Міністерство освіти і науки України

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

П. Б. Олійник

ІНФОРМАТИКА ТА ПРОГРАМУВАННЯ.

ЧАСТИНА 4. ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ. МОВА ПРОГРАМУВАННЯ C#.

Курс лекцій

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньою програмою «Комп'ютерне моделювання фізичних процесів»
спеціальності 104 «Фізика та астрономія»

Електронне мережеве навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2024

УДК 004.43 (004.4’2)
О54

Автор: *Олійник Павло Борисович*, канд. техн. наук

Рецензент Аушева Н.М., д.т.н., професор
КПІ ім.. Ігоря Сікорського, кафедра цифрових технологій в енергетиці

Відповідальний редактор Залевський С.В., канд. техн. наук, доцент

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № X від DD.MM.YYYY р.)
за поданням вченої ради фізико-математичного факультету
(протокол № X від DD.MM.YYYY р.)

Олійник П. Б.

О54 Інформатика та програмування. Частина 4. Об'єктно-орієнтоване програмування. Мова програмування C# [Електронний ресурс] : курс лекцій : навч. посіб. для здобувачів ступеня бакалавра за освіт. програмою «Комп’ютерне моделювання фізичних процесів» спец. 104 «Фізика та астрономія» / П. Б. Олійник; КПІ ім. Ігоря Сікорського. – Електрон. текст. дані (1 файл). – Київ : КПІ ім. Ігоря Сікорського, 2024. – 123 с.

Посібник містить 9 лекцій, в яких викладено курс основ мови програмування C#. Увагу приділено особливостям мови C#, в т.ч. особливостям об’єктно-орієнтованого програмування, наведено приклади, які доступно пояснюють концепції мови та особливості її практичного використання. Посібник призначений для здобувачів ступеня бакалавра за спеціальністю 104 «Фізика та астрономія» (освітня програма «Комп’ютерне моделювання фізичних процесів»). Буде також корисним для студентів, аспірантів та спеціалістів, які бажають освоїти програмування мовою C#.

УДК 004.43 (004.4’2)

Реєстр. № НП XX/XX-XXX. Обсяг 8,7 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Берестейський, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

ЗМІСТ

Вступ

У сучасних обчислювальних системах актуальним є забезпечення кросплатформеної розробки, тобто розробки програм, які можуть працювати на різних апаратних засобах під різними операційними системами. На комп'ютерах під операційною системою Microsoft Windows поряд із Java ключовою технологією кросплатформеної розробки є Microsoft .NET. Програми, написані під .NET, можна використовувати під Windows, Free BSD, MacOS X і Linux (із використанням Mono).

Флагманською мовою програмування .NET є мова C#, розроблена Андерсом Гейлсбергом, Скотом Вілтамутом та Пітером Гольде. Мова C# – об'єктно-орієнтована мова з C-подібним синтаксисом. Використання C# дає змогу виконувати складні розрахунки, писати додатки з графічним інтерфейсом, системи керування базами даних, вимірювальні системи та системи керування обладнанням.

Базовим засобом розробки програм мовою C# є Microsoft Visual Studio. Також C# використовується в кросплатформеному інструменті Unity, завдяки якому можна швидко розробляти застосунки та ігри.

Цей курс із 9 лекцій є вступом до програмування мовою C# (версія C#12, .NET 8.0), і містить відомості про:

– Microsoft .NET і основні концепції цієї технології;

– базове середовище розробки – Microsoft Visual Studio;

– додаткові пакети бібліотек NuGet, їх встановлення та використання;

– структуру програми мовою C#, константи і літерали;

– базові типи даних і керуючі конструкції мови;

– консольне введення-виведення даних;

– масиви C# і особливості їх реалізації;

– методи, перезавантаження методів;

– основні концепції об'єктно-орієнтованого програмування і особливості їх реалізації мовою C#;

– інтерфейси і делегати;

– основи Windows Forms і WPF, швидке створення інтерфейсу користувача;

– обробку виключних ситуацій;

– роботу з файлами;

– узагальнення (generics).

Курс лекцій викладено на основі матеріалів сайтів abitap.com, wpf-tutorial.com та learn.microsoft.com. Всі лекції ілюстровано прикладами, які показують особливості використання конструкцій мови. Для кращого розуміння курсу бажане (хоча й не обов'язкове) володіння мовою C++.

Після освоєння курсу студент зможе писати мовою C# прикладні програми, як консольні, так і з графічним інтерфейсом, використовувати мову для проведення обчислювальних експериментів і моделювання фізичних процесів і систем, автоматизації фізичних досліджень та вирішення інших наукових і практичних задач.

При розгляді конструкцій мови у посібнику застосовано такі позначення:

<a> - обов’язковий аргумент;

[a] - необов’язковий аргумент.

Якщо в конструкціях мови використовуються <> або [], вони виділяються жирним шрифтом.

Лекція 1. Microsoft .Net і основні концепції цієї технології. Microsoft Visual Studio IDE. Перша програма на C#. Поняття про простори імен. Пакети NuGet

1.1 Microsoft .NET і основні концепції цієї технології. JIT-компіляція. Збірки (assemblies). Керований код.

З часу появи перших комп'ютерів програмне забезпечення для більшості із них було орієнтовано на виконання лише на тому типі комп'ютера, для якого воно написане. Це було зумовлено несумісністю машинних мов різних процесорів, використанням різних форматів виконуваних файлів у різних операційних системах, особливостями реалізації системних бібліотек та іншими причинами. Як наслідок, і досі програма, написана для Windows, не може виконуватися під Linux або MacOS без додаткового програмного забезпечення, яке забезпечує використання такою програмою стандартних функцій операційної системи (наприклад, введення-виведення, роботи з файлами, виділення і звільнення пам'яті і т.п.).

Проблему переносу програм дещо пом'якшує використання мов програмування на кшталт C/C++, які забезпечують сумісність програм на рівні вихідного коду. Це дає змогу переносити програму між різними комп'ютерами у вигляді тексту програми (вихідного коду), після перекомпіляції якого програма однаково працює на різних операційних системах та архітектурах комп'ютерів. Недоліком такого підходу є необхідність перекомпіляції програм, а, іноді, і їх адаптації до різних систем. Це потребує часу і певного рівня кваліфікації користувача, адже у деяких випадках необхідно модифікувати текст програми, що для більшості користувачів недоступно.

Першою мовою програмування, відразу призначеною для забезпечення переносу програм між комп'ютерами з різними архітектурами та операційними системами на рівні виконуваного коду, була мова Java від Sun Microsystems. Основою забезпечення такої функції у Java стало використання для виконання скомпільованого коду віртуальної машини. Компілятор Java створює виконуваний код не на машинній мові процесора комп'ютера, а на байт-коді – машинній мові віртуальної машини, яка при виконанні на різних комп'ютерах і під різними операційними системами функціонує однаково. Як наслідок, з'явилася можливість написавши програму на IBM PC, перенести її на Mac або під Linux просто скопіювавши .jar-файл, причому функціонування програми на всіх платформах буде ідентичним – включаючи навіть графічний інтерфейс, який на різних платформах виглядатиме дещо по-різному, але функціонуватиме однаково. Для виконання таких програм необхідно використовувати пакет Java runtime, який, власне, і реалізує віртуальну машину Java, та відповідні бібліотеки класів Java.

На початку 2000-х почався бум використання мобільних пристроїв. Microsoft .NET з'явилася у 2002 році як прямий конкурент Java, у спробі перенести додатки Windows із персональних комп'ютерів на базі x86 та x64 на інші пристрої, включаючи планшети та мобільні телефони. Оскільки в

планшетах і смартфонах використовуються, на відміну від ПК, ARM процесори, безпосереднє виконання додатків із ПК на них неможливе – навіть за умови встановлення “мобільних” операційних систем Windows CE/Windows Mobile. Для вирішення цієї проблеми і було створено .NET.

В основі .NET лежить та сама ідея, що й в основі Java: виконуваний код створюється не на машинній мові процесора, який виконує програму, а на байт-коді віртуальної машини мовою CIL (Common Intermediate Language). Основою платформи .NET є віртуальна машина – так зване єдине середовище виконання Common Language Runtime (CLR), що забезпечує виконання скомпільованих у CIL модулів, написаних на різних мовах, що підтримують .NET. Це, зокрема, такі мови, як Microsoft Visual Basic .Net, C++/CLI (C++, адаптований під .NET), C#, F#, а також діалекти інших мов програмування, наприклад Python, FORTRAN, Java. **Підтримка кількох мов програмування** дає змогу реалізовувати великі проекти на кількох різних мовах, гнучко використовуючи переваги кожної, що є беззаперечною перевагою .NET.

Другою перевагою .NET є її **кросплатформеність** на рівні відкомпільованих модулів (за термінологією .NET – збірок (assemblies)): один і той же модуль однаково працюватиме на різних пристроях, незалежно від процесора і операційної системи. Для пришвидшення виконання коду при запуску збірки байт-код перетворюється вбудованим у середовище Just in time (JIT) компілятором на код цільового процесора «на льоту», причому компілюється лише частина коду, що безпосередньо виконуватиметься. Раніше відкомпільований код при цьому залишається в пам’яті.

Третьою перевагою є потужна базова бібліотека класів, та різноманітність технологій. Базова бібліотека класів є єдиною для всіх мов, що підтримують .NET, і дає змогу реалізувати будь-який додаток – від калькулятора до складних розподілених мережевих систем обробки даних. Базова бібліотека містить класи, які реалізують багато стандартних задач – обчислення математичних функцій, використання стандартних структур даних на кшталт списків, сортування списків і масивів, роботу з файлами, розрахунок контрольних сум, криптографію та інше. До того ж, базова бібліотека класів є основою стеку технологій, які розробники можуть задіяти при побудові тих чи інших додатків. Наприклад, для роботи із базами даних у .NET можна використати ADO.NET або Entity Framework Core. Для розробки графічних додатків у стилі Windows 8/10/11 – технології WPF та WinUI, а для створення графічних додатків із класичним інтерфейсом – Windows Forms. Для розробки кросплатформних мобільних застосунків і програм для персональних комп'ютерів – Xamarin/MAUI. Веб-сайти та веб-додатки в .NET можна створювати використовуючи ASP.NET і т.д.

Ще однією перевагою є використання при програмуванні під .NET так званого **керованого коду**. Керований код (managed code) відрізняється від класичного (некерованого) коду тим, що програма, написана під .NET управляється середовищем виконання CLR, на яке покладається очищення пам’яті. Як наслідок, у всіх мовах .NET немає операторів звільнення пам’яті –

уся виділена пам’ять звільняється в процесі виконання програми автоматично. Це дещо спрощує життя програмісту, оскільки, на відміну від класичних мов, а особливо C/C++, виключено виникнення ситуацій використання «диких» вказівників, які вказують на знищені об’єкти або виділення і не звільнення пам’яті («витоки пам’яті»). Цю концепцію повністю скопійовано із Java.

Недоліки платформи .NET є продовженням її переваг. По-перше, програма на .NET ніколи не буде виконуватися так швидко, як програма, скомпільована під цільовий процесор у його машинних кодах – оскільки JIT компілятор витрачає час на перекомпіляцію байт-коду CIL в машинні коди. По-друге, керований код не дає можливості прямого доступу до пам’яті, тому на відміну від тієї ж мови C++, мови .NET «віддаляються» від апаратури. Все це обмежує використання .NET у задачах, в яких необхідна безпосередня взаємодія з апаратним забезпеченням і робота в реальному масштабі часу. Щоправда, станом на 2024 рік існує адаптація платформи .NET nanoFramework, орієнтована на мікроконтролери та IoT (Internet of things), котра дає змогу писати програми на C# для мікроконтролерів і взаємодіяти з апаратурою на низькому рівні.

Для підвищення продуктивності програм при виконанні критичних за часом задач .NET допускає використання у програмах так званого native code, тобто коду, написаного під певний процесор, на якому виконується програма. Однак, використання коду в командах цільового процесора накладає на програму під .NET певні обмеження, і однозначно нівелює її переносимість між різними платформами.

.NET тривалий час розвивалася як платформа лише для Windows під назвою .NET Framework; остання версія цієї платформи – .NET Framework 4.8 – була випущена в 2019 році. У 2014 році Microsoft розпочала розробку нової платформи .NET Core, вже кросплатформеної. .NET Core призначена замінити .NET Framework, і поєднує можливості застарілої системи і нову функціональність. Поточна версія .NET Core – 8.0, програмування під неї підтримується середовищем Visual Studio 2022.

Станом на 2024 рік існує реалізація .NET для Windows, MacOS, а також Unix-подібних систем, зокрема FreeBSD (від Microsoft), та для Linux в проекті Mono (сумісна розробка Microsoft та Novell).

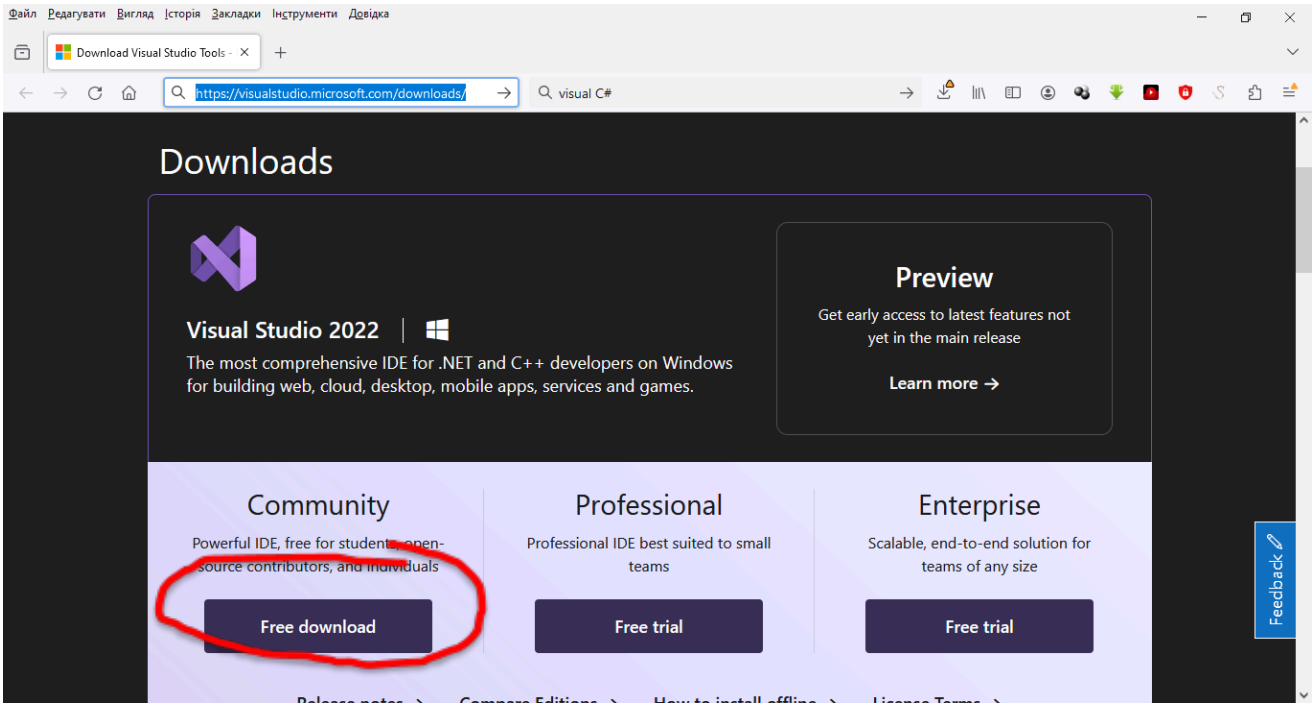
1.2 Мова C#. Microsoft Visual Studio IDE – основний засіб розробки. Перша програма на C#.

Флагманською мовою .NET є мова програмування C#, розроблена Андерсом Гейлсбергом, Пітером Гольде та Скотом Вілтамутом в Microsoft Research. Мова C# – об’єктно-орієнтована мова із C-подібним синтаксисом. Багато властивостей автори цієї мови запозичили з C++ і Java, а також із Object Pascal (Delphi), Modula та Smalltalk. C# підтримує поліморфізм, успадкування, перевантаження функцій та операторів, статичну типізацію, властивості, події, винятки (exceptions). Із кожною новою версією в мову додаються нові можливості. Детальний список змін від версії до версії можна побачити тут:

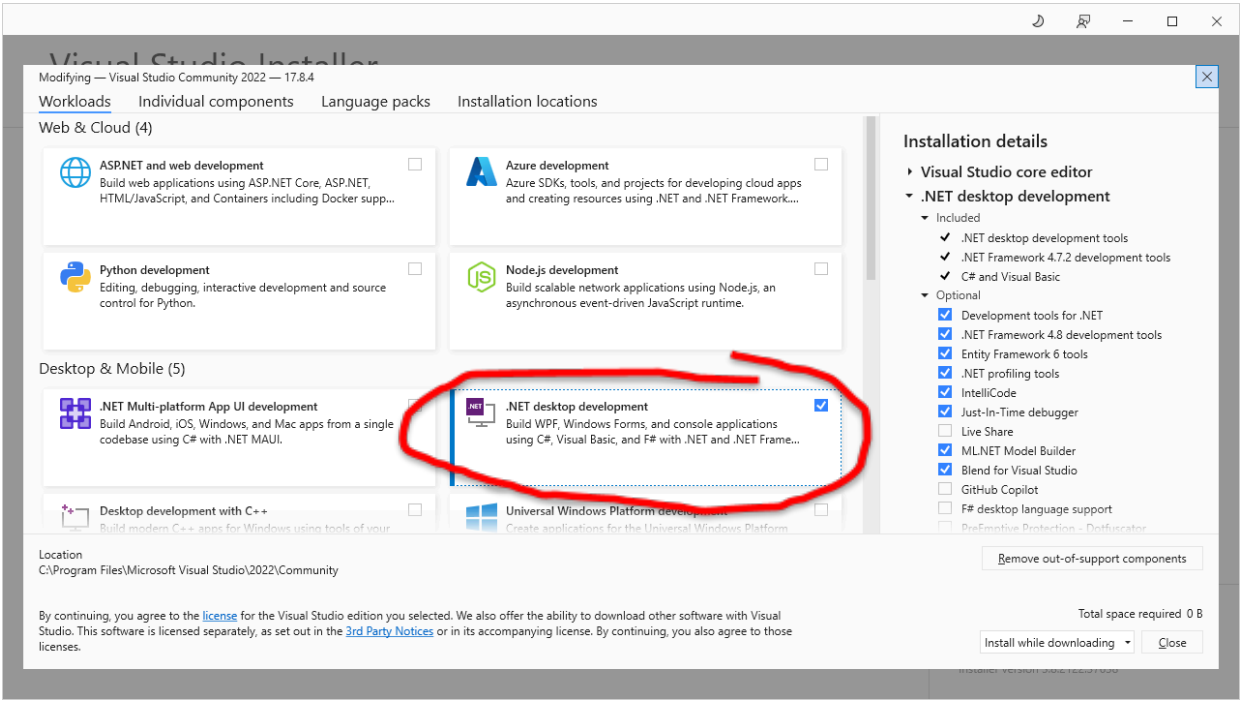
https://en.wikipedia.org/wiki/C_Sharp_(programming_language) або тут https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/tutorials/.

Станом на 2024 рік стабільною версією мови є C# 11 (розроблена для .NET 7.0); використовується також C#12 (.NET 8.0).

Основним засобом розробки під Windows та MacOS X є Microsoft Visual Studio. Це середовище розробки розвивається з 1997 року, і доступне в кількох редакціях; для навчання C# та розробки прикладних програм можна скористатися безплатною редакцією Community Edition. Для завантаження Microsoft Visual Studio слід перейти на сторінку https://visualstudio.microsoft.com/downloads/ і клацнути по кнопці «Free Download», після чого завантажити інсталятор VisualStudioSetup.exe та запустити його.

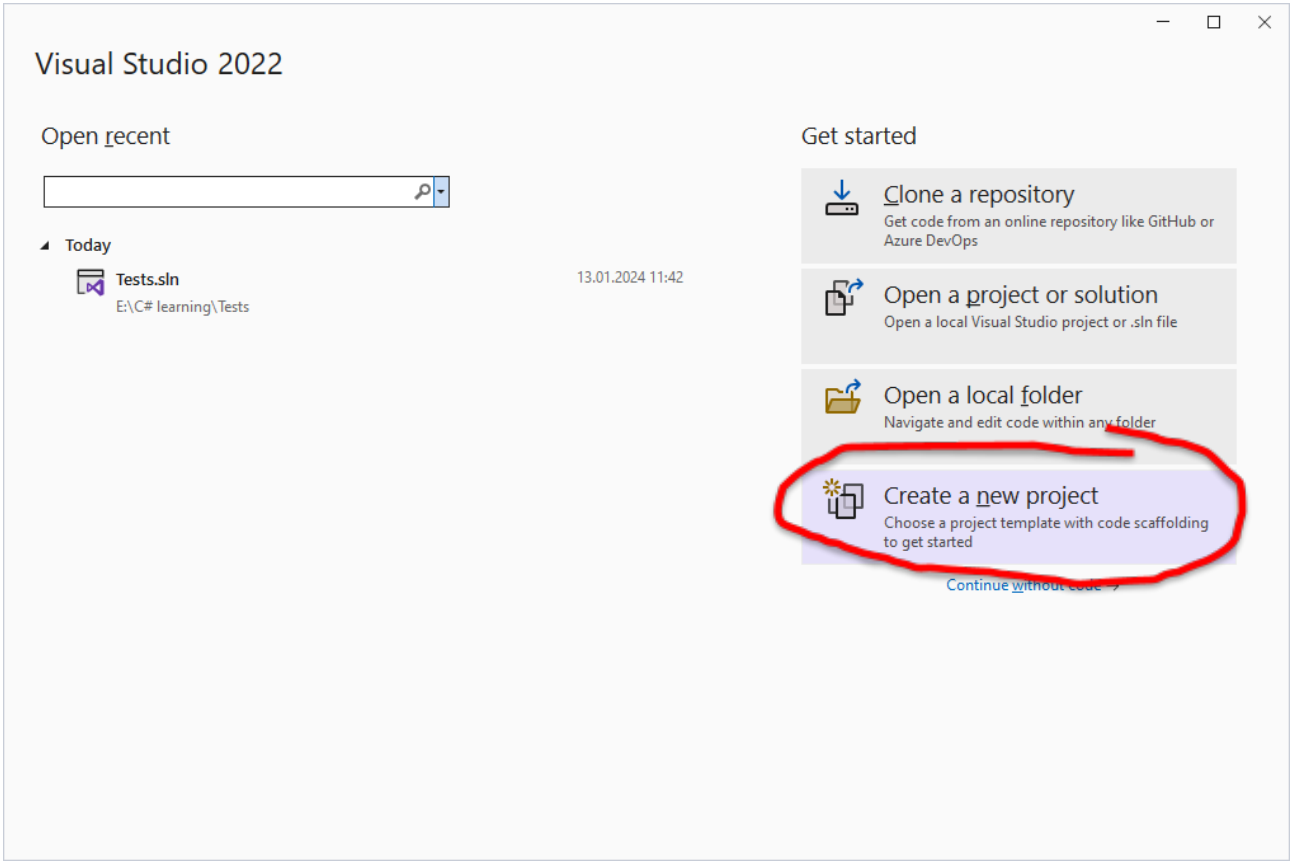


Якщо Ви вже встановили Visual Studio раніше, запустіть Visual Studio Installer із меню «Пуск» і у вікні інсталятора клацніть по кнопці «Modify».На екран буде виведено вікно вибору компонентів для установки.

