

КИЇВСЬКИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

Основи програмування. Мова C#

**Методичний посібник
для студентів радіофізичного факультету**

Київ – 2009

Рецензент: Ю.В. Бойко

Основи програмування. Мова С#. Методичний посібник для студентів радіофізичного факультету університету. / В.О. Грязнова, С.В.Єфіменко, К.Е.Юштин – К.: КНУ, 2009 – ??? с.

Посібник містить викладення основ програмування від історії програмування та представлення інформації у комп'ютері до огляду можливостей сучасної програмної технології Microsoft .NET Framework. Основні принципи об'єктно-орієнтованого програмування проілюстровані можливостями сучасної мови програмування С#. Кожний розділ містить велику кількість прикладів, готових до виконання.

Затверджено
Радою радіофізичного факультету,
Протокол № 8 від 16 лютого 2009 року

ЗМІСТ

Мови програмування. Представлення даних.	5
1. Вступ. Про обчислювальну техніку.	5
2. Історія мов програмування.	6
3. Поняття про платформу .NET	8
4. Створення мови програмування C#.	9
5. Представлення даних. Необхідність типізації. Двійкова арифметика.	10
6. Основні поняття програмування.	12
Поняття про інтегроване середовище розробки VisualStudio (на прикладі Visual Studio 2005). Структура C#-програми.	16
1. Основні можливості інтегрованого середовища розробки VisualStudio .NET.	16
2. Структура програми мовою C#.	18
Основні елементи мови C#.	22
1. Основні вбудовані типи мови C#	22
2. Визначення та ініціалізація змінних, область їх видимості.	24
3. Приведення типів.	26
4. Літерали (константи) мови C#.	28
5. Операції мови C#.	30
Основні інструкції керування мови C# – розгалуження та цикли.	37
1. Розгалуження у мові C#	37
2. Цикли у мові C#	43
3. Керування виходом із циклів C#	49
Масиви в мові C#.	50
1. Визначення та ініціалізація масиву.	50
2. Цикл foreach	55
3. Багатовимірні масиви.	56
4. Використання деяких методів класу System.Array	58
5. Масиви масивів. Непрямокутні масиви.	59
Структуровані типи даних (колекції) в мові C#.	60
1. Основні структури даних та їх призначення	60
2. Використання списку ArrayList та узагальненого списку List	62
3. Використання асоційованого списку Hashtable та узагальненого словника Dictionary	67
Класи в мові C#.	69
1. Визначення класу.	70
2. Методи класу.	72
3. Методи з параметрами.	75
4. Конструктор класу.	77
Методи в мові C#.	80
1. Передача об'єктів методам.	80

2. Використання модифікаторів для параметрів методів.	83
3. Методи, що повертають об'єкти.	86
Перевантаження методів в мові C#.....	89
1. Перевантаження методів.....	89
2. Перевантаження конструкторів.....	92
3. Використання ключового слова this.	94
4. Деструктор класу.	97
5. Метод Main ().....	99
Статичні члени класу.....	100
1. Статичні дані-члени класу.....	100
2. Статичні методи-члени класу.	104
3. Статичний конструктор класу.	107
4. Статичні класи, локалізація та глобалізація	110
Властивості та індексатори.	112
1. Властивості.	112
2. Індексатори.	117
Спадкування в мові C#.....	124
1. Поняття про спадкування та ієрархію класів.	124
2. Спадкування та правила доступу до членів класів.....	127
3. Конструктори базового та похідних класів.	129
4. Посилання на екземпляри базового та похідних класів.	133
5. Поняття про поліморфізм.	136
6. Віртуальні функції – більш детальний погляд.....	140
7. Абстрактні методи та класи.	146
Перевантаження операцій в мові C#.....	149
1. Загальні відомості.....	149
2. Перевантаження бінарних арифметичних операцій.	150
3. Перевантаження унарних операцій.....	153
4. Перевантаження операцій відношення.	154
5. Перевантаження логічних операцій.....	156
6. Підсумкові зауваження.	157
Структури та переліки в мові C#.....	158
1. Структури.....	158
2. Переліки.....	165
Делегати, події та обробники подій	168
1. Делегати (delegate).....	169
2. Події та їх обробники.	173
Атрибути та їх використання	180

Мови програмування. Представлення даних.

1. Вступ. Про обчислювальну техніку.

Історія обчислювальної техніки сягає давніх часів. Можна назвати імена відомих вчених, фізиків і математиків – Блезе Паскаля, Готфріда Лейбніца, які сконструювали перші обчислювальні пристрої. Відоме ім'я Чарльза Беббіджа, який у 1835 році винайшов та описав свою аналітичну машину (Analytical Engine). Це був проект комп'ютера загального призначення із перфокартами в ролі носіїв вхідних даних та парового двигуна в ролі джерела енергії. Архітектура Analytical Engine практично відповідала сучасним комп'ютерам. З іменем Беббіджа часто пов'язують ім'я леді Ади Лавлейс – доньки лорда Байрона, яка переклала та доповнила коментарями книгу про аналітичну машину Беббіджа. Стверджують подекуди, що вона була першим програмістом, проте це скоріше красиве історичне перебільшення. Тим не менше, назву «Ада» на її честь носить одна із мов програмування.

Зрозумілим чином роботи по створенню потужних обчислювальних пристроїв пожвавились за років другої світової війни. Великих здобутків досягла Велика Британія. Група англійських вчених (серед них був і Алан Тьюрінг – автор «машини Тьюрінга») намагалась дешифрувати коди німецької шифрувальної машини «Енігма», яку використовували німецькі військово-морські сили для передачі секретних повідомлень. Англійцям вдалось виключити ряд варіантів шляхом логічних виводів, реалізованих обчислюваннями за допомогою електрики. Для зламу кодів «Енігми» в обстановці цілковитої секретності крім того було створено машину “Colossus” («Колосс»), специфікацію якої розробив професор Макс Ньюман з колегами. Ці роботи успішно проводились у 1941-1943 рр. “Colossus” став першою цілком електронною машиною, яка містила велику кількість електровакуумних ламп, введення інформації відбувалось з перфострічки. Хвилюючись, щоб цими досягненнями не скористались союзники, Уїнстон Черчилль особисто підписав у 1945 році наказ про руйнацію цієї машини на частини, що не перебільшували за розміром долонь, а інформація про неї трималась у секреті до 70-х років, тому “Colossus” майже не згадується серед перших обчислювальних машин.

Першим електронним комп'ютером загальновизнано вважають американський «Еніак», який довів можливість застосування електроніки для масштабних обчислень, сягнувши високої на той час швидкості обчислень – близько 5000 операцій в секунду. Роботи над цим проектом почались у 1943-1945 роках. Багато сучасників були налаштовані досить скептично і не без підстав – вони вважали, що

тисячі тендітних електроламп будуть згорати настільки часто, що машина буде майже безперервно перебувати у ремонтах. Тим не менше «Еніаку» вдавалось працювати безперебійно до кількох годин до чергового збою із-за лампи, що вийшла з ладу.

Аналізуючи результати цього проекту, Джон фон Нейман у своєму широко відомому звіті розробив ідею проекту комп'ютера, в якому і дані, і програма зберігаються в єдиній універсальній пам'яті. Принципи побудови такої машини одержали назву «архітектури фон Неймана» і лягли в основу розробки перших гнучких універсальних цифрових комп'ютерів.

Слід віддати належне і досягненням вітчизняних вчених. Перший універсальний комп'ютер у Європі був створений командою Сергія Олексійовича Лебедева із Київського інституту електротехніки. Його Мала електронна лічильна машина (рос. - МЭСМ) запрацювала у 1950 році, виконуючи до 3000 операцій у секунду. Лебедев пізніше працював у Москві і помер у 1973 році, вже після його смерті був реалізований започаткований ним проект суперкомп'ютера «Ельбрус».

Наступне покоління комп'ютерів базувалось на транзисторах, а ще наступне – на інтегральних мікросхемах. Розвиток комп'ютерної техніки відбувається за експонентою, буквально щороку змінюючи уявлення людей про можливості обчислювальної техніки.

2. Історія мов програмування.

Нині відомо більше 8500 мов програмування. Що не дає спокою програмістам, примушуючи їх шукати і створювати «ідеальну» мову програмування, залишаючи за собою за влучним виразом Кріса Касперського «ціле кладовище мов, парадигм, вмерлих концепцій, ідей, що випередили свій час»? Навіщо взагалі потрібні різні мови програмування?

Справа в тому, що комп'ютер здатен розуміти лише команди, записані у двійкових кодах. Тобто кожен символ команди комп'ютера може набувати лише двох значень: 0 або 1. Історичний аспект цього питання полягає в тому, що перші обчислювальні машини сприймали лише наявність або відсутність сигналу у певній комірці. Перші програмісти безпосередньо задавали команди комп'ютерам у вигляді низки двійкових кодів. Вхідна інформація у вигляді таких команд наносилась на так звані «перфострічки» або «перфокарти». На них у певних місцях пробивались дірочки, наявність або відсутність яких і означала відповідно 0 або 1. Можна уявити собі, на який кошмар перетворювався пошук помилок у перших програмах. Тому абсолютно слушною видалась ідея про те, що програма має бути написана якомога більш зрозумілим людині чином, тобто двійкові команди

необхідно замінити словами або скороченнями слів і фраз природної мови. Це викликало необхідність у виникненні «посередника» – спеціальної програми, яка б перекладала такий текст у комп'ютерні команди. Такий посередник одержав назву компілятора або транслятора. За своїм призначенням це дещо різні речі, проте ми не будемо зараз заглиблюватись у деталі.

