісний будинок та житловий будинки є різновидами будівлі, для них можна визначити похідні класи **OfficeBuilding** та **LivingBuilding**, додавши у клас **Building** необхідну функціональність.

```
namespace DerivativeClass
{
   class Building
   {   // базовий клас - будинок
      public int floors;      // кількість поверхів
      public double area;     // площа
   }
   class OfficeBuilding : Building
   {   // похідний клас - офісний будинок
      public int numOffices;  // кількість офісів
   }
   class LivingBuilding : Building
```

```
    {   // ще один похідний клас - житловий будинок
        public int numFlats;    // кількість квартир
        public int numPeople;   // кількість жителів
    }
    class Program
    {
        static void Main()
        {
            Building b = new Building();  // створюємо будинок
            b.area = 1000;              // визначаємо площу
            b.floors = 10;      // визначаємо кількість поверхів
                            // створюємо офісний будинок
            OfficeBuilding ob = new OfficeBuilding();
            ob.area = 500;      // визначаємо площу
            ob.floors = 3;      // визначаємо кількість поверхів
            ob.numOffices = 8;   // визначаємо кількість офісів
                            // створюємо житловий будинок
            LivingBuilding lb = new LivingBuilding();
            lb.area = 2000;     // визначаємо площу
            lb.floors = 5;      // визначаємо кількість поверхів
            lb.numFlats = 20;   // визначаємо кількість квартир
            lb.numPeople = 75;  // визначаємо кількість мешканців
            Console.WriteLine(
            "Будинок : поверхів {0} площа {1}", b.floors, b.area);
            Console.WriteLine(
            "Офісний будинок : поверхів {0} площа {1} офісів {2}",
            ob.floors, ob.area, ob.numOffices);
            Console.WriteLine(
            "Житловий будинок : поверхів {0} площа {1} квартир {2}

             мешканців { 3}
            ",  lb.floors, lb.area, lb.numFlats,
           lb.numPeople);
        }
    }
}
```

Як видно з даного прикладу, об’єкти класів **OfficeBuilding** та **LivingBuilding** крім власних членів мають і всі члени базового класу **Building**. Хоча в даному прикладі розглядались лише дані-члени, все сказане переноситься і на інші члени класів – методи, властивості, індексатори. Правда, такий спосіб спадкування має місце лише у випадку, коли члени базового класу відкриті – **public**.

Зауваження.

1. Правила спадкування мови C# забороняють використовувати два або більше класів як базові – множинне спадкування, подібне до існуючого в C++, мова C# не підтримує.
2. Якщо при визначенні класу не вказано базовий клас, то компілятор вважає, що базовим класом є **System.Object**. Тобто наступні два визначення класів еквівалентні :

class MyClass : Object// клас успадкований від System.Object

```
   {
       // визначення класу
   }
```

та

```
class MyClass    // клас успадкований від System.Object
   {
       // визначення класу
   }
```

Таким чином, всі створені класи успадковують методи від класу **System.Object**.

1. Якщо при визначенні класу вказано службове слово **sealed**, це означає заборону спадкування від такого класу, тобто наступний клас не може бути використаний як базовий:

```
         // від цього класу не можна утворити похідний клас
sealed class MyClass
   {
       // визначення класу
   }
```

2. Спадкування та правила доступу до членів класів.

З'ясуємо, що відбувається при спадкуванні із закритими (**private**) членами базового класу. Нагадаємо, що використання закритих членів класу – одна з основ інкапсуляції даних. При спадкуванні закриті члени класу залишаються ***недоступними*** для похідних класів. Такий підхід зрозумілий, адже в іншому разі для «зламу» закритих членів класу досить було б утворити похідний від нього клас. Переконайтесь самостійно, що якщо у базовий клас **Building** включити закрите поле, наприклад, **private double cost**; то при спробі звернутись до нього із екземпляру похідного класу, вказавши **lb.cost**, одержимо синтаксичну помилку:

«**… is inaccessible due to its protection level**» – недоступний із-за його рівня захисту.

Проте, у випадках, коли необхідно дозволити похідним класам використовувати закриті поля базового класу, останні можна визначити із модифікатором доступу **protected** – захищений. Такий член класу відкритий лише у своїй ієрархії класу і закритий для інших частин програми.

У наступному прикладі в базовому класі **Base** визначені: захищене (protected) поле **field** та відкрита властивість **Field**, яка керує доступом до цього поля (адже у функції **Main(),** зокрема, неможливо звернутись до поля

field безпосередньо). У похідному класі **SubBase** створимо метод **ChangeField()**, призначення якого – змінити (в даному прикладі подвоїти) захищене поле **field** базового класу **Base**.

```
using System;namespace ProtectedClass
{
   class Base
   {
      protected int field;
      public int Field
      {
         get { return field; }
         set { field = value; }
      }
   }
   class SubBase : Base
   {
      public void ChangeField()
      {   // Метод має доступ до захищеного поля базового класу
         field = 2 * field;
      }
   }
   class Program
   {
      static void Main()
      {
         Base b = new Base();
         b.Field = 100;
         Console.WriteLine("Клас Base: Field = {0}", b.Field);
         SubBase sb = new SubBase();
         sb.Field = 200;
         Console.WriteLine(
                  "Клас SubBase: Field = {0}", sb.Field);
         // Метод змінює захищене поля базового класу
         sb.ChangeField();
         Console.WriteLine("Клас SubBase після ChangeField() :

             Field = { 0}
         ",  sb.Field);
      }
   }
}
```

Після запуску проекту на екрані побачимо наступні повідомлення:

Клас Base: Field = 100
Клас SubBase: Field = 200
Клас SubBase після ChangeField() : Field = 400

Таким чином, похідні класи мають відкритий доступ до захищених членів базового класу. Проте для інших частин коду поле **field** у екземплярах похідних класів залишається, як і раніше, недоступним.

12.3. Конструктори базового та похідних класів.

Як відомо, конструктори відіграють важливу роль при створенні об'єктів. Навіть, якщо у класі конструктор не визначений, компілятор створює конструктор за замовчуванням, який присвоює нульові значення всім полям. Важливо зрозуміти, що при створенні екземпляру похідного класу недостатньо роботи лише конструктору цього класу, адже він несе відповідальність тільки за члени похідного класу, який у свою чергу містить всі члени свого базового класу. Отже, необхідно визначити спочатку всі значення полів базового класу. Тому при створенні екземпляру похідного класу компілятором ***спочатку викликається конструктор базового класу*** і лише потім конструктор похідного класу. Наступний простий приклад наочно демонструє порядок виклику конструкторів базового та похідних класів.

```
using System;
namespace InheritanceConstructor
{
   class Base              // базовий клас
   {
      public Base()
      {
         Console.WriteLine(
            "Створюємо екземпляр базового класу");
      }
   }
   class SubBase : Base       // похідний  клас
   {
      public SubBase()
      {
         Console.WriteLine("Створюємо екземпляр похідного класу");
      }
   }
   class SubSubBase : SubBase  // похідний  від похідного класу
   {
      public SubSubBase()
      {
         Console.WriteLine(
          "Створюємо екземпляр похідного від похідного класу");
      }
   }
   class Program
   {
      static void Main()
      {
         Base b = new Base();
         Console.WriteLine("---------------------");
         SubBase sb = new SubBase();
         Console.WriteLine("---------------------");
         SubSubBase ssb = new SubSubBase();
      }
   }
}
```

На екрані одержимо наступні повідомлення:

Створюємо екземпляр базового класу

Створюємо екземпляр базового класу

Створюємо екземпляр похідного класу

Створюємо екземпляр базового класу

Створюємо екземпляр похідного класу

Створюємо екземпляр похідного від похідного класу

Як бачимо, при створенні екземпляру найнижчого в ієрархії класу послідовно спрацьовують всі конструктори, починаючи від конструктору базового класу. Спробуємо змінити визначення базового класу **Base**, додавши в нього деяке приватне поле та конструктор з параметром для його ініціалізації:

```
class Base              // базовий клас
{
   private int field;
   // конструктор ініціалізує поле field
   public Base(int field_)
   {
      field = field_;
      Console.WriteLine(
    "Створюємо екземпляр базового класу field = {0}", field);
   }
}
```

Спроба скомпілювати цей приклад призведе до помилки «No overload for method 'Base' takes '0' arguments» – немає перевантаженого конструктора 'Base' без аргументів. Поява такого повідомлення компілятора цілком зрозуміла, адже конструктор похідного класу **SubBase** автоматично намагається викликати конструктор класу **Base** без параметрів, а клас **Base** такого конструктору не надає. Можливих виходів із такої ситуації існує декілька. Один із варіантів – створити у класі **Base** конструктор без параметрів, використавши уже знайомий нам синтаксис виклику одним конструктором класу іншого:

public Base() : this (1) { }

Правда, в цьому випадку прийдеться задовольнитись викликом конструктору із деяким значенням параметру за замовчуванням, в даному прикладі це 1.

Інший варіант – викликати у похідному класі конструктор базового класу, передавши йому потрібні значення аргументів. Для цього використовується наступний синтаксис:

<специфікатор_доступу>
<ідентифікатор_конструктора_похідного_класу>

(<список_параметрів_конструктора_похідного_класу >) :

base (<список_параметрів_конструктора_базового_класу>)

{

// код конструктора похідного класу

}

Внесемо такі зміни у наш приклад.

```
using System;
namespace InheritanceConstructor
{
   class Base                 // базовий клас
   {
      private int field;
      // конструктор ініціалізує поле field
      public Base(int field_)
      {
         field = field_;
         Console.WriteLine("Створюємо екземпляр базового класу

               field = { 0}
         ", field);
      }
   }
   class SubBase : Base       // похідний  клас
   {
      // виклик конструктору базового класу
      public SubBase(int field_) : base(field_)
      {
         Console.WriteLine(
              "Створюємо екземпляр похідного класу");
      }
   }
   // похідний  клас від похідного класу
   class SubSubBase : SubBase
   {
      public SubSubBase(int field_) : base(field_)
      {
         Console.WriteLine(
        "Створюємо екземпляр похідного від похідного класу");
      }
   }
   class Program
   {
      static void Main()
      {
         Base b = new Base(100);
         Console.WriteLine("-------------------------");
```

```
            SubBase sb = new SubBase(200);
            Console.WriteLine("------------------------");
            SubSubBase ssb = new SubSubBase(300);
        }
    }
}
```

Тепер кожний похідний клас звертається не до конструктора базового класу, передаючи йому свій аргумент. Результатом роботи будуть наступні повідомлення:

Створюємо екземпляр базового класу field = 100

Створюємо екземпляр базового класу field = 200

Створюємо екземпляр похідного класу

Створюємо екземпляр базового класу field = 300

Створюємо екземпляр похідного класу

Створюємо екземпляр похідного від похідного класу

Використання службового слова **base** в такому контексті дозволяє викликати будь-який із перевантажених конструкторів базового класу відповідно до списку аргументів виклику.

Зауваження. Важливо розуміти також, що якщо конструктор деякого класу визначений із модифікатором **private**, звертання до нього у похідному класі неможливо, тобто від такого класу неможливо створити екземпляр похідного.

12.4. Посилання на екземпляри базового та похідних класів.

Важливо пам’ятати, що ідентифікатор, який ми вказуємо при визначенні екземпляру класу, є змінною-посиланням на область пам’яті, де буде розміщений об’єкт. Раніше ми використовували приведення типів при використанні змінних скалярних типів. Ці дії керувались низкою правил «просування по сходинках типів». З’ясуємо, як відбувається приведення типів при використанні посилань на екземпляри базового та похідних класів. Головне правило, яке тут діє, полягає у тому, що ***посиланню на базовий клас можна присвоїти посилання на будь-який похідний від нього клас***. Тобто, якщо розглянути два класи – клас **A** та похідний від нього клас **B**, то правильність визначень та присвоєнь, виконаних у функції **Main()**, позначена у коментарях:

```
  class A               // базовий клас
{
   // визначення базового класу
}
class B : A             // похідний  клас
```

```
{
   // визначення похідного класу
}
static void Main()
{
   A a = new A();   // вірно
   A ab = new B();  // вірно – посиланню на базовий клас
                 // присвоєне посилання на похідний
   B b = new B();   // вірно
   B ba = new A();  // помилка – посиланню на похідний
                 // клас присвоєне посилання на базовий
   a = b;          // вірно – посиланню на базовий клас
                 // присвоєне посилання на похідний
   b = a;          // помилка – посиланню на похідний
                 // клас присвоєне посилання на базовий
   ab = b;         // вірно
   b = (B)ab;      // вірно, явне приведеня
   b = (B)a;       // вірно, явне приведеня
   a = (A)b;       // вірно, явне приведеня
}
```

Дуже важливо розуміти, що саме тип змінної-посилання на клас, визначає доступні для використання члени класу. Тобто якщо змінній **ab** типу **A** присвоєне посилання на об'єкт **B** , як це зроблено в інструкції

A ab = new B(); то **ab** є змінною-посиланням на клас **A**, тому через **ab** доступні **лише** члени класу **A**, адже змінній базового класу невідомий склад похідних класів. Для більш прозорого розуміння цього факту розглянемо ще один приклад – варіацію на тему класів **Base**, **SubBase** та **SubSubBase**. Тут у кожний із класів даної ієрархії включене відкрите поле, щоб мати змогу прослідкувати за їх доступністю при використанні посилань на об'єкти даної ієрархії класів. Метод **fun()** перевантажений тричі – він приймає відповідно параметри типів **Base**, **SubBase** та **SubSubBase**. Результат цього прикладу підтверджує, що змінна **b** має тип **Base**, незважаючи на створення операцією **new SubBase(200)** , а змінна **sb** має тип **SubBase**, незважаючи на створення операцією **new SubSubBase(300)**.

```
using System;
namespace ConvertObject
{
   class Base                // базовий клас
   {
      public int field;
      public Base(int field_)   // конструктор
      { field = field_; }
   }
   class SubBase : Base       // похідний  клас
   {
      public int subfield;
      public SubBase(int field_) : base(field_)
      { subfield = field_; }
   }
   // похідний  клас від похідного класу
```

```
    class SubSubBase : SubBase
    {
        public int subsubfield;
        public SubSubBase(int field_) : base(field_)
        { subsubfield = field_; }
    }
    class Program
    {
        static void fun(Base b)
        {
            Console.WriteLine("об\'єкт Base: field = {0}", b.field);
        }
        static void fun(SubBase b)
        {
            Console.WriteLine("об\'єкт SubBase: field = {0}

                    subfield = { 1}
            ", b.field, b.subfield);
        }
        static void fun(SubSubBase b)
        {
            Console.WriteLine("об\'єкт SubSubBase: field = {0}

               subfield = { 1}
            subsubfield = { 2}
            ", b.field,
             b.subfield, b.subsubfield);
        }
        static void Main(string[] args)
        {
            object o = new Base(100);  // вірно - Base «Є» object
            Base b = new SubBase(200); // вірно - SubBase «Є» Base
                                 // вірно - SubSubBase «Є» SubBase
            SubBase sb = new SubSubBase(300);
            SubSubBase ssb = new SubSubBase(400);   // вірно
                                         //    SubSubBase ssb = new Base(1000);  // тут буде помилка
            fun(b);
            fun(sb);
            fun(ssb);
        }
    }
}
```

В результаті роботи даного прикладу одержимо наступні повідомлення:

об'єкт Base: field = 200

об'єкт SubBase: field = 300 subfield = 300

об'єкт SubSubBase: field = 400 subfield = 400 subsubfield = 400

Цікаво також перевірити, що буде, якщо закрити коментарем метод **fun(SubBase b)**, а потім і **fun(SubSubBase b)**. Перевірте самостійно, яким чином будуть викликатись методи **fun()** .

12.5. Поняття про поліморфізм.

Поліморфізм – наступний з трьох принципів ООП. Його застосування дозволяє створювати у похідних класах методи, які мають той самий інтерфейс, що й методи базового класу, проте відмінну функціональність.