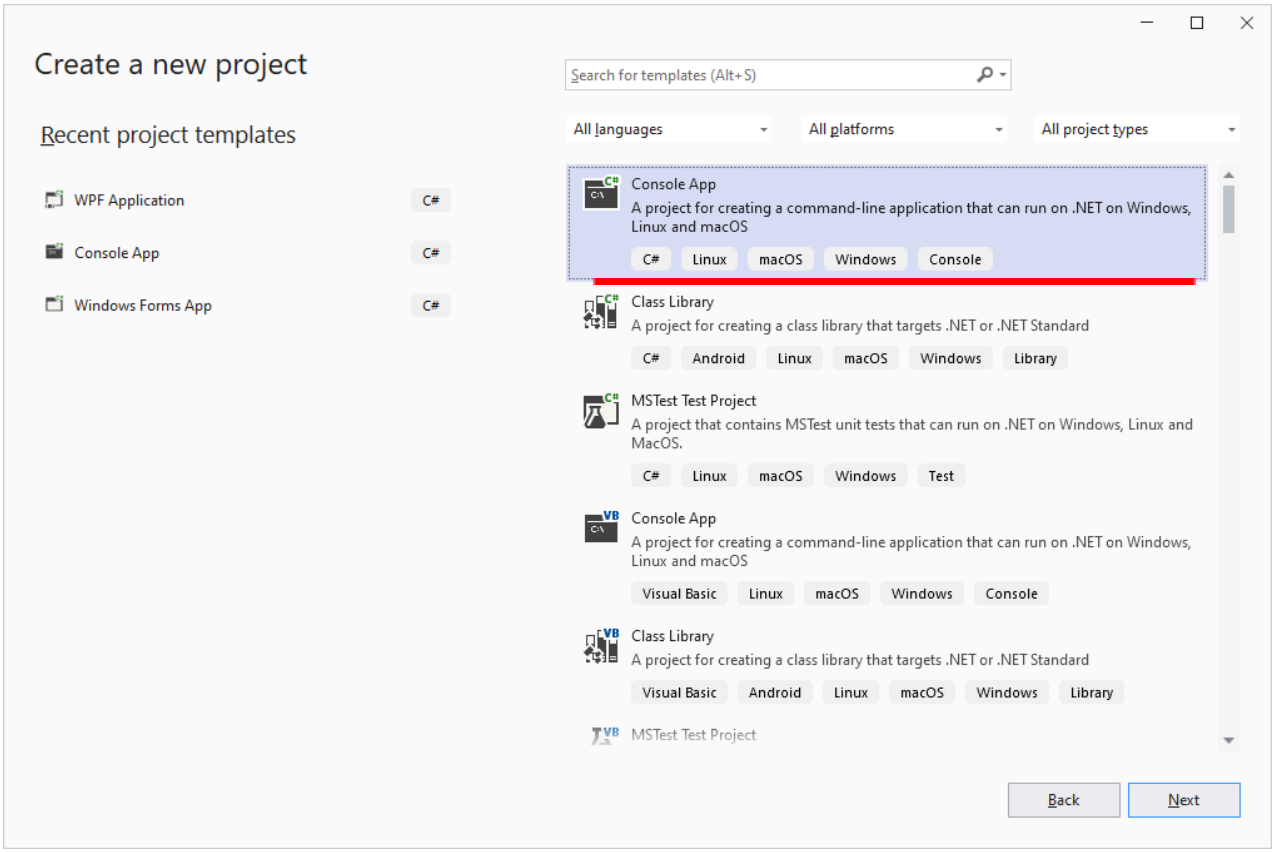


Виберіть у вікні вибору компонентів для установки «.NET desktop development» (це мінімально необхідний набір для програмування на C# під .NET) і клацніть по кнопці «Install»/ «Modify». Далі слідуйте за вказівками інсталятора.

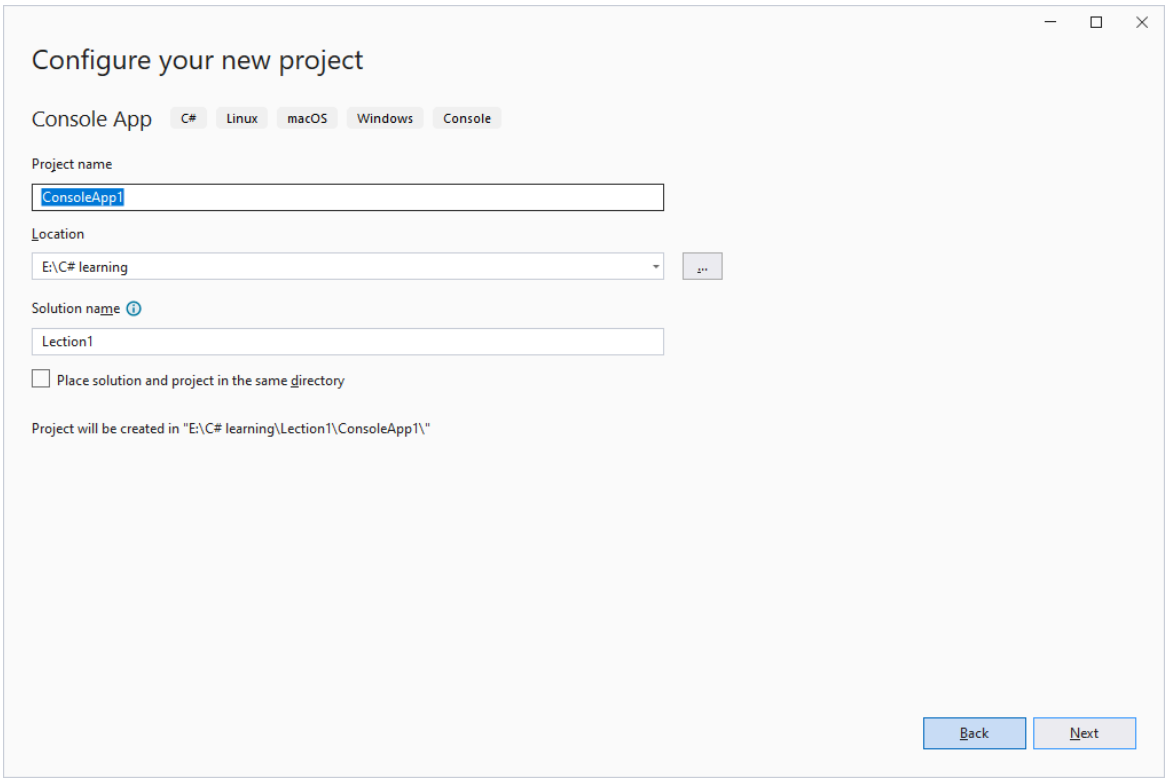
Після завершення інсталяції запустіть Visual Studio 2022 через меню «Пуск». У стартовому вікні виберіть кнопку «Create a new project».



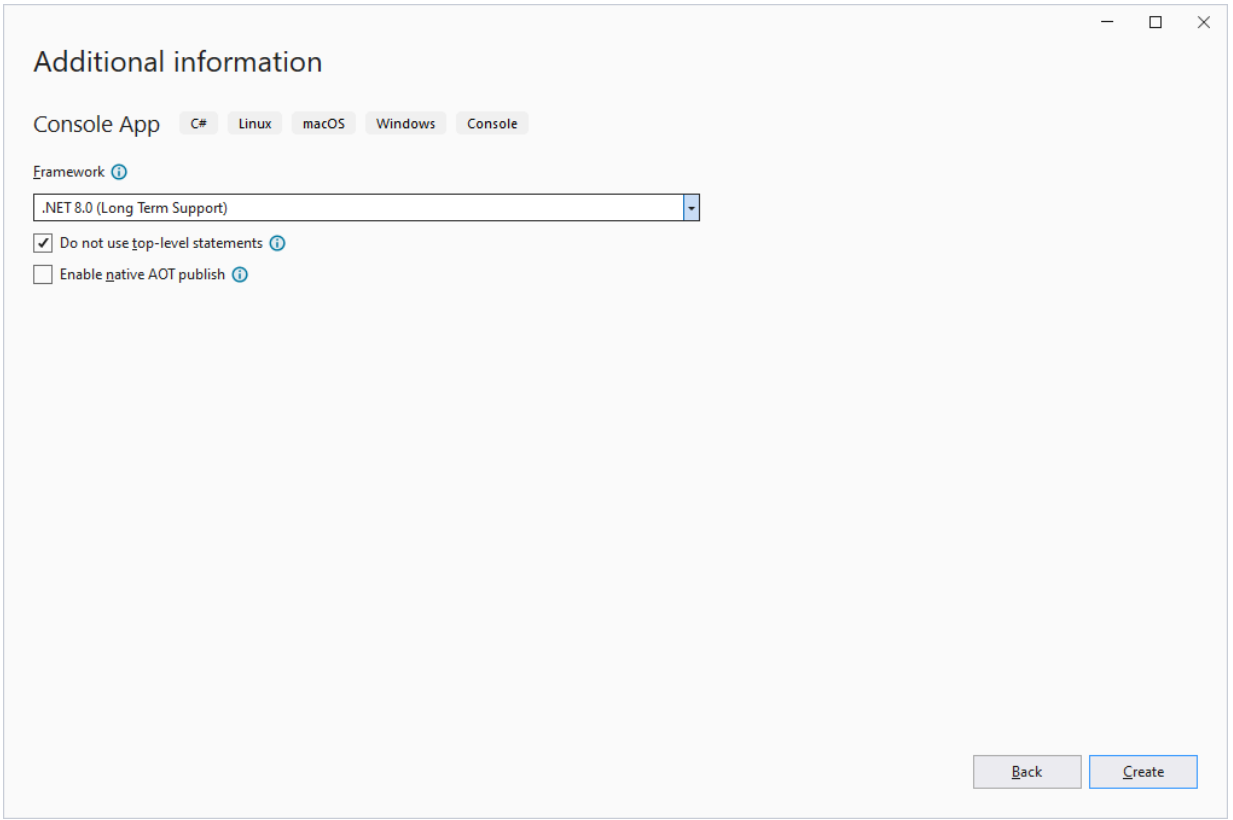
У наступному вікні виберіть шаблон Console App і клацніть по кнопці «Next».



Далі введіть назву проекту, папку для збереження та назву рішення і клацніть по кнопці «Next».



У наступному вікні виберіть у полі «Framework» .NET 8.0 , встановіть прапорець «Do not use top-level statements» і клацніть по кнопці «Create».



На екран буде виведено головне вікно Visual Studio. Зліва знаходиться текстовий редактор, в якому набирають текст програми, справа – Solution

Explorer, що показує структуру проекту. Під вікном редактора розміщено вікно Output, в яке видає повідомлення компілятор; під Solution Explorer – вікно Properties, про яке поговоримо при створенні графічного інтерфейсу.

Головне меню середовища містить такі пункти (вказано команди меню, які найчастіше використовують):

File – робота з файлами, зокрема створення нових проектів і файлів («File\New\Project», «File\New\File»), завантаження проекту («File\Open»), збереження поточного файлу («File\Save») і всіх змінених файлів у редакторі («File\Save all»), друк файлів («File\Print»), а також доступ до останніх завантажених файлів і проектів («File\Recent Files», «File\Recent Projects and Solutions»).

Edit – команди редактора, включаючи пошук і заміну: («Edit\Find and Replace\Quick Find» (Ctrl+F) – швидкий пошук, «Edit\Find and Replace\Quick Replace» (Ctrl+H) – швидка заміна, «Edit\Find and Replace\Find in Files» (Ctrl-Shift-F) – пошук у файлах, «Edit\Find and Replace\ Replace in Files» (Ctrl-Shift-H) – заміна у файлах), рефакторінг (підменю «Refactor») – автоматизована зміна назв, виділення фрагментів коду в методи і подібні зміни тексту програми без зміни її функціоналу.

View – команди відображення вікон, зокрема вікна коду («View\Code» (F7)).

Git – робота з GitHub;

Project – команди роботи з проектом: «Project\Add Module» – додавання до проекту нового модуля, «Project\Add Class» – додавання до проекту нового класу, «Project\Class Wizard» – додавання до проекту нового класу за допомогою майстра. Також підменю містить пункт «Project\Set as Startup Project» для встановлення поточного проекту як такого, що буде запускатися для налагодження (це актуально, коли у Вас у рішенні є кілька проектів), «Project\Manage NuGet Packages» для підключення до проекту пакетів NuGet, і, нарешті, пункт «Project\<ім’я проекту> Properties» – який викликає вікно властивостей поточного проекту.

Build – команди для компіляції та збирання проекту, найголовніші з яких: «Build\ Build Solution» (Ctrl-Shift-B) – збирає (повністю перекомпілює) всі проекти рішення; «Build\ Clean Solution» – очищує всі проекти рішення; «Build\ Build <ім’я проекту>» (Ctrl-B) – збирає (повністю перекомпілює) поточний проект; «Build\ Compile» (F7) – перекомпілює проект (відкомпільовані будуть лише ті файли, які змінилися з часу останньої компіляції).

Debug – команди для налагодження програми: підменю «Debug\Windows» – дає змогу викликати вікна точок зупинки «Breakpoints» та інші вікна, необхідні для налагодження програм; «Debug\Start Debugging» (F5) запускає програму для налагодження; «Debug\Start Without Debugging» (Ctrl-F5) запускає програму без налагодження; «Debug\Step Into» (F11) – робить крок у програмі із заходом у функції та методи; «Debug\Step Over» (F10) – робить крок у програмі без заходу у функції та методи; «Debug\Breakpoint» (F9) – встановлює або

знімає точку зупинки на поточному рядку програми; «Debug\Options» – викликає вікно налаштувань налагоджувача.

Test – дає змогу автоматизувати тестування проектів. Складання тестів для автоматизованої перевірки програм не входить до даного курсу.

Analyze – містить команди аналізу коду проекту, за результатами якого програміст може спростити й оптимізувати програми. Для малих проектів не використовується.

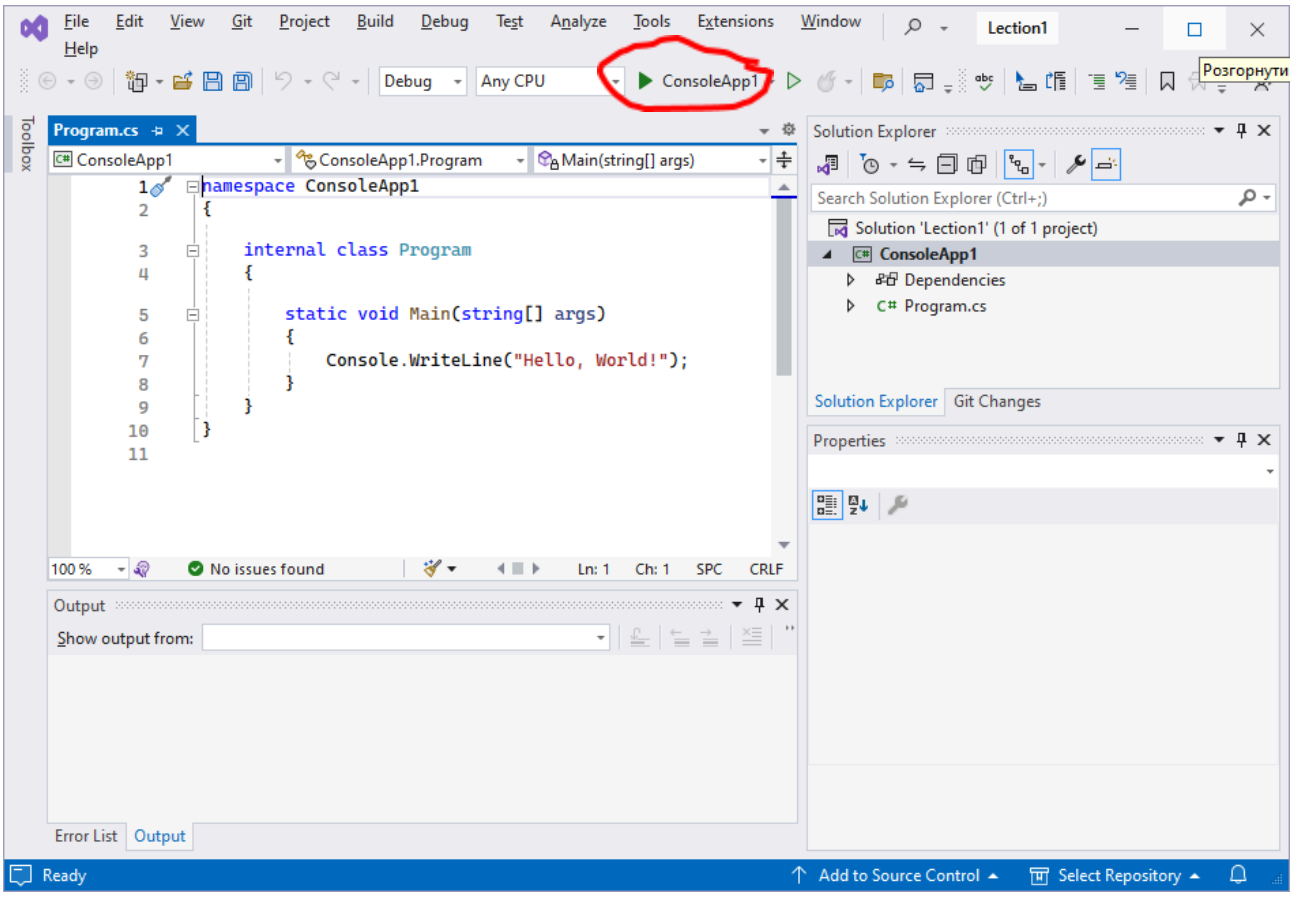
Tools – містить команди меню для роботи із допоміжними програмами, що спрощують розробку. Наприклад, «Tools\Connect to database» викликає майстер підключення до бази даних, «Tools\Spy++» – викликає утиліту, яка відображає вікна, процеси і потоки в системі, «Tools\Customize…» дає змогу вибрати видимі панелі інструментів і налаштувати їх, а пункт меню «Tools\Options…» викликає вікно глобальних налаштувань середовища розробки.

Extensions – слугує для встановлення, видалення та виклику додатків (розширень) Visual Studio. Такі додатки спрощують роботу із середовищем розробки, наприклад, спрощують текст програми, роботу з БД, зокрема із SQL Lite/SQL server та ін.

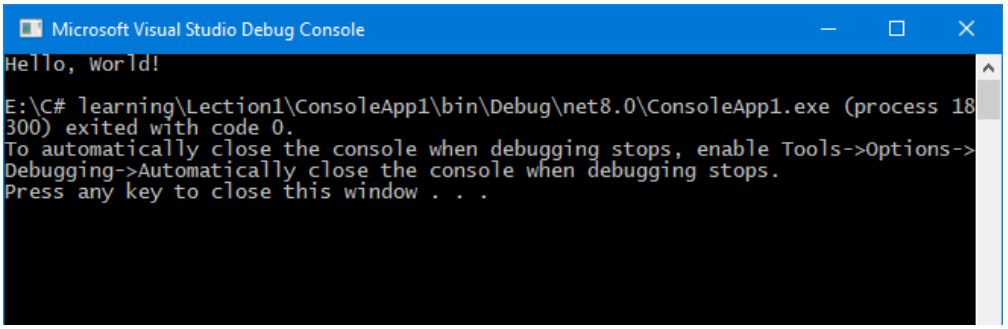
Window – містить команди роботи із вікнами середовища, зокрема створення нового вікна та впорядкування вікон.

Help – отримання доступу до системи допомоги. У ранніх версіях Visual Studio ця система встановлювалася на комп’ютер користувача, в останніх – допомога за замовчуванням працює в онлайн-режимі та потребує доступу до Internet (сайт https://learn.microsoft.com/). Також меню «Help» традиційно містить команду «About Microsoft Visual Studio», яка виводить на екран вікно з інформацією про Visual Studio.

Для запуску програми на виконання виберіть команду меню «Debug\Start Debugging» або клацніть по кнопці, виділеній на рисунку.



Програма запуститься і виведе у вікно консолі Hello, world!. Для закриття консолі натисніть будь-яку клавішу.



Для збереження змін використовуйте пункт меню «File\Save» або «File\Save all».

1.3 Структура першої програми. Пакети NuGet. Пошук і додавання потрібних пакетів з NuGet до проекту. Поняття про простори імен.