Перші мови програмування були точно зав'язані на вузько спеціальні задачі тієї чи іншої програмної системи. Наприклад, для наукових цілей використовувалась мова ФОРТРАН, для бухгалтерії – переважно мова КОБОЛ. Проте для системних робіт використовувалася мова програмування низького рівня – Асемблер. Мови програмування високого рівня відрізняв певний рівень абстракції в інтерпретації команд та даних, вони доволі наближені до природної людської мови.

Програмування не раз переживало важкі часи. І кожен раз це призводило до створення нових, більш абстрактних, парадигм програмування та мов, що їх реалізовували. Однією з революційних концепцій програмування було **структурне програмування**, що базувалось на процедурній абстракції. Процедури (або просто підпрограми) дозволяли структурувати текст програми, допускали повторне використання окремих частин програми (реалізоване у вигляді виклику процедури з будь-якого місця програми, в тому числі і з самої процедури). Тим самим ця концепція наводила певний порядок у програмах (які часто порівнювали із спагеті-кодом через багатократні переходи та заплутаність текстів програм), долаючи «демонів складності». Найбільш поширеними мовами програмування високого рівня в межах цієї концепції були мови ПЛ/1, Pascal, C, Modula-2.

Структурні мови програмування дозволяли писати досить складні програми. Проте фатальним для концепції структурного програмування ставав об'єм програми. В той час як важливою підтримкою складних інженерних проектів виступало програмування, написання реально складної та великої програми в межах цієї концепції ставало неможливою.

Один із нових підходів до програмування одержав назву **об'єктно-орієнтованого програмування** (ООП). Для реалізації цієї концепції були створені нові мови програмування. Одним із найбільш вдалих проектів була мова C++, яка повністю підтримувала весь парк програм, створених для компіляторів з мови C. Саме цей факт і забезпечив мові C++ комерційний успіх. Іншими прикладами об'єктно-орієнтованих мов програмування служать мови Smalltalk, Java та інші.

Концепція об'єктно-орієнтованого програмування базується на трьох основних принципах. По-перше, об'єкт **інкапсулює** (приховує) у собі дані та методи, які ними оперують, залишаючи назовні лише можливість послати або одержати повідомлення – дозволену інформацію про себе. По-друге, існує можливість створювати нові об'єкти з розширеним списком можливостей шляхом **спадкування** від існуючих об'єктів. І нарешті, різні об'єкти можуть мати методи з однаковим інтерфейсом, проте різним функціонуванням. Ця властивість реалізує підхід, що одержав назву **поліморфізму**. Його ідея : один інтерфейс – багато методів.

Сучасні мови програмування, C++ зокрема, доросли таким чином до так званого «метапрограмування», або програмування з допомогою шаблонів – можна створювати програми, які створюють інші програми як результат своєї роботи. Таким чином, мова програмування одержує певну самостійність, яка з одного боку значно спрощує розв'язання багатьох проблем, проте й вносить певний елемент некерованості, незалежності від авторського задуму. Як страшну аналогію можна навести ідеї сучасних авторів-фантастів про війни роботів із людством. Крім того, мова диктує стиль мислення програміста. І доволі часто трапляється, що вся міць і потужність мови використовуються там, де в цьому зовсім мало потреби.

3. 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. Ця технологія (версії .NET 1, 1.1, 2, 3 та 3.5) працює під Windows, а для переносу на інші операційні системи існує Open Source проект Mono, який на даному етапі має задекларовану повну сумісність на рівні .NET версії 2.0 (<http://www.mono-project.com/>).

4. 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 напряду 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.

5. 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 А до F. Їх відповідність між собою показано у наступній таблиці.

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

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

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:

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

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 помножити 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!) $0,4567_{(10)} \approx 0,01110100_{(2)} \approx 0,74_{(16)}$.

6. 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 говорилося 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, символічні, текстові, тощо), за що вони ще називаються змінними **value-muny**, або просто змінними-значеннями, існують змінні **reference-muny**, 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 звуться **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 адреса 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.

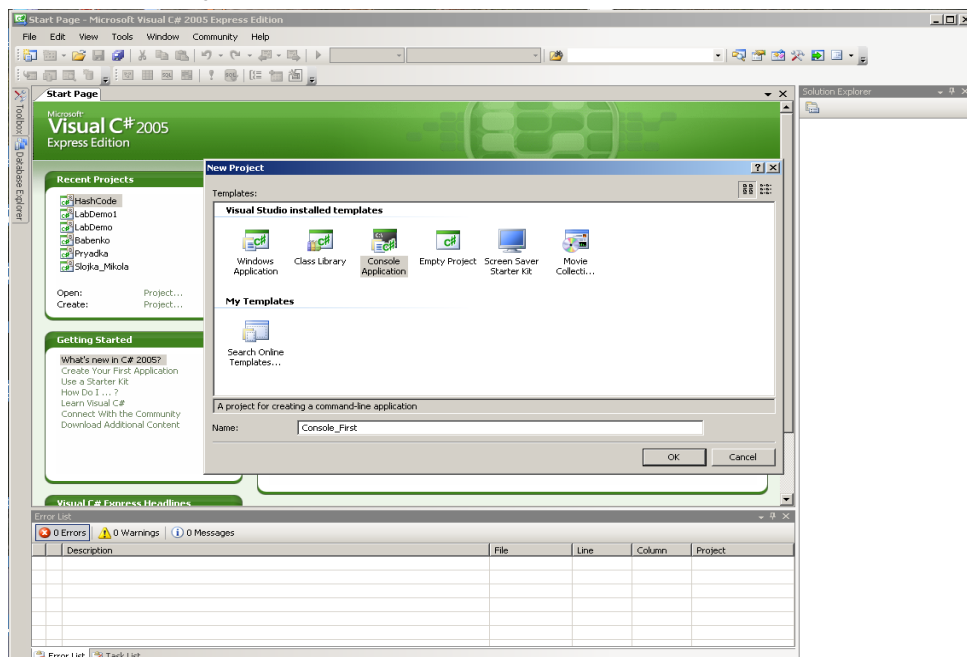
Поняття про інтегроване середовище розробки VisualStudio (на прикладі Visual Studio 2005). Структура C#-програми.

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.

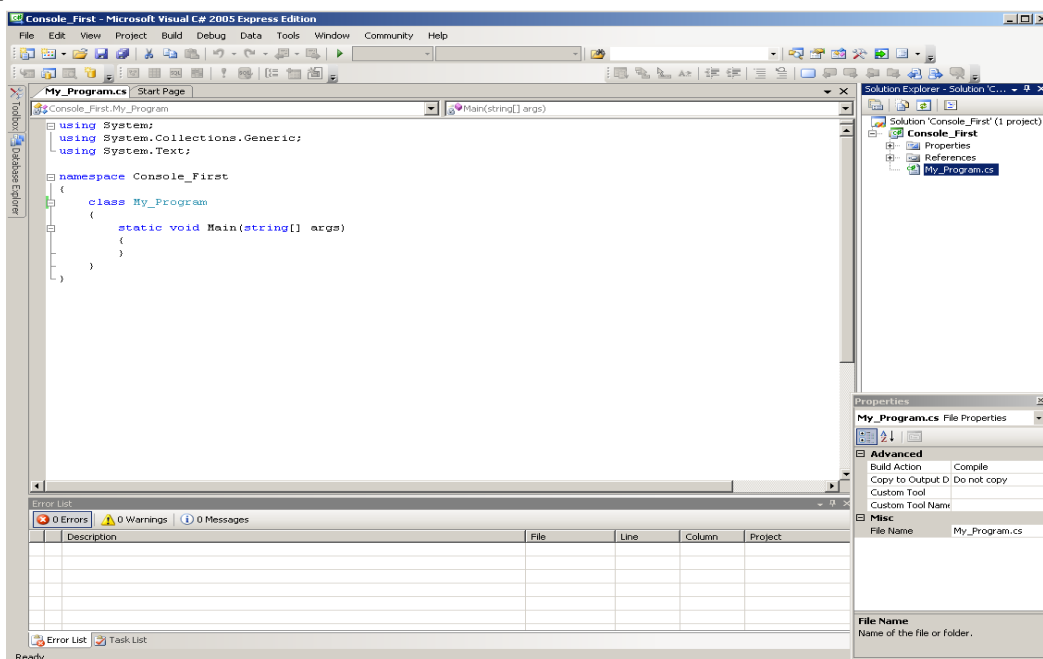
Для створення проекту використаємо послідовність команд меню : File -> New Project. Далі у діалоговому вікні необхідно вибрати один із запропонованих типів проектів. Зупинимо свій вибір на консольному застосуванні – Console Application. Маємо можливість обрати ім'я для свого застосування, наприклад,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



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.