Припустимо, що ми маємо клас **BankAccount**, що описує функціонування банківського рахунку. Крім звичайного банківського рахунку можна створити і інші – наприклад, депозитний рахунок **DepositAccount**, як похідний клас від **BankAccount**.

```
using System;
namespace BankAccount
{
   class BankAccount // Це банківський рахунок
   {
      protected double money;     // розмір вкладу
      private long numAccount;    // номер рахунку
                             // властивість - повертає розмір вкладу
      public double Money
      {
         get { return money; }
      }
      // конструктор
      public BankAccount(long numAccount_, double money_)
      {
         money = money_;
         numAccount = numAccount_;
      }
      // поповнення рахунку
      public void setMoney(long numAccount_, double sum)
      {
         if (numAccount_ != numAccount)
            Console.WriteLine("Неправильний номер рахунку");
         else money += sum;
      }
      // одержання грошей
      public void getMoney(long numAccount_, double sum)
      {
         if (numAccount_ != numAccount)
            Console.WriteLine("Неправильний номер рахунку");
         else
           if (money >= sum)
            money -= sum;
         else Console.WriteLine("Ви перевищили свій баланс");
      }
      // нарахування відсотків - бонус 6 %
      public void setBonus()
      { money += money * 0.06; }
   }
   // Це депозитний банківський рахунок
   class DepositAccount : BankAccount
```

```
    {
        private int termin;     // термін депозиту
        public DepositAccount(long numAccount_, double money_,
          int termin_) : base(numAccount_, money_)
        { termin = termin_; }
    }
    class Program
    {
        static void Main()
        {
            // відкрили рахунок
            BankAccount myBankAccount = new BankAccount(1, 1000);
            Console.WriteLine("На рахунку - {0} грн.",
                    myBankAccount.Money);
            myBankAccount.getMoney(1, 200); // зняли гроші
            Console.WriteLine("На рахунку - {0} грн.",
                    myBankAccount.Money);
            myBankAccount.setMoney(1, 200); // поповнили рахунок
            Console.WriteLine("На рахунку - {0} грн.",
                    myBankAccount.Money);
            myBankAccount.setBonus();      // одержали бонус
            Console.WriteLine("На рахунку - {0} грн.",
                    myBankAccount.Money);
            DepositAccount myDepositAccount =
                      new DepositAccount(1, 5000, 3);
            Console.WriteLine("На депозитному рахунку - {0} грн.",
                    myDepositAccount.Money);
            myDepositAccount.setBonus();
            Console.WriteLine("На депозитному рахунку - {0} грн.",
                    myDepositAccount.Money);
        }
    }
}
```

В результаті роботи цієї програми на екрані одержимо такі повідомлення:

На рахунку – 1000 грн.

На рахунку – 800 грн.

На рахунку – 1000 грн.

На рахунку – 1060 грн.

На депозитному рахунку - 5000 грн.

На депозитному рахунку - 5300 грн.

Очевидно, що невірно нараховувати бонус однаковим чином по різних видах внесків на банківські рахунки. Отже, у кожному похідному класі необхідно мати для цього окремий метод. Зручно мати дійсно різні функції проте з ідентичним інтерфейсом. Досягти цього можна двома шляхами. Один з них можливий завдяки підтримці поліморфізму у мові C#. Ідея полягає у тому, що деякий метод базового класу може ***заміщатись*** у похідних класах. Таким чином, однакове звертання до екземплярів різних похідних класів

призводить до різних відгуків – адже викликаються різні методи хоча й з однаковою сигнатурою. Це доволі складний механізм, і щоб підключити його підтримку, метод (або властивість) базового класу, який буде заміщатись у похідному класі, позначають службовим словом **virtual**. Відповідний метод у похідному класі позначають **override**:

```
class Base
{
   public virtual void DoSomething() { }
   public virtual int SomeProperty
   { get { return 0; } }
}
class SubBase : Base
{
   public override void DoSomething() { }
   public override int SomeProperty
   { get { return 0; } }
}
```

Інший шлях – визначити у похідному класі метод із модифікатором **new**, який буде ***перекривати*** (приховувати) відповідний метод базового класу. Тоді для екземплярів базового та похідних класів викликатимуться відповідно різні методи.

```
class Base
{
   public void DoSomething() { }
   public int SomeProperty
   { get { return 0; } }
}

class SubBase : Base
{
   public new void DoSomething() { }
   public new int SomeProperty
   { get { return 0; } }
}
```

Коли при визначенні члену класу використовується модифікатор **new**, цей новий член класу викликається замість відповідного члену базового класу, який виявляється перекритим. Тим не менше і перекритий член класу може бути викликаний, якщо екземпляр похідного класу приводиться до типу базового класу:

```
SubBase sb = new SubBase ();

sb.DoSomething ();  // викликається новий метод

Base b = (Base)sb;

b.DoSomething ();  // викликається старий метод
```

Зробимо поліморфним метод **setBonus()**. Для цього у базовому класі **BankAccount** змінимо його визначення наступним чином (з'являється службове слово **virtual**):

```
public virtual void setBonus()  // бонус 6 %
{  money += money * 0.06;  }
```

У похідному класі **DepositAccount** визначимо свій метод **setBonus()** із службовим словом **override**:

```
        // бонус залежить від терміну
public override void setBonus()
{  money += money * 0.15 * termin / 4;  }
```

Тепер звертання **myDepositAccount.setBonus()** означає виклик методу **setBonus()** похідного класу **DepositAccount**, а звертання **myBankAccount.setBonus()** – виклик однойменного методу базового класу **BankAccount**. Проте це ще далеко не всі можливості віртуальних функцій.

12.6. Віртуальні функції – більш детальний погляд.

Придивимось більш уважно до віртуальних функцій. З цією метою розглянемо досить загальний клас **Base** з віртуальним методом **DoSomething()**, який просто надсилає повідомляє про себе. У похідному класі **SubBase** визначимо відповідний метод із модифікатором **override**. Створимо також похідний клас наступного рівня спадкування **SubSubBase**. Крім того створимо зовнішню функцію **fun(Base sb)**, яка приймає аргумент типу **Base**. Всередині цієї функції міститься звертання до методу **DoSomething()**. Оскільки функція **fun** може бути викликана як для екземпляру класу **Base**, так і для екземплярів класів **SubBase** та **SubSubBase**, рішення про те, який саме метод **DoSomething()** (базового чи одного з похідних класів) буде викликаний, неможливо прийняти у момент компіляції програми, а лише під час виконання. Цей механізм називається ***динамічним поліморфізмом*** або ***динамічною диспетчеризацією методів***. (Відповідний механізм в мові C++ називають ***пізнім зв'язуванням***.)

```
using System;
namespace UseOverride1
{
   class Base
   {
      public virtual void DoSomething()
      { Console.WriteLine("Ми у базовому класі Base"); }
   }
   class SubBase : Base
   {
      public override void DoSomething()
```

```
        { Console.WriteLine("Ми у похідному класі SubBase"); }
    }
    class SubSubBase : SubBase
    {
        public override void DoSomething()
        { Console.WriteLine("Ми у похідному класі SubSubBase"); }
    }
    class Program
    { // метод може приймати аргумент типу Base (або SubBase,
     //  SubSubBase)
        static void fun(Base sb)
        { sb.DoSomething(); }
        static void Main()
        {
            Base b = new Base();
            SubBase sb = new SubBase();
            SubSubBase ssb = new SubSubBase();
            // тут працює звичайний поліморфізм
            b.DoSomething();        // метод класу Base
            sb.DoSomething();       // метод класу SubBase
            ssb.DoSomething();      // метод класу SubSubBase
            Console.WriteLine("-------------------------");
            // тут працює динамічний поліморфізм
            fun(b); // викликається метод DoSomething() класу Base
                    // викликається метод DoSomething() класу SubBase
            fun(sb);
            // викликається метод DoSomething() класу SubSubBase
            fun(ssb);
        }
    }
}
```

Запуск цієї програми призведе до наступних повідомлень на екрані:

Ми у базовому класі Base

Ми у похідному класі SubBase

Ми у похідному класі SubSubBase

Ми у базовому класі Base

Ми у похідному класі SubBase

Ми у похідному класі SubSubBase

Таким чином, результат роботи функції **fun** визначається типом аргументу, переданого їй ***під час виклику***, тобто у момент виконання програми. В залежності від типу аргументу – **Base**, **SubBase** чи **SubSubBase** викликається метод **DoSomething()** відповідного класу.

Поля класів не можуть бути віртуальними. Заміщеними можуть бути тільки методи, властивості, індексатори та події (останній тип членів класів не обговорювався).

Крім того, якщо похідний клас заміщає деякий віртуальний член базового класу, то коли екземпляр похідного класу приводиться до типу базового класу, для нього все рівно **буде викликаний (override) метод похідного класу**. Для перевірки включимо у попередній приклад наступне визначення: **Base bb = new SubBase();** та виклик функції **fun(bb);** і методу **bb.DoSomething();**. Екземпляр **bb** безперечно має тип **Base**, проте повідомлення, які додатково з'являться на екрані, переконують – для нього в обох випадках викликається метод **DoSomething()** похідного класу **SubBase**.

Що станеться, якщо у базовому та похідних класах визначити методи з однаковою сигнатурою, проте не помітити їх модифікаторами **virtual** та **override** відповідно? Внесемо такі зміни у текст попереднього прикладу. При компіляції одержимо зауваження від компілятора щодо методів **DoSomething()** похідних класів «…hides inherited member DoSomething()…» – приховує успадкований метод **DoSomething()**. Тим не менше програма виконується, і на екрані побачимо:

Ми у базовому класі Base

Ми у похідному класі SubBase

Ми у похідному класі SubSubBase

Ми у базовому класі Base

Ми у базовому класі Base

Ми у базовому класі Base

Це означає, що безпосереднє звертання до методів **DoSomething()** через екземпляри базового та похідних класів, тобто звичайний поліморфізм, спрацьовує очікуваним чином – ми дійсно звертаємось до методів різних класів. А от функція **fun** у даному разі ***завжди*** викликатиме лише метод базового класу, адже таким є тип її параметра – динамічного поліморфізму немає. Щоб позбавитись від зауваження компілятора у випадку, коли ви ***свідомо*** використовуєте приховування методу базового класу, досить при визначенні відповідного методу у похідних класах додати службове слово **new**, як уже завважувалось у попередньому розділі і як радить у своєму зауваженні компілятор.

Цілком зрозуміло також, що віртуальна функція не може мати модифікатор **private** – адже тоді вона не буде доступна, а отже, і не може бути заміщена у похідних класах.

Будь-який клас може зупинити заміщення віртуального методу у своїх похідних класах. Для цього необхідно використати службове слово **sealed** (закритий печаткою) перед словом **override** у визначенні такого методу. Наприклад,

```
class SubBase : Base
  {  // Цей метод не буде заміщатись у похідних класах
     public sealed override void DoSomething()
     {
       // код методу
     }
  }
```

Клас також може бути визначений із модифікатором **sealed** - такий клас не може бути базовим, таким чином від нього неможливо утворювати похідні класи. Заборона спадкування може дещо прискорити звертання до методів такого класу.

Якщо вас продовжує турбувати питання, а навіщо взагалі заміщати методи базового класу, можливо, ви не до кінця усвідомлюєте основні ідеї об'єктно-орієнтованого програмування. Заміщення методів – дуже потужний механізм, який дозволяє в ієрархії класів лише додавати та уточнювати функціональність уже існуючим класам, а не дублювати методи базових класів. До речі, код методів, які заміщають деякий віртуальний метод, найчастіше включає безпосередній виклик цього віртуального методу базового класу, саме з метою позбавитись від дублювання фрагментів коду. Для звертання до будь-якого члену базового класу (не лише віртуального) використовується службове слово **base**:

```
public class Base
{
   public virtual void DoSomething()
   { // код методу
   }
}
public class SubBase : Base
{
   public override void DoSomething()
   { // код методу
   }
}
public class SubSubBase : SubBase
{
   public override void DoSomething()
   {
      // викликається метод DoSomething() класу SubBase
```

```
      base.DoSomething();
      // далі код, що визначає особливості класу SubSubBase
    }
}
```

Розглянемо дещо більш складний приклад. Тут в базовому класі **Base** визначені 3 віртуальні методи – **Meth1()**, **Meth2()** та **Meth3()**. У похідному класі **SubBase** методи **Meth1()** та **Meth2()** заміщаються (**override**), а метод **Meth3()** перекривається (**new**). У наступному похідному класі **SubSubBase** заміщається лише метод **Meth1()**, а **Meth2()** та **Meth3()** – перекриваються.

```
using System;
namespace UseOverride1
{
  class Base
  {
    public virtual void Meth1()
    { Console.WriteLine("Ми у базовому класі Base - Meth1"); }
    public virtual void Meth2()
    { Console.WriteLine("Ми у базовому класі Base - Meth2"); }
    public virtual void Meth3()
    { Console.WriteLine("Ми у базовому класі Base - Meth3"); }
  }
  class SubBase : Base
  {
    public override void Meth1()
    {
      Console.WriteLine(
      "Ми у похідному класі SubBase - Meth1");
    }
    public override void Meth2()
    {
      Console.WriteLine(
      "Ми у похідному класі SubBase - Meth2");
    }
    public new virtual void Meth3()
    {
      Console.WriteLine(
      "Ми у похідному класі SubBase - Meth3");
    }
  }
  class SubSubBase : SubBase
  {
    public override void Meth1()
    {
      Console.WriteLine(
       "Ми у похідному класі SubSubBase - Meth1");
    }
    public new virtual void Meth2()
    {
      Console.WriteLine(
       "Ми у похідному класі SubSubBase - Meth2");
    }
```

```
        public new virtual void Meth3()
        {
            Console.WriteLine(
             "Ми у похідному класі SubSubBase - Meth3");
        }
    }
    class Program
    {
        static void fun(Base b)
        { b.Meth1(); b.Meth2(); b.Meth3(); }
        static void Main()
        {
            Base b = new Base();
            SubBase sb = new SubBase();
            SubSubBase ssb = new SubSubBase();
            // тут працює ранній поліморфізм
            b.Meth1(); b.Meth2(); b.Meth3();   // b має тип Base
            Console.WriteLine("----------------------------");
            // sb має тип SubBase
            sb.Meth1(); sb.Meth2(); sb.Meth3();
            Console.WriteLine("----------------------------");
            // ssb має тип SubSubBase
            ssb.Meth1(); ssb.Meth2(); ssb.Meth3();
            Console.WriteLine("-------- Функція fun : ---------");
            // тут працює пізній поліморфізм
            fun(b);
            fun(sb);
            fun(ssb);
        }
    }
}
```

Результати роботи цього прикладу будуть такими:

Ми у базовому класі Base - Meth1

Ми у базовому класі Base – Meth2

Ми у базовому класі Base – Meth3

Ми у похідному класі SubBase - Meth1

Ми у похідному класі SubBase – Meth2

Ми у похідному класі SubBase – Meth2

Ми у похідному класі SubSubBase - Meth1

Ми у похідному класі SubSubBase – Meth2

Ми у похідному класі SubSubBase – Meth3

------------- Функція fun : ----------------

Ми у базовому класі Base - Meth1

Ми у базовому класі Base – Meth2

Ми у базовому класі Base – Meth3

Ми у похідному класі SubBase - Meth1

Ми у похідному класі SubBase – Meth2

Ми у базовому класі Base – Meth3

Ми у похідному класі SubSubBase - Meth1

Ми у похідному класі SubBase – Meth2

Ми у базовому класі Base – Meth3

Перші три блоки вихідних повідомлень означають, що на рівні екземплярів **b**, **sb** та **ssb** відповідно класів **Base**, **SubBase** та **SubSubBase** викликаються методи **Meth1()**, **Meth2()** та **Meth3()**саме цих класів. Коли ж ці методи викликаються опосередковано через деяку функцію **fun()**, яка приймає параметр типу **Base** (а отже, і будь-якого похідного класу), то метод **Meth3()** заміщається для аргументів будь-якого похідного класу, метод **Meth2()** заміщається лише для аргументу класу **SubBase**, а метод **Meth3()** завжди викликається для базового класу, оскільки в кожному похідному класі він був перекритий власними неполіморфними методами.