Подивимося уважно на програму:

```
namespace ConsoleApp1
{
    internal class Program
    {
        static void Main(string[] args)
```

```
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

Перший рядок *namespace ConsoleApp1* оголошує **простір імен** поточної програми. Простори імен в C# відіграють таку ж роль, як і модулі в Pascal або файли заголовків у C/C++. Вони слугують для того, щоб оголосити область дії, що містить набір пов'язаних об'єктів (тобто, типів даних, класів та ін.). Усі класи стандартної бібліотеки розміщено по просторам імен відповідно до їх призначення.

За замовчуванням проект використовує такі простори імен:

System – системні класи, зокрема Math (математичні функції);

System.IO – класи введення-виведення, як через консоль, так і в файли;

System.Threading – класи для створення багато-потокових додатків.

Далі оголошується клас Program, який власне представляє саму програму, і містить єдиний метод (функцію, яка є членом класу Program) Main. Виконання програми на C# починається завжди із виконання методу Main, за аналогією з C/C++. У методі Main виконано виклик методу виведення на консоль Console.WriteLine, який виводить на екран рядок "Hello world!".

Детально простори імен і введення-виведення в C# буде розглянуто в наступних лекціях.

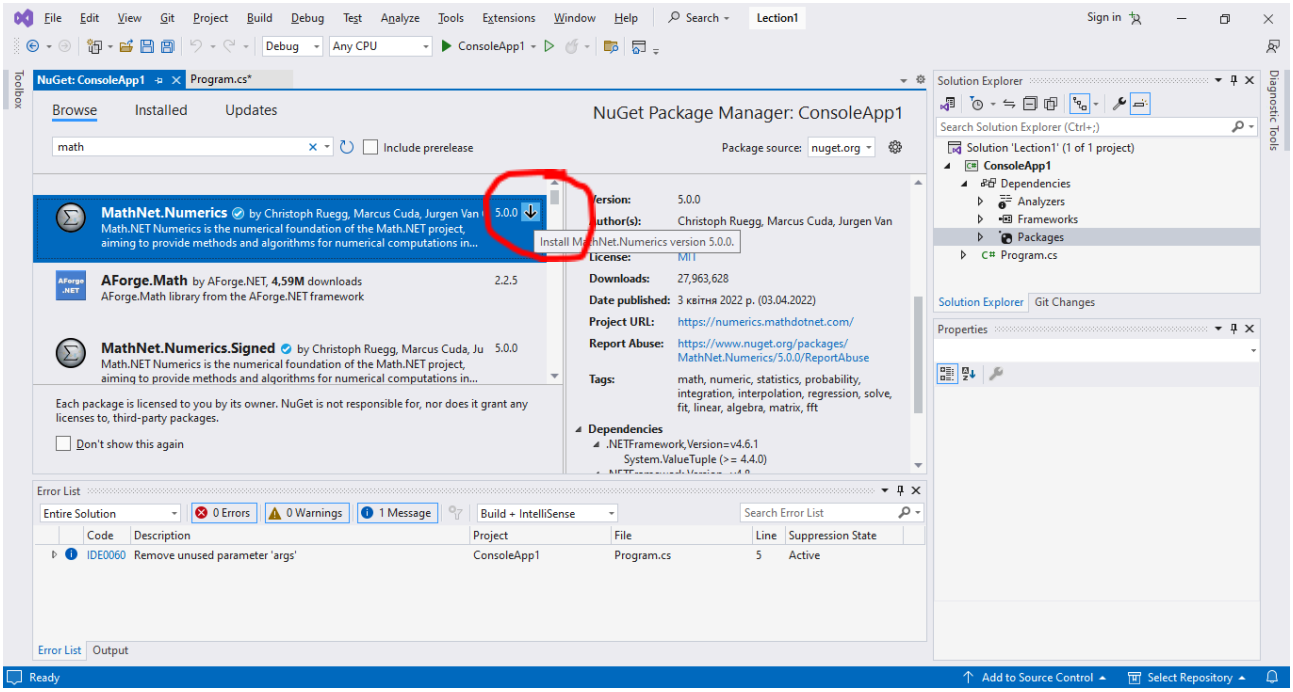
Примітка: Якщо при створенні проекту прапорець «Do not use top-level statements» не буде встановлено, то файл Program.cs міститиме один рядок коду після коментаря:

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

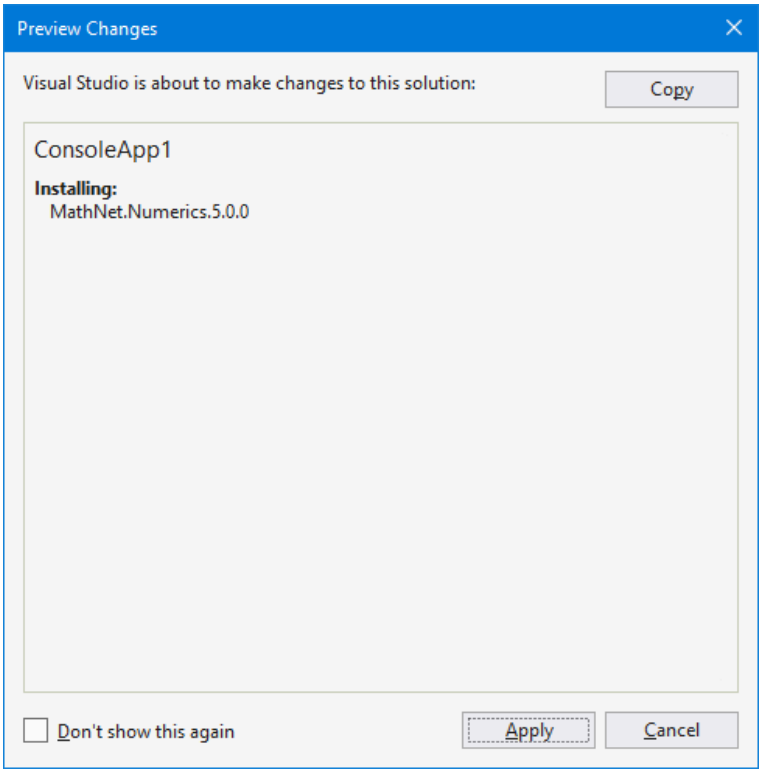
У цьому випадку всі описи класу програми, простору імен та ін. будуть виконані компілятором неявно. Функціонал програми при цьому не зміниться.

Якщо для написання програми не вистачає функціоналу стандартної бібліотеки класів, є можливість скористатися онлайн-бібліотекою пакетів NuGet, яка містить тисячі класів, розроблених іншими розробниками.

Для цього в меню Visual Studio слід вибрати пункт «Project\Manage NuGet packages». У головному вікні з'явиться вкладка «NuGet : ConsoleApp1», в якій можна задати у рядку пошуку назву пакету або ключове слово (наприклад, Math). Вибравши пакет зі списку і клацнувши по ньому, справа від списку можна подивитися інформацію про пакет, у тому числі ліцензію, сайт проекту пакету та інші дані. Для підключення пакету до проекту слід клацнути по стрілочці вниз.



Visual Studio виведе на екран вікно, в якому слід клацнути по кнопці «Apply» для встановлення і підключення пакета до проекту.



Деталі використання пакету описуються на його сайті, наприклад, для пакету MathNet.Numerics5.0.0 – на сайті https://numerics.mathdotnet.com.

Для використання функціоналу пакету слід додати на початку .cs файлу програми відповідний рядок підключення простору імен пакету, в даному випадку *using MathNet.Numerics*.

Програма, що обчислює значення функції Бесселя 0 в точці 0.5, і виводить його на екран, виглядає так:

```
using MathNet.Numerics;

namespace ConsoleApp1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(SpecialFunctions.BesselI0(0.5));
        }
    }
}
```

Запитання для самоперевірки

1. Коли і навіщо була створена Microsoft .NET?
2. Які основні переваги Microsoft .NET?
3. Що таке керований код і чим він відрізняється від класичного коду?
4. Що таке JIT компіляція?
5. Які основні недоліки Microsoft .NET?
6. Як створити найпростішу програму на C#?
7. Як встановити пакети NuGet?

Питання з теми, що виносяться на самостійне опрацювання

1. Меню та команди меню і панелей інструментів Microsoft Visual Studio.

Лекція 2. Структура програми. Змінні, константи і літерали. Базові типи даних та їх перетворення. Операції. Пріоритет та асоціативність операторів. Консольне введення-виведення у C#.

2.1 Структура програми.

Як і в інших мовах програмування, текст програми C# міститься в файлі. Розширення файлу із текстом програми на C# – .cs. Програма – послідовність інструкцій (statement), які вказують, що робити комп'ютеру. Інструкції в C# розділяються крапкою з комою (;). Набір інструкцій може поєднуватися в блок коду, який виділяється фігурними дужками, як у C/C++:

```
{
   <інструкція 1>;
       ...
   <інструкція 1>;
}
```

Як і в C/C++, блок коду при використанні в операторах вважається за одну інструкцію. Блоки коду можуть бути вкладеними.

Компілятор C# ігнорує символи пробілу, табуляцію і коментарі. Коментарі в C# пишуть так само, як і в C++.

Коментар до кінця рядка починається з двох /:

```
//Це коментар до кінця поточного рядка, що починається з двох /
```

Коментар, що може займати кілька рядків, починається з /*, а закінчується */ :

```
/* Це
   коментар на кілька
   рядків
 */
```

2.2 Змінні, константи і літерали. Базові типи даних.

Змінні і константи в C# позначаються іменами-ідентифікаторами. Ці імена мають відповідати таким умовам:

1) ім'я може містити будь-які цифри, букви (справді будь-які букви – латиницю, кирилицю, букви інших алфавітів, хірагану, катакану та ієрогліфи), а також символ підкреслення, однак перший символ в імені не може бути цифрою (тобто має бути буквою або символом підкреслення);

2) ім'я не повинно містити знаків пунктуації та пропусків;

3) ім'я не може бути ключовим словом C#. Список ключових слів C# наведено тут: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ .

Мова C# успадкувала від C/C++ чутливість до регістру літер. Тому змінні з іменами Console, console та CoNsoLe – це три різних змінних.

Змінна у C# оголошується так:

```
<тип> <ім'я> [= <значення>];
```

де <значення> – значення, яким ініціалізується змінна при створенні. Якщо значення не вказано, змінна ініціалізується значенням по замовчуванню (залежно від типу, це нуль, пусте значення null та ін.).

Константа в C# оголошується так:

```
const <тип> <ім'я> = <значення>;
```

де <значення> – значення константи, яке не може бути змінено.

Значення, якими ініціалізують константи та змінні, називають літералами. Літерали бувають логічними, цілочисельними, дійсними, символьними і рядковими. Окремий літерал (пусте значення) є ключовим словом null.

Логічні літерали – це значення true і false.

Цілочисельні – цілі числа, знакові та беззнакові: 10, 0, -25. Цілочисельні літерали можуть бути також виражені у шістнадцятковій системі (починаються з 0x):

```
0x10 //16
0xA1 //161
```

та двійковій системі (починаються з 0b):

```
0b0101 //5
0b10001 //17
```

На відміну від C/C++, вісімкова система у C# не підтримується.

Дійсні літерали традиційно записуються в формі із фіксованою крапкою, наприклад 2.71, -5.25, та з плаваючою, наприклад 12e-4, -1.3e+8 (e*x* традиційно означає $\times 10^x$). Для відділення одиниць від десятих у тексті програм ЗАВЖДИ використовується крапка.

Символьні літерали – одиночні символи, виділяються знаком апострофа, наприклад 'v', 'Ю'. Спеціальним випадком є керуючі послідовності, котрі, як і в C/C++, починаються з символу слеш (\). Нижче наведено керуючі послідовності, які використовуються при виведенні в консоль:

'\n' – новий рядок,

'\r' – повернення курсору на початок рядка,

'\t' – табуляція,

'\a' – звуковий сигнал (bell),

'\b' – перемістити курсор на символ вліво і стерти символ, якщо він існував (backspace),

'\v' – вертикальна табуляція, як правило, еквівалентна переходу на новий рядок,

'\\' – слеш.

Також символи можна задати їх кодом у шістнадцятковій системі, наприклад '\x59' (символ 'Y') або кодом Unicode '\u0422' (символ 'Т').

Рядкові літерали – це рядки, які подаються в подвійних лапках. Якщо в рядок необхідно включити символ лапок, його записують як послідовність \" :

```
"Введіть число"
"Вілла \"Смерека\""
```

Розрив рядкового літерала за зразком C++

```
"Це один розірваний \
рядок" //У C# це не працює!
```

у C#не працює. Його слід прописувати так:

```
"Це один розірваний "+
"рядок" //У C# треба так!
```

У C# існують перераховані нижче базові типи даних, тісно пов'язані з системними типами .NET.

Логічний тип bool. Зберігає значення true або false, займає 1 байт. Системний тип .NET – System.Boolean.

Цілочисельні типи. У C# існують перераховані нижче цілочисельні типи даних – розміром від 1 (8 біт) до 8 байт (64 біти). На відміну від C/C++, де розміри типів і діапазони значень залежать від реалізації мови, у C# вони чітко визначені.

Тип	Пам'ять	Діапазон значень	Системний тип .NET
byte	1 байт	0..255	System.Byte
sbyte	1 байт	-127..128	System.SByte
short	2 байти	-32767…32768	System.Int16
ushort	2 байти	0..65536	System.uInt16
int	4 байти	-2147483648...2147483647	System.Int32
uint	4 байти	0.. 4294967295	System.uInt32
long	8 байт	-9223372036854775808 … 9223372036854775807	System.Int64
ulong	8 байт	0 … 18446744073709551615	System.uInt64

Усі цілі літерали за замовчуванням представляють значення типу int. Щоб явно вказати, що цілий літерал представляє значення типу uint, треба використовувати суфікс U/u, для типу long – суфікс L/l, а для типу ulong – суфікс UL/ul:

```
long a = 10;
uint b = 10U;
ulong c = 10UL;
```

Дійсні типи. C# підтримує два базових дійсних типи, що відповідають вимогам IEEE 754 – зі звичайною точністю (float) та подвійною (double). На

відміну від C++, типи float і double ніколи не еквівалентні (і це не залежить від реалізації).

Тип	Пам'ять	Min/Max значення	Системний тип .NET
float	4 байти	1.175494351 E – 38 3.402823466 E + 38	System.Single
double	8 байт	2.2250738585072014 E – 308 1.7976931348623158 E + 308	System.Double

Тип float гарантує при обчисленнях точність 5-6 знаків після коми, тип double – 10-12 знаків після коми.

Крім базових типів, починаючи з .NET 5, в C# з'явився також тип Half – 16-бітне число з плаваючою комою (float16 за IEEE 754). На практиці цей тип використовується при роботі з обчисленнями на процесорах графічних карт. 128-бітну арифметику з плаваючою комою (чотирикратну точність) C# не підтримує без використання сторонніх бібліотек, які, як правило, емулюють всі операції над такими числами і тому повільні.

Тип decimal є чимось середнім між цілим і дійсним типами. Тип decimal зберігає десяткове дробове число. Якщо використовується без десяткової коми, має значення від $\pm 1.0*10^{-28}$ до $\pm 7.9228*10^{28}$, може зберігати 28 знаків після коми і займає 16 байт. Системний тип .NET – System.Decimal. Цей тип використовується при фінансових обчисленнях.

Всі дійсні літерали за замовчуванням представляють значення типу double. Щоб вказати, що дробове число представляє тип float або тип decimal, необхідно до літералу додавати суфікс: F/f – для float і M/m – для decimal:

```
float d = 10.4f;
decimal e = 12M;
```

Тип char відповідає системному типу System.Char і містить один символ Unicode. Займає 2 байти.

Тип string відповідає системному типу System.String і містить рядок символів Unicode.

Тип object відповідає системному типу System.Object і займає 4 байти на 32-бітній платформі і 8 байт на 64-бітній. Цей тип є базовим для всіх типів .NET, і в цьому сенсі є аналогом вказівника на void у C++.

Для змінних у C# можливе використання неявної типізації, коли тип змінної визначається компілятором за типом значення виразу – ініціалізатора. Оголошення змінної за допомогою неявної типізації виглядає так:

```
var <ім'я> = <значення>;
```

Наприклад, фрагмент коду

```
int i = 10; //Ця змінна має тип int
var j = 10; //Ця змінна теж має тип int
```

визначає дві змінні типу int. У випадку неявної типізації не можна використовувати літерал null – у цьому випадку компілятор не знає, якого типу повинна бути змінна.

Розмір змінної (і типу) в байтах можна визначити за допомогою оператора **sizeof**. Наприклад, виклик sizeof(int) поверне 4.

2.3 Арифметичні операції. Побітові операції. Операції присвоювання.

Всі арифметичні операції в C# перейшли із C++. Стандартні арифметичні операції – бінарні + і – (сума і різниця двох чисел), множення і ділення виконуються за тими ж правилами, що в C++. Якщо в операції хоча б один з аргументів – дійсний, результат буде дійсним. Якщо обидва аргументи цілі – результат цілий, тому результатом ділення 10.0/4 буде 2.5, а 10/2 в результаті дасть 2. Операція остачі від ділення %, на відміну від C++ працює як із цілими, так і з дійсними числами; наприклад 10.0 % 4.2 в остачі дасть 1.6.

Унарний + (знак операнду залишається без змін) і – (зміна знаку операнду) порівняно з C++ змін не зазнали. Так само як в C++ працюють постфіксний і префіксний інкремент ++ і декремент --. Постфіксна операція повертає значення операнду, а потім його збільшує (чи зменшує) на 1, префіксна – спочатку змінює значення операнду, потім повертає його. Приклад:

```
int i = 10;
int j;

j = i++; //j==10, i=11
j = ++i; //j==12, i=12
```

Побітові операції в C# виконуються над цілими числами, і нічим не відрізняються від C++:

a & b – побітове AND (І) чисел a і b;

a | b – побітове OR (АБО) чисел a і b;

a ^ b – побітове XOR (виключаюче АБО) чисел a і b;

~ a – побітове NOT (інверсія) числа a;

a << b – зсув числа a на b біт вліво (рівноцінно множенню a на 2^b);

a >> b – зсув числа a на b біт вправо (рівноцінно діленню a на 2^b).

Операції присвоєння встановлюють значення. В операції беруть участь два операнди, при цьому лівий операнд має бути lvalue – модифікованим іменованим виразом, наприклад змінною або полем об’єкта (простіше кажучи, повинен бути тим, чому можна присвоїти значення).

Найпростіша операція присвоювання, як і в C++, задається знаком =. Можливе як одинарне, так і множинне присвоювання (як у рядку x=y=z=2.5):

```
int n, i;
double x, y, z;

n = 5;
x = y = z = 2.5; //Всі три змінні рівні 2.5
i = n * n; //i=25
```

Як і в C++, оператор присвоювання може містити арифметичні та побітові операції, що скорочує запис операцій:

a+=b; еквівалентно a=a+b;
a-=b; еквівалентно a=a-b;
a*=b; еквівалентно a=a*b;
a/=b; еквівалентно a=a/b;
a&=b; еквівалентно a=a&b;
a|=b; еквівалентно a=a|b;
a^=b; еквівалентно a=a^b;
a>>=b; еквівалентно a=a>>b;
a<<=b; еквівалентно a=a<<b.

Операції присвоювання в C# мають найнижчий пріоритет.

2.4 Перетворення базових типів даних. Явні і неявні перетворення типів в C#.

Як вже було сказано вище, за замовчуванням усі цілі літерали мають тип int. Тому наведений нижче код,

```
byte a = 4;
byte b = a + 10;
int c = a;
```

на відміну від C++, не відкомпілюється через те, що результат обчислення виразу a+10 матиме тип int, а змінна b – тип byte. Для вирішення цієї проблеми слід застосувати явне приведення типів (<тип>) <вираз>:

```
byte a = 4;
byte b = (byte) (a + 10);
int c = a;
```

Перетворення типу бувають звужуючі (narrowing) та розширюючі (widening). Розширюючі перетворення розширюють об’єм об’єкта пам’яті, наприклад, присвоєння значення змінної a змінній c є розширюючим перетворенням, адже збільшує розрядність із 1 байту до 4. Приведення (byte) (a + 10) приводить значення результату обчислень з типу, що займає 4 байти, до типу, що займає 1, і тому є звужуючим перетворенням. Зверніть увагу на те, що у випадку розширюючого перетворення компілятор виконує його за програміста, таким чином, перетворення є неявним.

Неявне перетворення виконується з урахуванням знаковості числа. Якщо відбувається перетворення від беззнакового типу меншої розрядності до беззнакового типу більшої розрядності, додаються додаткові біти, які мають значення 0. Якщо відбувається перетворення до знакового типу, бітове представлення доповнюється нулями якщо число додатне і одиницями, якщо число від’ємне (при розширенні в додані розряди копіюється знаковий біт).

Усі безпечні автоматичні неявні перетворення можна описати так:

- тип byte безпечно перетворюється на short, ushort, int, uint, long, ulong, float, double, decimal;

- тип sbyte безпечно перетворюється на short, int, long, float, double, decimal;
- тип short безпечно перетворюється на int, long, float, double, decimal;
- тип ushort безпечно перетворюється в на int, uint, long, ulong, float, double, decimal;
- тип int безпечно перетворюється на long, float, double, decimal;
- тип uint безпечно перетворюється на long, ulong, float, double, decimal;
- тип long безпечно перетворюється на float, double, decimal;
- тип ulong безпечно перетворюється на float, double, decimal;
- тип float безпечно перетворюється на double;
- тип char безпечно перетворюється на ushort, int, uint, long, ulong, float, double, decimal.

В інших випадках необхідно використовувати явне приведення типу.

2.5 Умовні вирази і тип bool.

Для керування логікою роботи програми часто необхідна перевірка умов.

В мові С логічний тип взагалі був відсутнім, і умови перевірялися шляхом обчислення цілочисельного значення виразу. Якщо значення виразу було рівне 0, умова – не істинна, якщо не 0 – істинна. У С++ з'явився тип bool, однак для сумісності з С обчислення логічного значення як 0=false/не 0 =true залишилось, тому код типу

```
while (1) {
 //тут якісь оператори
}
```

компілювався і працював. У С# позбавилися цього анахронізму, і в усіх місцях, де потрібне використання логічних умов, використовується лише тип bool.

В операціях порівняння С# порівнюються два операнди і повертається значення типу bool – true, якщо вираз вірний, і false, якщо вираз невірний:

== Перевіряє два операнди на рівність. Якщо вони рівні, операція повертає true, якщо не рівні, то повертається false.

!= Порівнює два операнди і повертає true, якщо операнди не рівні, і false, якщо вони рівні.

< Операція «менше ніж». Повертає true, якщо перший операнд менший за другий, і false, якщо перший операнд більший за другий.

> Операція «більше ніж». Порівнює два операнди і повертає true, якщо перший операнд більший за другий, інакше повертає false.

<= Операція «менше або дорівнює». Порівнює два операнди і повертає true, якщо перший операнд менше або дорівнює другому. Інакше повертає false.

>= Операція «більше або дорівнює». Порівнює два операнди і повертає true, якщо перший операнд більше або дорівнює другому, інакше повертається false.

Для всіх цілих та дійсних типів операції порівняння виконуються за правилами арифметики. Тип char порівнюється за значенням кодів, тобто

забезпечується порівняння цифр за правилами арифметики, латинських і кириличних букв за алфавітом, причому малі букви більше великих.

Для рядків (string) порівняння за допомогою операторів порівняння передбачає лише рівність (==) або нерівність (!=). Решту порівнянь реалізовано через методи класу System.String.

Над двома виразами типу bool також передбачено виконання логічних операцій:

a && b – логічне AND (І) виразів a і b;

a || b – логічне OR (АБО) виразів a і b;

!a – логічне NOT.

2.6 Пріоритет та асоціативність операторів.

Пріоритет розглянутих в лекції операторів зведено в таблицю. Спершу виконуються оператори з вищим пріоритетом, потім – із нижчим; пріоритет зменшується до низу таблиці.

Оператор	Категорія
x.y, f(x), a[i], x?.y, x?[y], x++, x--, sizeof	Основні
+x, -x, !x, ~x, ++x, --x, , (T)x, true and false	Унарні
x * y, x / y, x % y	Множення/ділення
x + y, x – y	Додавання/віднімання
x << y, x >> y, x >>> y	Зсув
x < y, x > y, x <= y, x >= y, is, as	Відношення
x == y, x != y	Рівність
x & y	Побітове І
x ^ y	Побітове виключаюче АБО
x | y	Побітове АБО
x && y	Логічне І
x || y	Логічне АБО
x ?? y	Оператор об’єднання з null
c ? t : f	Тернарний оператор (див. лекцію 3)
x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &= y, x |= y, x ^= y, x <<= y, x >>= y	Присвоювання

Повну таблицю пріоритету операторів C# наведено тут: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/#operator-associativity.

Більшість операторів C# з однаковим пріоритетом виконуються зліва направо. Винятками є оператори присвоювання, тернарний оператор та оператори об’єднання з null (оператори ?? та ??=).

Для зміни порядку виконання операторів у виразі використовують дужки. Так, якщо необхідно обчислити значення виразу $x = \frac{a \cdot b}{c \cdot d}$, правильний запис на C# матиме такий вигляд:

```
X=a*b/(c*d);
```

оскільки операції * і / мають однаковий пріоритет.

2.7 Консольне введення-виведення у C#.

Розглянемо таку програму.