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. 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 додалися обчислення та виведення значень двох виразів. В першому визначається $\sqrt{i^2 + f^2}$, а в другому – $\text{tg}(\sqrt{i^2 + f^2})$. 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 (параметр **x** функцій має тип **double**).

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

Основні елементи мови C#.

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.

	Назва типу	Назва системного типу	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	8	-128 +127

	Назва типу	Назва системного типу	Опис типу	Розмір у бітах	Діапазон значень
			цілий		
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 +18 448 744 073 709 551 615
10	float	Single	дійсний	32	1,401298E-45 3,402823E+38
11	double	Double	дійсний подвоєної точності	64	E-324 E+308
12	decimal	Decimal	числовий для фінансових розрахунків	96	29 значущих розрядів
13	char	Char	символьний	16	
14	string	String	Набір символів Unicode		

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 = 1.55555555555555555555555555555M;
            // 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 =
1,55555555555555555555555555555
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 сприйняття 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 сходах 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`;

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 правила 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

5. Операції мови C#.

Розглянемо основні групи операцій, завдяки яким в мові C# 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 C# 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 інкременту 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 визначити, 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 нижче у таблиці для різних значень операндів (op1 та op2).

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 i не
зміниться
            // Цей оператор не виконується
            Console.WriteLine(
                "Скорочена форма &&: тут i не зміниться"
            );

        Console.WriteLine("i = " + i.ToString());
        i = 10;                                // Відновлюємо i
        if (!(k > 10) & (++i < 100))           // Тут i
зміниться
            // Цей оператор не виконується
            Console.WriteLine("Звичайна форма &: тут i
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 заповнюються одиницями. Це обов'язково слід враховувати. У наведеній нижче таблиці вказані результати бітових операцій для різних значень операндів (op1 та op2).

op1	op2	op1	op1	op1	~op1
-----	-----	-----	-----	-----	------

		& op2	 op2	^ op2	
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
            Console.WriteLine("x = {0} y = {1} x & y = {2}", x, y, z);
            z = x | y; // Логічне додавання
            Console.WriteLine("x = {0} y = {1} x | y = {2}", x, y, z);
            z = x ^ y; // Логічне виключне
            Console.WriteLine("x = {0} y = {1} x ^ y = {2}", x, y, z);
            z = x << 1; // Зсув на 1 позицію
            Console.WriteLine("x = {0} x << 1 = {1}", x, z);
            z = x >> 1; // Зсув на 1 позицію
            Console.WriteLine("x = {0} x >> 1 = {1}", x, z);
            z = ~y; // Логічне заперечення
            Console.WriteLine("y = {0} ~y = {1}", y, z);
            z = y | 0X7; // Встановлюємо одиниці в останні 3
            Console.WriteLine("y = {0} y|0X7 = {1}", y, z);
            z = x & ~0X7; // Встановлюємо нулі в останні 3
            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.

Основні інструкції керування мови C# – розгалуження та цикли.

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` приходить 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 // Hi!
        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;
            }
        }
    }
}
```

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.

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);

```

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);
        }
    }
}

```

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 клавіші 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 клавіші
                conKey = Console.ReadKey(true);
                } // Порівнюємо його з ESC
            while (conKey.Key != ConsoleKey.Escape);
        }
    }
}
```

Нижче – інший варіант реалізації тієї ж ідеї, програма 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; j <= 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**.

3. Керування виходом із циклів C#

Іноколи 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;
```

Масиви в мові C#.

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 елемента 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 програмування, 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 елементів 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 С# 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.

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
        // максимальний 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);
    }
}
}

```

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.

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]);
    }
}
}

```

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]`.

Структуровані типи даних (колекції) в мові C#

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 елемента 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'явилися в .NET версії 1.1) та коли дані є типізованими (узагальнені колекції (Generic), що з'явилися їм на зміну в версії .NET 2.0).

Нижче будуть розглянуті 4 типи структур даних:

1. Набір даних із цілими ключами
 - a. **ArrayList** – масив-список, елементи такої структури даних належать до типу **object**.
 - b. **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
 - a. **Hashtable** – дані та ключі даних є довільними (належать до типу **object**)
 - b. **Dictionary <T1,T2>** – дані та ключі є типізованими і належать до типів **T1** та **T2**.

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. Змінимо цю програму для використання типізованої колекції наступним чином:

```

using System;
using System.Collections.Generic;
namespace UseCollectionSpeed
{
    // 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 рефлексору).

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]);
        }
    }
}

```

Зауваження до обох прикладів:

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. Перевірка наявності ключа в обох типах списків відбувається за 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.

Класи в мові С#.

Концепція ООП реалізована в мові С# 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 С#, 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.

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);
        }
    }
}

```

```
}
```

Проаналізуємо 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 функції **Main**, спочатку декларуються, а потім і створюються 2 екземпляри класу **Student** – **s1** та **s2** з допомогою оператора **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 клас **Student** є «пасивним», адже члени даного класу використовуються відкрито «без участі та нагляду» класу.

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();
```

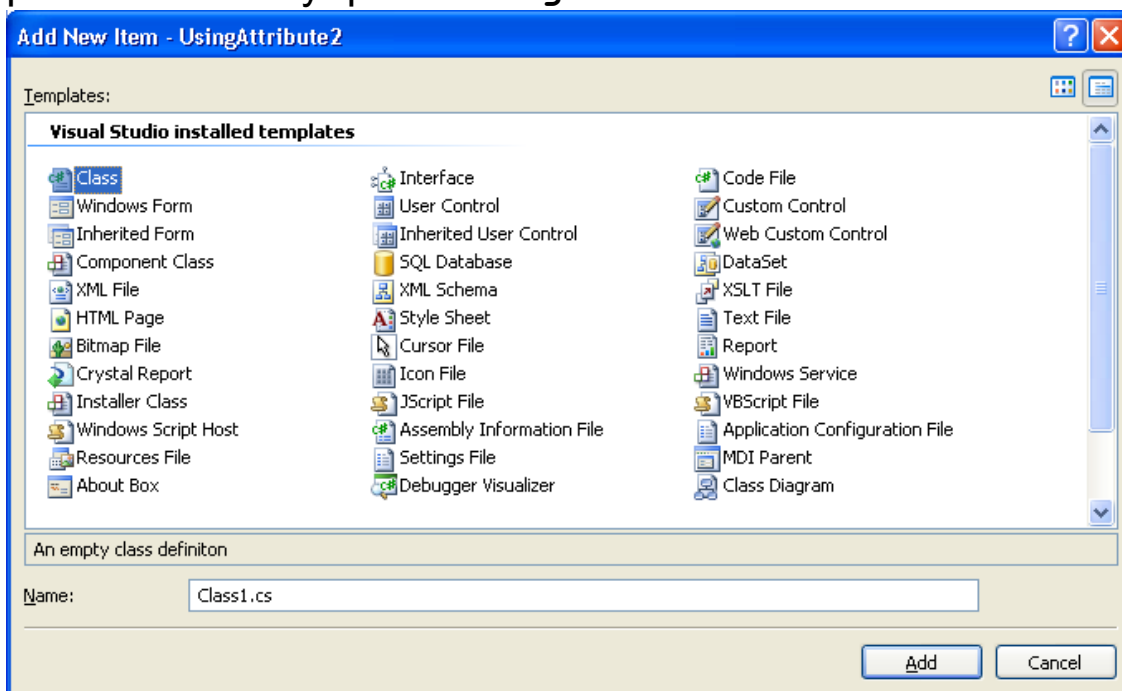
У ньому визначається змінна `p1` `reference`-типу, яка відразу ініціалізується адресою області пам'яті, виділеної під об'єкт (екземпляр) `p1` класу `Polar_Point`. Той самий результат можна було одержати за два кроки подібно до попереднього прикладу:

```
Polar_Point p1;           // 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`.



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 адресу змінної, 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 ініціалізації 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 клас `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 `OurMethod`. Її список включає 3 параметри `x_` типу `double`, `i_` типу `int`, та `c_` типу `char`. У виклику цього методу `cl.OurMethod(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 `OurMethod`. 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.

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 `OurMethod`, 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.

Методи в мові C#.

1. 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 клас **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-reference
        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 фрази, 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 помінялись
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.

2. 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.

Модифікатор	Призначення
Відсутній	Параметр вважається вхідним , тобто передається методу за значенням
<code>out</code>	Параметр вважається вихідним , тобто визначається всередині методу; параметр передається за посиланням
<code>ref</code>	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.

3. 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();
}
}
}

```

Перевантаження методів в мові C#.

1. 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 використанні стандартної функції мови C++ 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:

```
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.

2. 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.

3. 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>` 3

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 викликалися 3 різних конструктори, проте повідомлення про створення кола, насправді надсилав лише один з них, до якого два інших звертались завдяки посиланню `this`.