12.7. Абстрактні методи та класи.

Можлива ситуація, коли у базовому класі неможливо реалізувати деякий метод. Проте зрозуміло, що подібний метод зі своєю особливою функціональністю буде присутній у похідних класах. В такому випадку є сенс визначити «порожній» віртуальний метод у базовому класі (свого роду «заглушку»), визначивши його інтерфейс, а у похідних класах слушним чином визначити коди відповідних методів, що його заміщають. Засіб, який примушує всі похідні класи обов'язково замістити такий «невизначений» метод, полягає у використанні модифікатору **abstract**:

abstract <тип_результату> <ідентифікатор_методу> (<параметри_методу>);

Такий метод автоматично є віртуальним і його ***не треба*** додатково помічати службовим словом **virtual**.

Абстрактний метод, таким чином, взагалі не має тіла. Абстрактними можуть бути не лише методи, але й властивості. Клас, який містить принаймні один абстрактний член класу, теж має бути визначений як **abstract**, адже від нього не можна утворювати екземпляри – такі класи використовуються лише для спадкування. Класичним прикладом демонстрації абстрактного класу є клас геометричних форм – не існує загальної формули для визначення площі

абстрактної геометричної фігури. Проте всім відомо як визначити площі, наприклад, квадрата або прямокутника чи кола. Розглянемо приклад, у якому визначається абстрактний клас **Shape**, що містить закрите текстове поле **shapeType** – тип фігури, та відкриту властивість **ShapeType**, яка реалізує доступ до нього. Ця властивість використовується у конструкторі класу. Властивість **Area** базового класу визначається як абстрактна. Крім того, вказується, що властивості, які будуть заміщати її у похідних класах, будуть мати лише аксесор **get** – властивість дозволяє лише читання. Зверніть увагу, що у базовому класі заміщається також метод **ToString()**, успадкований від класу **System.Object**. Тепер він буде виводити на екран тип фігури та її площу.

```
using System;
namespace AbstractClass
{   // абстрактний клас - геометрична форма
   abstract class Shape
   {
      private string shapeType;
      public string ShapeType
      {
         get { return shapeType; }
         set { shapeType = value; }
      }
      public Shape(string s)  // конструктор
      { ShapeType = s; }  // працює set-аксесор властивості
                     // Площа Area є властивістю read-only –
                     // буде потрібний лише get-аксесор
      public abstract double Area // абстрактна властивість
      { get; }
      // заміщаємо метод класу Object
      public override string ToString()
      {
         return ShapeType + ": площа = " + String.Format("{0:F2}",
              Area);
      }
   }
   class Square : Shape // похідний клас - квадрат
   {
      private double side;
      // працює конструктор базового класу
      public Square(double side_, string type) : base(type)
      { side = side_; }
      // заміщаємо абстрактний метод
      public override double Area
      {
         get    // повертає площу квадрата із стороною side
         { return side * side; }
      }
   }
   class Circle : Shape // похідний клас - коло
   {
      private double radius;
```

```
        public Circle(double radius_, string type) // конструктор
          : base(type)       // працює конструктор базового класу
        { radius = radius_; }
        // заміщаємо абстрактний метод
        public override double Area
        {
            get        // повертає площу кола радіуса radius
            { return radius * radius * Math.PI; }
        }
    }
    class Rectangle : Shape  // похідний клас - прямокутник
    {
        private double width;
        private double height;
        public Rectangle(     // конструктор
             double width_, double height_, string type)
             : base(type)  // працює конструктор базового класу
        {
            width = width_;
            height = height_;
        }
        // заміщаємо абстрактний метод
        // повертає площу прямокутника ширини width і висоти height
        public override double Area
        {
            get
            { return width * height; }
        }
    }
    class Program
    {
        static void Main()
        {
            Shape[] shapes = {     // масив геометричних форм
        new Square(10, "Квадрат"),
        new Circle(10, "Коло"),
        new Rectangle( 10, 5, "Прямокутник")
                         };
            Console.WriteLine("Масив геометричних фігур");
            foreach (Shape s in shapes)
                // тут викликається заміщений метод ToString()
                Console.WriteLine(s);
        }
    }
}
```

Результатом роботи цієї програми буде наступний вигляд екрану:

Масив геометричних фігур

Квадрат: площа = 100

Коло: площа = 314,16

Прямокутник: площа = 50

Завдання для самоконтролю

1. Використовуючи інтерфейси та спадкування реалізуйте еволюцію: дисковий телефон -> кнопочний телефон –>мобільний телефон з чорно-> білим екраном – >мобільний телефон з кольоровим екраном – >смартфон.
2. Створити клас Автомобіль. Створити інтерфейси автомеханік, водій, заправщик. Реалізувати код, якщо автомобіль потрапляє в руки автомеханіку, автомеханік ремонтує двигун, якщо водію – водій ключає фари і жме на клаксон, якщо заправщику то заправляє бензин.

Розділ 13. Структури та переліки в мові C#.

13.1. Структури.

Структури в мові C# відносяться до value-типу, а за своєю реалізацією багато в чому нагадують класи, які відносяться до reference-типу. Так само, як і класи можуть містити дані-члени та методи для їх обробки. В деяких випадках перевагою використання структур перед класами є прямий доступ, а не через адресну змінну, як це має місце у випадку класів. Ця обставина спрощує доступ, та часом може призводити до деякого прискорення виконання програми.

Для визначення структури використовується наступний синтаксис:

```
struct <ідентифікатор_структури> {
// декларації даних-членів структури
<специфікатор_доступу> <тип> <ідентифікатор_змінної_1>;
<специфікатор_доступу> <тип> <ідентифікатор_змінної_2>;
//...
// декларації методів-членів класу
<специфікатор_доступу> <тип_результату>
<ідентифікатор_методу_1> (<параметри>)
{
// код методу
}
//...
<специфікатор_доступу> <тип_результату>
<ідентифікатор_методу_N> (<параметри>)
{
// код методу
}
}
```

Структури не підтримують спадкування, хоча самі успадковані від класу **System.Object** – тому мають всі методи цього класу. Так само, як і у класів, членами структур можуть бути поля, методи, властивості, індексатори (а також делегати та події, які залишились поза межами нашого обговорення мови C#). Проте в зв'язку з відсутністю спадкування члени структури не можуть мати модифікаторів **protected**, **virtual** або **abstract**. Крім того, для

структур можна визначати і конструктори з єдиним обмеженням – конструктор за замовчуванням (без параметрів) для будь-якої структури визначений автоматично і не може бути змінений, отже, конструктор структури повинен мати параметри.

Розглянемо приклад. Нижче визначені дві структури: **MyStruct**, яка містить приватне поле **field** та відкриту властивість **Field**, пов'язану з цим полем. Друга структура реалізує комплексне число та містить два відкритих поля **Re** та **Im**. В принципі, оскільки структура є типом-значенням, для її створення операція **new** не є обов'язковою, проте в цьому випадку перед використанням такої структури її поля мають бути ***повністю*** проініціалізовані – саме так в методі **Main()** створений екземпляр **c** структури **Complex**. Для створення екземпляра структури **MyStruct** операція **new** виявляється необхідною, оскільки її поле **field** закрите, а отже, не може бути проініціалізоване безпосередньо. При спробі створити екземпляр **MyStruct m** без **new**, звертання до аксесора **set** властивості **Field** цієї структури спровокує синтаксичну помилку «**Use of unassigned local variable 'm'**» . Саме тому при визначенні структур всі їх поля як правило визначають відкритими (**public**).

```
using System;
namespace UseStruct0
{
   struct MyStruct
   {
      private int field;
      public int Field
      {
         get { return field; }
         set { field = value; }
      }
   }
   struct Complex
   {
      public double Re, Im;
   }
   class Program
   {
      static void Main()
      {
         // без new не вдається створити структуру
         MyStruct m = new MyStruct();
         m.Field = 100;
         Console.WriteLine("зміст структури - {0}", m.Field);
         Complex c;       // ця структура створена без new
                          // і може використовуватись, оскільки
         c.Re = 1; c.Im = 2;  // повністю ініціалізована
         Console.WriteLine(
               "Комплексне число: {0} + {1}i", c.Re, c.Im);
      }
   }
}
```

В результаті виконання цієї програми на екрані побачимо:

зміст структури - 100

Комплексне число: 1 + 2i

Підсумуємо те, що відомо про структури, які часто вважають спрощеною моделлю класів:

1. Структура – це тип-значення, а не посилання, тому зберігається у стеку або є вбудованою у клас, якщо членом класу є екземпляр даної структури. Звідси обмеження на час існування таких об'єктів, а отже, спрощений процес їх знищення без застосування системи GC.
2. Структури не підтримують спадкування.
3. Конструктор за замовчуванням (без параметрів) автоматично генерується компілятором і не може бути перевизначеним. Проте структура може мати перевантажені конструктори із параметрами.
4. Структура може бути створена і використана без оператора **new** в разі, коли всі її поля проініціалізовані безпосередньо.

Вище зазначалось про сенс використання структур – адже звертання до них відбувається не опосередковано через адресну змінну, як у випадку із класами, а безпосередньо. З одного боку це дійсно спрощує та прискорює процес створення та знищення екземплярів структур, з іншого боку передача структури у метод відбувається ***за значенням***, тобто у стеку має створюватись копія аргументу-структури, що може уповільнювати виклик методів. Для запобігання цьому варто передавати структури у функції як ref-аргументи. Проте при цьому не варто забувати, що метод у такому разі має прямий доступ до структури, а отже, впливатиме на значення її членів.

У наступному прикладі розглядається структура, що містить інформацію про пацієнта медичного закладу. Зверніть увагу, в цій структурі визначений конструктор для ініціалізації членів структури. При цьому частина полів з іменем (**name**), віком (**age**) та адресою пацієнта є відкритою (**adress**). Діагноз (**diagnosis**) – закрите поле, яким керує властивість **Diagnosis**. Проте попередня ініціалізація цього поля у конструкторі деяким значенням, наприклад, порожнім стрінгом, є обов'язковою. Інакше, використання властивості **Diagnosis** викличе помилку компіляції, пов'язану із використанням неініціалізованої структури.

```
namespace UseStruct1
{
   struct HelthCard
   {
      public HelthCard(string name_, string adress_, int age_)
      {
         name = name_; adress = adress_; age = age_;
         diagnosis = "";       // це присвоєння обов'язкове
      }
      public string name;
```

```
        public string adress;
        public int age;
        private string diagnosis;
        public string Diagnosis
        {
            get { return diagnosis; }
            set { diagnosis = value; }
        }
    }
    class Program
    {
        static void Main()
        {
            HelthCard men = new HelthCard("Іванов", "гурт. 1", 18);
            men.Diagnosis = "здоровий";
            HelthCard women = new HelthCard("Петрова", "гурт.1", 18);
            women.Diagnosis = "ОРВІ";
            Console.WriteLine("Пацієнт - {0}, адреса - {1},

              вік - { 2}, діагноз - { 3}
            ", men.name, men.adress,
             men.age, men.Diagnosis);
            Console.WriteLine("Пацієнт - {0}, адреса - {1}, вік –
             { 2}, діагноз - { 3}
            ", women.name, women.adress,
              women.age, women.Diagnosis);
        }
    }
}
```

Після виконання цього прикладу побачимо наступні повідомлення:

Пацієнт – Іванов, адреса - гурт. 1, вік - 18, діагноз – здоровий

Пацієнт – Петрова, адреса - гурт. 1, вік - 18, діагноз - ОРВІ

У наступному прикладі членами класу **ComplexVector** є екземпляри структури **Complex**, подібної до розглянутої вище. Для структури визначений конструктор, а також заміщений метод **ToString()** класу **System.object**, таким чином, щоб комплексне число виводилось у прийнятому в математиці форматі.

```
using System;
namespace UseStruct2
{
    struct Complex
    {
        public double Re, Im;
        // конструктор структури
        public Complex(double Re_, double Im_)
        {
            Re = Re_; Im = Im_;
        }
        // заміщаємо метод ToString() із класу object
        public override string ToString()
```

```
        {
            if (Im > 0)
                return String.Format("{0} + {1}i ", Re, Im);
            else if (Im < 0)
                return String.Format("{0} - {1}i ", Re,
                                    Math.Abs(Im));
            else return String.Format("{0}", Re);
        }
    }
    class ComplexVector
    {
        public Complex x, y, z;  // вбудовані структури
                              // конструктор класу
        public ComplexVector(Complex x_, Complex y_, Complex z_)
        {
            x = new Complex(x_.Re, x_.Im);
            y = new Complex(y_.Re, y_.Im);
            z = new Complex(z_.Re, z_.Im);
        }
    }
    class Program
    {
        static void Main()
        {
            Complex a = new Complex(1, 1);
            Complex b = new Complex(-2, -2);
            Complex c = new Complex(3, 0);
            ComplexVector cv = new ComplexVector(a, b, c);
            Console.WriteLine("Координати комплексного вектора:");
            Console.WriteLine("x = " + cv.x.ToString());
            Console.WriteLine("y = " + cv.y.ToString());
            Console.WriteLine("z = " + cv.z.ToString());
        }
    }
}
```

Результат роботи цієї програми наступний:

Координати комплексного вектора:

x = 1 + 1i

y = -2 - 2i

z = 3

Важливо усвідомлювати різницю між присвоєнням екземплярів класів та структур. У наступному прикладі визначена структура комплексних чисел **ComplexSruct** та клас комплексних чисел **ComplexClass**. Їх визначення ідентичні між собою, крім службових слів **struct** та **class**. Проте це визначає принципову різницю між екземплярами структури та класу. Присвоєння екземплярів структур **st1 = st2** означає копіювання значень полів структур. Присвоєння екземплярів класів **cl1 = cl2** означає дещо більше – тепер ці змінні вказують на одну область пам’яті. Це видно у наступних інструкціях –

після зміни значень полів структури **st2**, друга структура **st1** зберігає своє попереднє значення. Ті самі інструкції для екземплярів **cl1** та **cl2** змінюють їх одночасно, адже ці змінні вказують на одну область пам'яті!

```
using System;
namespace AssignStruct
{
   struct ComplexSruct
   {
      public double Re, Im;
      // конструктор структури
      public ComplexSruct(double Re_, double Im_)
      { Re = Re_; Im = Im_; }
      //заміщаємо метод ToString() із System.Object
      public override string ToString()
      {
         if (Im > 0)
            return String.Format("{0} + {1}i ", Re, Im);
         else if (Im < 0)
            return String.Format("{0} - {1}i ", Re,
                              Math.Abs(Im));
         else return String.Format("{0}", Re);
      }
   }
   class ComplexClass
   {
      public double Re, Im;
      // конструктор класу
      public ComplexClass(double Re_, double Im_)
      { Re = Re_; Im = Im_; }
      //заміщаємо метод ToString() із System.Object
      public override string ToString()
      {
         if (Im > 0)
            return String.Format("{0} + {1}i ", Re, Im);
         else if (Im < 0)
            return String.Format("{0} - {1}i ", Re,
                              Math.Abs(Im));
         else return String.Format("{0}", Re);
      }
   }
   class Program
   {
      static void Main()
      {
         ComplexSruct st1 = new ComplexSruct(1, 1);
         ComplexSruct st2 = new ComplexSruct(2, 2);
         ComplexClass cl1 = new ComplexClass(1, 1);
         ComplexClass cl2 = new ComplexClass(2, 2);
         st1 = st2;      // Тут копіюються значення структур
         cl1 = cl2;     // Тут копіюються посилання на клас
         Console.WriteLine("------------------------");
         Console.WriteLine("Структура st1: " + st1.ToString());
         Console.WriteLine("Класс     cl1: " + cl1.ToString());
```

```
            st2.Re = 10; st2.Im = 20;   // st2 не залежить від st1
                                // cl2 та cl1 - один і той самий клас
            cl2.Re = 10; cl2.Im = 20;
            Console.WriteLine("------------------------");
            Console.WriteLine("Структура st1: " + st1.ToString());
            Console.WriteLine("Структура st2: " + st2.ToString());
            Console.WriteLine("Класс     cl1: " + cl1.ToString());
            Console.WriteLine("Класс     cl2: " + cl2.ToString());
        }
    }
}
```

Після виконання цього прикладу на екрані побачимо:

Структура st1: 2 + 2i

Класс   cl1: 2 + 2i

Структура st1: 2 + 2i

Структура st2: 10 + 20i

Класс   cl1: 10 + 20i

Класс   cl2: 10 + 20i

13.2. Переліки.