```
namespace ConsoleApp1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.OutputEncoding = System.Text.Encoding.Unicode;
            Console.InputEncoding = System.Text.Encoding.Unicode;
            Console.WriteLine("Введіть чило:")
            string? st = Console.ReadLine();
            int n = Convert.ToInt32(st);
            Console.WriteLine("Квадрат числа {0}={1}",n,n*n);
        }
    }
}
```

Наведена вище програма вводить з клавіатури число і виводить його квадрат. Рядок Console.OutputEncoding = System.Text.Encoding.Unicode; встановлює кодування консолі Unicode (щоб запобігти проблемам із виведенням в консолі українських букв і та І). Для встановлення кодування консолі для введення використовується рядок Console.InputEncoding = System.Text.Encoding.Unicode;.

Для введення даних використовується метод Console.ReadLine, який вичитує з консолі і повертає рядок (тип string). Знак ? після string означає, що st може містити і рядок, і null (пусте значення). Рядок int n = Convert.ToInt32(st); перетворює введений рядок на ціле число і присвоює його значення змінній n. Введення значень double виконується аналогічно, викликом Convert.ToDouble.

Ще один варіант введення цілих і дійсних чисел із консолі – використання методів <тип>.Parse(string st) і <тип>.TryParse(string st, <тип> d):

```
//Якщо введено не число - буде помилка виконання
int i2 = int.Parse(Console.ReadLine());

//Якщо введено не число - TryParse повертає false
double dd;
if (!double.TryParse(Console.ReadLine(),dd))
    Console.WriteLine("Введено не дійсне число!");
```

Метод Parse сканує рядок і перетворює його, якщо це можливо, на число відповідного типу. У випадку, коли перетворення не можливе, результатом роботи Parse буде помилка виконання (а точніше – виключна ситуація) і програма звершиться.

Метод TryParse сканує рядок і повертає результат перетворення в параметрі d. У випадку, коли перетворення не можливе, результатом методу

буде false, інакше – true. При цьому не виникає виключних ситуацій і тому рекомендовано використовувати TryParse.

Нарешті, для виведення числа і квадрата використано Console.WriteLine з трьома аргументами. Перший аргумент – рядок, в якому у фігурних дужках вказано номери аргументів після рядка, у тих місцях, де їх слід вивести (плейсхолдери). Нумерація аргументів починається з нуля і виконується зліва направо. Два наступних за рядком аргументи – вирази, значення яких необхідно вивести на екран.

Якщо необхідно вивести значення кількох змінних, то можна скористатися так званою інтерполяцією:

```
string name = "Іван";
int age = 19;
Console.WriteLine($"Ім'я: {name} Вік: {age}");
```

У даному випадку виклику Console.WriteLine в рядку вказано імена змінних, а перед початком рядку – знак $. В результаті при введенні замість name та age буде підставлено їх значення, і на екран буде виведено «Ім’я: Іван Вік: 19».

У фігурних дужках при інтерполяції можна також використовувати складні вирази, наприклад,

```
Console.WriteLine($"{n*n}");
```

виведе квадрат числа n.

Крім Console.WriteLine, можна використовувати метод Console.Write, який працює так само, але після виведення не переходить на наступний рядок консолі.

Запитання для самоперевірки

1. Якою є структура програми на C#?
2. Як в C# створюються змінні і константи? Які в C# можуть бути літерали?
3. Які цілочисельні і дійсні типи існують в C#?
4. Яке призначення типів bool, char, string, object?
5. Як визначити розмір змінної і типу?
6. Які існують арифметичні операції в C#? Які існують побітові операції в C#?
7. Які існують операції присвоювання C#?
8. Які в C# є перетворення базових типів даних? В чому їх особливості?
9. Як використовувати тип bool і умовні вирази в C#?
10. Які пріоритет та асоціативність основних операторів C#?
11. Як виконується консольне введення-виведення даних в C#?

Питання з теми, що виносяться на самостійне опрацювання

1. Пріоритет операторів C#.

2. Консольне введення-виведення даних: особливості роботи з кирилицею в Unicode.

Лекція 3. Умовні конструкції. Цикли. Рядки і символи в C#. Операції над рядками. Перелічувані типи (enum).

3.1 Умовні конструкції (оператори if, switch, тернарний)

Для керування ходом виконання програми в C#, як і в C++, передбачені умовні оператори та оператори циклів. Найпростішим оператором для керування ходом виконання програми є **оператор if**:

```
if (<умова>)
   <інструкція 1>;
```

Якщо значення виразу <умова>, наведеного в дужках після if, рівне true, то виконується <інструкція 1>, інакше <інструкція 1> ігнорується, і виконується наступна за оператором інструкція. <Інструкція 1> може бути як однією інструкцією, наприклад присвоєнням значення змінній, так і блоком, що містить кілька інструкцій. Можливе також застосування оператора if у формі

```
if (<умова>)
   <інструкція 1>;
    else
   <інструкція 2>;
```

Якщо значення виразу <умова>, наведеного в дужках після if, рівна true, то виконується <інструкція 1>, інакше <інструкція 2>. Приклад:

```
Console.WriteLine("Введіть чило:");
string? st = Console.ReadLine();
double x = Convert.ToDouble(st);
if (x >= 0)
    Console.WriteLine("x={0}, sqrt(x)={1}", x, Math.Sqrt(x));
else
{ //Блок інструкцій виконується при (x>=0)==false
    Console.WriteLine("X<0, дійсного квадратного кореня немає ");
    Console.WriteLine("x={0}, sqrt(x)=1i*{1}", x, Math.Sqrt(-x));
}
```

У цьому випадку при x>=0 програма видає на екран значення квадратного кореня. Якщо (x>=0) == false (тобто x<0), програма видає повідомлення про це і виводить значення комплексного кореня від'ємного числа. Зверніть увагу на те, що одинична інструкція в if, як і в C/C++, повинна обов'язково бути закритою крапкою з комою, а блок інструкцій можна писати без «;».

Тернарний оператор дає змогу перевірити деяку умову і залежно від її істинності обчислити значення результату. Синтаксис тернарного оператора такий:

```
<умова> ? <вираз 1> : <вираз 2>
```

Якщо <умова> == true, обчислюється <вираз 1>, і отримане значення виразу 1 є результатом оператора, інакше обчислюється <вираз 2>, і результатом є його значення. Приклад:

```
double y= x < 0 ? -x : x; //Результат - модуль x
```

При виконанні оператора обчислюється лише <вираз 1> чи <вираз 2>. Результати обох виразів повинні мати однаковий тип. Якщо в інструкції з тернарним оператором використано неявну типізацію (ключове слово var), потрібне явне приведення типів <виразу 1> і <виразу 2>.

Оператор switch в C# має синтаксис:

```
switch (<вираз>)
{
    case <значення 1>: <оператори 1> break;
    case <значення 2>: <оператори 2> break;
    //.............
    case <значення N>: <оператори N> break;
    default: <код, який виконується, якщо вираз не має жодного із
значень> break;
}
```

Вираз, на відміну від C++, може бути не цілочисельним і не типу, який однозначно перетворюється на цілочисельний (як-от, наприклад, тип enum). Більше того, замість порівняння на рівність виразу із заданим константою <значенням і>, можливе порівняння результату обчислення виразу із заданою константою на відношення (на кшталт case > 2).

Як наслідок, якщо в C++ доводилося писати код для керування регулятором температури, заданої змінною temperature типу double, комбінуючи кілька операторів if так:

```
if (temperature == -273.1)
    cout <<  "Абсолютний нуль";
else if (temperature < 15.0)
{
    cout << "Температура ”  << temperature << “ низька.";
    HeaterOn = true;
    CoolerOn = false;
}
else if (temperature > 35.0)
{
    cout << "Температура ”  << temperature << “ висока.";
    HeaterOn = false;
    CoolerOn = true;
}
else if (isnan(temperature))
    cout << "Збій вимірювання";
else
{
    cout << "Температура ”  << temperature << “ в нормі.";
    HeaterOn = false;
    CoolerOn = false;
}
```

то в C# можливий такий код:

```
switch (temperature)
{
    case -273.1:
        Console.WriteLine("Абсолютний нуль");
        break;

    case < 15.0:
        Console.WriteLine($"Температура {temperature} низька.");
```

```
            HeaterOn = true;
            CoolerOn = false;
            break;

        case > 35.0:
            Console.WriteLine($"Температура {temperature} висока.");
            HeaterOn = false;
            CoolerOn = true;
            break;

        case double.NaN:
            Console.WriteLine("Збій вимірювання");
            break;

        default:
            Console.WriteLine($"Температура {temperature} в нормі.");
            HeaterOn = false;
            CoolerOn = false;
            break;
    }
```

Як видно з наведеного коду, у C# оператор switch став набагато потужніше, ніж в C++, і в більшості випадків може замінити зв'язку кількох операторів if, що значно спрощує програму і її сприйняття людиною.

Як <вираз> у switch можна використати також не чисельний тип, наприклад, рядок:

```
switch (UserName)
{
    case "Іван":
        Console.WriteLine("Адміністратор");
        break;

    case "Петро": goto case "Марія";

    case "Марія":
        Console.WriteLine("Користувач");
        break;

    default:
        Console.WriteLine("Такого користувача немає!");
        break;
}
```

Зверніть увагу на те, що у C# кожен case повинен закінчуватися оператором break, return (повернення з функції) або throw (створення виключення). «Провали» (fall-through) через case, не закриті break, return або throw, які часто призводили до помилок в C/C++, в C# недопустимі, тому код, наведений нижче, не відкомпілюється:

```
    case "Петро": //В C# це не працює

    case "Марія":
        Console.WriteLine("Користувач");
        break;
```

Для того, щоб скористатися «провалом» через case, слід закрити його оператором goto case <значення>. У наведеному нижче прикладі реалізовано «провал» через case "Петро":

```
case "Петро": goto case "Марія";

case "Марія":
    Console.WriteLine("Користувач");
    break;
```

В результаті для обох випадків («Петро» і «Марія») на екран буде виведено «Користувач».

Як і в C++, у C# в операторі switch є варіант default, інструкції якого виконуються лише якщо <вираз> у заголовку оператора не задовольняє жодному варіанту case. Цей варіант теж повинен бути закритим break, return або throw; за необхідності default можна опустити.

3.2 Цикли (do…while, while, for, foreach).

Цикли в C# аналогічні C++. **Цикл do...while** (цикл із післяумовою) виконує інструкції в тілі циклу доти, поки <умова> ==true;

```
do {
   <інструкція 1>;
       ...
   <інструкція N>;
}
while (<умова>);
```

Цикл do…while виконується один раз навіть у випадку, якщо умова ==false. Приклад (виведення на екран чисел від 1 до 10):

```
int i = 0;

do
{
    i++;
    Console.WriteLine(i);
} while (i<10);
```

Цикл while (цикл із передумовою) спочатку перевіряє істинність умови. Якщо <умова>==true, цикл виконує інструкцію доти, поки умова не стане false.

```
while (<умова>)
   <інструкція>;
```

Зрозуміло, <інструкція> (тіло циклу) може бути блоком інструкцій. Приклад (стартовий відлік при старті космічної ракети):

```
i = 10;
while (i>0)
{
    Console.WriteLine(i);
    i--;
}
Console.WriteLine("0. Пуск.");
```

Якщо <умова> рівна false, то цикл із передумовою не виконуватиметься жодного разу.

Цикл for, як і в C/C++, має синтаксис:

```
for (<вираз 1>; <умова>; <вираз 2>)
   <інструкція>;
```

і аналогічний такому фрагменту коду:

```
<вираз 1>;
while (<умова>) {
   <інструкція>;
   <вираз 2>;
}
```

Типовим використанням <виразу 1> є ініціалізація змінних, а <виразу 2> – зміна значень змінних, яка повинна відбуватися в кінці виконання кожного циклу. Цикл for виконується доти, поки <умова>==true:

```
for (int i = 1; i < 10; i++)
    Console.WriteLine(i);
```

Цей цикл знов-таки виводить на екран числа від 1 до 10. Зверніть увагу на те, що взагалі <вираз1> і <вираз 2> можуть бути довільними виразами:

```
var i = 1;
for (Console.WriteLine("Початок виконання циклу"); i < 10;
      Console.WriteLine($"i = {i}"))
    i++;
```

У даному випадку виведення чисел реалізовано у заголовку циклу, а у тілі – виконано зміну змінної-лічильника.

Нескінчений цикл на основі for записується так: for (;;) { <інструкції>; }

Зрозуміло, <інструкція> (тіло циклу) може бути блоком інструкцій:

```
for (int i = 1; i < 10; i++)
{
    Console.WriteLine(i);
    Console.WriteLine(i * i);
}
```

Цикл foreach призначений для перебору набору або колекції елементів. Цей цикл має синтаксис:

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

Цикл застосовується до рядків, керованих масивів і об’єктів-колекцій. Цикл foreach перебирає всі елементи колекції і поміщує їх у змінну, з якою можна виконати певні дії – наприклад, вивести на екран, використати у виразі, присвоїти її значення іншій змінній. При цьому присвоювати значення самій цій змінній заборонено.

Приклад застосування – виведення рядка на екран посимвольно, кожен символ у своєму рядку:

```
String ss = "Name";
foreach (char c in ss)
    Console.WriteLine(c);
```

Цикл **foreach** дуже корисний при обробці динамічних масивів та колекцій, число елементів у яких змінюється під час виконання програми, оскільки можна не задумуватися про кількість елементів масиву/колекції та їх індекси. Основним недоліком такого циклу є те, що він послідовно перебирає всі елементи, тобто крок лічильника циклу завжди рівний 1, а сам лічильник циклу програмісту недоступний. Тому, наприклад, присвоїти нулю кожен третій елемент масиву за допомогою foreach не можна.

Часто буває, що потрібно перервати виконання циклу або перейти до наступної ітерації циклу, пропустивши поточну. Як і в C++, для першої мети служить оператор **break**, для другої – **continue**.

Приклади:

1) Тут виконання циклу переривається, якщо i==5:

```
int i = 0;
do
{
    i++;
    if (i == 5) break;
    Console.WriteLine(i);
} while (i < 10);
```

В результаті на екран буде виведено числа з 1 по 4, і подальше виконання програми продовжиться за циклом.

2) Тут, якщо i==5, вивід на консоль не виконується, і цикл переходить на наступну ітерацію (i==6):

```
int i = 0;
do
{
    i++;
    if (i == 5) continue;
    Console.WriteLine(i);
} while (i < 10);
```

В результаті на екран буде виведено всі числа з 1 по 10, крім 5.

Оператори break і continue можна використовувати в будь-якому типі циклів.

Для складних випадків (якщо, наприклад, треба перервати багато вкладених циклів) в C# можна застосувати оператор **goto** <мітка>, який передає виконання на оператор, помічений <міткою>:

```
foreach (Matrix m in StiffnessMatrices)
    for (int Row = 0; Row < m.RowCount; Row++)
        for (int Col = 0; Col < m.ColCount; Col++)
            if (double.IsInfinity(m[Row,Col]))
            {
```

```
                errMat = m;
                goto error;
            }

        error: Console.WriteLine($"Matrix {m.Name} has errors!");
```

Правила використання міток в C# аналогічні правилам C++: ідентифікатор мітки повинен відповідати правилам мови, мітка видима лише всередині метода, в якому описана, і перехід за міткою в інший метод неможливий. Після мітки обов'язково ставиться двокрапка.

Використання оператора goto в C# не рекомендоване, оскільки це знижує читабельність програми і часто є причиною логічних помилок в алгоритмах.

3.3 Рядки і символи в C#. Операції над рядками.

Рядок у C# – це впорядкована колекція символів типу char, доступна лише для читання. На відміну від C++, у C# рядок не завершується символом '\0', тому може містити довільну кількість таких символів. Літерали рядків було детально розглянуто в лекції 2; тепер звернімо увагу на операції над рядками.

Слід зазначити, що рядок у C# незмінний. Всі описані нижче функції при виконанні операцій над рядками знищують старий об'єкт рядка і створюють новий.

Перш за все, рядки можна зчіплювати в один (конкатенація). Для цього служить оператор +:

```
st1 ="Say:";
st2 ="Hello world";
st2 = " " + st2+"!"; //Додати пробіл і знак оклику
st1 += st2; //Результат: st1=="Say: Hello world!"
```

Кожен символ рядка доступний за індексом, індекс першого символу рядка рівний 0:

```
char c = st1[1]; //c=='a'
```

Довжина рядка міститься у його властивості Length:

```
Console.WriteLine(st1.Length);
```

Порівняння рядків проводиться за їх значенням, причому визначені лише операції == і != (перевірка на рівність/не рівність).

Починаючи з C# 11 можливе задання багаторядкового тексту за допомогою трьох лапок, причому як просто тексту, так і тексту з інтерполяцією:

```
string text = $"""
  <element attr="content">
    <body>
    {val}
    </body>
  </element>
  """;
```

Основна функціональність роботи з рядками реалізована через методи класу String:

Compare: порівнює два рядки з урахуванням поточної культури (локалі, тобто таблиці символів) на комп'ютері користувача.

CompareOrdinal: порівнює два рядки без урахування локалі.

Contains: визначає, чи міститься підрядок у рядку.

Concat: з'єднує рядки (реалізує оператор +).

CopyTo: копіює частину рядка, починаючи з певного індексу до масиву.

EndsWith: визначає, чи збігається кінець рядка з підрядком.

Format: форматує рядок.

IndexOf: знаходить індекс першого входження символу або підрядка у рядку.

Insert: вставляє в рядок підрядок.

Join: з'єднує елементи масиву рядків.

LastIndexOf: знаходить індекс останнього входження символу або підрядка у рядку.

Replace: заміщує у рядку символ або підрядок іншим символом або підрядком.

Split: розділяє один рядок на масив рядків.

Substring: витягує з рядка підрядок, починаючи із зазначеної позиції.

ToLower: перекладає всі символи рядка у нижній регістр.

ToUpper: перекладає всі символи рядка у верхній регістр.

Trim: видаляє початкові та кінцеві пробіли з рядка.

Детальне вивчення параметрів та функціоналу цих методів виноситься на самостійне опрацювання, ознайомитися з матеріалом можна за посиланням https://abitap.com/18-2-operacziyi-z-ryadkamy/

Далі розглянемо форматування. Для виведення в консоль ми вже застосовували інтерполяцію та плейсхолдери. За допомогою використання методу string.Format і плейсхолдерів можна форматувати вивід даних безпосередньо у змінну типу string:

```
string output = string.Format("Ім'я: {0}  Вік: {1}", name, age);
```

У рядку форматування після плейсхолдера можна вказати такі формати:

C/c – задає формат грошової одиниці, вказує кількість десяткових розрядів після коми;

D/d – цілочисельний формат, вказує мінімальну кількість цифр;

E/e – експоненційне подання числа, вказує кількість десяткових розрядів після коми;

F/f – формат дробових чисел із фіксованою комою, вказує кількість десяткових розрядів після коми;

G/g – задає більш короткий із двох форматів: F або E;

N/n – також задає формат дробових чисел із фіксованою комою, визначає кількість розрядів після коми;

P/p – задає відображення знаку відсотків поряд з числом, вказує кількість десяткових розрядів після коми;

X/x – шістнадцятковий формат числа.

Приклади:

```
//Гроші
double number = 29.7;
Console.WriteLine(string.Format("{0:C0}", number)); // 30 грн.
string result = string.Format("{0:C2}", number); // 29,70 грн.
Console.WriteLine(result);

//Ціле число
int n = 25;
Console.WriteLine(string.Format("{0:d}", n)); // 25
result = string.Format("{0:d5}", n); // 00025
Console.WriteLine(result);

//Ціле число, шістнадцяткова система
n = 16;
Console.WriteLine(string.Format("{0:x}", n)); // 10
result = string.Format("{0:x5}", n); // 00010
Console.WriteLine(result);

//Дійсне число
Console.WriteLine(string.Format("{0:f}", n)); // 16,00
number = 22.07;
Console.WriteLine(string.Format("{0:f4}", number)); // 22,0700
number = 1.245e+10;
result = string.Format("{0:e3}", number); // 1,245e+10
Console.WriteLine(result);

//Дробове число
number = 10.37;
Console.WriteLine(string.Format("{0:n}", number)); //10,37
Console.WriteLine(string.Format("{0:n4}", number)); //10,3700

//Проценти
number = 0.075;
Console.WriteLine(string.Format("{0:P2}", number)); //7,50%
```

Використовуючи знак #, можна налаштувати формат виведення. Наприклад, виведення номера телефону можна реалізувати так:

```
long phone = 380444549470L;
//+38 (044) 454-94-70
Console.WriteLine(string.Format("{0:+## (###) ###-##-##}", phone));
```

Метод ToString(), який отримує рядковий опис об’єкта, може здійснювати форматування і підтримує ті ж описувачі, що й у методі Format:

```
Console.WriteLine(phone.ToString("+## (###) ###-##-##"));
```

Всередині фігурних дужок інтерполяції також можна використовувати форматування:

```
Console.WriteLine($"{phone:+## (###) ###-##-##}");
```

До того ж, можна задати кількість пробілів до та після виведеного значення:

```
string name = "Іван";
int age = 19;
Console.WriteLine($"Ім'я: {name -5} Вік: {age 5}");
```

В результаті буде виведено 5 пробілів після значення імені «Іван» і 5 – до значення віку «19».

3.4 Перелічувані типи (enum).

Перерахування або перелічуваний тип у C# – це тип користувача, що є послідовністю логічно пов'язаних констант. Синтаксис оголошення такого типу:

```
enum <назва> {
    <значення1>, <значення2>, ..... <значенняN>
}
```

Кожне перерахування визначає новий тип даних, за допомогою якого можна визначати змінні, константи, параметри методів та інше. Як значення змінної, константи або параметра методу, які мають тип перерахування, повинно виступати одне з перерахованих у дужках оголошення enum значень.

Константи перерахування можуть мати явно вказаний тип. Тип вказується після назви перерахування через двокрапку:

```
enum LineType :byte
{
  Soild,
  Dash,
  Dot,
  DashDot
}
```

Тип перерахування обов'язково повинен бути цілим (byte, sbyte, short, ushort, int, uint, long, ulong). Якщо тип явно не вказано, за замовчуванням використовується тип int. За замовчуванням кожному елементу enum присвоєно ціле значення, починаючи з 0. Так, у прикладі LineType.Soild має значення 0, LineType.Dash – значення 1 і т.д. Можна явно задати значення для константи перелічуваного типу, тоді всі константи за нею матимуть наступні за ним цілі значення:

```
enum LineType :byte
{
  Soild, //0
  Dash = 2, //2
  Dot, //3
  DashDot //4
}
```

При цьому константи перерахування можуть мати однакові значення, або навіть можна надавати одній константі значення іншої константи:

```
enum LineType :byte
{
  Soild = 1, //1
```

```
    Dash = 1,  //1
    Dot,       //2
    DashDot = Dash //1
}
```

Можна визначити чисельне значення змінної типу enum приведенням типу:

```
enum LineType :byte
{
  Soild,
  Dash,
  Dot,
  DashDot
}

LineType lt1 = LineType.Dash; //Змінна типу LineType
int i1 = (int)lt1; //i1 == 1
lt1 = (LineType)2;  //lt1 == LineType.Dot
Console.WriteLine(lt1 < LineType.Dot); //False
```

Над змінними типів enum визначено операції присвоювання і порівняння, бінарні + і –, ++, -- (префіксні і постфіксні), побітові операції (~, ^, &, |) та sizeof. Різні типи enum, навіть з однаковими індексами констант, між собою не сумісні. Можливе, проте, приведення типу до базового типу констант (див. наведений вище приклад із приведенням до int) і навпаки, а також приведення між типами enum

```
lt1 = (LineType)HardwareStats.ThermalStab;
```

хоча таке приведення часто не має жодного практичного сенсу.

Для чого потрібні типи enum? На практиці enum використовують для символьного позначення констант, пов’язаних із цілими значеннями, і зв’язаних логічно. Типовий приклад – стилі ліній, стиль шрифту на екрані (звичайний, жирний, похилий, підкреслений, закреслений) та інші подібні значення, зокрема, значення кольорів. Наприклад, Color.AliceBlue і шістнадцяткове число 0xFFF0F8FF означають один і той самий колір, однак запам’ятати символьні значення легше, ніж набір цифр. Також такими типами задаються бітові маски при доступі до низькорівневих можливостей апаратури.

У випадку, коли типом enum задається бітова маска, перед визначенням типу слід писати атрибут [Flags], далі такі значення можна комбінувати і перевіряти побітовими операціями:

```
[Flags]
public enum HardwareStats {
  IsOn = 0b0001,
  ThermalStab = 0b0010,
  Interrupt = 0b0100,
  Error =0b1000
};

HardwareStats Stats = HardwareStats.Interrupt | HardwareStats.ThermalStab;
i1 = (int)Stats;
Console.WriteLine(Stats); //ThermalStab, Interrupt
Console.WriteLine(Stats & HardwareStats.Error); //0
Console.WriteLine(Stats & HardwareStats.Interrupt); //Interrupt
```

Запитання для самоперевірки

1. Які в C# є умовні конструкції?

2. В чому полягають особливості оператора switch у C# (зокрема порівняно з C++)?

3. Які оператори циклу є в C#?

4. Якому циклу еквівалентний оператор for в C#?

6. У чому полягають особливості оператора foreach?

7. Для чого потрібні оператори break, continue та goto і як їх застосовувати?

8. Які основні особливості рядків у C#?

9. Які існують основні операції над рядками в C#?

10. Як форматувати виведення даних безпосередньо в змінну-рядок?

11.Що таке перелічуваний тип, як його оголосити і як можна використовувати?

Питання з теми, що виносяться на самостійне опрацювання

1. Детальне вивчення параметрів та функціоналу методів класу System.String.

Лекція 4. Масиви. Методи. Параметри методів.

4.1 Масиви

Масив в C#, як і в C++ – набір однотипних даних. Оголошення масиву в C# суттєво відрізняється від C++:

```
<тип>[] <ідентифікатор>;
```

Як бачимо, квадратні дужки, на відміну від C++, вказують після типу, і в них немає довжини масиву. Тому, оголошений масив буде пустим (не міститиме елементів), що в C++ теж неможливо; змінна <ідентифікатор> при такому оголошенні буде рівною null. Таким чином, масив у C# завжди динамічний, на відміну від C/C++. При цьому пам'ять, зайняту масивом, .NET звільнить автоматично, як тільки масив перестане бути потрібним у програмі.

Для того, щоб у масив C# можна було записати дані, його слід ініціалізувати. Найпростіший метод ініціалізації – виділення пам'яті під масив за допомогою оператора **new** із вказуванням розміру масиву:

```
int[] arr; //Це пустий масив

arr = new int[5]; //Масив отримує розмір 5 та ініціалізується 0
```

При такій ініціалізації всі елементи масиву будуть ініціалізовані значеннями типу за замовчуванням (у випадку int це 0). Значення елементів такого масиву доведеться встановлювати в коді програми. Можливо також ініціалізувати масив наперед заданими значеннями при оголошенні. Приклади такого оголошення і одночасної ініціалізації масивів наведено нижче:

```
int[] arr1 = new int[5] { 5, 4, 3, 2, 1 };
int[] arr2 = new int[] { 5, 4, 3, 2, 1 };
int[] arr3 = new[] { 5, 4, 3, 2, 1 };
int[] arr4 = { 5, 4, 3, 2, 1 };
```

Усі 4 варіанти ініціалізації рівнозначні: масив буде заповнено числами від 5 до 1 у порядку спадання.

Для того, щоб звернутися до елементу масиву, використовується доступ за індексами: arr[i] – i-й елемент масиву arr. Перший елемент масиву, як і в C/C++, має індекс 0, другий – 1 і т.д. Спроба звернутися до елементу масиву з індексом, більшим за довжину масиву-1, призведе до виникнення помилки виконання. Це значно зменшує ймовірність виникнення і не виявлення помилок виходу за межі масиву, які часто допускають при програмуванні на C++, особливо при одночасному використанні масивів та вказівників. Подібні помилки погано виявляються і тому їх досить важко виправляти.

Для присвоєння елементу масиву значення константи чи змінної (і навпаки) використовується звичайний оператор присвоювання:

```
int tmp;

tmp = arr[2];
arr[2] = arr[1];
```

```
arr[1] = tmp; //В результаті arr[1] та arr[2] поміняються місцями
```

На відміну від C++, масив в C# не можна присвоїти вказівнику, якщо не використовується так званий небезпечний код (unsafe code), тому перехід до і-го елемента масиву слід виконувати змінюючи індекс елементу масиву. За замовчуванням інструкції на кшталт x=*(arr+i) в C# не тільки не працюватимуть – вони навіть не відкомпілюються.

Небезпечним кодом в C# вважається використання вказівників, виділення і звільнення пам'яті «вручну» (не через механізми .NET) і виклик функцій через вказівники на них. Використання небезпечного коду (тобто коду, безпечність якого не може перевірити .NET) виходить за межі даного курсу. Детальніше про нього можна прочитати тут: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/unsafe-code.

Кожен масив C# має властивість Length, що зберігає довжину масиву. Індекс останнього елементу масиву при цьому рівний <масив>.Length-1. Для спрощення доступу до елементів масиву з його кінця, щоб не доводилося писати код на зразок arr[arr.Length-2], починаючи з C# 8, можна використати звернення arr[^2]:

```
int j = 2;
//Ці дві інструкції рівноцінні
arr[arr.Length-j] = 0;
arr[^j] = 0;
```

Для перебору масивів можна використовувати будь-які цикли: do…while, while, for. Найчастіше використовується цикл for, наприклад:

```
//Обчислення суми елементів масиву з парними індексами
int sum_even = 0;
for (int i = 0; i < arr.Length; i += 2)
    sum_even += arr[i];
```

Цикл foreach теж працює з масивом, але його використання іноді незручне через ряд обмежень. По-перше, цикл foreach проходить всі елементи масиву підряд, і неможливо «перескочити» елементи простою зміною індексу. По-друге, змінна циклу, ідентифікатор якої стоїть перед in, отримує в кожній ітерації значення елементу масиву; доступу ж до самого елементу масиву в цьому циклі немає, і змінити його значення неможливо. З іншого боку, для обчислення суми масиву і подібних операцій простіше скористатися циклом foreach, ніж for:

```
int sum = 0;
foreach (int element in arr)
    sum += element;
```

В C#, як і в C++, можливе використання ***багатовимірних масивів***. Оголошення такого масиву містить в квадратних дужках коми. Так, оголошення двовимірного масиву з ініціалізацією слід робити так:

```
int[,] arr2D = { { 1, 2, 3 }, { 4, 5, 6 } };
```

```
int[,] arr2D1; //Пустий масив
int[,] arr2D2 = new int[2, 3];
int[,] arr2D3 = new int[2, 3] { { 1, 2, 3 }, { 4, 5, 6 } };
int[,] arr2D4 = new int[,] { { 1, 2, 3 }, { 4, 5, 6 } };
int[,] arr2D5 = new[,] { { 1, 2, 3 }, { 4, 5, 6 } };
```

а оголошення тривимірного – так

```
int[,,] arr3D = new int[2, 3, 4];
```

Рангом масиву називають кількість його вимірів. Так, звичайний (одновимірний) масив має ранг 1, двовимірний масив – ранг 2, тривимірний – 3 і т.д. На практиці найчастіше використовують одновимірні і двовимірні масиви.

Для перебору елементів багатовимірного масиву необхідно знати кількість елементів в кожному його вимірі. Цю кількість повертає метод GetUpperBound(int dimension), де dimension=0 відповідає першому виміру, dimension=1 відповідає другому виміру і т.д. Тому, для виведення двовимірного масиву (матриці) можна скористатися таким кодом:

```
int[,] matrix = { { 1, 2, 6 }, { 4, 7, 8 } };
int rows = matrix.GetUpperBound(0) + 1; // кількість рядків
int columns = matrix.Length / rows; // кількість стовпців
                                    // або так
            // int columns = numbers.GetUpperBound(1) + 1;

for (int i = 0; i < rows; i++)
{
    for (int j = 0; j < columns; j++)
    {
        Console.Write($"{matrix[i, j]} \t");
    }
    Console.WriteLine();
}
```

В C# також легко можна створити так званий «**зубчатий масив**», або **масив масивів**. Синтаксис оголошення такого масиву виглядає так:

```
<тип>[][] <ідентифікатор>;
```

Як приклад розглянемо оголошення масиву із чотирьох підмасивів типу int:

```
int[][] ToothArr = new int[4][]; //Масив з чотирьох підмасивів
ToothArr[0] = new int[2] { 1, 2 }; // Ініціалізація першого підмасиву
ToothArr[1] = new int[3] { 1, 2, 3 }; // Ініціалізація другого підмасиву
ToothArr[2] = new int[5] { 1, 2, 3, 4, 5 }; // Ініціалізація третього
підмасиву
ToothArr[3] = new int[3] { 5, 6, 7 }; // Ініціалізація четвертого підмасиву
```

Той самий масив можна оголосити так:

```
int[][] ToothArr = {
   new int[2] { 1, 2 },
   new int[3] { 1, 2, 3 },
   new int[5] { 1, 2, 3, 4, 5 },
   new int[3] { 5, 6, 7 }
```

```
};
```

Ці оголошення рівнозначні.

Перебір «зубчатих масивів» слід виконувати за допомогою вкладених циклів:

```
// Перебір за допомогою циклу foreach
foreach (int[] row in ToothArr)
{
    foreach (int number in row)
    {
        Console.Write($"{number} \t");
    }
    Console.WriteLine();
}

// Перебір за допомогою циклу for
for (int i = 0; i < ToothArr.Length; i++)
{
    for (int j = 0; j < ToothArr[i].Length; j++)
    {
        Console.Write($"{ToothArr[i][j]} \t");
    }
    Console.WriteLine();
}
```

Зверніть увагу на те, що в зовнішньому циклі foreach змінна циклу має тип масиву, а також на те, що довжину кожного підмасиву «зубчатого» масиву слід визначати використавши його властивість Length; ToothArr.GetUpperBound(1) у цьому випадку не спрацює.

4.2 Методи. Параметри методів.

Методи в C# – аналог методів і функцій C++. **Метод** – іменований блок коду, що виконує певну функцію, і повертає результат.

```
[<модифікатори>] <тип результату> <ім'я>([<формальні
параметри>])
{
    //тіло методу
}
```

Модифікатори та формальні параметри в оголошенні методу – **необов'язкові.**

Формальні параметри методу – опис (типи та імена) параметрів, які приймає метод. Формальні параметри записуються в круглих дужках у такому вигляді:

```
<тип> <параметр1>, <тип> <параметр2>,  …  <тип> <параметрN>
```

Імена формальних параметрів у методі доступні як локальні змінні.

Тіло методу – блок операторів, що може містити як інструкції, так і оголошення даних. Як приклад, розглянемо метод OutputName, який виводить на екран (в консоль) ім'я користувача:

```
void OutputName(string Name)
{
    Console.WriteLine("Ім'я користувача: "+Name);
}
```

Цей метод приймає один параметр Name типу string (ім'я користувача) і виводить його на екран. Метод OutputName нічого не повертає (тип результату – void, тобто пустий). Метод в C# (як і метод в C++) є частиною класу, в якому його оголошено. При оголошенні в рамках консольної програми OutputName цей метод належатиме до класу Program.

Для коротких методів можна застосувати спрощений запис:

```
void OutputName(string Name) => Console.WriteLine("Ім'я користувача: " + Name);
```

Звернення до методу в межах класу, в якому його оголошено виконується так:

```
<ім'я методу>([<фактичні параметри>]);
```

а поза межами класу (якщо створено об'єкт відповідного класу) так:

```
<ім'я об'єкту>.<ім'я методу>([<фактичні параметри>]);
```

Фактичні параметри методу – це значення чи змінні, які передаються методу для обробки в кожному конкретному виклику. Метод OutputName приймає один параметр – рядок з іменем. Тому виклик цього методу для виведення імені Діана матиме вид

```
OutputName("Діана");
```

У даному прикладі формальному параметру Name метода OutputName відповідає фактичний параметр "Діана".

Передаються параметри в метод за позицією. Якщо оголошено метод, який обчислює, наприклад, значення амплітуди гармонічного коливання в заданий час

```
double Harmonic(double t, double A, double w, double Ph)
{
    return A * Math.Sin(w * t);
}
```

при його виклику першим буде передано час t, потім – амплітуду коливань A, після якої – кутову швидкість w, а за нею – фазу Ph. Якщо параметрам методу передаються значення змінних, то цим змінним має бути присвоєно значення – інакше програма не відкомпілюється. Формальні та фактичні параметри повинні мати однакові типи. Можна використовувати фактичні параметри, тип даних яких може бути автоматично приведений до типу відповідних формальних параметрів:

```
double d;
int A1 = 10, f1 = 11;
```

```
d = Harmonic(10.2, 11.0, 25.3, 10); // всі параметри дійсні
d = Harmonic(24.3, A1, f1, 0); // частина параметрів – цілі числа
```

За замовчуванням у виклику методу треба вказувати всі його параметри. В C# можливе також використання **необов'язкових параметрів**. Наприклад, перевизначимо функцію обчислення амплітуди гармонічного коливання так:

```
static double Harmonic(double t, double A, double w = 10, double Ph = 0)
{
    return A * Math.Sin(w * t + Ph);
}
```

У даному методі два останніх параметри є необов'язковими: Якщо при виклику методу необов'язковий параметр опустити, методу буде передано його значення за замовчуванням, вказане в оголошенні методу після оператора присвоювання. При оголошенні методу із необов'язковими параметрами, всі параметри після першого оголошеного необов'язкового (із заданим значенням за замовчуванням) повинні бути необов'язковими.

При виклику методу з необов'язковими параметрами можна не передавати дані як усіх необов'язкових параметрів, так і лише частини таких параметрів. При цьому не можна пропустити частину необов'язкових параметрів, а потім задати значення, наприклад, останнього параметра:

```
d = Harmonic(25.4, 10, 12); //Фаза Ph за замовчуванням
d = Harmonic(25.4, 10); //Частота w і фаза Ph за замовчуванням
d = Harmonic(25.4, 10, , 1); //Спроба задати Ph, не задавши w.
                             //Не відкомпілюється!
```

Для того, щоб обійти це обмеження, з Python запозичили концепцію **іменованих параметрів**: при передачі даних можна вказати ідентифікатор формального параметра, а потім через символ «:» вказати значення фактичного, причому порядок параметрів може бути довільним:

```
d = Harmonic(12, w: 23, Ph: 11, A: 5); //Перший параметр – час t
d = Harmonic(w: 23, A: 5, t: 35);  // тут фаза Ph=0 за замовчуванням
```

Метод може повертати значення будь-якого типу. Якщо метод нічого не повертає, то тип результату методу – void. Для повернення результату з методу, як і в C++, в C# служить оператор **return**:

```
return <значення результату>;
```

Цей оператор повертає значення результату і негайно припиняє виконання методу. Між типом методу, що повертається, і значенням, що повертається після оператора return, повинна бути відповідність; спроба повернути значення типу, що не приводиться автоматично до типу результату метода, призведе до помилки компіляції. Операндом return може бути константа, змінна або вираз, як, наприклад, показано вище у методі Harmonic.

При скороченому записі методів return не вказують:

```
double Modulus(double x) => x > 0 ? x : -x; //Обчислення модуля
```

Далі розглянемо методи передачі параметрів. Звичайний метод передачі параметрів, описаний вище – **передача за значенням**, при якій метод отримує копію фактичного параметру і працює з нею, а не зі змінною, значення якої передано в метод. Після виходу з методу, локальна копія параметру знищується.

```
int i = 2;

void MultiplyBy2(int i)
{
    i *= 2;
    Console.WriteLine($"В методі i = {i}");
}

Console.WriteLine($"До виклику метода i = {i}");  //i=2
MultiplyBy2(i);  //i=4 – всередині метода
Console.WriteLine($"Після виклику метода i = {i}"); //i=2
```

Для того, щоб метод міг змінювати значення фактичного параметру, застосовують **передачу за посиланням**. На відміну від C++, де така передача виконувалася шляхом передачі в метод вказівника на змінну (наприклад, int* par) або посилання на неї (int &par), в C# перед типом параметра слід просто поставити ключове слово **ref**. Те саме ключове слово слід вказати і при виклику такого метода. Таким чином, коректне визначення метода із наведеного вище прикладу, що множить значення змінної на 2 і повертає результат в ту саму змінну, і виклик такого методу виглядають так:

```
int i = 2;

void MultiplyBy2(ref int i)
{
    i *= 2;
    Console.WriteLine($"В методі i = {i}");
}

Console.WriteLine($"До виклику метода i = {i}");  //i=2
MultiplyBy2(ref i);  //i=4 – всередині метода
Console.WriteLine($"Після виклику метода i = {i}"); //i=4
```

Ключове слово **out** слугує для позначення того, що параметр методу приймає результат виконаних методом дій:

```
double f1, f2;

void FreqMultiples(double f, out double f2x, out double
f3x)
{
    f2x = f * 2;
    f3x = f * 3;
}

FreqMultiples(25, out f1, out f2);
```

Метод FreqMultiples повертає значення частоти f, помноженої на 2 і на 3. Ключове слово out, як і ключове слово ref, потрібно вказувати і при оголошенні, і при виклику метода.

Використання параметрів out виправдане у випадку, коли метод повинен повертати кілька значень. При цьому, кожному параметру out всередині метода слід присвоїти значення, інакше програма не відкомпілюється.

Можна визначати змінні, які передаються out-параметрами, безпосередньо при виклику методу; якщо тип параметрів, які повертає метод, невідомий, можна використати ключове слово var (неявну типізацію):

```
FreqMultiples(25, out double f1, out double f2); //змінні визначено в методі
FreqMultiples(25, out var f1, out var f2); //Неявна типізація
```

Ключове слово **in** використовується аналогічно слову **out** і вказує, що параметр передаватиметься в метод за посиланням, проте всередині методу його значення не можна буде змінити:

```
void FreqMultiples(in double f, out double f2x, out double f3x)
{
    //f=f*0.5; //Цей рядок не відкомпілюється!
    f2x = f * 2;
    f3x = f * 3;
}
```

Сенс використання ключового слова in полягає у підвищенні продуктивності при передачі деяких типів параметрів (наприклад великих масивів) за посиланням замість передачі за значенням, оскільки в такому випадку локальна копія масиву не створюється, що економить час і пам'ять.

Ключове слово **params** слугує для того, щоб визначити метод, який може приймати будь-яку кількість параметрів:

```
int SumByModulus(params int[] list)
{
    int s = 0;
    foreach (int n in list)
        s += n > 0 ? n: -n;
return s;
}
```

При виклику такого метода в нього можна передати або окремі значення, або масив:

```
int[] numbers = { -1, 2, -3, -4 };
Console.WriteLine(SumByModulus(numbers)); // = 10
Console.WriteLine(SumByModulus(1, 2, 3, 4, 5, 6)); // = 21
Console.WriteLine(SumByModulus(1)); // = 1
Console.WriteLine(SumByModulus(0)); // = 0
```

Параметр методу з модифікатором params повинен бути останнім у оголошенні методу. Спроба задати параметри після нього не допускається (спричинить помилку компіляції).

Зверніть увагу на те, що якщо Ви передаєте методу як формальний параметр масив,

```
int SumArray(in int[] list, bool ByModulus = false)
{
    int s = 0;
    foreach (int n in list)
        if (ByModulus)
            s += n > 0 ? n: -n;
        else
            s += n;
    return s;
}
```

відповідним йому фактичним параметром обов'язково повинен бути масив. Передати методу список розділених комою чисел тут не можна. Також, на відміну від випадку з використанням params, після такого параметру-масиву в оголошенні методу може бути задано інші параметри.

Метод може викликати сам себе (**рекурсія**). Наприклад, метод для обчислення значення n-го члену ряду Фібоначчі

```
int Fibonacci(int n)
{
    if (n == 0 || n == 1)
        return n;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

у кожному виклику викликає сам себе двічі для обчислення значень n-1-го та n-2-го членів ряду. При написанні рекурсивних функцій слід обов'язково забезпечити, щоб рекурсивна функція обов'язково завершувала перехід на наступний рівень рекурсії (тобто виходила з рекурсії, повертаючи виконання на попередній рівень виклику) за певних умов – оскільки інакше стек переповниться і програма зруйнується. У наведеному вище прикладі такою умовою є рівність n 0 або 1 – при цих значеннях n функція не викликатиме саму себе і поверне результат на вищий рівень виклику. Це уможливлює виконання оператора return Fibonacci(n - 1) + Fibonacci(n - 2); і функція врешті-решт завершить виконання.