4. 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 змодельюємо 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 C++, в якій об'єкти створюються командою **new**, а знищуються командою **delete**, яка гарантує в цей момент

виклик деструктора, в мові C# 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).

5. 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.

Статичні члени класу.

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 компіляції
    /// 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
екземляру
            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`.

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 -
            Console.WriteLine(d);    // Статичний метод -
            // Статичний метод - ToString()
            string s = Convert.ToString(129, 2);
            Console.WriteLine(s); //s - зображення двійкового
            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
```

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 спрацював конструктор за замовчуванням. Якщо в такому класі необхідна попередня ініціалізація певних даних, визначте

статичний конструктор. Іншим вирішенням може бути проголошення всього класу статичним.

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")));
    }
}

```

Властивості та індексатори.

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 i, 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 С# 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 компіляції 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            // асесор get
            {
                return field;
            }
            set            // асесор 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 з 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.

Ознаки схожості **властивості** проявляють і з **методами** класу, тому порівняємо і їх.

1. Властивість, як і метод, містить код.
2. Властивість, як і метод, приховує деталі свого функціонування.
3. Властивість, як і метод, може бути статичною.
4. Властивість, на відміну від методу, не може мати тип `void`.
5. Властивість, на відміну від методу, не має дужок із списком параметрів.
6. 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.

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 `x` коренів, який може містити 0, 1 або два корені. У випадку нескінченної 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 x . Індикатор, 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 нескінченної 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]    // індиксатор
    {
        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 елемента.
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. Індиксатор 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 `x` може бути взагалі не визначений при деяких наборах коефіцієнтів `a`, `b`, `c`.

Спадкування в мові C#.

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** 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`.

3. 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.

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.

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 А та похідний від нього клас В, 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 приведення
    b = (B) a;           // вірно, явне приведення
    a = (A) b;           // вірно, явне приведення
}

```

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()`.

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 С#. 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.

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 не поліморфними методами.

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 заміщують. 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 = " +
tring.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

Перевантаження операцій в мові C#.

1. Загальні відомості.

При реалізації класів часто зручно передбачати можливість виконання певних операцій над екземплярами класів. Наприклад, ви створюєте клас, що реалізує простір геометричних векторів. Необхідно мати можливість виконувати над об'єктами прийняті в математиці операції, як то: додавання та віднімання векторів, множення їх на число та між собою, тощо. Звісно для реалізації подібних операцій можна створити методи-члени класу, проте значно зручніше мати змогу застосування цих операцій безпосередньо до об'єктів класу. Для цього в мові передбачене **перевантаження операцій**.

Для перевантаження операцій в класі створюється операторний метод – член класу, позначений службовим словом **operator**. Як відомо, операції бувають унарні та бінарні. Для перевантаження унарної операції використовується наступний синтаксис:

```

public static <ідентифікатор_класу> operator op
(<ідентифікатор_класу> <операнд>)

```

```
{
    // код операції
}
```

Для перевантаження бінарної операції синтаксис операторної функції буде наступний:

```
public static <тип_результату_операції> operator op
(<тип_операнду_1>      <операнд_1>,      <тип_операнду_2>
<операнд_2>)
{
    // код операції
}
```

Тут символами `op` позначений знак операції, що перевантажується. Для бінарної операції тип принаймні одного операнду має бути типом класу, для якого перевантажується операція. Таким чином, операції можливо перевантажувати лише для класів, створюваних користувачем, а не для існуючих класів. Крім того, неможливо змінити пріоритет операцій, а також використати якісь нові знаки крім існуючих для операцій. Існує також ряд операцій, заборонених для перевантаження, але до них ми повернемося пізніше.

2. Перевантаження бінарних арифметичних операцій.

Створимо клас `d2Vector` геометричних векторів на площині, для якого пізніше і перевантажимо ряд операцій. Визначення класу помістимо в окремий файл проекту, наприклад, `d2Vector.cs`.

```
using System;
namespace ReloadOperators {
    class d2Vector
    {
        double x, y;
        public d2Vector(double x_, double y_) // конструктор
        {    x = x_; y = y_;    }
                                // конструктор без параметрів
        public d2Vector() : this(1, 1) { }
                                // конструктор
        public d2Vector(d2Vector v) : this(v.x, v.y) { }
                                // заміщений метод із
    }
}
System.Object
    public override string ToString()
    {    return String.Format("x = {0} y = {1}", x, y);    }
}
```

Тут заміщається метод `ToString()` із класу `System.Object` – це дозволить пізніше зручно виводити координати векторів.

У файл `Program.cs` помістимо метод `Main()`. У ньому створимо вектори `a` та `b` (які саме конструктори спрацьовують при цьому?):

```
using System;
namespace ReloadOperators
{
    class Program
    {
        static void Main()
        {
            d2Vector a = new d2Vector(1, 2);
            Console.WriteLine("a: " + a.ToString());
            d2Vector b = new d2Vector(3, 5);
            Console.WriteLine("b: " + c.ToString());
        }
    }
}
```

Завдяки заміщеному методу `ToString()` координати тепер легко виводити на екран:

`a: x = 1 y = 2`

`b: x = 3 y = 5`

Наша мета – мати змогу написати, наприклад, `a = a + b` або `a = a * b`. Це і буде означати перевантаження операцій для класу.

Додамо у визначення класу `d2Vector` операторні методи для операцій додавання та віднімання двох векторів:

```
                // бінарний плюс
public static d2Vector operator +(d2Vector v1, d2Vector
v2)
{
    d2Vector temp = new d2Vector(v1.x + v2.x, v1.y +
v2.y);
    return temp;
}

                // бінарний мінус
public static d2Vector operator -(d2Vector v1, d2Vector
v2)
{
    d2Vector temp = new d2Vector(v1.x - v2.x, v1.y -
v2.y);
    return temp;
}
```

Код цих методів цілком зрозумілий – необхідно створити новий вектор, координати якого є відповідно сумою або різницею координат двох векторів-параметрів. Це зовсім просто завдяки тому, що у нас у класі є відповідний конструктор. Далі одержаний вектор повертається як результат кожного з цих методів. Помістивши тепер у `Main()` інструкції

```
d2Vector c = new d2Vector();  
c = a + b;  
Console.WriteLine("c = a + b: " + c.ToString());  
c = a - b;  
Console.WriteLine("c = a - b: " + c.ToString());
```

одержимо на екрані наступні результати:

```
c = a + b: x = 4 y = 7
```

```
c = a - b: x = -2 y = -3
```

Далі реалізуємо у класі `d2Vector` можливість множення векторів. Тут слід мати на увазі, що вектори можна множити між собою скалярно, а також множити вектор на число і навпаки – число на вектор (для операторного методу **порядок операндів суттєвий!**). Таким чином нам необхідно **три перевантажених методи** множення векторів – при скалярному множенні результатом буде дійсне число, при множенні вектора та числа між собою – результатом буде вектор.

// скалярний добуток

```
public static double operator *(d2Vector v1, d2Vector  
v2)
```

```
{    return v1.x * v2.x + v1.y * v2.y;    }
```

// множення вектора на число

```
public static d2Vector operator *(d2Vector v1, double k)  
{  
    d2Vector temp = new d2Vector(v1.x * k, v1.y * k);  
    return temp;  
}
```

// множення числа на вектор

```
public static d2Vector operator *(double k, d2Vector v1)  
{  
    d2Vector temp = new d2Vector(v1.x * k, v1.y * k);  
    return temp;  
}
```

Операторний метод множення числа на вектор після того, як визначене множення вектора на число можна було створити і в інший спосіб:

// множення числа на вектор

```
public static d2Vector operator *(double k, d2Vector v1)  
{
```



```
return v1 * k;      // тут викликаємо попередній метод
}
```

Тут просто **викликається попередній операторний метод**, який множить вектор на число.

Зверніть вагу, що при множенні числа та вектору для результату також створюється новий вектор, адже координати вектора-множника не повинні змінюватись, принаймні так це відбувається із справжніми векторами.

Тепер можемо протестувати новий фрагмент коду – включимо у метод `Main()` наступні інструкції:

```
double res = a * b;
Console.WriteLine("a * b = {0} ", res);
c = 2 * b;
Console.WriteLine("c = 2 * b: " + c.ToString());
c = a * 5;
Console.WriteLine("c = a * 5: " + c.ToString());
```

В результаті одержимо:

```
a * b = 13
c = 2 * b: x = 6 y = 10
c = a * 5: x = 5 y = 10
```

Таким чином ми створили можливість використання вбудованих операцій визначених для числових операндів також і для екземплярів нашого класу. При цьому зміст та результат операцій ми визначаємо, виходячи із змісту класу.

3. Перевантаження унарних операцій.

У множині геометричних векторів існує також поняття протилежного вектору – це вектор з координатами, що мають протилежні знаки. Крім того, хоча подібної операції у математиці не визначено, реалізуємо операції інкременту та декременту для векторів. Результуючий вектор матиме координати на одиницю більші або відповідно менші за координати заданого вектору. Це унарні операції, тобто пов'язані лише з одним операндом. Їх результатом також має бути вектор. При цьому операція унарний мінус не повинна змінювати початковий вектор на відміну від операцій інкременту та декременту. Це необхідно врахувати у коді операторних методів. Зауважимо також, що для перевантажених операцій ++ та – неможливо реалізувати постфіксний та префіксний варіанти цих операцій – прийдеться задовольнитись тим, що **обидві форми інкременту та декременту працюють однаково чином**. Включимо у клас відповідні операторні функції:

```
public static d2Vector operator -(d2Vector v) // унарний
мінус
```

```

{
    d2Vector temp = new d2Vector(-v.x, -v.y);
    return temp;
}
public static d2Vector operator ++(d2Vector v) //
інкремент
{
    v.x += 1; v.y += 1;
    return v;
}
public static d2Vector operator --(d2Vector v) //
декремент
{
    v.x -= 1; v.y -= 1;
    return v;
}

```

Внесемо також у метод `Main()` код для тестування новостворених методів:

```

b = -a;
Console.WriteLine("b = -a: " + b.ToString());
a++;
Console.WriteLine("a++: " + a.ToString());
a--;
Console.WriteLine("a--: " + a.ToString());
// префіксна та постфіксна форми працюють однаково
++a;
Console.WriteLine(++a: " + a.ToString());
--a;
Console.WriteLine(--a: " + a.ToString());

```

Результати цих дій будуть наступними:

```

b = -a: x = -1 y = -2
a++: x = 2 y = 3
a--: x = 1 y = 2
++a: x = 2 y = 3
--a: x = 1 y = 2

```

4. Перевантаження операцій відношення.

Як відомо, вектори можна перевіряти на рівність та нерівність між собою – це означає порівняння їх координат. Отже, маємо перевантажити і операції `==` та `!=`. До речі, стосовно операцій відношення діє правило: **якщо ви перевантажуєте одну з операцій `==` та `!=`, то обов'язково маєте перевантажити і іншу**. Те саме

стосується і пар операцій $< i >$ та $<= i >=$. Результатом відповідних операторних методів обов'язково має бути `true` або `false`. Створимо операторні методи для перевантаження операції `==` та `!=` у класі `d2Vector`.

```
        // перевірка векторів на рівність
public static bool operator ==(d2Vector v1, d2Vector v2)
{    return ((v1.x == v2.x) && (v1.y == v2.y));    }
```

```
        // перевірка векторів на нерівність
public static bool operator !=(d2Vector v1, d2Vector v2)
{    return ((v1.x != v2.x) || (v1.y != v2.y));    }
```

Програма з цими методами буде скомпільована та готова до виконання, проте ми одержимо зауваження від компілятора, щодо того, що при перевантаженні операцій `==` та `!=` не заміщаються методи `Object.Equals()` та `Object.GetHashCode()`. Справа в тому, що метод `Object.Equals()` перевіряє рівність двох посилованих типів і успадкований будь-яким класом від класу `Object`. Проте у разі перевантаження операцій перевірки рівності та нерівності об'єктів, компілятор хоче мати певність, що перевантажена операція `==` та метод `Equals()` використовують однакову логіку порівняння. Саме тому компілятор і радить заміщення цього методу. Що стосується методу `GetHashCode()`, то він потрібний, якщо екземпляри класу можуть використовуватись в ролі ключів у `hash`-таблицях. Оскільки векторам така роль не загрожує, просто проігноруємо це зауваження компілятора, а от метод `Equals()` мусимо таки замінити. Проте потім ми використаємо його в операторних методах `==` та `!=`. Отже, замість запропонованого вище фрагменту коду включимо наступний:

```
public override bool Equals(Object obj)    // заміщений
метод із System.Object
{ // його перевантаження вимагає operator == та operator
!=
    if (obj == null || GetType() != obj.GetType())
        return false;
    d2Vector v = (d2Vector)obj;
    return ((x == v.x) && (y == v.y));
}

        // перевірка векторів на рівність
public static bool operator ==(d2Vector v1, d2Vector v2)
{
    return v1.Equals(v2);
}

        // перевірка векторів на нерівність
public static bool operator !=(d2Vector v1, d2Vector v2)
```

```
{
    return ((v1.x != v2.x) || (v1.y != v2.y));
}
```

Напишемо також код для тестування цих методів у `Main()`:

```
Console.WriteLine((a == b));
Console.WriteLine((a != b));
```

Після запуску одержимо наступні повідомлення:

```
false
true
```

5. Перевантаження логічних операцій.

Із всього комплексу логічних операцій (`&`, `|`, `!`, `&&`, `||`) перевантажувати можна лише перші три. Кожна логічна операція має повертати результат типу `bool`. Домовимось вважати, що вектор є хибним, якщо він має всі нульові координати. Включимо в клас логічні операції:

```
public static bool operator &(d2Vector v1, d2Vector v2)
// логічне множення
```

```
{
    if ((v1.x != 0)&&(v1.y != 0) & (v2.x != 0)&&(v2.y !=
0) )
        return true;
    else return false;
}
```

`// логічне додавання`

```
public static bool operator |(d2Vector v1, d2Vector v2)
{
```

```
    if ((v1.x != 0)|| (v1.y != 0) | (v2.x != 0)|| (v2.y !=
0))
        return true;
    else return false;
}
```

`// логічне заперечення`

```
public static bool operator !(d2Vector v1)
```

```
{
    if ((v1.x != 0) || (v1.y != 0)) return false;
    else return true;
}
```

Для тестування цього фрагменту додамо у метод `Main()` наступний код

```
c = new d2Vector(0, 0);
Console.WriteLine("a & b: " + (a & b));
```

```

Console.WriteLine("a & c: " + (a & c));
Console.WriteLine("a | b: " + (a | b));
Console.WriteLine("a | c: " + (a | c));
Console.WriteLine("c | c: " + (c | c));
Console.WriteLine("!a: " + (!a));
Console.WriteLine("!c: " + (!c));

```

Одержимо наступні результати:

```

a & b: True
a & c: False
a | b: True
a | c: True
c | c: False
!a: False
!c: True

```

6. Підсумкові зауваження.

Які операції варто перевантажувати? Очевидними претендентами на роль класів, в яких повинні бути перевантажені операції, є класи, що описують математичні об'єкти – вектори, матриці, комплексні числа, точки на площині, тощо. Якщо ж клас описує, наприклад, персону чи книжку, важко уявити, що можуть означати операції множення чи додавання. Окрім того, слід мати на увазі, що не всі мови платформи .NET підтримують перевантаження операцій. Отже, якщо ваш клас планується широко використовувати, варто мати в ньому крім операторних звичайні методи, які реалізують такі операції.

Важливо також відзначити, що не всі операції допускають перевантаження. І як вже згадувалось раніше, при перевантаженні операцій їх пріоритет залишається незмінним. Перелік операцій мови C#, не дозволених до перевантаження, наведений нижче:

```

&&      ||      []      ()      new      is
sizeof
typeof      ?      ->      .      =

```

Разом із операцією присвоєння не можуть перевантажуватись і всі складені операції присвоєння на кшталт +=. Проте можете бути впевненими, що якщо в класі перевантажена операція, наприклад, додавання (+), то автоматично стає доступною для екземплярів класу і операція +=.

Структури та переліки в мові C#.

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** закрито, 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 = "ОРВИ";
            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:
 Пациєнт – Іванов, адреса – гурт. 1, вік – 18, діагноз – здоровий

Пациєнт – 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 **ComplexStruct** 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 **c11 = c12** 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 **c11** та **c12** 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("Клас      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("Клас      cl1: " +
cl1.ToString());
        Console.WriteLine("Клас      cl2: " +
cl2.ToString());
    }
}
}

```

Після виконання цього прикладу на екрані побачимо:

```

Структура st1: 2 + 2i
Клас      cl1: 2 + 2i
-----
Структура st1: 2 + 2i
Структура st2: 10 + 20i
Клас      cl1: 10 + 20i
Клас      cl2: 10 + 20i

```

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, шістнадцятковому 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.

Делегати, події та обробники подій

В цьому розділі будуть розглянуті деякі питання, пов'язані з реалізацією реакції програми на події, що можуть виникати під час її роботи, – це так звана модель подій в мові C#. Події (events) найчастіше виникають при розробці програм, орієнтованих на роботу під Windows. Ними можуть бути натискання певних клавіш або кнопок, «кліки» миші, тощо. З технічної точки зору при цьому необхідні засоби, які дозволяють викликати деякий конкретний метод, як реакцію на певну подію, що сталась в системі під час роботи. При цьому, який саме метод має зреагувати на подію, звісно, не відомо в момент компіляції програми. В багатьох мовах програмування, зокрема в C++, цій меті слугують вказівники на функцію, які виступають в ролі параметрів функції-обробника події. Проте використання прямої адресації, пов'язаної із застосуванням вказівників, не завжди

бездоганне з точки зору безпеки програми. Мова С# має безпечний засіб реалізації подієорієнтованої моделі програми – це делегати.

1. Делегати (delegate).