Перелік в мові C#, як і у багатьох інших мовах, це визначений користувачем злічений тип-значення. Іншими словами – це визначений список імен із зумовленими цілими значеннями. Синтаксис визначення переліку – наступний:

enum <ідентифікатор_переліку> {<список_переліку>};

Тут **<список_переліку>** являє собою список констант, відокремлених комами. За замовчуванням значенням першої константи є 0, а кожна наступна константа має значення на одиницю більшу від попередньої. Втім будь-якій із констант можна присвоїти довільне ціле значення. Розглянемо приклади.

```
using System;
namespace Use_enum_0
{
    class Program
    {
        // Цей перелік задає дні тижня
        public enum Days { Sun, Mn, Tu, We, Th, Fr, St };
        static void Main()
        {
            for (Days d = Days.Sun; d <= Days.St; d++)
                Console.WriteLine("День - {0}, значення - {1}", d,
                                (int)d);
        }
    }
```

}

Тут визначений перелік **Days** для іменування днів тижня. В циклі змінна **d** типу **Days** пробігає значення всіх днів тижня. На екран виводиться назва дня та його значення. Як видно з прикладу доступ до членів переліку відбувається з допомогою крапкової нотації із ідентифікатору переліку та дня тижня: **Days.Sun** або **Days.St**. На екрані будуть наступні повідомлення:

День - Sun, значення - 0

День - Mn, значення – 1

День - Tu, значення – 2

День - We, значення – 3

День - Th, значення – 4

День - Fr, значення – 5

День - St, значення - 6

Як вже зазначалось, члени переліку можуть мати довільні значення, вказані при визначенні списку переліку. У наступному прикладі визначається подібний перелік.

```
using System;
namespace Use_enum_1
{
   class Program
   {
      public enum Flags { ok = 0, error = 128, avost = 1024 };
      static void Main()
      {
         Flags flag;
         // Тут має бути ініціалізація змінної flag
         switch (flag)
         {
            case Flags.ok:
               Console.WriteLine("Успішне завершення");
               break;
            case Flags.error:
               Console.WriteLine("Помилка виконання");
               break;
            case Flags.avost:
               Console.WriteLine("Аварійне завершення");
               break;
            default:
               Console.WriteLine("неприпустиме значення");
               break;
         }
      }
   }
}
```

Перелік **Flags** містить прапорці завершення деякого процесу. Можливим варіантам **ok**, **error** та **avost** приписані відповідно константи **0**, **128** та **1024**. У перемикачі **switch** аналізується значення змінної **flag** та виводиться відповідне повідомлення. Змінна **flag** може бути проініціалізована різними способами, наприклад, безпосередньо: **flag = (Flags) 1024;**. При цьому ініціалізація константою, якої немає у переліку, призведе до активації гілку **default** перемикача **switch**.

Оскільки переліки походять від типу **System.Enum**, вони мають низку корисних методів. Зокрема статичний метод **GetName**() повертає назву заданої у переліку константи. Наприклад, інструкція

Console.WriteLine(Enum.GetName(typeof(Flags), 128));

у попередньому прикладі призведе до появи на екрані наступного повідомлення: **error**. Метод **Format()** дозволяє визначити, якому саме елементу переліку відповідає значення змінної. При цьому це значення можна одержати у десятковому, шістнадцядковому або текстовому форматі в залежності від того, яким символом заданий вид форматування: **"d"**, **"x"** чи **"g"**. Так, задавши низку інструкцій:

```
flag = (Flags) 1024;
Console.WriteLine(Enum.Format(typeof(Flags), flag, "d"));
Console.WriteLine(Enum.Format(typeof(Flags), flag, "x"));
Console.WriteLine(Enum.Format(typeof(Flags), flag, "g"));
```

на екрані побачимо:

1024

00000400

avost.

Корисним також є метод **Enum.GetNames**(). Він повертає масив назв констант у вказаному переліку. Цей масив далі може бути використаний, наприклад, у циклі **foreach**. Таким чином, задавши код:

```
Console.WriteLine("Значення прапорців наступні:");
  foreach (string s in Enum.GetNames(typeof(Flags)))
    Console.WriteLine(s);
```

на екрані одержимо всі назви констант переліку **Flags**:

ok

error

avost

System.Enum також має методи, що повертають кількість констант у переліку, сам масив констант, результат перевірки, чи є дана константа елементом переліку та інші корисні методи.

Розділ 14. Класи колекцій і протоколи ітерації

14.1. Використання колекцій.

Будь-який набір даних, об'єктів можна назвати колекцією.

Масив з простору імен System.Array - це теж свого роду колекція, але статична, його не можна ні розширити, ні стиснути при необхідності.

.NET Являє собою безліч класів колекцій, що дозволяють працювати з наборами даних більш гнучко ніж масив.

Кожен об'єкт, що міститься в колекції, називається її елементом. Деякі колекції зберігають дані як прямий список елементів, інші містять списки пар ключів і значень.

Популярні класи колекцій:

System.Collections.ArrayList class – подібний масиву, але на відміну від масиву може розширюватися.

System.Collections.Stack class – для тимчасового розміщення об'єктів, які після використання будуть видалені. Ця колекція працює за принципом LIFO (last-in-first-out) - послідовність вилучення об'єктів протилежна послідовності їх приміщення в колекцію.

System.Collections.Queue class – головною відмінністю даної колекції від Stack полягає в принципі FIFO (first-in-first-out), за яким працює черга. Елементи вилучаються з колекції в тій же послідовності, в якій вони були в її розміщені. Індексація до елементів даної колекції не застосовується.

System.Collections.SortedList class – колекція пар об'єктів ключ-значення, які сортуються по ключу і за індексом. За замовчуванням ємність колекції дорівнює 0. При додаванні першого елемента ємність встановлюється в 16 елементів.

System.Collections.Hashtable class – являє собою набори даних, що містять пари ключ-значення, причому обидва вони відносяться до типу Object, а це означає, що тип і ключа і його парного значення може бути зовсім будь-який. Ця колекція відноситься до так званої групи словників.

Всі класи колекцій знаходяться в просторі імен System.Collections.

У доповненні до цього простору існує ще один простір імен System.Collections.Specialized, який містить не настільки широко відомих строго типізованих класів колекцій.

System.Collections.Specialized.ListDictionary class - клас поводиться як Hashtable, але перевершує його за швидкістю при роботі з 10 і менш елементами. При роботі з великою кількістю елементів його використовувати не рекомендується.

System.Collections.Specialized.HybridDictionary class – складається з двох вбудованих колекцій, List Dictionary і Hashtable. Одночасно

використовується тільки один з цих класів. ListDictionary використовується поки колекція містить 10 або менше елементів, а потім відбувається перемикання на використання Hashtable. Якщо колекція вже переключалася на Hashtable, переключитися назад вже не можливо.

System.Collections.Specialized.CollectionsUtil class – створює колекції, що ігнорують регістр в рядках. Містить 2 статичних методи: один для створення. регістронезалежного Hashtable, а інший для створення регістронезалежного SortedList.

System.Collections.Specialized.NameValueCollection class – это коллекция состоит из пар ключ – значение, оба из которых относятся к типу string. Особенность: может хранить множество значений при единственном ключе.

System.Collections.Specialized.StringCollection class – це звичайний список елементів типу string. У ньому можна поміщати null елементи і дубльовані елементи. Цей список чутливий до регістру.

System.Collections.Specialized.StringDictionary class – це Hashtable, містять ключі і значення типу string. Перш ніж бути доданими до колекції, ключі приводяться до нижнього регістру.

Інтерфейси колекцій

System.Collections.ICollection interface – визначає розмір, нумератори і синхронізовані методи для узагальнених колекцій.

System.Collections.IDictionary interface – це базовий інтерфейс не узагальнених колекцій, що представляють набори пари ключ-значення. Кожна пара значень повинна мати унікальний ключ. Значення може бути null і йому не обов'язково бути унікальним.

System.Collections.IDictionaryEnumerator interface – перебирає елементи узагальненої колекції. Нумератор може бути використаний для читання даних з колекції, але не змінювати їх.

System.Collections.IList interface – представляє узагальнену колекцію об'єктів, доступ до яких можна отримати за індексом. Реалізація інтерфейсу ділиться на три категорії: read-only, fixed-size, variable-size.

Об'єкт IList read-only не може бути модифікований.

Об'єкт IList fixed-size не дозволяє видалення і додавання елементів, але дозволяє модифікувати існуючі елементи.

Об'єкт IList variable-size дозволяє видалення, додавання і модифікацію елементів.

System.Collections.IComparer interface – представляє метод для порівняння двох об'єктів.

System.Collections.IEnumerable interface – підтримує нумератори, який виробляє ітерації по не узагальненій колекції.

System.Collections.IEnumerator interface – підтримує просту ітерацію по не узагальненій колекції. Є базовим інтерфейсом для не узагальнених нумераторів.

14.2. Generics.

У С # є два окремих механізму для створення коду, який повторно використовується: спадкування та узагальнення.

Спадкування висловлює можливість повторного використання базового типу, а узагальнення висловлюють ідею повторного використання "шаблону", що містить типи — "заглушки".

У .NET Framework, починаючи з версії 2.0, з'явилася можливість використовувати в якості параметрів типи даних. Ця можливість реалізується за допомогою Generics.

Generics дозволяють створювати узагальнені:

класи, структури, інтерфейси, делегати, методи.

Параметри типів використовуються для вказівки типів: змінних класу, параметрів функції, повертаємих значень функцій, локальних змінних.

Наприклад: у колекції ArrayList зберігаються дані типу object для того щоб в цю колекцію можна було помістити змінні будь-якого типу.

Найчастіше в колекції зберігаються змінні одного і того ж типу.

При отриманні даних з колекції треба вказувати явне приведення типу.

Можна легко допустити помилку приведення при отриманні даних з колекції, тобто помістити в колекцію змінну одного типу, а при добуванні виконати приведення до іншого типу.

Іншим недоліком є зниження продуктивності через виконання упаковки і розпаковки при зберіганні в колекції змінних структурних типів.

Методи Generics колекцій приймають і повертають параметри узагальненого типу. Тому при розміщенні даних у колекцію і при зверненні до цих даних упаковка і розпаковка не відбувається.

Якщо в неузагальнених колекціях зберігати об'єкти посилальних типів даних, то зниження продуктивності відбуватися не буде.

Використання Generics дає наступні переваги:

безпеку типів. З Generics колекцію можна помістити тільки об'єкти певного типу (зазначеного при розкритті шаблону);

більш простий і зрозумілий код;

підвищення продуктивності.

Створення Generic класів.

Параметри типу можна ввести в оголошення класів, структур, інтерфейсів, делегатів і методів.

Узагальнений тип або метод може мати декілька параметрів

class Dictionary <TKey, TValue> {…}

Створимо екземпляр

var myDic = new Dictionary <int, string>();

Імена узагальнених типів і методів можна перевантажувати, якщо вони відрізняються кількістю параметрів типу:

class A<T> { }

class A<T1, T2> { }

Правила спадкування від generic класів:

якщо від generic класу успадковується неузагальнений клас, клас-спадкоємець повинен конкретизувати параметр типу;

при реалізації generic віртуальних методів похідний неузагальнений клас повинен конкретизувати параметр типу;

якщо від generic класу успадковується інший generic клас, в ньому необхідно враховувати обмеження типу, зазначені в базовому класі.

З появою концепції Generics в FCL практично для всіх класів неузагальнених колекцій з'явилися відповідні узагальнені, таблиця 14.1.

Таблиця 14.1.

Відповілність між типами колекцій

Generic колекції	Неузагальнені колекції
Collection<T>	CollectionBase
List<T>	ArrayList
Dictionary<TKey, TValue>	Hashtable
SortedList<TKey, TValue>	SortedList
Stack<T>	Stack
Queue<T>	Queue

Microsoft рекомендує скрізь, де це можливо, використовувати узагальнені колекції замість неузагальнених.

Використання неузагальнених колекцій може бути виправданим тільки якщо дійсно існує необхідність зберігати в одній колекції об'єкти різних типів.

Усередині generic класу можна оголошувати вкладені класи. Ці класи також будуть узагальненими, навіть якщо вони не мають власних параметрів.

У вкладених класах можна використовувати параметр типу, зазначений у зовнішньому класі.

У вкладеному класі можна оголошувати власний список параметрів типу.

У цьому випадку вкладений клас буде параметризовано двома наборами типів: власним і типом зовнішнього класу.

Використання обмежень.

Для параметра типу можна вказати обмеження, що вказують яким вимогам повинен задовольняти тип даних, використовуваний замість цього параметра.

where T: struct — параметр типу повинен успадковувати system.ValueType (бути структурним типом);

where T: class — параметр типу не повинен успадковувати system.ValueType (бути посилальним типом);

where T: new() — клас повинен мати конструктор за замовченням (вказується останнім);

where T: BaseClass — параметр типу повинен бути похідним класом від вказаного базового класу;

where T: Interface — параметр типу повинен реалізовувати вказаний інтерфейс.

Створення Generic інтерфейсів.

При створенні інтерфейсів, як і при створенні класів, можна використовувати узагальнені параметри типу. Особливо зручно використовувати узагальнені інтерфейси при роботі зі структурними типами, тому робота зі структурними типами через неузагальнений інтерфейс привела б до упаковки і розпаковки, а також до втрати безпеки типів під час компіляції.

При реалізації generic інтерфейсу в неузагальненому класі необхідно конкретизувати аргументи типу.

У generic класі можна реалізувати інтерфейс, використовуючи узагальнені параметри типу.

Створення Generic делегатів.

Підтримка узагальнених делегатів дозволяє передавати методам зворотнього виклику будь-які типи об'єктів, забезпечуючи при цьому безпеку викликів.

Створення Generic методів.

Метод, параметризований якимось типом даних, може знаходитися в узагальненому класі.

При виклику методу можна після імені вказувати реальний тип даних, але можна цього і не робити, тому тип даних визначається автоматично за типом параметрів, переданих в метод.

14.3. Ітератори

Для циклічного звернення до елементів колекції найчастіше простіше (та й краще) організувати цикл foreach.

Якщо потрібно створити клас, що містить об'єкти, що перераховуються в циклі foreach, то в цьому класі слід реалізувати інтерфейси IEnumerator і IEnumerable.

Іншими словами, для того щоб звернутися в циклі foreach до об'єкта класу, що визначається користувачем, необхідно реалізувати інтерфейси IEnumerator і IEnumerable в їх узагальненій або неузагальненій формі.

Простіше скористатися ітератором, який являє собою метод, оператор або аксесуар, який повертає по черзі члени сукупності об'єктів від початку до кінця.

Реалізувавши ітератор, можна звертатися до об'єктів класу, визначеним користувачем, в циклі foreach.

Позначення yield служить в мові C # в якості контекстного ключового слова. Це означає, що воно має спеціальне призначення тільки в блоці ітератора — повернути.

Для передчасного переривання ітератора служить наступна форма оператора yield: yield break;

Коли цей оператор виконується, ітератор повідомляє про те, що досягнутий кінець колекції. А це, по суті, зупиняє сам ітератор.

У ітераторі допускається застосування декількох операторів yield. Але кожен такий оператор повинен повертати наступний елемент в колекції.

Іменований итератор являє собою метод.

Синтаксис:

```
public IEnumerable ім'я_ітератора(список_параметрів) {
  // ...
  yield return obj;
}
```

ім'я ітератора - конкретне ім'я методу;

список параметрів - від нуля до декількох параметрів, що передаються;

obj - наступний об'єкт, що повертається ітератором.