Якщо можливо, слід уникати використання рекурсивних функцій з великою глибиною рекурсії (кількістю необхідних вкладених викликів), оскільки кожен такий виклик функції займає частину стеку, що може призвести до його переповнення. Крім того, часто альтернативні методи обчислень працюють швидше за рекурсію.

На відміну від C++, у C# можливе використання всередині методів **локальних функцій**. Локальна функція – це функція, визначена всередині іншого методу або функції. Звернутися до такої функції можна лише в межах блоку коду того методу, в якому її оголошено. Приклад:

```
bool IsArrSumLarger(in int[] arr1, in int[] arr2)
{
    int Sum1 = Sum(arr1);
    int Sum2 = Sum(arr2);
    return Sum1 > Sum2;
```

```
    int Sum(in int[] arr)
    {
        int s = 0;
        foreach (int n in arr)
            s += n;
        return s;
    }
}
```

У наведеному прикладі, до локальної функції Sum звернутися за межами методу IsArrSumLarger не можна.

Локальна функція може бути оголошена з модифікатором static:

```
static int Sum(in int[] arr)
```

Такі функції не можуть звертатися до змінних оточення, тобто змінних і параметрів методу, у якому статична локальна функція визначена. Це гарантує відсутність побічних ефектів у вигляді модифікації локальною функцією змінних методу, в якому цю функцію оголошено.

Якщо у функції використовується оператор switch, то замість класичного запису

```
double CallTrigFcnClassic(int fcn, double arg)
{
    switch (fcn)
    {
        case 0: return Math.Sin(arg);
        case 1: return Math.Cos(arg);
        case 2: return Math.Tan(arg);
        default: return 1/Math.Tan(arg);
    }
}
```

можна використати скорочений:

```
//Варіант 1
double CallTrigFcn(int fcn, double arg)
{
    double result = fcn switch
    {
        0 => Math.Sin(arg),
        1 => Math.Cos(arg),
        2 => Math.Tan(arg),
        _ => 1 / Math.Tan(arg)
    };
    return result;
}

//Варіант 2
double CallTrigFcn1(int fcn, double arg)
{
    return fcn switch
    {
        0 => Math.Sin(arg),
        1 => Math.Cos(arg),
        2 => Math.Tan(arg),
        _ => 1 / Math.Tan(arg)
    };
}
```

```
//Варіант 3
double CallTrigFcn3(int fcn, double arg) => fcn switch
{
    0 => Math.Sin(arg),
    1 => Math.Cos(arg),
    2 => Math.Tan(arg),
    _ => 1 / Math.Tan(arg)
};
```

Зверніть увагу на те, що при використанні будь-якого з варіантів запису, метод повинен повертати результат за всіх можливих значень fcn, тобто використання варіанту default (у скороченому записі «_») є необхідною умовою. Це зумовлено тим, що функція повинна повертати значення за будь-яких значень аргументів.

Запитання для самоперевірки

1. Одновимірні масиви в C# : як оголосити, ініціалізувати, визначити розмір, перебирати елементи масиву?
2. Двовимірні масиви в C# : як оголосити, ініціалізувати, перебирати елементи масиву? Що таке ранг масиву?
3. Що таке зубчатий масив і як з ним працювати?
4. Що таке метод? Що таке формальні і фактичні параметри?
5. Що таке необов'язкові параметри? Які особливості їх використання?
6. Як використовувати іменовані параметри?
7. Чим відрізняється передача параметрів за значенням і за посиланням?
8. Навіщо потрібні ключові слова ref, in, out ? В чому різниця між ref і out?
9. Для чого використовується ключове слово params?
10. Що таке рекурсія? Що слід враховувати при написанні рекурсивних функцій?
11. Що таке локальні функції?
12. Якими способами можна записати функцію з використанням switch?

Питання з теми, що виносяться на самостійне опрацювання

1. Ознайомлення з використанням небезпечного коду.

Лекція 5. Об'єктно-орієнтоване програмування в C#. Класи та об'єкти. Структури (struct). Кортежі. Перезавантаження методів. Область видимості змінних. Іще раз про простори імен.

5.1 Об'єктно-орієнтоване програмування в C#. Класи та об'єкти. Модифікатори доступу. Конструктори. Ініціалізатори об'єктів.

C# – об'єктно-орієнтована мова програмування. Тому програму на C# можна представити у вигляді об'єктів, що взаємодіють між собою. Описом кожного такого об'єкту є клас, а екземпляром цього класу є відповідний об'єкт.

Для розуміння концепції візьмемо за приклад автомобіль. Загальна назва цього засобу пересування (автомобіль) – це клас. Кожен автомобіль має ряд характеристик: об'єм двигуна, максимальна швидкість, час розгону до 100 км/год, номер держреєстрації та ін. У кожного конкретного автомобіля (об'єкту) ці дані свої; але кожен автомобіль все одно є автомобілем, тобто кожен конкретний автомобіль є об'єктом класу «автомобіль».

У C# рядок – це клас; масиви обслуговує клас Array; консольне введення-виведення забезпечується класом Console. Класами у C# забезпечується робота з файлами, обробка виняткових ситуацій, а також створення графічного інтерфейсу програм. Наприклад, вікно на екрані, кнопка, поле введення, текст, меню та інші компоненти екранних форм – все це в програмі на C# реалізовано як класи. Бібліотеки для C# найрізноманітнішого призначення, представлені в NuGet – це теж бібліотеки класів.

Для оголошення власного класу використовують такий синтаксис:

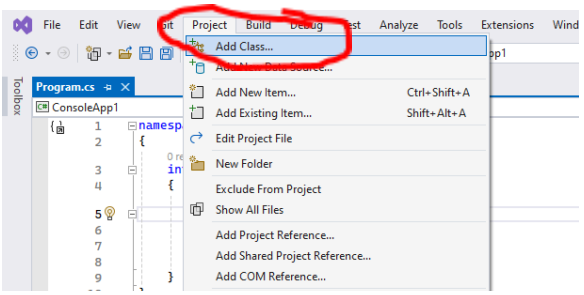
```
class <назва класу>
{
   [<поля>]
   [<методи>]
     ...
}
```

<Назва класу>, вказана після ключового слова class, є типом класу. Всі створені в програмі об'єкти класу матимуть цей тип. Кожен клас містить дані та методи їх обробки. Дані класу розміщуються в **полях** – змінних, які оголошені всередині класу (всередині фігурних дужок). Методи обробки даних – це методи, які належать до класу і мають доступ до його полів.

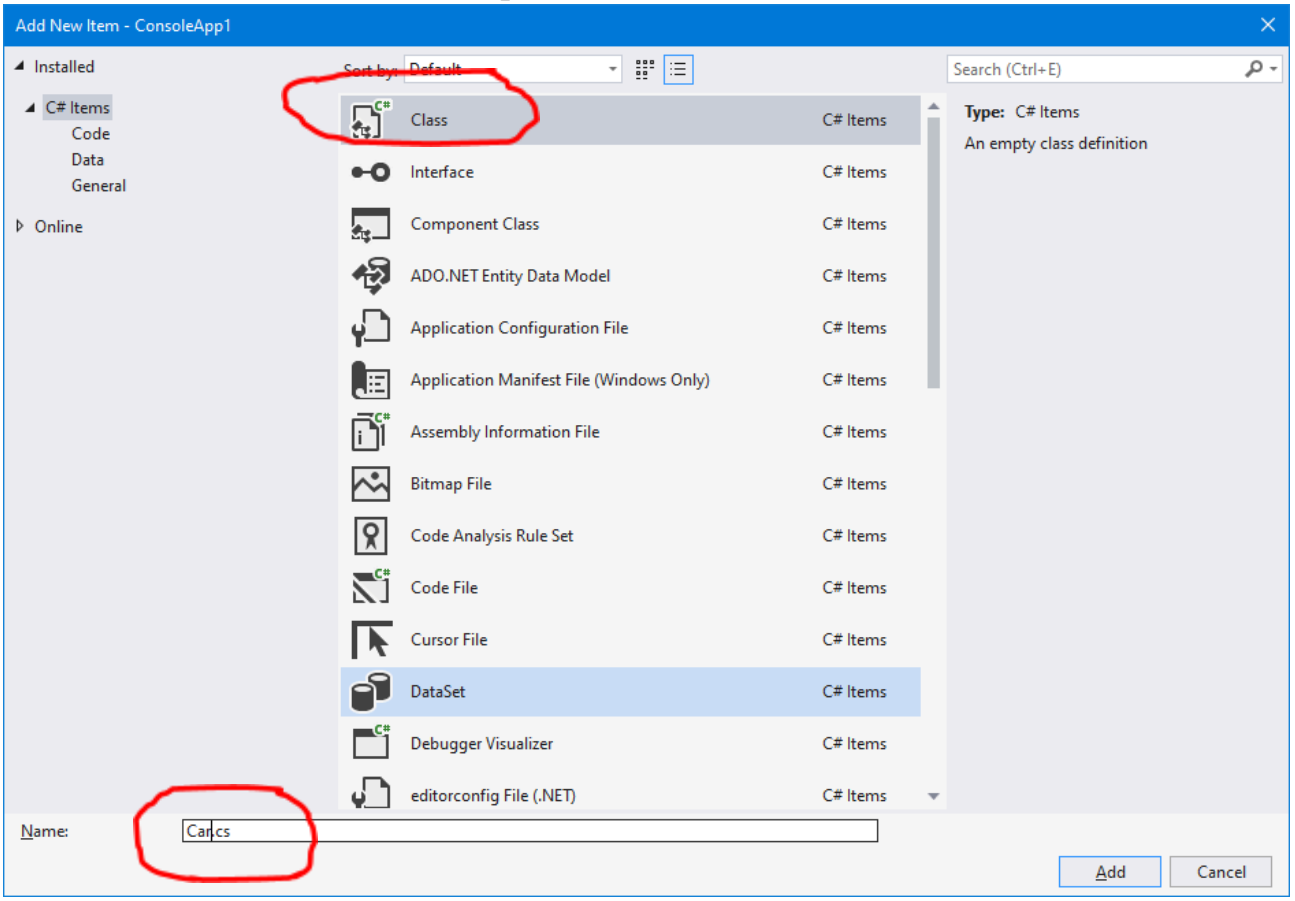
Наприклад, якщо говорити про автомобіль, то клас Car, призначений для використання в програмі автоматизації роботи автопідприємства, може містити такі поля: Model (рядок), Volume (double), Number (рядок), Mileage (int) – модель, об'єм двигуна, реєстраційний номер, пробіг. Також такий клас у найпростішому вигляді може мати метод Print без параметрів – для друку даних.

Поля кожного об'єкту (екземпляру класу) належать лише цьому об'єкту і характеризують його стан (наприклад, модель конкретної машини, реєстраційний номер). Методи класу призначені для обробки даних об'єкту та забезпечення його взаємодії з іншими об'єктами, операційною системою та користувачем.

Для того, щоб додати до проекту клас, потрібно в меню Visual Studio вибрати пункт меню «Project\Add class…»:



Visual Studio виведе на екран вікно,



в якому слід вибрати пункт Class і ввести назву файлу класу (у прикладі це Car.cs). В результаті у проект буде додано відповідний файл із пустим класом Car. Додавши поля і методи класу, отримаємо такий опис класу:

```
internal class Car
{
    public string Model = "Unknown";
    public double Volume;
    public string Number;
    public int Mileage;
    public void Print()
    { Console.WriteLine($"Автомобіль: {Model}, V={Volume}л,"+
                        $" Пробіг {Mileage}, Номер {Number}"); }
}
```

Visual Studio за замовчуванням створює клас із модифікатором доступу **internal**. Це означає, що клас Car буде доступний лише всередині збірки – програми, в якій його оголошено. Поглянемо уважніше на поля класу.

По-перше, поля класу можуть бути ініціалізовані явно, як, наприклад, Model. Якщо поля явно не ініціалізовано, вони отримують значення за замовчуванням залежно від типу (0, null, пустий рядок…).

По-друге, перед полями і методами класу, як і перед самим класом, може бути вказано **модифікатор доступу**. У C# можна застосувати такі модифікатори доступу:

public: Тип або член класу видимий у збірці, де його оголошено, та в будь-якій збірці, що посилається на неї.

private: Тип або член класу видимий лише для коду всередині самого класу або структури.

protected: Тип або член класу видимий лише для коду всередині самого класу, або класу, що походить від нього (створений на його базі).

internal: Тип або член класу видимий у збірці, де його оголошено, але не в іншій збірці.

protected internal: Тип або член класу видимий у збірці, де його оголошено, або з похідного класу в іншій збірці.

private protected: Тип або член класу видимий для класу, що походить від нього, і розташований у тій самій збірці.

Якщо модифікатор опущено, за замовчуванням поля і методи класу отримують модифікатор доступу **private**, а клас – модифікатор **internal**.

Методи класу мають доступ до всіх його полів незалежно від модифікатора доступу, а також до public, protected та protected internal полів класів-предків (класів, від яких клас походить).

Для використання класу слід створити об’єкт. Створення об’єкту виконується так:

```
<ім’я об’єкту> = new <конструктор класу>(<параметри_конструктора>);
```

У даному випадку викликається **конструктор** класу – спеціальний метод, який, як і в C++, слугує для ініціалізації об’єкту. Конструктор класу завжди називається так само, як клас. Якщо конструктор у класі не описано, компілятор автоматично створює **конструктор за замовчуванням**, який не приймає жодних параметрів і не виконує жодних дій із ініціалізації об’єкту.

Після створення об’єкту класу можна отримати доступ до його полів та викликати методи:

```
Car Car1 = new Car();

//Ініціалізуємо об'єкт
Car1.Model = "Ferrari 512 TR";
Car1.Volume = 4.9;
Car1.Number = "AA 1234 BB";
Car1.Mileage = 10000;
//Друк даних об'єкта
Car1.Print();
```

```
//Збільшимо пробіг
Car1.Mileage += 100;
//Друк даних об'єкта
Car1.Print();
```

Створення об’єкту і подальша ініціалізація полів у тексті програми незручна, оскільки ускладнює її сприйняття. Тому, для ініціалізації об’єкту створюються конструктори. Таких конструкторів може бути декілька – із різними наборами аргументів, причому в конструкторах можна використовувати параметри за замовчуванням:

```
internal class Car
{
    public const string type = "Car";
    public string Model = "Unknown";
    public double Volume;
    public string Number;
    public int Mileage;

    public Car() {
        Volume = 1.0;
        Number = "Unknown";
        Mileage = 0;
    }
    public Car(string model, double volume)
    {
        Model = model;
        Volume = volume;
        Number = "Unknown";
        Mileage = 0;
    }
    public Car(string model, double volume=1.0,
        string number="Unknown", int mileage=0)
    {
        Model = model;
        Volume = volume;
        Number = number;
        Mileage = mileage;
    }
    public void Print()
    { Console.WriteLine($"Автомобіль: {Model}, V={Volume}л,"+
                        $" Пробіг {Mileage}, Номер {Number}"); }
}
```

Клас Car у наведеному вище прикладі має три конструктори: перший не приймає параметрів, другий дає змогу вказати модель і об’єм двигуна автомобіля, третій має чотири параметри для ініціалізації усіх полів класу, причому об’єм двигуна, реєстраційний номер і пробіг мають значення за замовчуванням.

Якщо в класі оголошено хоча б один конструктор, конструктор за замовчуванням компілятором не створюється. При виклику конструктора класу, як і будь-якого іншого методу, можна застосовувати іменовані параметри:

```
Car Car2 = new Car(model: "Lamborghini Aventador", volume: 6.5 );

Car2.Print();
```

Починаючи із C# 8, при створенні об’єкта можна опускати ідентифікатор конструктора:

```
Car Car3 = new(model: "Lamborghini Aventador", volume: 6.5);
```

Для того, щоб всередині метода класу отримати доступ до полів об’єкту, служить ключове слово **this**. Це ключове слово є посиланням на конкретний об’єкт, для якого було викликано метод. Як правило, використання this необхідне у випадках, коли параметри методів та імена полів співпадають:

```
public Car(string Model, double Volume)
{
    this.Model = Model;
    this.Volume = Volume;
    Number = "Unknown";
    Mileage = 0;
}
```

Крім того, this використовується, якщо потрібно пов’язати два об’єкти, один з яких повинен містити посилання на інший. Наприклад, метод NewBuilder

```
internal class Table
{
    ...
    public Builder NewBuilder() { return CreateBuilder(this); }
```

створює об’єкт класу Builder шляхом виклику CreateBuilder(this), пов’язуючи таким чином новостворений об’єкт класу Builder з об’єктом класу Table, метод якого власне й було викликано.

Для ініціалізації створених об’єктів класів можна використовувати **ініціалізатори**. Ініціалізатори забезпечують передачу значень доступним полям і властивостям об’єкта; список полів зі значеннями вказують у фігурних дужках через кому:

```
Car Car4 = new Car { Model = "AAA", Volume = 0, Number = "___", Mileage = 10 };
```

При використанні ініціалізатора слід враховувати, що:

1) За допомогою ініціалізатора можна встановити властивості лише доступним поза класом полям і властивостям – наприклад, значення protected поля встановити не вийде;
2) Ініціалізатор завжди виконується ***після*** конструктора, і тому значення, встановлені конструктором, буде замінено значеннями, вказаними в ініціалізаторі.

У наведеному вище прикладі, модель буде виставлено рівною «AAA», об’єм двигуна рівним 0 і т.д.

Крім полів, клас може зберігати також константи:

```
internal class Car
{
    public const string type = "Car";
    public string Model = "Unknown";
     ...
```

Значення таких констант змінювати в тексті програми після їх оголошення не можна. До того ж, дані, що зберігаються в цих константах, належать не до створених об'єктів, а до класу взагалі, для доступу до них об'єкт класу створювати не потрібно. Для звернення до таких констант можна скористатися зверненням <клас>.<ім'я константи>:

```
Console.WriteLine(Car.type);
```

5.2. Структури (struct). Конструктори в структурах. Відмінності структур та класів.

Крім класів, C#, як і його предок C++, підтримує також структури. Багато примітивних типів (наприклад, int та інші цілочисельні типи, float, double та інші) є структурами.

Для оголошення структури використовують такий синтаксис:

```
struct <назва структури>
{
   [<поля>]
   [<методи>]
     ...
}
```

<Назва структури>, вказана після ключового слова struct, є типом структури. Всі створені в програмі об'єкти структури матимуть цей тип. Як і класи, структури можуть містити поля і методи, причому починаючи з C# 10, поля структур можна ініціалізувати безпосередньо при оголошенні. Наприклад, клас Car можна перевизначити в структуру, просто замінивши в оголошенні internal class на struct:

```
struct CarStruct
{
    public string Model = "Unknown";
    public double Volume;
    public string Number;
    public int Mileage;

    public void Print()
    {
        Console.WriteLine($"Автомобіль: {Model}, V={Volume}л," +
                          $" Пробіг {Mileage}, Номер {Number}");
    }
}
}
```

Створення структур, використання їх полів і методів аналогічно класам.

Як і клас, структура може мати конструктори. Якщо в структурі визначається конструктор, в ньому ***обов'язково*** слід ініціалізувати всі поля структури. Починаючи з C# 10, можна створити конструктор структури, який не буде приймати жодних параметрів.

Для ініціалізації структури можна використовувати ініціалізатори. Але крім цього, можна присвоїти значення однієї структури іншій із заміною значень полів за допомогою ключового слова **with**:

```
CarStruct CarSt1 = new CarStruct();

//Ініціалізуємо структуру
CarSt1.Model = "Ferrari 512 TR";
CarSt1.Volume = 4.9;
CarSt1.Number = "AA 1234 BB";
CarSt1.Mileage = 10000;

CarStruct CarSt2 = CarSt1 with { Number = "BB 0000 CC", Mileage = 12000 };
```

В чому ж полягають **відмінності між класом і структурою**?

1. Змінні структур зберігають не посилання на об'єкт, а сам об'єкт. Тобто структура – це тип-значення, а клас – це тип-посилання.

2. При присвоєнні одного об'єкта структури іншому об'єкту буде скопійовано всі значення полів структури, а не посилання на об'єкт, як у випадку з класами. Аналогічною ситуація буде з передачею об'єктів структур у методи – вони за замовчуванням передаються за значенням, а класи за посиланням (навіть якщо не вказано ключове слово ref).

3. На відміну від класів, змінна структури зберігає дані, а не посилання на них, і зберігає їх в стеці, а не в купі (heap). Тому доступ до полів структур здійснюється швидше.

4. Структури не підтримують успадкування, на відміну від класів, тому їх члени можуть мати тільки модифікатори доступу public, private або internal.

5. Як і клас, структура може мати конструктори. Однак, якщо в структурі визначається конструктор, в ньому обов'язково потрібно ініціалізувати всі поля структури.

6. Об'єкт структури можна створити і без конструктора, на відміну від об'єкта класу:

```
Painting Joconde;
Joconde.Author = "Da Vinci";
struct Painting
{
    public string Author;
    public int Year;
}
```

7. Якщо в структурі є визначені конструктори, то все одно може бути викликаний конструктор за замовчуванням (конструктор без параметрів):

```
CarStruct CarSt3 = new();
```

Результатом такого виклику стане ініціалізація полів структури значенням за замовчуванням.

5.3. Кортежі.

Кортеж дає змогу швидко згрупувати кілька елементів даних у спрощеній структурі. Варіанти визначення кортежів показано нижче:

```
//Варіант 1 – поля з іменами за замовчуванням
(double, int) t1 = (5.5, 2);
Console.WriteLine($"Кортеж з елементами {t1.Item1} і {t1.Item2}.");
```

```
//Варіант 2 – з визначенням імен полів
(double Sum, int Count) t2 = (5.5, 2);
Console.WriteLine($"Сума {t2.Count} елементів рівна {t2.Sum}.");
```

Для визначення типу кортежу слід вказати типи всіх його полів, і, за необхідності, їх імена. Звернення до імен кортежу аналогічне зверненню до полів структур і класів. Типи кортежів підтримують оператори == і !=, рівними вважають кортежі з однаковими значеннями полів; імена полів при цьому не враховуються.

Кортежі можна присвоювати один одному, якщо вони містять однакову кількість елементів і типи відповідних елементів кортежів співпадають, або ж тип присвоюваного елемента кортежу може бути неявно перетворений на тип елемента, якому присвоюється значення.

Найчастіше кортежі використовують як тип, який повертає метод:

```
(int min, int max) ArrMinMax(int[] input)
{
    if (input is null || input.Length == 0)
    {
        return (0, 0);
    }

    var min = input[0];
    var max = input[0];
    foreach (var i in input)
    {
        if (i < min)
        {
            min = i;
        }
        if (i > max)
        {
            max = i;
        }
    }
    return (min, max);
}
```

Із повернутим методом кортежем можна працювати як з єдиною змінною або деконструювати його в окремі змінні:

```
//limits.min відповідає minimum, limits.max – maximum
var limits = ArrMinMax(xs);
var (minimum, maximum) = ArrMinMax(xs);
```

5.4 Перезавантаження методів.