Отже, делегати призначені для збереження відомостей про метод, який може бути переданий у метод-обробник події в ролі його параметру. Фактично вони «делеґують» деякий метод для виконання іншим методом. З синтаксичної точки зору делегат – це посилальний тип даних (reference type), що визначає **повну** сигнатуру методу разом із його типом результату. При створенні екземплярів класу ми спочатку визначаємо клас, лише потім створюємо і використовуємо відповідні об'єкти. Так само маємо поступити і з делегатами – спочатку декларуємо тип делегата, потім створюємо і використовуємо його екземпляри, передаючи їх у метод для виконання ним.

Для декларації делегату використовується службове слово **delegate**, після якого просто вказується результат та сигнатура методу, що відповідає даному типу делегату. Наведемо приклади декларацій делегатів:

```
// делегат DoSth представляє функції, що приймають 1
рядковий
// параметр та не повертають результат
    delegate void DoSth(string x);
// делегат MyFun представляє функції, що приймають 1
дійсний
// параметр і повертає дійсне число
    delegate double MyFun(double x);
```

Делегат насправді являє собою клас, успадкований від класу **System.Delegate**. Тому декларувати делегат можна як у просторі імен, так і всередині іншого класу.

Для ілюстрації корисності делегатів, розглянемо типовий приклад. Нехай необхідно програмно побудувати графік скалярної функції виду $y = f(x)$. Для скінченного набору функцій можна написати метод, який повертає результат виклику функції $f(x)$. Програма з методом для обрахунку лише трьох функцій (синусу, косинусу та модуля) наведена нижче:

```
using System;
namespace Demo
{
    class Program
    {
        static void Main(string[] args)
```

```

        {
            // косинус в точці 1
            Console.WriteLine(Method(1, 1));
            // синус в точці 1
            Console.WriteLine(Method(2, 1));
            // модуль в точці 1
            Console.WriteLine(Method(3, 1));
        }
// Метод обраховує значення функції,
// визначеної номером iFunctionNumber, в точці x
static double Method(int iFunctionNumber, double
x)
    {
        if (iFunctionNumber == 1)
            return Math.Cos(x);
        if (iFunctionNumber == 2)
            return Math.Sin(x);
        if (iFunctionNumber == 3)
            return Math.Abs(x);
        else return 0;
    }
}
}

```

Як можна бачити, даний код є важко піддається подальшому розширенню кількості функцій, тому що всі можливі функції, значення яких можуть в ньому обраховуватись, повинні бути жорстко закодовані (hard coded) в метод `Method` класу `Program`. Подібні програми являють собою зразок поганого стилю програмування. Спробуємо покращити код програми, використавши в ролі параметру методу `Method()` не номер функції, а делегат для цього методу.

```

using System;
namespace Demo
{
// делегат MyFun представляє функції, що приймають 1
дійсний
// параметр та повертають дійсний результат
delegate double MyFun(double x);

class Program
{
    static void Main(string[] args)
    {

```

```

        // косинус в точці 1
        Console.WriteLine(Method(Math.Cos, 1));
        // синус в точці 1
        Console.WriteLine(Method(Math.Sin, 1));
        // модуль в точці 1
        Console.WriteLine(Method(Math.Abs, 1));
    }
    // Метод обраховує значення функції,
    // що визначається делегатом MyFun, в точці x
    static double Method(MyFun f, double x)
    { return f(x); }
}
}

```

Зауваження

1. Як можна помітити, тепер код програми значно скоротився.
2. Даний метод **Method()** має першим параметром функцію-делегат, значення якої буде повертатись як результат цього методу.
3. Аргументом для функції з параметром-делегатом може бути довільний метод з іншого класу, причому як статичний, так і метод екземпляру.
4. В ролі аргументу для параметра-делегата також може виступати метод, безпосередньо написаний в даному коді – так званий вбудований метод.

Вдосконалимо попередню програму, щоб проілюструвати зміст двох останніх зауважень та продемонструємо використання оператора **+=** для делегатів. Оператор **+=** для делегатів дозволяє побудувати послідовність методів-делегатів, які будуть виконані один за одним («ланцюжком») – таким чином буде організована та виконана ціла черга методів. (Так само можна використати і оператор **-=** для виключення членів черги із ланцюжка обробки). В реальному житті ця ситуація дійсно нагадує чергу, коли всі, хто в ній знаходяться, послідовно виконують одну й ту саму дію:

```

using System;
namespace Demo
{
    // делегат MyFun представляє функції, що приймають 1
    // дійсний
    // параметр та повертають дійсний результат
    delegate double MyFun(double x);
    class Program
    {

```

```

    static void Main(string[] args)
    {
        // Визначаємо делегат f, який поки що є нульовим
        // посиланням
        MyFun f = null;
        // включимо до ланцюжка делегатів статичний метод:
        f += Math.Sin;
        // включимо до ланцюжка деякий тестовий метод TestFun
        // відповідною сигнатурою:
        f += Program.TestFun;
        // включимо до ланцюжка вбудований метод – його код
        // визначений безпосередньо тут:
        f += delegate(double x)
        {
            return x * x - 1;
        };
        // запускаємо чергу-ланцюжок в метод Method():
        Console.WriteLine(Method(f, 1));
    }
    // Визначення тестового методу TestFun, сигнатура
    // відповідає делегату
    static double TestFun(double x)
    {
        return x * x * x;
    }
    // Метод обраховує значення функції,
    // визначеної делегатом f, в точці x
    static double Method(MyFun f, double x)
    { return f(x); }
}

```

В результаті виконання цього прикладу на екрані побачимо 0. Це результат виконання інструкції `Console.WriteLine (Method(f, 1))`; . В ній викликається метод `Method (f, 1)`, в якому змінна `f` одержує по черзі посилання на всі методи із ланцюжка делегатів, а на екрані ми бачимо лише результат звертання до останнього з них – вбудованого методу, який повертає значення $x^2 - 1$.

Зауваження

1. Оскільки делегат є представником reference-типу, то його значенням за замовченням є `null`. Таким чином, якщо в метод

замість функції буде переданий нульовий вказівник `null` (у випадку відсутньої ініціалізації делегату), відбудеться виключення `NullReferenceException`. Тому, коректним підходом до використання екземпляру-делегату є перевірка його значення на нерівність нульовій адресі `null`.

2. У випадку побудови ланцюжка з функцій-делегатів, що повертають значення, та виклику методу з аргументом-делегатом, послідовно виконуються всі функції, що входять до ланцюжка, а результатом всього ланцюжка буде результат виконання останньої функції в ланцюжку (last wins).

2. Події та їх обробники.

Як вже згадувалось вище, модель подій базується на тому, що процес виконання програми залежить від подій, що відбуваються в ній. Далі програма реагує на подію викликом відповідного обробника цієї події.

Події як елемент програми безпосередньо пов'язані з делегатами. **Подія** – це особливий член-класу (поле класу), його тип має бути деяким делегатом. Для визначення події використовується службове слово `event`. У наступному прикладі декларується делегат `MyFun`, який в класі-обробнику `Observer` визначає тип події `myEvent`.

```
// делегат MyFun представляє функції, що приймають 1
дійсний
// параметр та повертають дійсний результат
delegate double MyFun(double x);
// декларація класу, що містить подію
class Observer
{
// декларація події myEvent типу MyFun, яка
відбуватиметься
    public event MyFun myEvent;
}
```

Насправді можна обійтись і без `event` – у фрагменті програми, наведеному нижче, членом класу є делегат (незважаючи на ідейну некоректність цього коду, він може бути успішно відкомпільованим та виконаним):

```
// декларація некоректно визначеного класу, що містить
подію
class WrongObserver
{
// декларація події myEvent, яка відбуватиметься
```

```

    public MyFun myEvent;
}

```

Використаємо обидва класи наступним чином:

```

class Program
{
    static void Main(string[] args)
    {
        // робота з класом, що містить коректний обробник
        // подій
        Observer o = new Observer();
        // існує можливість як включити обробник до черги
        // обробки
        o.myEvent += Math.Sin;
        // так і виключити його звідти
        o.myEvent -= Math.Cos;
        // робота з класом, що містить некоректну подію
        WrongObserver wo = new WrongObserver();
        // тут також можна включити обробник в чергу
        wo.myEvent += Math.Cos;
        // тут також можна виключити обробник
        wo.myEvent -= Math.Cos;
        // проте тут є потенційна можливість монопольного
        // перехоплення обробки події, який демонструє
        // наступний
        // рядок коду
        wo.myEvent = Math.Tan;
    }
}

```

Таким чином ключове слово `event` забороняє монопольне перехоплення події якимось обробником. За відсутності декларування у класі поля `myEvent` з модифікатором `event` існує можливість реалізації заміщення побудованої послідовності обробників одним новим обробником (замість оператора `+=` можна використати звичайний оператор присвоєння `=`).

Для поля ж з модифікатором `event` існує можливість лише включення в кінець черги-ланцюжка, або виключення з нього. Монопольного захоплення поля з використанням оператора присвоєння компілятором забороняється.