Як тільки іменований ітератор буде створено, його можна використовувати, наприклад для управління циклом foreach.

14.4. LINQ

Мова інтегрованих запитів (Language Integrated Quary – LINQ) вбудовує синтаксис запитів в мову програмування C# і забезпечує можливість доступу до різних джерел даних за допомогою одного і того ж синтаксису.

Завдання для самоконтролю

1. Створіть клас CarCollection. Реалізуйте в найпростішому наближенні можливість використання його екземпляру для створення машин. Мінімальний інтерфейс взаємодії з екземпляром, повинен включати методи додавання машин з назвою і роком випуску, індексатор для отримання значення елементу по вказаному індексу і властивість тільки для зчитування і отримання загальної кількості елементів.
2. Створіть метод, який в якості аргументу приймає масив студентів і повертає колекцію студентів, які добре вчяться і вчасно здають лабораторні. Для формування колекції використовуйте оператор yield.
3. Створити клас AppleBucket (основна властивість: кількість яблук у корзині). Створити клас HungryStudent (основна властивість: кількість яблук які може з'їсти студент). Створити клас AppleDistributor, який дозволяє розподілити набір корзинок (IEnumerable<AppleBucket>) з яблуками серед голодних студентів (IEnumerable<HungryStudent>).

Розділ 15. Обробка винятків. Застосування конструкцій checked і unchecked

15.1. Ієрархія винятків

При виникненні помилки генерується виняток.

У C# виняток є об'єктом, який створюється і "викидається" (throw) у разі виникнення помилки.

Для перехоплення винятків і створення винятків існує базовий клас System.Exception.

В BCL (.NET Framework Class Library (FCL) існує багато класів винятків (спадкоємців System.Exception).

Розробник може створювати власні класи.

Усі винятки діляться на дві групи SystemExeption і ApplicationException.

SystemException – це клас винятків, які генеруються CLR або є винятками загальної природи і можуть бути згенеровані будь-яким додатком.

ApplicationException – від цього класу повинні успадковуватися користувальницькі винятків, специфічні для застосування. Однак успадкування спеціальних типів винятків від класу ApplicationException більше не є рекомендованим прийомом, оскільки це не дає ніяких відчутних переваг.

CompositionException і похідні від нього класи відносяться до простору імен System.ComponentModel.Composition. Цей простір імен відповідає за роботу з частинами і компонентами, які динамічно завантажуються.

15.2.Основи обробки винятків

Синтаксис:

```
try
{
    //код, в якому може
    //виникнути виняток
}
catch(тип винятку)
{
    //обробка винятку
}
finally
{
    //звільнення ресурсів
}
```

Синтаксис генерації винятку:

throw new Констр. класу винятку()

Блок **try** містить код, що формує частину нормальних дій програми, які потенційно можуть зіткнутися з серйозними помилковими ситуаціями

Блок **catch** містить код, який повинен виконуватися при виникненні винятку. При оголошення блоку catch вказується тип винятку, для обробки якого він призначений. Якщо блок try завершився без генерації виключення, блоки catch не виконуються

Блок **finally** зазвичай містить очистку ресурсів, а також інші дії, які необхідно гарантовано виконати після завершення блоку try і catch. Наприклад: закриття файлу, закриття з'єднання з базою даних

Блок **finally** виконується завжди незалежно від виникнення винятку в блоці try.

У одного блоку try може бути кілька блоків catch для обробки винятків різних типів.

При виникненні винятку пошук обробника починається з 1-го блоку catch, тому слід спочатку розміщувати обробники для винятків похідних класів, потім базових.

Вкладені блоки try.

```
try
{
    //точка A
    try
    {
        //точка B
    }
    catch(тип винятку)
    {
        //точка C
    }
    finally
    {
        //звільнення ресурсів
    }
}
catch(тип винятку)
{
    //обробка винятку
}
finally
{
    //звільнення ресурсів
}
```

Повторне генерування винятків.

У блоці catch крім виконання деякої обробки, наприклад відкат транзакцій в базі даних, може генеруватися повторно виняток для повідомлення методу, що здійснював виклик про виникнення проблеми.

Трасування стека при винятках.

Якщо в блоці catch звернутися до властивості Stack Trace, то CLR побудує рядок, що ідентифікує всі методи, викликані між виникненням виключення і його обробкою.

Інформація про компонент, який викликав.

Часто корисно отримувати інформацію про те, де виникла та або інша помилка.

У C# є можливість отримання цієї інформації за допомогою атрибутів і необов'язкових параметрів.

До параметрів можуть бути застосовані атрибути CallerLineNumber, CallerFilePath і CallerMemberName, визначені в просторі імен System.Runtime.CompilerServices.

Створення винятків користувача.

ApplicationException – від цього класу повинні успадковуватися винятки користувачів, специфічні для застосування.

Data

Повертає колекцію пар ключ / значення, що надають додаткові відомості про виняток, визначені користувачем.

HelpLink

Отримує або задає посилання на файл довідки, пов'язаний з цим винятком.

HResult

Повертає або задає HRESULT - кодоване числове значення, присвоєне певному винятку

InnerException

Повертає екземпляр класу Exception, який викликав поточний виняток.

Message Отримує повідомлення, яке описує поточний виняток.

Властивості *Date, Message, HelpLink і InnerException* повинні заповняться кодом, який генерує виняток, шляхом встановлення потрібних значень безпосередньо перед генерацією винятку.

Source

Повертає або задає ім'я програми або об'єкта, що викликав помилку.

StackTrace

Отримує строкове представлення безпосередніх кадрів в стеку виклику.

TargetSite

Повертає метод, який створив поточний виняток.

Свойства *Date, Massage, HelpLink и InnerException* должны заполнятся кодом, сгенерировавшим исключение, путём установки нужных значений непосредственно перед генерацией исключения.

15.3.Застосування конструкцій checked і unchecked.

При виконанні арифметичних операцій з цілочисельними типами даних може виникати переповнення, тобто ситуація при якій кількість двійкових розрядів отриманого результату перевищує розрядність змінної, в яку цей результат записується.

Переповнення виникає:

у виразах, які використовують арифметичні оператори:

++ - - (унарний) + - * /;

при виконанні явного перетворення цілочисельних типів.

CLR може:

проігнорувати переповнення і відкинути старші розряди (за замовчуванням);

згенерувати виключення OverflowException.

Можна явно задати режим контролю переповнення за допомогою ключових слів checked і unchecked.

Ключове слово checked задає режим контролю переповнення з генерацією винятку.

Ключове слово unchecked задає ігнорування виникнення переповнення.

Ключові слова checked і unchecked можуть використовуватися як оператори в рядку перетворення.

Якщо режим контролю переповнення не вказано явно, він визначається опцією компілятора /checked. Цю опцію можна задати за допомогою властивостей пректу.

Завдання для самоконтролю

1. Створити структуру Color з набором полів Red, Green, Blue з типом byte. Розробити статичний метод CreateFromRGB(), який на вхід приймає три параметру типу int і повертає новий екземпляр структури Color. Зробити контроль переповнення за допомогою: умовних операцій if/else, блоку try/catch, ключових слів checked/unchecked. Проаналізувати який з цих методів працює швидше.

Розділ 16.Взаємодія з файловою системою

16.1. Модель потоків в C#. Простір System.IO. клас Stream

Операції, пов'язані з введенням / виведенням в .NET здійснюються за допомогою потоків.

Потік даних (stream) - це об'єкт, який використовується для передачі даних (не плутати зі thread).

Дуже часто в ролі зовнішнього джерела виступає файл, але це не завжди так.

Наприклад:

читання або запис даних в мережі з використанням якогось мережевого протоколу, де метою є отримання або відправлення цих даних з іншого комп'ютера;

читання або запис даних в іменований канал;

читання або запис даних в певну область пам'яті.

Базовий клас для всіх потоків - System.IO.Stream.

Відділення концепції передачі даних від конкретного джерела істотно спрощує заміну джерела даних.

Об'єкти потоків самі містять безліч узагальненого коду, відповідального за переміщення даних між зовнішніми джерелами і змінними, які визначені в коді.

Утримуючи цей код окремо від концепції конкретного джерела даних, можна значно поліпшити ступінь його повторного використання в різних обставинах.

16.2. Аналіз байтових, символьних, бінарних класів потоків.

Багато потоків, які працюють безпосередньо з пристроями введення / виводу, вміють писати / читати тільки послідовності байтів.

У .NET існують класи-спадкоємці Stream, які здатні перетворити записані або прочитані дані в масив байт і, навпаки.

Такими класами є класи StreamReader, StreamWriter, BinaryReader, BinaryWriter та інші.

Для роботи з символами і рядками можуть служити класи System.IO.StreamReader і System.IO.StreamWriter.

Ці класи призначені для читання і запису даних у текстовий файл.

Класи BinaryWriter і BinaryReader призначені для запису простих типів даних в потік у вигляді бінарних значень, а також рядків в певному кодуванні.

Класи BinaryWriter і BinaryReader забезпечують додаткове форматування двійкових даних, дозволяючи безпосередньо читати або записувати вміст змінних C# в відповідний потік.

16.3. Використання класів FileStream; StreamWriter, StreamReader; BinaryWriter, BinaryReader; Directory, DirectoryInfo; File, FileInfo для файлових операцій.

Використання класу FileStream для файлових операцій.

Клас FileStream дозволяє проводити операції читання і запису з файлів.

Змінна типу FileMode (перерахування) описує яким чином файл повинен бути відкритий операційною системою.

Append

Відкриває файл, якщо він існує, і знаходить кінець файлу; або створює новий файл. Для цього потрібен дозвіл FileIOPermissionAccess.Append. FileMode.Append можна використовувати тільки разом з FileAccess.Write.

Create

Вказує, що операційна система повинна створювати новий файл. Якщо файл вже існує, він буде перезаписано. Для цього потрібен дозвіл FileIOPermissionAccess.Write. Значення FileMode.Create еквівалентно вимогу використовувати значення CreateNew, якщо файл не існує, і значення Truncate в іншому випадку. Якщо файл вже існує, але є прихованим, створюється виняток UnauthorizedAccessException.

CreateNew

Вказує, що операційна система повинна створювати новий файл. Для цього потрібен дозвіл FileIOPermissionAccess.Write. Якщо файл вже існує, створюється виняток IOException.

Open

Вказує, що операційна система повинна відкрити існуючий файл. Можливість відкрити даний файл залежить від значення, що задається перерахуванням FileAccess. Виняток System.IO.FileNotFoundException створюється, якщо файл не існує.

OpenOrCreate

Вказує, що операційна система повинна відкрити файл, якщо він існує, в іншому випадку повинен бути створений новий файл. Якщо файл відкритий за допомогою FileAccess.Read, потрібен дозвіл FileIOPermissionAccess.Read. Якщо доступ до файлу є FileAccess.Write, потрібен дозвіл FileIOPermissionAccess.Write. Якщо файл відкритий за допомогою FileAccess.ReadWrite, необхідні дозволи FileIOPermissionAccess.Read і FileIOPermissionAccess.Write.

Truncate

Вказує, що операційна система повинна відкрити існуючий файл. Якщо файл відкритий, він повинен бути усічений таким чином, щоб його розмір став дорівнювати нулю байтів. Для цього потрібен дозвіл FileIOPermissionAccess.Write. Спроби виконати читання з файлу, відкритого за допомогою FileMode.Truncate, викликають виняток ArgumentException.

Змінна типу File Access. Описує яким чином здійснюється доступ до файлу:

запис (FileAccess.Write),

читання (FileAccess.Read),

запис і читання (FileAccess.ReadWrite).

Змінна типу FileShare. Дозволяє управляти доступом, який інші об'єкти FileStream можуть здійснювати до цього файлу.

Delete

Дозволяє подальше видалення файлу.

Inheritable

Дозволяє спадкування дескриптора файлу дочірніми процесами. У Win32 безпосередня підтримка цієї властивості не забезпечена.

None

Відхиляє спільне використання поточного файлу. Будь-який запит на відкриття файлу (даним процесом або іншим процесом) не виконується до тих пір, поки файл не буде закритий.

Read

Дозволяє подальше відкриття файлу для читання. Якщо цей прапор не заданий, будь-який запит на відкриття файлу для читання (даним процесом або іншим процесом) не виконується до тих пір, поки файл не буде закритий. Однак, навіть якщо цей прапор заданий, для доступу до даного файлу можуть знадобитися додаткові дозволи.

ReadWrite

Дозволяє подальше відкриття файлу для читання або запису. Якщо цей прапор не заданий, будь-який запит на відкриття файлу для запису або читання (даним процесом або іншим процесом) не виконується до тих пір, поки файл не буде закритий. Однак, навіть якщо цей прапор заданий, для доступу до даного файлу можуть знадобитися додаткові дозволи.

Write

Дозволяє подальше відкриття файлу для запису. Якщо цей прапор не заданий, будь-який запит на відкриття файлу для запису (даним процесом або іншим процесом) не виконується до тих пір, поки файл не буде закритий. Однак, навіть якщо цей прапор заданий, для доступу до даного файлу можуть знадобитися додаткові дозволи.

Використання класу StreamWriter, StreamReader для файлових операцій.

TextWriter - клас, що представляє модуль запису, який може записувати послідовні набори символів. Це абстрактний клас.

StreamWriter - клас, що реалізує TextWriter для запису символів в потік в певному кодуванні.

TextReader - клас, що представляє засіб читання, що дозволяє зчитувати послідовні набори символів.

StreamReader - клас, що реалізує об'єкт TextReader, який зчитує символи з потоку байтів в певному кодуванні.

Використання класу BinaryWriter і BinaryReader для файлових операцій.

BinaryWriter - клас, що записує прості типи даних в потік як двійкові значення і підтримує запис рядків у певному кодуванні.

BinaryReader - клас, що зчитує примітивні типи даних як двійкові значення в заданому кодуванні.

Використання класів Directory, DirectoryInfo для файлових операцій.

Directory - клас, що надає статичні методи для створення, переміщення і перерахування в каталогах і вкладених каталогах. Цей клас не успадковується.

DirectoryInfo - клас, що надає методи екземпляра класу для створення, переміщення і перерахування в каталогах і підкаталогах. Цей клас не успадковується.

Класи *Directory і DirectoryInfo* призначені для роботи з папками.

Використання класів File і FileInfo для файлових операцій.

File - клас, що надає статичні методи для створення, копіювання, видалення, переміщення і відкриття файлів, а також допомагає при створенні об'єктів FileStream.

FileInfo - клас, що надає властивості і методи примірника для створення, копіювання, видалення, переміщення і відкриття файлів, а також дозволяє створювати об'єкти FileStream. Цей клас не успадковується.

Якщо ви виконуєте кілька операцій на тому ж файлі, то може бути більш ефективно використовувати T: System.IO.FileInfo методи примірника замість відповідних статичних методів класу T:System.IO.File, бо перевірка безпеки не буде завжди необхідно.

16.4. Використання класу Path.

Path - клас, що виконує операції для екземплярів класу String, що містять відомості про шлях до файлу або каталогу.

Екземпляри класу Path не створюються.

Він представляє статичні методи, які спрощують виконання операцій з маршрутними іменами.

Завдання для самоконтролю

1. Створити програму-шифрувач, яка шифрує будь-який файл збільшуючи/зменшуючи кожен байт файлу на певну кількість одиниць. Реалізувати можливість розшифрування зашифрованого файлу.
2. Створити програму-шифрувач, яка шифрує будь-який файл використовуючи XOR з певною бітовою маскою. Реалізувати можливість розшифрування зашифрованого файлу.
3. Створити програму-архівувач, яка архівує файл використовуючи алгоритм RLE. Реалізувати можливість розархівування заархівованого файлу.

Розділ 17. Делегати. Лямбда-вирази

17.1. Делегати

Поняття делегата.

Делегат - це об'єкт, який може посилатися на метод (за допомогою посилання викликати даний метод).