Часто виникає необхідність перевизначити метод класу так, щоб він працював із різними наборами параметрів. Як і в C++, у C# реалізовано перезавантаження методів, причому компілятор визначає перезавантажені методи класу автоматично. Для того, щоб розрізнити методи з однаковими іменами, компілятор використовує **сигнатуру**. **До сигнатури метода входять:**

1) ім’я (ідентифікатор) методу;
2) кількість параметрів методу;
3) типи параметрів;

4) порядок параметрів;
5) модифікатори параметрів (ref, in, out).

Імена параметрів і тип результату методу до сигнатури не входять.

Приклад. Оголошення методів, призначених для пошуку і видалення файлу за певним набором даних

```
void KillFile(string name) { }
void KillFile(string name, int Attrributes) { }
void KillFile(string name, string Content) { }
void KillFile(in int Size) { }
void KillFile(int AccessCount) { }
```

мають різні сигнатури, тому фрагмент без проблем компілюється. Однак, якщо спробувати додати метод

```
void KillFile(int ID) { }
```

виникне помилка компіляції, оскільки цей метод має таку ж сигнатуру, як і void KillFile(int AccessCount).

Виклик перезавантажених методів нічим не відрізняється від виклику інших методів.

5.5 Область видимості змінних. Іще раз про простори імен.

Кожна змінна в програмі доступна у межах певної **області видимості**; поза нею змінної вже немає. В C# існують такі області видимості:

Контекст класу. Змінні, визначені на рівні класу (поля і константи класу), доступні у будь-якому методі цього класу.

Контекст методу. Змінні, визначені на рівні методу, є локальними і доступні лише в рамках даного методу.

Контекст блоку коду. Змінні, визначені на рівні блоку коду (наприклад, внутрішні змінні циклу), є локальними і доступні тільки в рамках даного блоку.

Локальні змінні, визначені в методі або блоці коду, приховують змінні вищого рівня (класу, методу), якщо збігаються їх ідентифікатори.

```
class Demo
{
    public int i=1;

    public void PrintGlobal()
    {
        Console.WriteLine(i); //виведе 1
    }

    public void PrintLocal()
    {
    int i=3;

        Console.WriteLine(i); //виведе 3
    }

    public void Print()
    {
        //int i=3; // не відкомпілюється!
```

```
            Console.WriteLine(i); //виведе 1

            for (int j=0;j<5; j++)
            {
                int i = j * 2;
                Console.WriteLine(i); //виведе 0, 2, 4 ...
            }
        }
    }
```

В результаті виконання коду

```
Demo D = new Demo();
D.PrintGlobal();
D.PrintLocal();
D.Print();
```

в консоль буде виведено:

Перший виклик методу PrintGlobal() виведе 1, тому що виводиться значення з поля класу. Другий виклик PrintLocal() виведе 3 – значення локальної змінної. Виклик Print() виведе спершу 1 (значення глобальної змінної), а потім у циклі значення добутку i*2 (числа 0, 2, ... 8), де i – локальна змінна циклу for.

Слід зазначити, що в C# заборонено оголошувати змінні/методи з однаковими ідентифікаторами у вкладених локальних областях видимості. Як наслідок, метод Demo.Print() з оголошенням локальної змінної int i=3; не відкомпілюється, адже в методі і блоку коду всередині методу не можна оголосити змінні з однаковим ідентифікатором.

Тепер поговоримо про видимість класів і простори імен. Видимість класів визначається модифікаторами доступу, описаними вище.

Класи та інші типи в .NET для зручності поміщаються у спеціальні контейнери – простори імен. Простори імен дають змогу організувати програми в логічні блоки, відокремивши від решти програми код, який виконує певну задачу або містить пов’язані логічно об’єкти.

Синтаксис визначення простору імен такий:

```
namespace <ім'я простору імен>
{
   // Вміст простору імен
}
```

<Ім'я простору імен> повинно відповідати тим же умовам, що й інші ідентифікатори C#. Простори імен можуть містити різні типи, включаючи класи

і структури. Одні простори імен можуть містити інші. Простір імен може міститися як в одному файлі, так і бути розподіленим по кількох .cs файлам.

Якщо клас визначено в просторі імен, для звернення до класу слід використати повне ім’я класу, формат якого <ім'я простору імен>.<ім’я класу>. Так, якщо клас Car визначено в просторі імен Automotive, то для звернення об’єкту слід використовувати Automotive.Car:

```
Automotive.Car Car2 = new();
```

Повне ім’я класу з вказуванням простору імен ускладнює сприйняття коду, особливо коли використовується багато класів. Для спрощення тексту програми слід вказати директиву використання простору імен using <ім’я простору імен>:

```
using Automotive;

Car Car2 = new();
```

Починаючи з .NET 6 і C# 10 можна визначати простори імен на рівні файлу, вказавши namespace <ім'я простору імен>; на початку файлу:

```
namespace Automotive;

class Tractor
{
   string name;
   public Tractor(string name) => this.name = name;
   public void Print() => Console.WriteLine($"Трактор: {name} ");
}
```

Якщо необхідно використовувати простір імен у всіх файлах проекту, за замовчуванням потрібно його підключати до всіх файлів, де ми плануємо його використовувати. У C# є можливість підключити простір імен, як глобальний, директивою global using <простір імен>, помістивши її в будь-який .cs файл проекту.

Для зручності можна зібрати підключення усіх глобальних просторів імен в окремий .cs файл і в ньому визначити набір просторів імен, які повинні бути доступними у всіх файлах проекту:

```
global using System.Text;
global using System.Reflection;
global using Automotive;
```

Запитання для самоперевірки

1. Що таке клас і об’єкт?
2. Як оголосити клас у C#?
3. Що таке поля і методи класу? Яке їх призначення?
4. Які існують модифікатори доступу в C#?
5. Що таке конструктор класу і навіщо він потрібен? Коли компілятор створює конструктор за замовчуванням?

6. Навіщо в C# використовують ключове слово this?

7. Що таке ініціалізатор і як його використовувати для ініціалізації полів об’єкту?

8. Що таке структура в C#? Які відмінності між структурами і класами?

9. Як можна присвоювати структури одну іншій? Для чого потрібне ключове слово with?

10. Що таке кортеж? Як можна його використовувати в програмі?

11. Як у C# перезавантажити методи? Що входить до сигнатури метода?

12. Які існують області видимості змінних у C#? Що таке простори імен і як ними користуватися?

Питання з теми, що виносяться на самостійне опрацювання

1. Використання кортежів.
2. Використання просторів імен.

Лекція 6. Властивості та інкапсуляція. Автоматичні властивості. Статичні члени, статичний конструктор, статичні класи. Перезавантаження операторів.

6.1 Властивості та інкапсуляція. Автоматичні властивості. Ініціалізація автовластивостей.

Інкапсуляція – одна з основ об’єктно-орієнтованого підходу до програмування. Цей термін означає, що об’єкт вміщує як дані, так і правила їх обробки (методи), а доступ до стану об'єкта (тобто до всіх його полів) ззовні напряму заборонено. Як наслідок, із програми, що користується об’єктом, з даними цього об’єкту можна взаємодіяти лише через відкриті поля та методи. Використання інкапсуляції дає змогу програмісту контролювати звернення до даних класів і правильність їх ініціалізації, а також запобігти помилкам, пов'язаним із неправильними викликами методів.

Обмеження доступу до полів та методів класу забезпечується використанням модифікаторів доступу, детально розглянутих у попередній лекції. Зверніть увагу на те, що за замовчуванням поля і методи класу отримують модифікатор доступу **private** – тобто вони не доступні (невидимі) поза межами класу.

На відміну від класичного C++, в C# у класах крім полів можуть застосовуватися **властивості**. Концепція властивостей запозичена авторами C# з мов Visual Basic та Object Pascal (Delphi). Вона полягає в тому, що, на відміну від полів, властивості об’єкта при присвоєнні їм значення та читанні їх значення можуть ще виконувати певні дії. Наприклад, якщо клас – це кнопка, то вона може містити властивість – напис, і при зміні тексту напису кнопка перемалює себе автоматично. Це дуже зручно при створенні інтерфейсу користувача (і не тільки), оскільки:

1) позбавляє програміста необхідності слідкувати за зміною значень полів об’єкта для програмування реакції на них (наприклад, щоб перемальовувати кнопку викликом методу);
2) спрощує код програми і можна зосередитися не на програмуванні рутинних задач, на зразок перемальовування кнопок, а на написанні коду, який реалізовує функціональність програми;
3) дає змогу контролювати доступ до полів об’єкта, причому, можна легко контролювати значення властивості та/або зробити її доступною тільки для читання або тільки для запису. Використовуючи лише поля, таке обмеження або контроль даних зробити неможливо.

Синтаксис оголошення властивості в C# такий:

```
[<модифікатори>] <тип властивості> <назва властивості>
{
   get
   {
      <дії при отриманні значення властивості>
   }
   set
   {
      <дії при встановленні значення властивості>
```

```
            }
        }
```

До визначення властивості, крім вже знайомих модифікаторів, типу і імені, входять два блоки так званих **аксесорів (методів доступу)**, get і set. Перший метод називають гетером, другий сетером.

У блоці get (гетері) виконуються дії для отримання значення властивості, яке повертається за допомогою оператора return. У блоці set (сетері) виконуються дії для встановлення нового значення властивості, які обов'язково включають в себе присвоєння значення властивості внутрішньому полю об'єкта. Також, за необхідності, у сетері реалізують поведінку об'єкту при спробі зміни значення властивості. Наприклад, клас Car із попередньої лекції можна переписати так:

```
    internal class Car
    {
        //Поля – доступ лише в класі
        private string fModel;
        private double fVolume;
        private string fNumber;
        private int fMileage;

        //Властивості
        public double Volume
        {
            get { return fVolume; }
            set
            {
                if ((value <= 0.0) || (value > 100.0))
                    Console.WriteLine("Об'єм двигуна повинен бути більше нуля і до
100 л!");
                else
                    fVolume = value;
            }
        }
             ...
```

У цьому випадку всі поля класу Car мають модифікатор private, тобто недоступні за межами класу. До поля fVolume організовано доступ через властивість Volume: при зверненні до цієї властивості на читання (наприклад, для виведення або присвоювання якісь змінній) гетер повертає значення поля, а при запису (при присвоюванні властивості нового значення) виконується код контролю значення. Сетер властивості Volume присвоює передане сетеру значення value полю fValue, якщо value >0 і <=100, а при виході присвоюваного значення (value) за межі діапазону (0;100] видає повідомлення про помилку. Аналогічно, для класу кнопки в сетері властивості напису може бути запрограмовано автоматичну зміну розміру кнопки та її перемальовування на екрані.

Якщо в оголошенні властивості опущено гетер (get), то властивість стає доступною тільки для запису, якщо сетер (set) – тільки для читання.

Властивості можуть бути не пов'язаними з полями об'єкта напряму: значення властивості можна обчислювати на основі певного алгоритму або

виразу (так звані обчислювані властивості). Наприклад, залежно від об'єму двигуна можна обчислити і видати в гетері ставку податку на авто:

```
...
        public double TaxPercent //податок в процентах
        {
            get
            {
                switch (fVolume)
                {
                    case < 1.5: return 5;
                    case < 2: return 10;
                    case < 4: return 15;
                    default: return 20;
                }
            }
...
```

Обчислювані властивості можуть обчислюватися на основі відразу кількох полів, наприклад

```
public string EnergyConsumed
{
    get { return fPower*fOperationTime; }
}
```

Зрозуміло, обчислювані властивості – це завжди властивості тільки для читання.

Модифікатор доступу можна вказати не лише для всієї властивості, а й для гетера та сетера окремо. Наприклад, оголошення властивості

```
public string Number
{
    get { return fNumber; }
    private set { fNumber = value; }
}
```

робить її доступною для всіх класів на читання і лише всередині свого класу та його нащадків – на запис. При використанні модифікатора гетера та сетера слід враховувати такі обмеження:

1) модифікатор блоку set або get можна встановити лише якщо властивість має обидва блоки (і set, і get);
2) мати модифікатор доступу може лише один блок – або get, або set;
3) модифікатор доступу гетера/сетера повинен бути більш обмежуючим, ніж модифікатор доступу властивості. Наприклад, якщо властивість має модифікатор protected, зробити її блок get public не вийде (станеться помилка компіляції).

Для того, щоб спростити життя програмісту у випадку, коли в об'єкті багато полів, у C# запропоновано **автоматичні властивості**. Автоматичні властивості позбавляють необхідності описувати в оголошенні класу поля. Такі властивості мають скорочене оголошення:

```
internal class Car
{
```

```
    //Автоматичні властивості
    public string Model { get; set; } = "Unknown";
    public double Volume { get; set; }
    public string Number { get; set; }
    public int Mileage { get; set; }

    public Car(string model, double volume = 1.0,
        string number = "Unknown", int mileage = 0)
    {
        Model = model;
        Volume = volume;
        Number = number;
        Mileage = mileage;
    }

    public void Print()
    {
        Console.WriteLine($"Автомобіль: {Model}, V={Volume}л," +
                          $" Пробіг {Mileage}, Номер {Number}");
    }
}
```

У даному випадку поля, гетери і сетери автоматичних властивостей автоматично генеруються компілятором. Автоматично створений гетер повертає значення відповідного поля, а сетер його встановлює. Перевагою автоматичної властивості є те, що за необхідності її можна розгорнути у повноцінну властивість, додавши поле і необхідну логіку. Автоматична властивість повинна мати і гетер, і сетер; при цьому їй можна присвоїти значення за замовчуванням (властивість Model у прикладі). Якщо не вказати значення для ініціалізації автоматичних властивостей, вони отримають значення за замовчуванням.

Можна прибрати в оголошенні авто властивості блок set і зробити її доступною лише для читання. В цьому випадку початкове значення властивості слід задавати або в конструкторі, або при ініціалізації властивості:

```
class Tractor
{
    // Через ініціалізацію властивості
    public string Name { get; } = "XT3-240";
    // Через конструктор
    public Tractor(string name) => Name = name;
}
```

Починаючи з C# 9.0, сетер властивості можна визначити за допомогою ключового слова **init**. Для встановлення значення властивості з init можна використовувати ініціалізатор або конструктор, або при оголошенні вказати значення такої властивості. Різниця між властивістю з init та властивістю тільки для читання полягає в тому, що значення init-властивості ми також можемо встановити в ініціалізаторі (значення властивості для читання встановити в ініціалізаторі не можна):

```
// Через ініціалізатор
Tractor tractor = new() { Name = "Caterpillar D7E" };
class Tractor
{
    // Через ініціалізацію властивості
    public string Name { get; init; } = "XT3-240";
    // Через конструктор
```

```
        public Tractor(string name) => Name = name;
        public Tractor() { }
    }
```

Властивості можна записувати зі скороченим записом гетера і сетера:

```
class Tractor1
{
    string name;
    public string Name
    {
        get => name;
        set => name = value;
    }
}
```

Можна також скоротити запис властивості лише для читання:

```
class Tractor2
{
    string name;
    public string Name => name;
}
```

Зверніть увагу, що в наведених вище прикладах властивість Tractor1.Name доступна і для читання, і для запису, а Tractror2.Name – лише для читання (у цьому випадку сетер не визначено).

6.2 Статичні члени і модифікатор static. Статичний конструктор. Статичні класи.

Класи та структури в C# можуть мати статичні поля, методи та властивості. Статичні поля, методи та властивості належать до всього класу/типу структури, і характеризують клас як такий (або ж структуру як таку), а не об’єкти (екземпляри) цього класу/структури. Для звернення до статичних членів класу/структури необов’язково створювати об’єкт класу/структури. Статичні поля, методи та властивості мають модифікатор **static**.

Статичне поле містить дані, спільні для всього класу. Як приклад, для автомобіля таким полем може бути норма викидів шкідливих речовин. У самому класі таке поле використовується так само, як і будь-яке інше:

```
class Car
{
        //Поля
        //Норма забруднюючих речовин
        static public double PollNorm = 20;
        //Викид забруднюючих речовини для конкретної машини
        double Poll;

        public bool PassesPollTest()
        {
            return Poll <= PollNorm;
        }
```

Для звернення до статичного поля не потрібно створювати об’єкт класу. Достатньо звернутися до такого поля через ідентифікатор класу: Car.PollNorm. Аналогічно визначаються статичні властивості:

```
static double pollnorm = 20;
static public double PollNorm => pollnorm;
```

Статичні методи визначають спільну для всіх об’єктів класу поведінку, яка не залежить від конкретного об’єкта. Звернення до статичних методів відбувається через ім’я класу:

```
class Car
{
        //Поля
        //Норма забруднюючих речовин
        static public double PollNorm = 20;
        //Викид забруднюючих речовини для конкретної машини
        double Poll;

        static public bool PassesPollTest(Car car)
        {
            return car.Poll <= PollNorm;
        }
        ...
}

Car Ferrari=new() { Poll=10; };
if (Car.PassesPollTest(Ferrari))
   Console.WriteLine("Пройде тест!");
```

Звертатися до полів поточного об’єкта класу (через this) в статичних методах заборонено; можна звертатися тільки до статичних членів класу. При цьому статичні методи можуть приймати як параметри значення і змінні будь-яких типів, включаючи і об’єкти класу, до якого статичний метод належить, як це зроблено у методі static public bool PassesPollTest(Car car).

Крім звичайних конструкторів, у класі можуть бути визначені статичні конструктори. Статичні конструктори мають такі властивості:

1) Статичні конструктори не повинні мати модифікатора доступу і не приймають параметри.
2) Звертатися до полів поточного об’єкта класу (через this) в статичних конструкторах заборонено; можна звертатися лише до статичних членів класу.
3) Статичні конструктори не можна викликати у програмі вручну. Вони виконуються автоматично при першому створенні об’єкта даного класу або при першому зверненні до його статичних членів (якщо такі є).

Статичні конструктори використовують для ініціалізації статичних даних класу, або для виконання дій, які для даного класу потрібно виконати лише один раз.

```
static public double PollNorm { get; set; };
static Car()
{
    if (DateTime.Now.Year >= 2022)
        PollNorm = 15;
    else
        PollNorm = 20;
}
```

У наведеному вище прикладі в статичному конструкторі залежно від поточного року встановлюється різна норма забруднення. Зверніть увагу на те, що статичний конструктор, як вже сказано вище, буде виконано лише при першому створенні об'єкту даного класу або при першому зверненні до статичних членів класу. Якщо такого звернення не буде, то й виконуватися статичний конструктор теж не буде.

Статичні класи в C# оголошуються з модифікатором static і можуть містити лише статичні поля, властивості та методи. Створити об'єкт статичного класу неможливо, тому фрагмент коду

```
static class Test
{
    static void DoTest1(int x) { }
    static void DoTest2(double y) { }
}

Test t = new(); //Помилка! Не відкомпілюється.
```

не відкомпілюється. Статичний клас в C# не підтримує успадкування: породити новий клас від статичного класу неможливо. Також статичний клас сам не може бути успадкованим від будь-якого класу чи інтерфейсу, крім класу Object, що є базовим для всіх класів C#.

На практиці статичні класи використовують для реалізації класів, для яких необов'язково тримати стан об'єкту у внутрішніх змінних (полях). Типовий приклад статичного класу C# – клас System.Math, який містить методи, що реалізують обчислення математичних функцій аргументів, заданих як параметри його методів, і не має необхідності зберігати свій стан.

Поля будь-якого класу можуть мати також модифікатор **readonly**. Цей модифікатор явно вказує, що поле класу доступне лише для читання (хоча його й можна ініціалізувати). На відміну від полів const, поля readonly можуть бути ініціалізовані кілька разів – в оголошенні поля і в будь-якому конструкторі класу. Таким чином, значення поля readonly залежать від застосованого конструктора, і належать конкретному об'єкту класу (на відміну від статичних полів). Такі поля можна використовувати для зберігання констант, заданих під час виконання програми. Наприклад, поле readonly можна використати для фіксації номеру пакету в протоколі обміну:

```
class Packet
{
    public readonly int PacketNumber = 0;
    byte[] Data;

    public Packet(int PacketNumber, byte[] Data)
    {
        this.PacketNumber = PacketNumber;
        this.Data = new byte[Data.Length];
        for (int i = 0; i < Data.Length; i++)
            this.Data[i] = Data[i];
    }
}

    byte[] a = { 1, 2, 3 };
```

```
Packet Packet1 = new Packet(1, a);
Packet1.PacketNumber =5; //Помилка! Неможливо змінити readonly поле!
```

Зміна значення readonly поля поза межами ініціалізаторів полів та конструкторів неможлива – причому як у методах класу, так і у програмах, що клас використовують, що й ілюструє наведений вище приклад. Поля readonly можуть бути статичними.

6.3 Перезавантаження операторів.

Поряд із методами у класах та структурах можна також визначати оператори, тобто перезавантажувати оператори для об'єктів класу чи структури.

Наприклад, у нас є клас, що визначає покази електролічильника:

```
class EMeterOut
{
    public double Value { get; set; };
}
```

Для практичного використання такого класу було б зручно мати можливість порівняти значення показів (для визначення того, хто більше споживає енергії) та додавати спожиту енергію для визначення сумарного споживання. Для цього в C# передбачена можливість перезавантажити оператори. У даному випадку слід перезавантажити оператори порівняння та додавання.

Визначення оператора полягає в тому, що в класі (аналогічно C++) визначається спеціальний метод

```
public static <тип результату> operator <оператор>(<параметри>)
 {
   ...
 }
```

Для бінарних операторів задаються два параметри, для унарних – один. Тип результату бінарного оператора може відрізнятися від класу, наприклад допустимо обчислити суму EMeterOut і double, причому результатом бінарного оператора буде double.

У C# можна перезавантажити такі оператори:

1) унарні оператори +x, -x, !x, ~x, ++, –, true і false;
2) бінарні оператори +, -, *, /, % ;
3) операції порівняння ==, !=, <, >, <=, >= ;
4) порозрядні (побітові) оператори &, |, ^, <<, >> .

Є кілька операторів, які необхідно визначати парами:

1) true і false (вказують, що вираз має значення true або false);
2) == та != ;
3) < і > ;
4) <= та >= .

Неможливо перезавантажити операцію (оператор) присвоювання = або тернарний оператор ?:, а також низку інших. Повний список операторів, які можна перезавантажити, наведено тут:

https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/operator-overloading#overloadable-operators

Перезаватаження операторів && та || у C# не передбачено; щоб їх використовувати, слід перезавантажити для класу оператори &, |, true і false.