Далі, потрібно зауважити, що у випадку спроби видалення з ланцюжка за допомогою оператора `-=` обробника, якого в ньому немає, ніякої виключної ситуації не відбувається. У випадку якщо відбувається виключення з ланцюжка обробника методу, який там присутній,

декілька разів, (наприклад, двічі), кількість входжень його у чергу буде просто зменшена на одиницю. Насправді ніяких інших додаткових функцій службове слово `event` не має.

```
static void Main(string[] args)
{
    o.myEvent += new MyFun(o_myEvent);
    ....
}

static void Main(string[] args)
{
    // код методу
}
```

Зазвичай типові обробники подій мають два параметри. Спробуємо пояснити причину використання саме двох параметрів. Для цього уявимо віртуальну ситуацію-казочку. Нехай деякий рибалка пішов на риболовлю та використав декілька вудок для ловлі риби. Він поставив дзвіночки на вудки для того, що спостерігати, коли буде відбуватися клювання. З точки зору програмування мовою C# цю ситуацію можна абстрагувати наступним чином:

```
using System;
// делегат - обробник сигналу від риби
delegate void FishHandler();
// клас рибалки
class Fisher
{
    // метод обробки події від вудки, який відповідає типу
    // FishHandler
    public void TakeFish()
    {
        Console.WriteLine("Клює");
    }
}
// клас вудки
class Spinning
{
    // назва вудки
    public string Name = "";
    // конструктор класу
    public Spinning(string s) { this.Name = s; }
    // подія від риби
}
```

```

public event FishHandler ItHappens;
// емулятор спрацьовування - оскільки в програмі
// немає сторонніх джерел генерації події
public void Simulate()
{
    // якщо б ніхто не зареєструвався на подію, то
    // без перевірки буде виключення!
    if (this.ItHappens != null)
    {
        // запусимо ланцюжок обробників
        this.ItHappens();
    }
}
}
class Program
{
    static void Main(string[] args)
    {
        // створюємо екземпляр рибалки
        Fisher f = new Fisher();
        // створюємо обробник
        Spinning s1 = new Spinning("# 1");
        Spinning s2 = new Spinning("# 2");
        // зареєструємо один метод-обробник на декілька
        подій
        s1.ItHappens += f.TakeFish;
        s2.ItHappens += f.TakeFish;
        // емулюємо запуск на першій вудці
        s1.Simulate();
        // емулюємо запуск на другій вудці
        s2.Simulate();
    }
}

```

Результатом виводу цієї програми буде

Ключе

Ключе

Зверніть увагу, що в класі `Spinning` при генерації події в методі `Simulate()` ми перевірили наявність обробників події. Відсутність цієї перевірки могла б призвести до помилки під час виконання. Проте, як можна бачити з результату виконання програми, обробник всередині класу `Fisher` не має відомостей про те, яка з вудок запустила на виконання подію. Тому, для зручності використання клієнтами класу

Spinning потрібно використати делегат з параметром, що визначає джерело події (додати параметр типу класу-генератору події):

```
using System;
// делегат - обробник сигналу від риби
delegate void FishHandler(Spinning sender);
// клас рибалки
class Fisher
{
    // метод обробки події від вудки, який відповідає типу
    // FishHandler
    public void TakeFish(Spinning spin)
    {
        Console.WriteLine("Клює на спінінгу " +
spin.Name);
    }
}
// клас вудки
class Spinning
{
    // назва вудки
    public string Name = "";
    // конструктор класу
    public Spinning(string s) { this.Name = s; }
    // подія від риби
    public event FishHandler ItHappens;
    // емулятор спрацьовування - оскільки в програмі
    // немає сторонніх джерел генерації події
    public void Simulate()
    {
        // якщо б ніхто не зареєструвався на подію, то
        // без перевірки буде виключення!
        if (this.ItHappens != null)
        {
            // запустимо ланцюжок обробників
            this.ItHappens(this);
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        // створюємо екземпляр рибалки
```

```

    Fisher f = new Fisher();
    // створює обробник
    Spinning s1 = new Spinning("# 1");
    Spinning s2 = new Spinning("# 2");
    // зареєструємо один метод-обробник на декілька
подій
    s1.ItHappens += f.TakeFish;
    s2.ItHappens += f.TakeFish;
    // емулюємо запуск на першій вудці
    s1.Simulate();
    // емулюємо запуск на другій вудці
    s2.Simulate();
}
}

```

При виконанні ця програма виведе на екран наступні повідомлення:

Ключе на спінінгу #1

Ключе на спінінгу #2

У програмі змінилось визначення делегату, разом із сигнатурою методу-обробника. Зверніть увагу на те, що сигнатура функції генерації події `Simulate()` не змінилась, оскільки об'єкт передає обробникам посилання на себе завдяки використанню ключового слова `this`. Тепер, нехай потрібно передавати обробнику додаткову інформацію про подію, наприклад, гучність дзвіночка на вудці (у випадку Windows-програм, властивостями подій, що відбуваються в програмі, можуть бути координати миші, кнопка миші, яку натиснув користувач, код клавиші клавіатури тощо). В даному випадку нехай це буде певна рядкова змінна:

```

using System;
// делегат - обробник сигналу від риби
delegate void FishHandler(Spinning sender, string args);
// клас рибалки
class Fisher
{
    // метод обробки події від вудки, який відповідає типу
    // FishHandler
    public void TakeFish(Spinning spin, string info)
    {
        Console.WriteLine(
            "Ключе " + info + " на спінінгу " +
spin.Name);
    }
}
}

```

```

// клас вудки
class Spinning
{
    // назва вудки
    public string Name = "";
    // конструктор класу
    public Spinning(string s) { this.Name = s; }
    // подія від риби
    public event FishHandler ItHappens;
    // емулятор спрацьовування - оскільки в програмі
    // немає сторонніх джерел генерації події
    public void Simulate(string info)
    {
        // якщо б ніхто не зареєструвався на подію, то
        // без перевірки буде виключення!
        if (this.ItHappens != null)
        {
            // запустимо ланцюжок обробників
            this.ItHappens(this, info);
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        // створюємо екземпляр рибалки
        Fisher f = new Fisher();
        // створює обробник
        Spinning s1 = new Spinning("# 1");
        Spinning s2 = new Spinning("# 2");
        // зареєструємо один метод-обробник на декілька
        подій
        s1.ItHappens += f.TakeFish;
        s2.ItHappens += f.TakeFish;
        // емулюємо запуск на першій вудці
        s1.Simulate("сильно");
        // емулюємо запуск на другій вудці
        s2.Simulate("слабко");
    }
}

```

Таким чином, ми прийшли до класичного двопараметричного визначення делегату-обробника події. Найбільш розповсюдженим

типом обробників подій для Windows програм є тип `System.EventHandler`

```
public delegate void EventHandler(object sender,  
EventArgs e);
```

Атрибути та їх використання

Ідея використання атрибутів навіяна технологією COM-програмування і не має аналогів у класичних мовах програмування таких, як C або C++. **Атрибути** мови C# є певною мірою анотаціями до програми, їх призначення – зберігати інформацію про деякі елементи програми у так званих метаданих збірки (assembly). Пригадайте, що виконуваний файл C#-програми називається **збіркою** і крім власне відкомпільованого коду містить метадані. Спеціальні засоби програмування, такі як **відображення** (або **рефлексія** – reflection) дозволяють аналізувати зміст метаданих **просто під час виконання програми**. На жаль, ця тема залишається поза межами даного посібника, а про атрибути поговоримо лише оглядово.

З точки зору синтаксису мови C# атрибутом є певний текст, який поміщений **у квадратні дужки** і відноситься до частини програмного коду, що знаходиться безпосередньо після нього. Атрибути можуть застосовуватись до класів, до членів класу або до збірки в цілому. Цікаво, що ця «зона поширення» атрибуту визначається в свою чергу спеціальним вбудованим атрибутом `AttributeUsage`.

Атрибути .NET (і мови C# зокрема) є об'єктами, що походять від класу `System.Attribute`. Мова C# постачає певну кількість вбудованих атрибутів різного призначення і крім того, дозволяє користувачеві створювати власні атрибути.

Нижче наведений приклад, в якому ілюструється використання вбудованого атрибуту `DllImport`. Його призначення – виклик методів із системних dll-файлів.

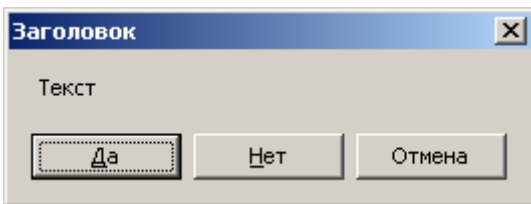
```
using System;  
using System.Runtime.InteropServices;  
namespace UsingAttribute2  
{  
    // Приклад виклику функції із зовнішньої бібліотеки  
    class Program  
    {  
        // Атрибут для забезпечення виклику функції  
        [DllImport("user32.dll")]  
        // Декларація зовнішньої функції, що викликається
```

```

extern static int MessageBox(int hWnd, string
s1,
                                string s2, int
buttons);
// Буде використано зовнішню функцію MessageBox()
створення
// діалогового вікна із бібліотеки user32.dll. Її
параметри:
// hWnd - вказівник на вікно
// s1 - текст у вікні
// s2 - заголовок вікна
// buttons - константа, що задає кількість кнопок
// результат - код натиснутої кнопки
static void Main(string[] args)
{
// виклик діалогового вікна, k - код результату виклику
int k = MessageBox(0, "Текст", "Заголовок",
3);
// виводимо код результату
Console.WriteLine(k);
}
}
}

```

Ця програма виведе системне вікно наступного вигляду (цифра 3 у виклику функції означає стандартний набір з трьох кнопок)



Після натискання однієї із трьох кнопок на екрані ми побачимо число, що відповідає коду даної кнопки у вікні.