Технічно делегат - посилальний тип, інкапсулює метод з вказаною сигнатурою і типом, що повертається.

Делегат пов'язує метод, що викликає з методом, що викликається.

Делегат має два аспекти: тип і екземпляр.

Тип делегата визначає протокол, якого дотримуються метод, що викликає, та метод, що викликається. Протокол складається зі списку параметрів типу і типу значення, що повертається.

Екземпляр делегата - це об'єкт, який посилається на один (або декілька) методів, що викликаються і які підтримують цей протокол.

Синтаксис оголошення делегата.

delegate type DelegateName (list)

delegate - ключове слово для оголошення делегата

type - тип, що повертається

DelegateName - ім'я делегата

list - список параметрів, необхідних методам при виклику їх за допомогою делегата.

Оголошений делегат може викликати тільки ті методи, які повертають вказаний тип type і приймають список параметрів list.

До делегатів можна застосовувати модифікатори доступу.

Цілі і завдання делегатів.

Делегат пов'язує метод, що викликає з методом, що викликається.

Делегати використовуються коли методу необхідно передати в якості параметрів не дані, а деякий метод, і під час компіляції невідомо, який саме метод потрібно викликати для обробки даних.

Наприклад:

багатопоточність;

графічний інтерфейс.

Виклик декількох методів через делегат.

Делегати в C# підтримують множинне делегування (multicasting), а саме можливість викликати з одного делегата кілька методів.

Для реалізації множинного делегування делегат повинен повертати тип void.

Додавання методів до списку викликів делегата здійснюється за допомогою операції +=.

За допомогою операції -= метод можна видалити з ланцюжка делегата.

Можна об'єднати делегати за допомогою операції +, отримуючи новий делегат, який буде викликати методи об'єднаних делегатів.

Базові класи для делегатів.

Клас System.Delegate є базовим для типів делегатів.

Тільки система і компілятори можуть явно наслідувати класи Delegate і MulticastDelegate

Неприпустимо створювати нові типи, похідні від типу делегата.

Методи делегатів.

Є три загальнодоступні методи.

Invoke () - можливо, головний з них, оскільки він використовується для синхронного виклику кожного з методів, підтримуваних об'єктом делегата; це означає, що код, який викликає повинен очікувати завершення виклику, перш ніж продовжити свою роботу.

Може здатися дивним, що синхронний метод Invoke () не повинен викликатися явно в коді C#. Invoke () викликається "за лаштунками", коли застосовується відповідний синтаксис C#.

Методи BeginInvoke () і EndInvoke () пропонують можливість виклику поточного методу асинхронним чином, в окремому потоці виконання.

Хоча в бібліотеках базових класів .NET передбачено цілий простір імен, присвячене багатопоточність програмуванню (System.Threading), делегати пропонують цю функціональність в готовому вигляді.

17.2. Анонімні методи

Анонімний метод - це блок коду, який застосовується в якості параметра делегата.

Правила використання анонімних методів:

всередині анонімного методу не можна застосовувати оператори управління потоком виконання (break, goto або continue) для переходу до коду, що знаходиться за межами тіла анонімного методу;

передача керування із зовнішнього коду всередину анонімного методу недопускається;

в тілі анонімного методу не може бути присутнім небезпечний код;

параметри ref і out, які застосовуються поза анонімного методу, в його тілі не доступні.

Починаючи з версії C# 3.0, замість анонімних методів можна використовувати лямбда-вирази.

17.3. Лямбда-вирази

Лямбда-вирази підтримуються в багатьох мовах програмування (Common Lisp, Ruby, Python, PHP, C#, F#, Visual Basic .NET, C++, Java, Scala та інш.)

Зазвичай лямбда-вираз допускає замикання на лексичний контекст, в якому цей вираз використано.

Лямбда-вирази в .NET приймають дві форми:

форма, яка найбільш прямо замінює анонімний метод, являє собою блок коду, укладений у фігурні дужки це - пряма заміна анонімних методів.

лямбда-вирази, з іншого боку, надають ще більш скорочений спосіб оголошувати анонімний метод і не вимагають ні коду у фігурних дужках, ні оператора return.

Обидва типи лямбда-виразів можуть бути перетворені в делегати.

Лямбда-вирази - це вбудовані вирази, аналогічні анонімним методам, але більш гнучкі.

Вони широко використовуються в запитах LINQ, виражених з використанням синтаксису методів.

Лямбда-оператор.

У всіх лямбда-виразах використовується лямбда-оператор =>, який читається як «переходить в».

Ліва частина лямбда-оператора визначає параметри введення (якщо такі є), а права частина містить вираз або блок оператора.

Лямбда-вираз x => x * 5 читається як «функція x, яка переходить у x, помножене на 5».

Маркер => називається лямбда-оператором. Він використовується в лямбда-виразах для відділення вхідних змінних з лівого боку від тіла лямбда-виразу з правого сторони.

Замикання.

Замикання (з точки зору реалізації) - це особливий вид функції. Вона визначена в тілі іншої функції і створюється кожного разу під час її виконання.

Синтаксично це виглядає як функція, що знаходиться цілком в тілі іншої функції. При цьому вкладена внутрішня функція містить посилання на локальні змінні зовнішньої функції. Щоразу при виконанні зовнішньої функції відбувається створення нового екземпляра внутрішньої функції, з новими посиланнями на змінні зовнішньої функції.

Замикання (англ. Closure) в програмуванні - функція, в тілі якої присутні посилання на змінні, оголошені поза тілом цієї функції в навколишньому коді і які не є її параметрами. Кажучи іншою мовою, замикання - функція, яка посилається на вільні змінні у своєму контексті.

Замикання, так само як і екземпляр об'єкта, є спосіб представлення функціональності і даних, пов'язаних і упакованих разом.

Замикання змінної, це захоплення значення (посилання) якогось об'єкта за допомогою лямбда-виразу. Замикання дозволяє використовувати і змінювати значення об'єкта без додаткових зусиль, таких як "out" або "ref" модифікаторів до параметрів.

Завдання для самоконтролю

1. Існує конвеєр з можливостями: зняти розміри, відрізати, заточить, нарізати різьбу, просверлити, пофарбувати, протестувати, запакувати. Створити комбіновані делегати, які будуть виконувати весь, або частину функціоналу.
2. Створіть масив делегатів. Непарні методи, пов'язані з делегатами із масива, повертають на консоль повідомлення. «Перший». Парні метоли, пов'язані з делегатами з масива повертають на консоль повідомлення «Другий»

Розділ 18.Події

18.1. Події

Подія - це повідомлення про виникнення деякої дії (натискання кнопки, вибір елемента списку і таке інше).

Відправником може бути будь-який компонент форми, або сам додаток.

Відправник - це об'єкт який генерує подію.

Об'єкт-відправник оповіщає інші об'єкти про те, що відбулася подія. Отримати повідомлення про подію може будь-який інший об'єкт, якому це необхідно. Такі об'єкти називаються одержувачем подій.

Для того, щоб об'єкт знав про виникнення події, його необхідно підписати на цю подію.

Відправник повідомлення не знає про існування одержувача.

Для реалізації події використовуються делегати.

Об'єкт відправник визначає делегат, а кожен об'єкт одержувач додає свій метод-обробник в ланцюжок посилань делегата.

Виклик ланцюжка методів здійснює об'єкт відправник.

Синтаксис оголошення події.

event DelegateName EventName;

event - ключове слово для оголошення події;

DelegateName - ім'я використовуваного делегата;

EventName - ім'я події.

Існує угода, за якою делегати обробники подій повертають тип void і мають два параметри: посилання на об'єкт відправника (генератор події) і посилання на об'єкт, що містить інформацію і параметри події.

18.2. Застосування подій

Головне достоїнство подій - об'єкт-відправник і об'єкт-одержувач незалежні один від одного.

Це робить код більш гнучким, дозволяє розробнику міняти реакцію на подію, додавати нових передплатників.

У C# існує форма event-інструкції, яка дозволяє використовувати засоби доступу до подій. Ці засоби доступу дають можливість управляти реалізацією списку обробників подій.

18.3. Стандартна модель подій

У платформі .NET Framework в основі моделі події лежить існування делегата події, що зв'язує подію з його обробником. Для виклику події потрібні два елементи:

делегат, який посилається на метод, що викликається у відповідь на подію;

клас, який містить дані події, якщо подія містить дані.

Делегат - це тип, який визначає сигнатуру, іншими словами тип значення, і тип списку параметрів методу. Тип делегата можна

використовувати для оголошення змінної, яка може посилатися на будь-який метод з тією ж сигнатурою, що і у делегата.

Стандартна сигнатура делегата обробника подій визначає метод, який не повертає значення. Перший параметр цього методу типу Object і відноситься до примірника, який ініціює подія. Його другий параметр є похідним від типу EventArgs і містить дані події. Якщо подія не створює дані події, то другий параметр буде просто значенням поля EventArgs.Empty. В іншому випадку другий параметр тип, похідний від EventArgs і надає всі поля або властивості, необхідне для зберігання даних події.

У бібліотеці класів .NET Framework події базуються на делегаті EventHandler і базовому класі EventArgs.

EventHandler<TEventArgs> - делегат, який представляє метод, який буде обробляти подію, коли подія надає дані.

Завдання для самоконтролю

1. Використовуючи події, напишіть клас Car.В метод Accekerate (прискорення) при наборі швидкості 200км/год, спрацьовує подія і екземпляр класу Бортовий комп'ютер отримує повідомлення: «Будьте обережні на дорозі». При наборі швидкості 250км/год спрацьовує подія і бортовий комп'ютер отримує повідомлення «Двигун перегрівся».
2. В проекті Windows Form створіть калькулятор. Спробуйте для кнопок 0,1….9 використовувати один і той же обробник події натиснення на кнопку. А вже в середині обробника провіряти яка кнопка була нажата. Спробуйте реалізувати даний проект не використовуючи конструктор форм(власноруч написати метод InitializeComponent(). Причому використовуйте свої класи Button і TextBox(наслідуючись від системних, додайте їм якийсь колір і т.д.)

Розділ 19. Багатопоточність

19.1. Багатопоточність.

Для застосування багатопоточності існує декілька причин.

Припустимо, у додатку робиться звернення до якогось сервера в мережі, яке може зайняти певний час. Не хочеться, щоб користувальницький інтерфейс через це блокувався, і користувачеві довелося просто чекати моменту, коли від сервера повернеться відповідь.

Користувач може виконувати в цей час якісь інші дії або взагалі скасувати відправлений серверу запит.

Для всіх видів активності, що вимагають очікування, наприклад, через отримання доступу до файлу, бази даних або мережі, може запускатися новий потік, що дозволяє виконувати в цей же час інші завдання. Численні потоки одного і того ж процесу можуть одночасно виконуватися різними ЦП або, що частіше зустрічається в наші дні, різними ядрами одного багатоядерного ЦП.

Через те, що багато потоків виконуються в один і той же час, при отриманні ними доступу до одних і тих же даних можуть виникати проблеми. Щоб цього не відбувалося, повинні бути реалізовані механізми синхронізації.

Потік (thread) являє собою незалежну послідовність інструкцій в програмі.

Приміром, під час введення якогось коду C# у вікні редактора Visual Studio проводиться аналіз на предмет різних синтаксичних помилок. Цей аналіз здійснюється окремим фоновим потоком. Те ж саме відбувається і в засобі перевірки орфографії в Microsoft Word.

Один потік очікує введення даних користувачем, а інший в цей час виконує у фоновому режимі деякий аналіз. Третій потік може зберігати записувані дані в тимчасовий файл, а четвертий - завантажувати додаткові дані з Інтернету.

Потоки плануються до виконання операційною системою. У будь-якого потоку мається пріоритет, лічильник команд, який вказує на місце в програмі, де відбувається обробка, і стек, в якому зберігаються локальні змінні потоку. Стек у кожного потоку виглядає по-своєму, але пам'ять для програмного коду і купа розділяються серед всіх потоків, які функціонують усередині одного процесу.

Це дозволяє потокам усередині одного процесу швидко взаємодіяти між собою, оскільки всі потоки процесу звертаються до однієї і тієї ж віртуальної пам'яті. Однак це також і ускладнює справу, оскільки дає можливість безлічі потоків змінювати одну і ту ж область пам'яті.

19.2. Основи багатопоточної обробки.

Багатозадачність-властивість операційної системи або середовища програмування забезпечувати можливість паралельної (або псевдопараллельной) обробки декількох процесів

Розрізняють два різновиди багатозадачності: на основі процесів і на основі потоків(рис.19.1.-19.2.).

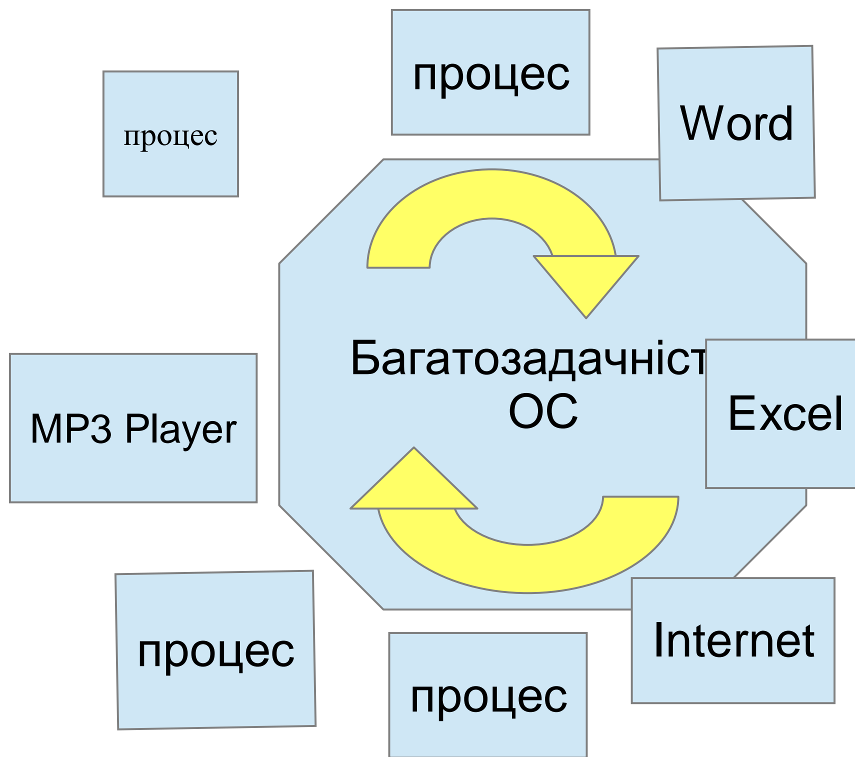

Рисунок 19.1 Багатопоточність на основі процесів

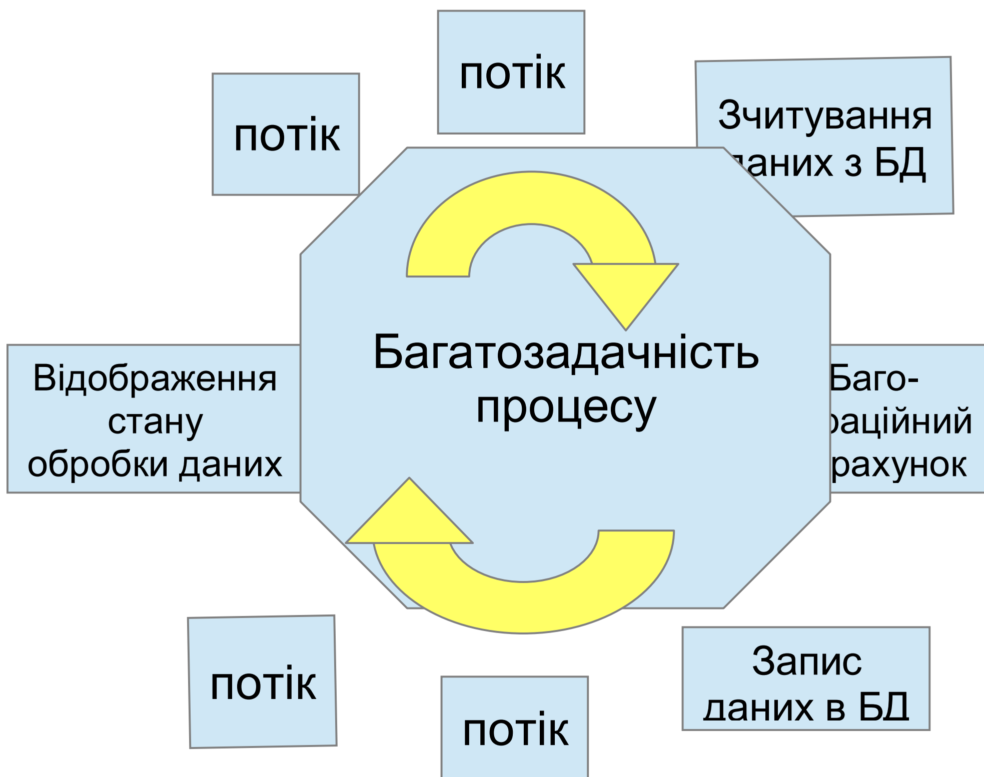

Рисунок 19.2 Багатопоточність на основі процесів

Процес відповідає за управління ресурсами, до числа яких належить віртуальна пам'ять і дескриптори Windows, і містить як мінімум один потік.