Таким чином, можна викликати функції, які написані на мові відмінній від мов .NET Framework, наприклад з динамічних бібліотек написаних на класичному C++ або C. Далі наведений приклад коду на мові C++, який можна додати до коду довільної DLL бібліотеки, для того, щоб функцію можна було б використовувати з C#:

```

extern "C"
{
    __declspec(dllexport) int TestFunction(int j)
    {
        return(j*j*j);
    }
}

```

```

    }
}

```

Таким чином, ми маємо функцію, що написана на мові C (unmanaged). Нехай вона компілюється до динамічної бібліотеки **MyDll.dll**. Тоді для використання її в довільному класі необхідно додати наступний код до класу:

```

[DllImport("MyDll.dll")]
extern static int TestFunction(int x);

```

Ніяким іншим шляхом без використання атрибутів, неможливо досягти **інтеграції managed та unmanaged коду**. Файл **MyDll.dll** повинен знаходитись або в поточному каталогі виконуючої програми, або в каталогах, що визначаються змінною оточення PATH.

Ще одним цікавим вбудованим атрибутом мови C# є атрибут `[System.Diagnostics.Conditional("ідентифікатор")]`. Він дозволяє реалізувати так званий умовний метод. Метод, перед звертанням до якого визначений даний атрибут, буде викликаний лише за умови, коли у програмі визначена директива **#define ідентифікатор**.

У наступному прикладі визначені та викликаються функцією **Main()** два методи: **DoMethod1()** та **DoMethod2()**. Проте кожному з них передуює атрибут **Conditional** (Кожний з атрибутів застосовується до метода, перед яким він вказаний). Перший атрибут пов'язаний із іменем **DEBUG** (це вбудований системний ідентифікатор, який діє у випадку, коли програма запускається в режимі налагоджування) і відноситься до методу **DoMethod1()**. Отже, реально передача управління цьому методу функцією **Main()** відбудеться лише тоді, коли програма буде запущена в режимі налагоджування. Другий атрибут відноситься до методу **DoMethod2()** та пов'язаний із ідентифікатором **COND**. Цей ідентифікатор визначений у програмі директивою **#define COND**, отже метод **DoMethod2()** буде викликаний. Перевірте, якщо рядок з декларацією імені **COND** виключити з програми, то цей метод викликатись не буде.

```

// декларація ідентифікатора, що визначає умовний метод
#define COND
using System;
namespace UsingAttributes
{
    class Program
    {
        // умовна компіляції - у випадку виконання

```

```
// в режимі налагоджування (DEBUG) метод працює
[System.Diagnostics.Conditional("DEBUG")]
static void DoMethod1(string s)
{
    Console.WriteLine("Режим DEBUG " + s);
}
// цей метод виконається у випадку декларації
змінної COND
[System.Diagnostics.Conditional("COND")]
static void DoMethod2(string s)
{
    Console.WriteLine("Режим REALISE " + s);
}
static void Main(string[] args)
{
    // викликаємо обидва методи
    DoMethod1("1");
    DoMethod2("2");
    Console.ReadLine();
}
}
```

Отже, у випадку виконання програми в режимі налагоджування будуть виведені повідомлення від обох методів:

Режим DEBUG 1

Режим REALISE 2

а у випадку виконання в звичайному режимі (Release) – лише від одного:

Режим REALISE 2

І на завершення цієї теми розглянемо створення власного атрибуту. Даний приклад для повного осягнення використання переваг атрибутів потребує значного розуміння програмування в цілому, проте є яскравим та наочним прикладом сукупного використання наслідування та декларативного програмування в мові С#. Це призводить до значного зменшення об'єму коду та збільшення у порівняння з іншими підходами.

Приклад використовує створення класу, похідного від `System.Attribute`, в якому визначаються члени класу, що підтримують даний атрибут. Перед визначенням класу слід помістити атрибути, які задають, до яких програмних елементів та яким чином може бути застосований атрибут, що визначається.

Зауваження.

1. Якщо треба вказати не один атрибут, а декілька, то їх можна визначити через кому в квадратних дужках або кожний з них окремо у власній парі дужок.
2. При визначенні класу атрибуту його ідентифікатор має завершуватись суфіксом `Attribute`, наприклад, `MyAttribute`, `RemarkAttribute` тощо.

```
using System;
namespace UsingAttribute
{
    // атрибут перед декларацією класа атрибуту вказує,
    // що цей атрибут буде застосовуватись до класів
    // та може використовуватись декілька разів
    [AttributeUsage(AttributeTargets.Class,
        AllowMultiple=true)]
    class MyAttribute: Attribute
    {
        private int x, y;
        // перша властивість
        public int X
        {
            get { return x; }
            set { x = value; }
        }
        // друга властивість
        public int Y
        {
            get { return y; }
            set { y = value; }
        }
    }
}
```

В цьому класі атрибуту визначені два закритих поля `x` та `y`, а також відповідні їм відкриті властивості. На перший погляд, це просто звичайний клас, проте він може бути використаний як атрибут до ієрархії класів (базовий клас `Animal` та два похідних від нього класи `Bear` та `Rabbit`).

```
using System;
namespace UsingAttribute
{
    // визначення базового класу з атрибутами
    // до базового класу застосовується
```



```

// атрибут MyAttribute зі значенням 10 та 20
[MyAttribute(X = 10, Y = 20)]
class Animal
{
    // два поля: швидкість та запас здоров'я (життя)
    public int Speed = 0, Health = 0;
    // конструктор базового класу
    public Animal()
    {
        // отримаємо всі атрибути цього класу
        object[] obj =
this.GetType().GetCustomAttributes(true);
        // перебір всіх атрибутів
        foreach (object o in obj)
        {
            // якщо це атрибути потрібного типу
            if (o is MyAttribute)
            {
                // то додаємо значення параметрів атрибутів
                this.Speed += (o as MyAttribute).X;
                this.Health += (o as MyAttribute).Y;
            }
        }
    }
    // перевантажений метод ToString() для виводу
екземплярів
    public override string ToString()
    {
        return string.Format("Speed = {0}, Health = {1} ",
            this.Speed, this.Health);
    }
}
// У екземплярів класу Bear буде швидкість 13,
// та кількість життя 40
[MyAttribute(X=3, Y=30)]
class Bear : Animal
{
}
// У екземплярів класу Rabbit буде швидкість 30,
// та кількість життя 20
[MyAttribute(X = 20, Y = 10)]
class Rabbit : Animal
{

```

```
}  
}
```

Таким чином, застосування атрибутів у даному класі дещо збільшує кількість коду у базовому класі, та суттєво зменшує кількість коду у похідних класах. Наступна програма виводить дані цих класів:

```
using System;  
namespace UsingAttribute  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            // створемо об'єкти класів  
            Animal a = new Animal();  
            Animal b = new Bear();  
            Animal r = new Rabbit();  
            // вивід даних  
            Console.WriteLine(a);  
            Console.WriteLine(b);  
            Console.WriteLine(r);  
        }  
    }  
}
```

Після виконання одержимо:

```
Speed = 10, Health = 20  
Speed = 13, Health = 50  
Speed = 30, Health = 30
```

Таким чином, в даному посібнику розглянуті основні базові питання, пов'язані із програмуванням мовою C#.

Додаток. Ключові (зарезервовані) слова мови C#

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

Рекомендована література

1. Б. Керниган, Р. Пайк. Практика программирования – СПб.; М.: «Невский диалект», 2001
2. В.О.Грязнова, С.В. Єфіменко. Основи методології програмування. - К.: ВПЦ "Київський університет", 2005 р.
3. Г. Шилдт. Полный справочник по C#. – М.: Издательский дом "Вильямс", 2008 г.
4. Т.А. Павловская. C#. Программирование на языке высокого уровня. – СПб. : Питер, 2007
5. Э. Троелсен. C# и платформа .NET. Библиотека программиста. – СПб. : Питер, 2007

Навчальне видання

Основи програмування. Мова C#.

Методичний посібник для студентів радіофізичного факультету університету

Автори:

**Грязнова Віра Олександрівна,
Єфіменко Світлана Володимирівна
Юштин Костянтин Едуардович**

Друкується за авторською редакцією

Підписано до друку **??.**2009. Формат 60x80¹⁶.
Гарнітура Arial. Папір офсетний. Друк офсетний.
Наклад **???** примірників. Ум. друк. арк. **??**.

Видавнича лабораторія радіофізичного факультету
Київського національного університету імені Тараса Шевченка