Наявність хоча б одного потоку є обов'язковим для виконання будь-якої програми. Тому багатозадачність на основі процесів - це засіб, завдяки якому на комп'ютері можуть паралельно виконуватися дві програми і більше.

Так, багатозадачність на основі процесів дозволяє одночасно виконувати програми текстового редактора, електронних таблиць і перегляду вмісту в Інтернеті. При організації багатозадачності на основі процесів програма є найменшою одиницею коду, виконання якої може координувати планувальник завдань.

Потік являє собою координуючу одиницю виконуваного коду. Своїм походженням цей термін зобов'язаний поняттю "потік виконання".

При організації багатозадачності на основі потоків у кожного процесу повинен бути принаймні один потік, хоча їх може бути і більше. Це означає, що в одній програмі одночасно можуть вирішуватися два завдання і більше.

Наприклад, текст може форматироваться в редакторі тексту одночасно з його висновком на друк, за умови, що обидва ці дії виконуються у двох окремих потоках.

Відмінності в багатозадачності на основі процесів і потоків можуть бути зведені до наступного:

багатозадачність на основі процесів організовується для паралельного виконання програм, а

багатозадачність на основі потоків - для паралельного виконання окремих частин однієї програми.

Головна перевага многопоточной обробки полягає в тому, що вона дозволяє писати програми, які працюють дуже ефективно завдяки можливості вигідно використовувати час простою, що неминуче виникає в ході виконання більшості програм.

Як відомо, більшість пристроїв введення-виведення, будь то пристрої, підключені до мережевих портів, накопичувачі на дисках або клавіатура, працюють набагато повільніше, ніж центральний процесор (ЦП). Тому більшу частину свого часу програмою доводиться очікувати відправки даних на пристрій вводу-виводу або прийому інформації з нього.

А завдяки многопоточной обробці програма може вирішувати якусь іншу задачу під час вимушеного простою.

Наприклад, у той час як

одна частина програми відправляє файл через з'єднання з Інтернетом,

інша її частина може виконувати читання текстової інформації, що вводиться з клавіатури, а

третя - здійснювати буферизацію чергового блоку даних, що відправляються.

Потік може перебувати в одному з декількох станів.

В цілому, потік може бути що виконуються;

готовим до виконання, як тільки він отримає час і ресурси ЦП;

зупиненим, тобто тимчасово не виконуються;

відновленим надалі;

заблокованим в очікуванні ресурсів для свого виконання; а також

завершеним, коли його виконання закінчено і не може бути відновлено.

19.3. Простір імен System.Threading.

Класи, які підтримують багатопоточний програмування, визначені в просторі імен System.Threading.

using System.Threading;

Простір імен System.Threading містить різні типи, що дозволяють створювати багатопотокові програми.

Головним серед них є клас Thread, що представляє окремий потік.

Різні потоки можуть використати одні й ті ж методи.

Локальні змінні метода зберігаються окремо в різних потоках.

Властивість Thread.Priority.

Повертає або задає значення, яке вказує на планований пріоритет потоку.

Потоку може бути присвоєно одне з таких значень Thread Priority:

- Highest
- AboveNormal
- Normal
- BelowNormal
- lowest

Але операційній системі не завжди потрібне надання пріоритету потоку.

Властивість Thread.IsBackground.

Властивість Thread.IsBackground - Повертає або задає значення, що показує, чи є потік фоновим.

Значення true, якщо цей потік є або стане фоновим потоком;

в іншому випадку - значення false.

Потік може бути фоновим або потоком переднього плану.

Фонові потоки ідентичні потокам переднього плану, за винятком того, що фонові потоки не запобігають завершенню процесу.

Коли всі основні потоки, що належать процесу, завершилися, загальномовне середовище виконання завершує процес. Всі фонові потоки, які залишилися зупиняються не закінчивши роботу.

19.4. Делегати ThreadStart, ParameterizedThreadStart.

Делегат ThreadStart представляє метод, який виконується в зазначеному потоці Thread.

ThreadStart не дозволяє передавати дані в потік.

Делегат ParameterizedThreadStart представляє метод, який виконується в зазначеному потоці Thread.

ParameterizedThreadStart дозволяє передавати дані в потік.

При створені потоків можна використовувати анонімні методи та лямда-вирази

19.5. Оператор lock. Monitor - клас.

Ключове слово lock не дозволить ні одному потоку увійти в важливий розділ коду в той момент, коли в ньому знаходиться інший потік. При спробі

входу іншого потоку в заблокований код потрібно дочекатися зняття блокування об'єкта.

Monitor - клас - Надає механізм для синхронізації доступу до об'єктів.

Monitor Клас управляє доступом до об'єктів, надаючи блокування об'єкта одному потоку. Блокування об'єктів надає можливість обмеження доступу до частини коду, зазвичай званої критичної секції. Поки потік володіє блокуванням об'єкта, ніякий інший потік не може отримати блокіровку. Можна також використовувати Monitor клас, щоб гарантувати, що ніякому іншому потоку не дозволено доступ до розділу додатка коду, що виконується власником блокування, поки інший потік не буде виконувати код, використовуючи інший об'єкт з блокуванням.

19.6. Асинхронне програмування. Async, Await.

Ви можете уникнути появи вузьких місць продуктивності і збільшити швидкість відгуку вашого застосування за допомогою асинхронного програмування. Однак традиційні методи для створення асинхронних додатків можуть бути складними, що робить їх важкими для написання, налагодження і супроводу.

Visual Studio 2012 вводить спрощений підхід асинхронного програмування, який використовує асинхронні можливості .NET Framework 4.5 і середовища виконання Windows. Компілятор виконує складну роботу, яку зазвичай робив розробник, при цьому ваш додаток зберігає логічну структуру, яка схожа з синхронним кодом. В результаті ви отримуєте всі переваги асинхронного програмування з малими зусиллями.

Асинхронність необхідна для дій, які потенційно є блокуючими, наприклад, коли додаток отримує доступ до інтернету. Доступ до веб-ресурсів може бути повільним або здійснюватися з затримками. Якщо така активність заблокована всередині синхронного процесу, все додаток повинен чекати відповіді. У асинхронному процесі, додаток може продовжити виконувати іншу роботу, яка не залежить від веб-ресурсу, поки потенційно блокуюча завдання завершиться.

Основні області, де асинхронне програмування зменшує час відгуку. Перераховані API з .NET Framework 4.5 і середовища виконання Windows містять методи, що підтримують асинхронне програмування:

Доступ до Інтернету HttpClient, SyndicationClient
Робота з файлами StorageFile, StreamWriter, StreamReader, XmlReader
Робота з зображеннями MediaCapture, BitmapEncoder, BitmapDecoder
Програмування з використанням WCF Синхронні і асинхронні операції

Завдання для самоконтролю

1. Створіть програму, яка буде виводити на екран ланцюжки падаючих символів. Перший символ ланцюжка – білий, другий – світло зелений, решта символів темно-зелені. Дійшовши до кінця, ланцюжок зникає і з низу формується новий ланцюжок. Див знімок екрану, як приклад:

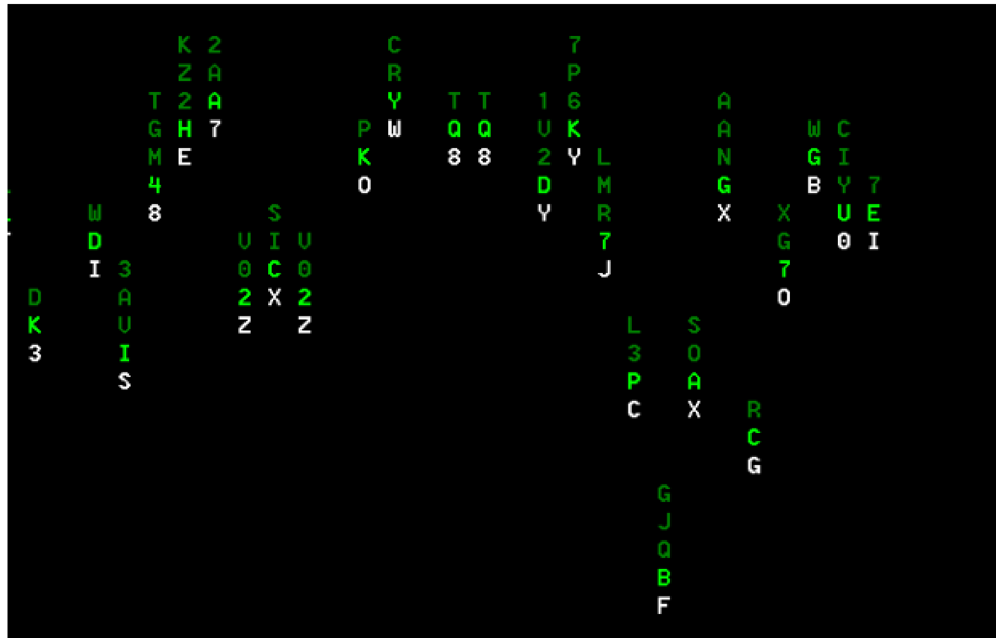

Рисунок 19.3. Приклад виконаного завдання

Розділ 20. Серіалізація

20.1. Поняття атрибутів.

Атрибути - це механізм розширення для додавання інформації користувача в елементи коду (збірки, типи, члени, значення, що повертаються і параметри).

Яскравим прикладом використання атрибутів є сериалізація - процес перетворення довільних об'єктів в конкретний формат, і навпаки.

Основне призначення серіалізації - зберегти стан об'єкта для того, щоб мати можливість відтворити його.

У цьому сценарії атрибут поля може задати відповідність між поданням поля в мові C# і поданням поля в зазначеному форматі.

Компілятор .NET при компіляції компановочного блоку генерує метадані для всіх типів, що використовуються в збірці. Крім цих стандартних метаданих платформа .NET дає можливість програмісту вбудувати в компановочний блок додаткові метадані, використовуючи атрибути.

Самі атрибути в платформі .NET є типами, що розширюють абстрактний базовий клас System.Attribute.

20.2. Що таке серіалізація?

Серіалізація - це збереження стану об'єкта в байтовий потік, з метою його (об'єкта) подальшого відновлення. Збережена послідовність байт електронних даних містить всю необхідну інформацію для відновлення об'єкта.

У багатьох випадках збереження даних програми за допомогою сервісу серіалізації виявляється набагато зручнішим, ніж пряме використання засобів читання \ запису, що пропонуються в рамках простору імен System.IO

Використання серіалізації також грає ключову роль при копіюванні об'єкта на віддалений комп'ютер.

20.3. Відносини між об'єктами

Процеси, що відбуваються при серіалізації є досить складними.

При серіалізації буде задіяно весь ланцюжок успадкування.

Безліч взаємопов'язаних об'єктів можна представити у вигляді об'єктного графа.

Сервіс серіалізації дозволяє зберігати і об'єктний граф.

Крім двійкового (бінарного) формату, використовуючи BinaryFormatter, об'єкт можна зберігати в форматі SOAP (Simple Object Access Protocol - простий протокол доступу до об'єктів) або в форматі XML. Ці формати корисні, коли збережені об'єкти повинні бути перенесені на інші комп'ютери з іншими операційними системами, мовами та архітектурою.

Об'єктний граф можна зберігати в будь-якому похідному від System.IO.Stream типі.

20.4. Графи відносин об'єктів.

При серіалізації об'єкту середовище CLR враховує стан усіх пов'язаних об'єктів.

Множина пов'язаних об'єктів представляється об'єктним графом.

Кожному об'єкту в об'єктному графі призначається середовищем CLR унікальне числове значення. Ці числові значення довільні.

Після призначення всім об'єктам числових значень можна записати безліч залежностей кожного об'єкта.

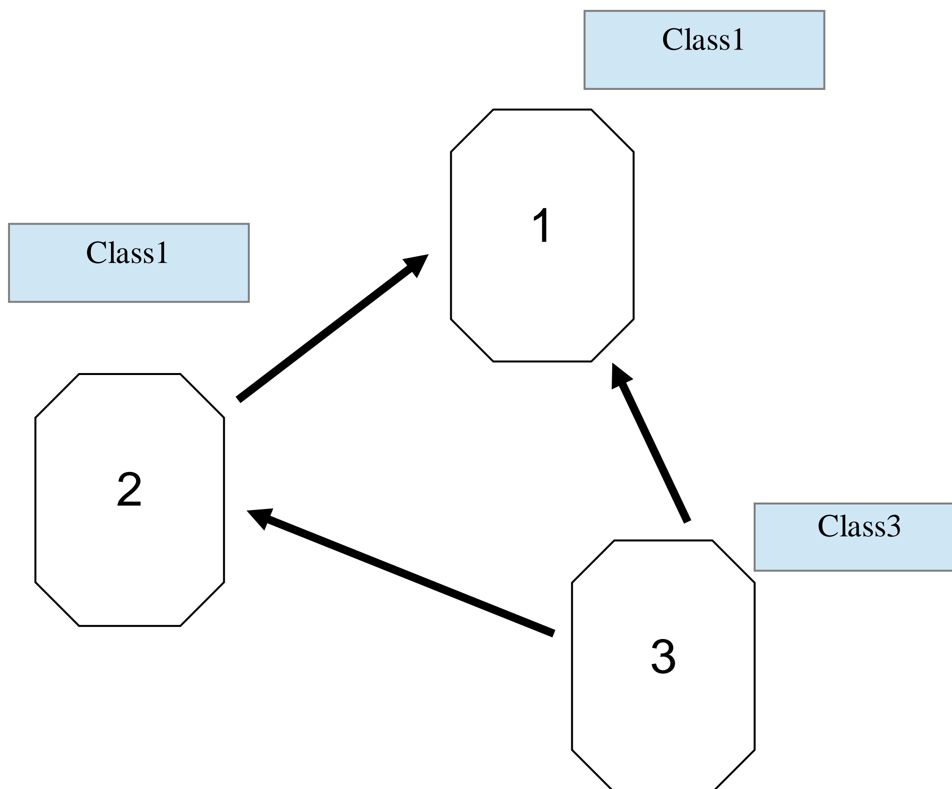

Рисунок 20.1. Середовище CLR

Взаємозв'язки, зазначені в рис. 20.1., представляються середовищем CLR математичною формулою:

[Class1 2, ref 1], [Class1 1], [Class3 3, ref 2, ref 1] ,

де Class1 1, Class1 2 – об'єкти 1 та 2 типу Class1, а Class3 3 – об'єкт 3 типу Class3.

20.5. Атрибути для серіалізації [Serializable] і [NonSerialized].

Щоб зробити об'єкт доступним сервісу серіалізації, досить помітити кожен пов'язаний клас атрибутом [Serializable].

Якщо необхідно, щоб деякі члени даного класу не брали участь в процесі серіалізації, то потрібно позначити відповідні поля атрибутом [NonSerialized].

Атрибут [Serializable] не успадковується. Таким чином, якщо ви отримуєте клас із типу, позначеного атрибутом [Serializable], дочірній клас теж слід позначити атрибутом [Serializable].

При спробі виконати серіалізацію об'єкта, що не позначено атрибутом [Serializable] - генерується виняток SerializationException.

20.6. Формати серіалізації.

Двійкове форматування. Клас BinaryFormatter.

Платформа .NET надає можливість здійснювати серіалізацію в трьох різних форматах:

серіалізація в двійковий формат. Здійснюється об'єктами класу BinaryFormatter, який знаходиться в просторі імен System.Runtime.Serialization.Formatters.Binary;

серіалізація в формат XML. Здійснюється об'єктами класу XmlSerializer, який знаходиться в просторі імен System.Xml.Serialization;

серіалізація в формат SOAP. Здійснюється об'єктами класу SoapFormatter, який знаходиться в просторі імен System.Runtime.Serialization.Formatters.Soap.

BinaryFormatter зберігає тип абсолютно точно.

SoapFormatter і XmlSerializer не зберігають тип точно - вони не записують абсолютні імена типів і компановочних блоків. Ці формати серіалізації призначені для збереження стану об'єктів так, щоб вони могли використовуватися в будь-якій операційній системі для будь-якого каркаса застосувань (.NET, Java і т.і.) в будь-якій мові програмування.

У класі BinaryFormatter визначено два методи, за допомогою яких і виконується серіалізація / десеріалізація об'єктних графів в двійковий потік:

public void Serialize (Stream serializationStream, Object graph);

Зберігає об'єктний граф graph в зазначеному потоці serializationStream у вигляді послідовності байтів.

public Object Deserialize (Stream serializationStream);

Перетворює збережену послідовність байтів з потоку serialization Stream в об'єктний граф.

BinaryFormatter серіалізує як відкриті, так і закриті поля і властивості об'єктів.

BinaryFormatter забезпечує сумісність між версіями .NET Framework.

Серіалізація і десеріалізація повинні виконуватися тільки .NET додатками.

SOAP форматування. Клас SoapFormatter.

SoapFormatter зберігає об'єктний граф в повідомленні SOAP (Simple Object Access Protocol), що робить цей варіант форматування прекрасним вибором при передачі об'єктів засобами віддаленої взаємодії за протоколом HTTP.

SOAP протокол заснований на XML.

SOAP визначає стандартний процес, за допомогою якого можна викликати методи незалежним від платформи і ОС способом для WEB-сервісів XML.

Щоб виконати серіалізацію \ десеріалізацію об'єкта в форматі SOAP, потрібно виконати ті ж дії, що і при двійковій серіалізації.

SoapFormatter не підтримує сумісність по серіалізації між версіями .NET Framework.

XML Серіалізація.

Підходить для серіалізації відкритих типів і членів типів.

Для виконання XML серіалізації не обов'язково використовувати атрибут Serializable.

JSON Серіалізація.

Механізм серіалізації перетворює дані JSON в екземпляри типів .NET Framework і назад в дані JSON.

Формат даних JSON буває особливо корисний при написанні веб-додатків AJAX (Asynchronous JavaScript and XML).

Створення формату серіалізації користувача. Інтерфейс Iserializable.

Метод інтерфейсу ISerializable

void GetObjectData(SerializationInfo info, StreamingContext context) – розміщує дані, необхідні для коректної серіалізації об'єкта в об'єкт класу SerializationInfo.

Завдання для самоконтролю

1. Створити застосування WindowsForms, що складається з двох чекбоксів і текстбоксу. За допомогою серіалізації потрібно зберігати параметри форми такі як розмір вікна, значення чекбоксів і текстбоксу, щоб після закриття форми і відкриття заново всі параметри відновлювались.

Розділ 21. Перевантаження операторів

21.1. Перевантаження операторів.

Припустимо, що визначено клас, що представляє математичну матрицю.

Матриці можна складати або перемножать, подібно до звичайних чисел.

Тому виникає бажання написати код подібно такому:

```
Matrix a, b, c;
// припустимо, що a, b і c вже ініціалізовані
Matrix d = c * (a + b);
```

У багатьох ситуаціях можливість перевантаження операторів дозволяє записувати більш читабельний і інтуїтивно зрозумілий код.

Наприклад:

при математичному чи фізичному моделюванні, при використанні класів, що представляють такі об'єкти як координати, вектори, матриці, функції і т.і.;

при обчисленні позицій на екрані в графічних програмах;

при обробці і аналізу тексту за допомогою класів, які представляють речення, висловлювання. При комбінації речень.

Перевантаження операторів (перевантаження операцій) дозволяє вказати, як стандартні оператори будуть використовуватися з об'єктами класу.

Вимоги до перевантаження операторів:

перевантаження операторів повинно виконуватися відкритими статичними методами класу;

у методу - оператора тип значення або одного з параметрів повинен збігатися з типом, в якому виконується перевантаження оператора;

параметри методу оператора не повинні включати модифікатор out і ref.

Перевантаження операторів має обмеження:

перевантаження не може змінити пріоритет операторів;

при перевантаженні неможливо змінити число операндів, з якими працює оператор;

не всі оператори можна перевантажувати.

Перевантаження операторів можна використовувати як в класах, так і в структурах, див таблицю 21.1.

Синтаксис перевантаження:

```
public static <тип результату> operator <символ операції> (параметри).
```

Таблиця 21.1.

Можливість перевантаження операцій

<table>
<tr><th>Операція C#</th><th>Можливість перевантаження</th></tr>
<tr><td>+, -, !, ++, --, true, false</td><td>Цей набір унарних операцій може бути перевантажений</td></tr>
<tr><td>+, -, *, /, %, &, |, ^, <<, >></td><td>Ці бінарні операції можуть бути перевантажені</td></tr>
<tr><td>==, !=, <, >, <=, >=</td><td>Ці операції порівняння можуть бути перевантажені. C# вимагає сумісного завантаження "подібних" операцій (тобто < і >, <= і > =, == і !=)</td></tr>
<tr><td>[]</td><td>Операція [] не може бути перевантажена. Однак, аналогічну функціональність пропонують індексатори</td></tr>
<tr><td>()</td><td>Операція () не може бути перевантажена. Однак ту ж функціональність надають спеціальні методи перетворення</td></tr>
<tr><td>+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=</td><td>Скорочені операції присвоювання не можуть перевантажуватися; проте ви отримуєте їх автоматично, перевантажуючи відповідну бінарну операцію</td></tr>
</table>

21.2. Перевантаження унарних операторів

Унарні оператори отримують єдиний операнд. Цей операнд повинен мати тип класу, в якому виконується перевантаження оператора.

21.3. Перевантаження бінарних операторів.

Доповнимо розроблений раніше клас CPoint новими можливостями:

виконання перетворення зсуву - складання точки з вектором, що задає зміщення;

виконання перетворення масштабування - множення точки на коефіцієнт, що задають масштаб;

обчислення відстані між двома точками - виконується шляхом вирахування відповідних координат. Виходить вектор.

21.4. Перевантаження операторів порівнянь.

При перевантаженні методу Equals() слід також перевантажувати метод GetHashCode() інакше виникає попередження компілятора.

Крістіан Нейгел, Білл Івьен, Джей Глінн, Карлі Уотсон, Морган Скіннер в книзі "З # 5.0 і платформа .NET 4.5 (для професіоналів)" стверджують: "Не

піддавайтеся спокусі перевантажувати операцію порівняння шляхом виклику методу Equals(), успадкованого від System.Object "

Пропонується інший варіант.

При компіляції коду може згенеруватися попередження компілятора (не був перевизначений метод Equals()). Але в даному випадку це не має значення, і попередження можна проігнорувати.

21.5. Перевантаження операторів true і false.

При перевантаженні операторів true і false розробник задає критерій істинності для свого типу даних.

Перевантаження виконується за такими правилами:

оператор true повинен повертати значення true, якщо стан об'єкта істинно і false в іншому випадку;

оператор false повинен повертати значення true, якщо стан об'єкта false і false в іншому випадку;

оператори true і false треба перевантажувати в парі.

При цьому можлива ситуація, коли стан не є ні true ні false, тобто обидва оператори можуть повернути результат false.

21.6. Перевантаження логічних операторів.

Як вам повинно бути вже відомо, в C # передбачені наступні логічні оператори: &, |, !, && і ||.

З них перевантаження, безумовно, підлягають тільки оператори &, | і !. Проте, дотримуючись певних правил, можна отримати також користь з укорочених логічних операторів && і ||.

Якщо не користуватися укороченими логічними операторами, то перевантаження операторів & і | можна виконувати абсолютно природним шляхом, отримуючи в кожному випадку результат типу bool. Аналогічний результат, як правило, дає і перевантажується оператор !.

21.7. Перевантаження операторів перетворення.

У власних типах можна визначити оператори, які будуть використовуватися для виконання приведення.

Приведення може бути двох типів:

з довільного типу у власний тип;

з власного типу в довільний тип.

Для довідкових і значущих типів приведення виконується однаково.

Приведення може виконуватися явним і неявним чином. Явне приведення типів потрібно, якщо можлива втрата даних в результаті приведення.

Синтаксис:

public static {implicit | explicit} operator <цільовий тип> (вихідний тип).

Завдання для самоконтролю

1. Створіть клас, в якому буде інформація про дату(день, місяць, рік). З допомогою механізму перегрузки операторів, визначіть операцію різності

двох дат(результат в вигляді кількості днів між датами), а також операцію збільшення дати на визначену кількість днів.

2. Створити клас комплексних чисел. За допомогою перевантаження операторів визначити арифметичні операції над комплексними числами.
3. Створити клас звичайних дробових чисел. За допомогою перевантаження операторів визначити арифметичні операції над дробовими числами. Також реалізувати можливість скорочення дробових чисел після арифметичних операцій.

Розділ 22. Збірка сміття

22.1. Життєвий цикл об'єктів

Життєвий цикл будь-якого об'єкта можна представити таким чином:

виділення пам'яті;

ініціалізація виділеної пам'яті (установка об'єкта в початкове значення, виклик конструктора);

використання об'єкта в програмі;

руйнування стану об'єкта;

звільнення пам'яті.

У C# розроблений автоматичний механізм управління пам'яттю - збір сміття.

Програмування в середовищі, що підтримує автоматичну збірку сміття, значно спрощує розробку додатків.

Доручивши системі автоматичного збору сміття знищення об'єктів, ви знімаєте з себе відповідальність за управління пам'яттю і перекладаєте її на середу - CLR.

22.2. Збирач сміття

Спеціальний код, який здійснює збір сміття називається "Збирач сміття" (Garbage Collector).

Збирач сміття відстежує і знищує об'єкти, що знаходяться в керованій пам'яті. Періодично збирач виконує збирання сміття, щоб вивільнити пам'ять, відведену під об'єкти, на які немає дійсних посилань.

Прибирання сміття автоматично запускається в разі, якщо необхідний обсяг пам'яті більше доступного обсягу вільної пам'яті.

Крім того, додаток може примусово запустити збірку сміття за допомогою методу Collect класу System.GC.

22.3. Деструктор і метод Finalize

Збирач сміття вміє руйнувати прості об'єкти (такі як String, Attribute, Exception, Delegate та інші)

Є складні типи, які збирач сміття правильно руйнувати не вміє, і тому необхідно писати спеціальний код для правильного знищення об'єкта

До таких типів належать типи:

які використовують не тільки оперативну пам'ять, а й інші машинні ресурси;

які представляють собою "оболонку" для так званого "некерованого коду" (System.IO.FileStream, System.Threading.Mutex та інші)

Для таких типів CLR підтримує спеціальний механізм - фіналізація об'єктів (finalization).

Для виконання механізму фіналізації об'єктів, об'єкт повинен реалізовувати однойменний метод (Finalize), який буде викликаний складальником сміття безпосередньо перед його знищенням.

Для реалізації даної функціональності (завершувача) використовується спеціальний синтаксис - тильда ~ та ім'я класу.

Перевантажувати фіналізатор і використовувати в ньому параметри не можна.

22.4. Метод Dispose і інтерфейс IDisposable

Метод Finalize() безпосередньо викликати не можна - його викликає Garbage Collector при руйнуванні об'єкта під час збірки сміття.

У деяких випадках виникає необхідність надати можливість явно звільнити зовнішні ресурси, захоплені об'єктом, перед тим як збирач сміття знищить об'єкт.

Найкращої продуктивності можна досягти при явному звільнення ресурсів, коли вони вже не використовуються.

Така поведінка реалізується за допомогою інтерфейсу IDisposable, який визначає наявність методу Dispose() у своїх спадкоємців.

Користувач об'єкта повинен викликати цей метод при завершенні використання об'єкта.

Але, при наявності явного управління за допомогою методу Dispose слід надати і неявне звільнення ресурсів з використанням методу Finalize, як додатковий спосіб запобігання втрати ресурсів.

Код, що викликає метод Dispose, повинен бути захищений блоком try \ finally.

Блок try \ finally в разі зі звільненням ресурсів буде практично однаковий.

Розробники C # для спадкоємців інтерфейсу IDisposable пропонують спрощений сиснтаксис (з використанням оператора using), який вказує компілятору згенерувати ідентичний код.

Microsoft не рекомендує втручатися в роботу збирача сміття, але така можливість є.

Клас, що надає збирач сміття, називається System.GC

Методи цього класу впливають на те, коли виконується прибирання сміття та коли вивільняються ресурси, виділені об'єктом.

Властивості цього класу надають відомості про загальний обсяг доступної пам'яті системи і про те до якого покоління належить пам'ять, що виділена об'єкту.

22.5. Поняття поколінь при збиранні сміття.

Проблема: Збирач сміття, після очищення пам'яті, щоб не допустити дефрагментації, він "стискає" ("зрушує") всі об'єкти так, щоб вони розташовувалися послідовно. При цьому змінюється значення посилань на об'єкти, що залишилися. Тому збирачеві необхідно у всьому додатку оновлювати посилання.

Під час цього процесу всі потоки додатку очікують оновлення посилань і додатки як би "зависають".

Тому збір сміття повинен відбуватися якомога швидше.

Керована купа розділена на 3 покоління: 0, 1, 2.

Розмір пам'яті збільшується з ростом номера покоління.

Новостворені об'єкти потрапляють в перше покоління.

Збір автоматично викликається після закінчення пам'яті в одному з поколінь.

Збір сміття відбувається не тільки в цьому поколінні, а й у всіх молодших поколінь.

Об'єкти, які пережили збір сміття переходять в наступне покоління.

Чим вище номер покоління, в якому знаходиться об'єкт, тим менше ймовірність його знищення збирачем сміття, навіть якщо на нього вже давно немає посилань в додатку.

Завдання для самоконтролю

1. Створити клас з деструктором. В деструкторі виводити на екран повідомлення про знищення об’єкту збірником сміття. А в контсрукторі повідомлення про створення об’єкту. В циклі створити величезну кількість об’єктів даного класу і проаналізувати роботу збірника сміття.

Список використаної літератури.

1. A. Hejlsberg, M. Torgersen, S.Wiltamuth, P. Gold, "The C# Programming Language (Covering C# 4.0) (4th Edition)", Addison-Wesley Professional; 4 edition - 864 p., 2010.
2. J. Albahari, B. Albahari, "C# 7.0 in a Nutshell", O'Reilly Media— 1090 p., 2017.
3. H. Schildt, "C# 4.0 The Complete Reference McGraw Hill Education; 1st edition" – 984 p., 2017.
4. J. Sharp, "Microsoft Visual C# Step by Step", Pearson Education – 878p., 2018.
5. A. Troelsen, "Pro C# 7 With .NET and .NET Core" - Apress, — 1372 p., 2017.
6. C# Reference, сайт розробників MSDN. [Електронний ресурс] – Режим доступу: https://docs.microsoft.com/uk-ua/dotnet/csharp/language-reference/index