На практиці перезавантаження операторів <, > і + для класу EMeterOut виглядатиме так:

```
class EMeterOut
{
    public double Value { get; set; };

    public static EMeterOut operator +(EMeterOut c1, EMeterOut c2) =>
        new EMeterOut { Value = c1.Value + c2.Value };

    public static bool operator >(EMeterOut c1, EMeterOut c2) =>
        c1.Value > c2.Value;

    public static bool operator <(EMeterOut c1, EMeterOut c2) =>
        c1.Value < c2.Value;

}
```

Як видно із наведеного прикладу, при перезавантаженні операторів, що повертають як результат клас, слід обов'язково створювати новий об'єкт класу (new EMeterOut), який і отримає результат роботи оператора. Метод, що реалізує перезавантаження оператора, можна записувати в скороченій формі.

Використання об'єкту з перезавантаженими операторами виглядатиме так:

```
EMeterOut Meter1 = new() { Value = 1000 };
EMeterOut Meter2 = new() { Value = 1250 };

Console.WriteLine((Meter1 + Meter2).Value); //2250
if (Meter1 < Meter2)
    Console.WriteLine("Другий лічильник більше"); //Це буде виведено
else
    Console.WriteLine("Перший лічильник більше");
```

У даному прикладі обчислюється і виводиться на екран сума показів двох лічильників, а також порівнюються їх покази. Зверніть увагу на вираз (Meter1 + Meter2).Value : тут відбувається звернення до поля .Value створеного методом перезавантаження оператора + об'єкту EMeterOut.

Оператор + можна перезавантажити також для операторів різних типів, наприклад EMeterOut і double:

```
public static double operator +(EMeterOut c1, double d) =>
     c1.Value + d;
public static double operator +(double d, EMeterOut c2) =>
     c2.Value + d;

...

    Console.WriteLine(Meter2 + 23.45); //1273.45
    Console.WriteLine(23.45 + Meter2); //1273.45
```

Зверніть увагу на те, що для використання операцій Meter2+23.45 і 23.45+Meter2 доводиться перезавантажувати оператор двічі, оскільки різний порядок параметрів EMeterOut та double вимагає написання методів, що реалізують оператор +, із різними сигнатурами.

У коді перезавантажуваного оператора не повинні змінюватися об'єкти, які передаються в оператор через параметри. Наприклад, оператор інкременту EMeterOut повинен бути визначений так:

```
public static EMeterOut operator ++(EMeterOut c1) =>
    new EMeterOut { Value = c1.Value + 0.1 }; //на 0.1 кВт*год
```

Цей оператор коректно працюватиме і для префіксного, і для постфіксного інкременту; на відміну від C++, двічі перезавантажувати оператор інкременту в C# не потрібно. Аналогічно оператору інкременту можна перезавантажити оператор декременту.

Окремо слід згадати оператори true і false. Ці оператори використовують, якщо об'єкт повинен використовуватися як умова (наприклад в операторі if):

```
public static bool operator true(EMeterOut c1) =>
    c1.Value>0;
public static bool operator false(EMeterOut c1) =>
    c1.Value <= 0;
```

Таке визначення уможливлює, наприклад, такий код:

```
if (Meter1)
    Console.WriteLine("Лічильник 1 активний.");
```

Якщо є необхідність використання оператора ! (not), його теж треба перезавантажити:

```
public static bool operator !(EMeterOut c1) =>
    c1.Value <= 0;
```

Умова цього оператора повністю збігається з умовою оператора false.

Запитання для самоперевірки

1. Що таке інкапсуляція? Які модифікатори доступу за замовчуванням мають поля і методи класу?
2. Що таке властивості і які вони мають переваги над полями? Як оголосити властивості? Що таке методи доступу властивостей і навіщо вони потрібні?
3. Що таке обчислювана властивість?
4. Які є обмеження на модифікатор доступу гетерів і сетерів?
5. Яке призначення автоматичних властивостей? Як оголосити автоматичну властивість?
6. Навіщо необхідне ключове слово init?
7. Що таке статичні члени класів і навіщо вони потрібні? Які є особливості статичних членів класів?
8. Для чого використовується статичний конструктор?

9. Що таке статичний клас і які його особливості?

10. Перезавантаження операторів у C#. Які особливості написання методів, що реалізують перезавантаження операторів?

Питання з теми, що виносяться на самостійне опрацювання

1. Перезавантаження операторів. Особливості перезавантаження операторів.

Лекція 7. Успадкування. Конструктори в похідних класах, порядок виклику конструкторів. Приведення типів. Оператори is, as. Поліморфізм. Абстрактні класи. Інтерфейси. Делегати.

7.1 Успадкування. Доступ до членів базового класу з класу-спадкоємця. Ключове слово base. Конструктори в похідних класах, порядок виклику конструкторів.

Успадкування є однією з ключових концепцій об'єктно-орієнтованого програмування. Завдяки успадкуванню один клас може успадкувати функціональність іншого класу. У C#, як і в C++, успадкування класу задається синтаксисом:

```
class <ім'я класу> : <ім'я базового класу >
{
    //зміст опису класу
}
```

Як приклад розглянемо клас Car, визначений у лекції 6.

```
class Car
{
    //Поля – доступ лише в класі
    private double fVolume;

    //Автоматичні властивості
    public string Model { get; set; } = "Unknown";
    public string Number { get; set; }
    public int Mileage { get; set; }
   //Властивості
   public double Volume
   {
       get { return fVolume; }
       set
       {
           if ((value <= 0.0) || (value > 100.0))
               Console.WriteLine("Об'єм двигуна повинен бути більше нуля
і до 100 л!");
           else
               fVolume = value;
       }
   }

    public Car(string model, double volume = 1.0,
        string number = "Unknown", int mileage = 0)
    {
        Model = model;
        Volume = volume;
        Number = number;
        Mileage = mileage;
    }

    public void Print()
    {
        Console.WriteLine($"Автомобіль: {Model}, V={Volume}л," +
                          $" Пробіг {Mileage}, Номер {Number}");
    }
}
```

Нам необхідно вести облік автомобілів в автотранспортній компанії, яка експлуатує вантажні автомобілі та автобуси. На основі класу Car можна створити відповідні класи Truck і Bus, які успадковують всі властивості, методи та поля класу Car:

```
class Truck : Car
{
}

class Bus : Car
{
}
```

У C# всі класи за замовчуванням є нащадками класу Object, навіть якщо успадкування не встановлено явно. Як наслідок, всі класи мають крім своїх методів, методи класу Object: ToString(), Equals(), GetHashCode() і GetType().

При успадкуванні класів є кілька обмежень:

1. На відміну від C++, множинне успадкування (успадкування від кількох класів відразу) заборонене.
2. Тип доступу до похідного класу (нащадка) повинен бути таким же або суворішим, ніж у базового класу (предка). Наприклад, якщо базовий клас має тип доступу internal, його нащадок не може мати тип доступу public. Якщо базовий та похідний класи розміщено в різних збірках, похідний клас може успадковуватися лише від класу з типом доступу public.
3. Якщо клас оголошено з модифікатором **sealed**, від нього не можна створювати похідні класи.

Клас-нащадок має доступ тільки до тих членів базового класу, які визначені з модифікаторами **private protected** (якщо базовий та похідний клас визначено в одній збірці), **public, internal** (якщо базовий та похідний клас визначено в одній збірці), **protected та protected internal**. Тому, спроба надрукувати об’єм двигуна автобуса таким кодом

```
class Bus : Car
{
    public void PrintVolume()
    {  //НЕ працюватиме!
        Console.WriteLine($"V={fVolume}л");
    }
}
```

завершиться помилкою компіляції. Для друку замість поля з модифікатором private слід використати властивість Volume, оголошену як public:

```
public void PrintVolume()
{
    Console.WriteLine($"V={Volume}л");
}
```

Додаймо в клас Bus властивість Passengers типу int (кількість пасажирів). Для ініціалізації оголосимо конструктор:

```
class Bus : Car
{
```

```
        public int Passengers { get; set; }
        public Bus(string model, int passengers, double volume = 1.0,
            string number = "Unknown", int mileage = 0) :
              base(model, volume, number, mileage)
        {
            Passengers = passengers;
        }
    }
```

Щоб у похідному класі не повторювати ініціалізацію полів і властивостей базового класу (предка), передбачено ключове слово **base**. Використання його в оголошенні конструктора дає змогу передати значення успадкованих полів/властивостей на встановлення в конструктор базового класу, і встановлювати лише значення полів, оголошені в класі- нащадку.

Зауважимо, що під час успадкування конструктори похідному класу від базового не передаються. Якщо в базовому класі не визначений конструктор без параметрів, а є лише конструктори з параметрами, у похідному класі ми обов'язково маємо викликати один із цих конструкторів через ключове слово base.

Якщо в базовому класі визначено конструктор без параметрів, то в будь-якому конструкторі похідного класу, де немає звернення до конструктора базового класу, цей конструктор викликатиметься неявно. Тому конструктор

```
public Bus(int passengers = 50)
{
    Passengers = passengers;
}
```

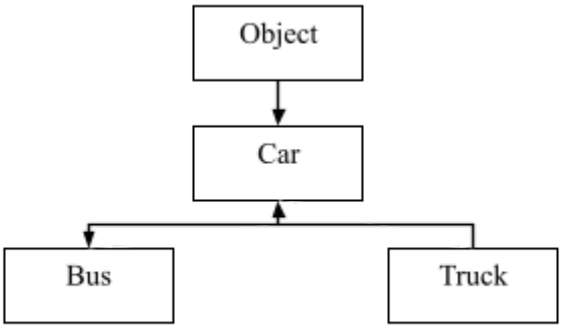
еквівалентний такому конструктору

```
public Bus(int passengers = 50) : base()
{
    Passengers = passengers;
}
```

При виклику конструктора похідного класу спершу викликаються конструктори базових класів у порядку успадкування, тобто від найдальшого предка. Якщо клас BigBus є нащадком Bus, який в свою чергу є нащадком Car, спершу буде виконано конструктор класу Object (який є предком всіх класів C#), потім класу Car, потім класу Bus, потім класу BigBus.

7.2. Приведення типів. Оператори is, as.

Якщо у нас від класу Car породжено два класи – Bus і Truck, це можна виразити у формі такої ієрархії (предки вгорі, похідні класи внизу):



Об’єкт похідного типу внизу ієрархії, одночасно є і об’єктом базового типу. Таким чином, можливе неявне **висхідне перетворення** від типів Bus і Truck до типу Car

```
Car Auto1 = new Bus("Cetra", 20);
Car Auto2 = new Truck("Renault")
```

Оскільки всі класи є нащадками класу Object, всі об’єкти всіх класів можна привести до цього типу.

Зворотне перетворення (**низхідне перетворення**) від базового класу до класу-нащадка неявним бути не може. Низхідне перетворення виконується лише явно, аналогічно приведенню простих типів:

```
Car Auto1 = new Bus("Cetra", 20);
Bus Bus1 = (Bus)Auto1; //Приведення до типу Bus
```

Якщо змінна Auto1 не є об’єктом класу Bus, то наведений вище код призведе до помилки виконання (виключної ситуації або винятку (exception)) – компілятор таку ситуацію не відслідковує. Для того, щоб уникнути подібних ситуацій, в C# передбачено оператори **is, as** (ще одне запозичення з Delphi).

Оператор **is** перевіряє, чи є даний об’єкт об’єктом певного класу, і повертає true, якщо приведення можливе:

```
Car Auto1 = new Bus("Cetra", 20);
if (Auto1 is Bus)
{ //Приведення можливе
    Bus Bus1 = (Bus)Auto1;
    //Якісь дії над об’єктом Bus1
}
```

Оператор is також дає змогу автоматично привести значення до типу:

```
Car Auto1 = new Bus("Cetra", 20);
if (Auto1 is Bus Bus1)
{ //Приведення можливе
  //Якісь дії над об’єктом Bus1
}
```

Оператор **as** виконує приведення об’єкту до вказаного класу, і повертає об’єкт цього класу. Якщо приведення неможливе, результатом буде null:

```
Car Auto1 = new Bus("Cetra", 20);
```

```
Bus? Bus1 = Auto1 as Bus;
if (Bus1 != null) { //приведення успішне
                    //Якісь дії над об'єктом Bus1
}
```

7.3. Перевизначення віртуальних методів. Поліморфізм. Приховування методів.

При успадкуванні класів часто виникає необхідність перевизначити в класі-спадкоємці функціонал методу базового класу. В цьому випадку клас-спадкоємець може перевизначати методи і властивості базового класу.

Методи і властивості, які слід зробити доступними для перевизначення, в класі позначаються модифікатором **virtual**. Такі методи і властивості називають віртуальними. У класі-нащадку такий метод визначається з модифікатором **override**. Перевизначений (його ще називають перекритим) метод повинен мати той самий набір параметрів, що й віртуальний метод у базовому класі:

```
    class Car
    {
        //Поля - доступ лише в класі і нащадках
        protected double fVolume;

        //Автоматичні властивості
        public string Model { get; set; } = "Unknown";
        public string Number { get; set; }
        public int Mileage { get; set; }
        //Властивості
        public virtual double Volume
        {
            get { return fVolume; }
            set
            {
                if ((value <= 0.0) || (value > 100.0))
                    Console.WriteLine("Об'єм двигуна повинен бути більше нуля і до 100
л!");
                else
                    fVolume = value;
            }
        }

        public Car(string model, double volume = 1.0,
            string number = "Unknown", int mileage = 0)
        {
            Model = model;
            Volume = volume;
            Number = number;
            Mileage = mileage;
        }

        public virtual void Print()
        {
            Console.WriteLine($"Автомобіль: {Model}, V={Volume}л," +
                              $" Пробіг {Mileage}, Номер {Number}");
        }
    }

    class Bus : Car
    {
        public int Passengers { get; set; }
```

```
    public Bus(string model, int passengers, double volume = 1.0,
        string number = "Unknown", int mileage = 0) :
            base(model, volume, number, mileage)
    {
        Passengers = passengers;
    }

    public override void Print()
    {
        Console.WriteLine($"Автобус: {Model}, Пасажирів={Passengers}");
    }

}
```

Віртуальні методи базового класу визначають інтерфейс усієї ієрархії, тобто у будь-якому похідному класі, який не є прямим спадкоємцем від базового класу, можна перевизначити віртуальні методи. Наприклад, можна визначити клас BigBus, який є нащадком Bus, і в ньому також перевизначити метод Print.

Тепер згадаємо, що об'єкт класу можна привести до його предка. Розглянемо код:

```
Car Auto1 = new Bus("Setra", 20);
Auto1.Print();
```

Результатом виклику буде виведення на екран рядка "Автобус: Setra, Пасажирів=20". Це означає, що викликається не метод Print базового класу, а метод похідного класу, для якого власне створено об'єкт. Така поведінка називається **поліморфізмом**: об'єкти класів-нащадків реалізують поведінку кожен зі своїми особливостями, але виклик методу для всіх об'єктів можливий з одними параметрами. Це третя основна концепція об'єктно-орієнтованого програмування. Для реалізації такої поведінки компілятор створює таблицю віртуальних методів (virtual method table), і при виклику віртуального метода його адреса вибирається з таблиці під час виконання програми. Це незначно сповільнює роботу програми.

При перевизначенні віртуальних методів існує ряд обмежень:

1. Віртуальний і перевизначений методи повинні мати той самий модифікатор доступу.
2. Не можна перевизначити чи оголосити віртуальним статичний метод.

Ключове слово base можна використовувати не лише в конструкторах. Можна звертатися також до інших членів базового класу:

```
public override void Print()
{
    base.Print();
    Console.WriteLine($"Пасажирів={Passengers}");
}
```

Аналогічно методам можливе перевизначення властивостей:

```
//Властивості
public override double Volume
{
    get { return fVolume; }
    set
```

```
            {
                if ((value < 10.0) || (value > 100.0))
                    Console.WriteLine("Об'єм двигуна автобуса повинен бути від 10 до
100 л!");
                else
                    fVolume = value;
            }
        }
```

Для заборони перевизначення методів та властивостей слід вказувати модифікатор **sealed**.

Крім оголошення віртуальних методів, існує також можливість приховування (shadowing/hiding) методів. Приховування метода базового класу виконується за допомогою модифікатора **new** перед оголошенням нової реалізації методу в похідному класі.

```
    class Car
    {
        //Поля - доступ лише в класі і нащадках
        protected double fVolume;

        //Автоматичні властивості
        public string Model { get; set; } = "Unknown";
        public string Number { get; set; }
        public int Mileage { get; set; }
        //Властивості
        public double Volume
        {
            get { return fVolume; }
            set
            {
                if ((value <= 0.0) || (value > 100.0))
                    Console.WriteLine("Об'єм двигуна повинен бути більше нуля і до 100
л!");
                else
                    fVolume = value;
            }
        }

        public Car(string model, double volume = 1.0,
            string number = "Unknown", int mileage = 0)
        {
            Model = model;
            Volume = volume;
            Number = number;
            Mileage = mileage;
        }

        public void Print()
        {
            Console.WriteLine($"Автомобіль: {Model}, V={Volume}л," +
                              $" Пробіг {Mileage}, Номер {Number}");
        }
    }

    class Bus : Car
    {
        public int Passengers { get; set; }

        public Bus(string model, int passengers, double volume = 1.0,
            string number = "Unknown", int mileage = 0) :
                      base(model, volume, number, mileage)
```

```
    {
        Passengers = passengers;
    }

    public new void Print()
    {
        Console.WriteLine($"Автобус: {Model}, Пасажирів={Passengers}");
    }
}
```

Приховування методів використовується у випадках, коли у похідному класі метод перевизначити не можна, а функціонал його слід змінити. У методі похідного класу можна звернутися до приховуваного методу базового класу через ключове слово base.

Аналогічно можна виконати приховування властивостей, і навіть статичних і нестатичних констант.

На відміну від перевизначення віртуальних методів, при приховуванні методу Print() код

```
Car Auto1 = new Bus("Setra", 20);
Auto1.Print();
```

викличе метод Print() класу Car, а не класу Bus. Це зумовлено тим, що клас Bus не перевизначає метод Print(), а створює новий, ніяк не пов'язаний із відповідним методом базового класу.

7.4. Абстрактні класи.

У C# можливе оголошення абстрактних класів. Класи слугують для опису деяких сутностей. Наприклад, клас може описувати автомобіль, людину або геометричну фігуру. Деякі сутності при цьому можуть не мати конкретного втілення: наприклад, існують коло, прямокутник і квадрат, але не існує геометричної фігури «взагалі». Притому і коло, і прямокутник мають спільні риси і є геометричними фігурами. Абстрактні класи необхідні для того, щоб описувати саме сутності, що не мають конкретного втілення. Приклад:

```
abstract class Shape
{
    public abstract double Perimeter();
    public abstract double Area();
}

class Circle : Shape
{
    public double CenX { get; set; }
    public double CenY { get; set; }
    public double Radius { get; set; }

    public override double Perimeter()
    {
        return 2 * Math.PI * Radius;
    }
    public override double Area()
    {
        return Math.PI * Radius * Radius;
    }
}
```

```
class Rectangle : Shape
{
    public double X1 { get; set; }
    public double Y1 { get; set; }
    public double X2 { get; set; }
    public double Y2 { get; set; }

    public override double Perimeter()
    {
        return 2 * (Math.Abs(X2 - X1) + Math.Abs(Y2 - Y1));
    }
    public override double Area()
    {
        return Math.Abs((X2 - X1) * (Y2 - Y1));
    }
}
```

Абстрактний клас Shape (фігура) містить два методи, які повинні повертати периметр і площу фігури. Класи-нащадки Circle і Rectangle містять поля, що описують конкретну фігуру (координати центру і радіус - для кола, координати двох протилежних вершин діагоналі - для прямокутника) і реалізацію методів для розрахунку периметру і площі кожної фігури.

Абстрактний клас може містити абстрактні члени класу, зокрема методи і властивості. Такі члени класу містять модифікатор abstract і не повинні містити реалізації:

```
abstract class Shape
{
    public abstract double Perimeter();
    public abstract double Area();
}
```

Абстрактні методи і властивості не можуть бути віртуальними і не можуть мати модифікатор private. Всі абстрактні методи і властивості повинні бути реалізовані в класах-нащадках. При перевизначенні у похідному класі такий метод або властивість оголошуються з модифікатором **override**.

При виклику абстрактні методи працюють так само, як і віртуальні: викликається метод того класу, до якого належить об’єкт.

```
Shape Shape1 = new Circle();
//Буде виведено периметр кола
Console.WriteLine(Shape1.Perimeter());
```

Таким чином, **використання абстрактних класів та їх членів – іще один механізм реалізації поліморфізму.**

Визначення абстрактних властивостей схоже на визначення автовластивостей:

```
public abstract string ShapeName { get; set; }
```

Реалізація абстрактних властивостей у похідних класах нічим не відрізняється від віртуальних властивостей.

Якщо в класі є хоча б один абстрактний метод або властивість, такий клас повинен бути визначений як абстрактний. Є можливість відмовитися від

реалізації властивості/методу, оголосивши похідний клас абстрактним, однак всі абстрактні методи, властивості і класи в кінцевому підсумку повинні бути реалізовані в класах-нащадках.

7.5. Інтерфейси.

Інтерфейс – тип-посилання, який може визначати набір методів і властивостей без реалізації. Потім функціонал інтерфейсу реалізують класи та структури, які застосовують дані інтерфейси.

Інтерфейс оголошується за допомогою ключового слова **interface**. Хорошою практикою вважається починати ідентифікатор (ім'я) інтерфейсу з латинської I (IComparable, IMovabe, …), хоча це не обов'язково. Інтерфейс може містити константи, статичні поля та статичні константи (починаючи з C# 8.0), властивості і методи. В інтерфейсах не можна визначати нестатичні поля. Як приклад нижче наведено інтерфейс, призначений для доступу до приладу:

```
interface IInstrument
{
    //Константа – час прогріву
    const int WarmupTime = 30;
    //Статична змінна – температура калібровки
    // 20 C (нормальні умови)
    static int CalibrationTemp = 20;
    //Властивість Назва приладу
    string Name { get; set; }
    //Метод – вимірювання
    void Measure();
}
```

Методи та властивості інтерфейсу можуть не мати реалізації, як абстрактні методи та властивості абстрактних класів. Зверніть увагу, що властивість Name у інтерфейсі – не автовластивість, а визначення властивості, яка не має реалізації.

В інтерфейсі всі методи і властивості за замовчуванням мають модифікатор доступу public, оскільки мета інтерфейсу – визначення функціоналу для реалізації класом. Так само модифікатор public за замовчуванням мають константи і статичні змінні. Починаючи з C# 8, у членів інтерфейсу можна вказувати модифікатори доступу:

```
interface IInstrument
{
    //Константа – час прогріву
    public const int WarmupTime = 30;
    //Статична змінна – температура калібровки
    // 20 C (нормальні умови)
    static int CalibrationTemp = 20;
    //Властивість – кількість вимірювань
    int MeasCount { get; set; }
    //Властивість Назва приладу
    protected string Name { get; set; }
    //Метод – вимірювання
    void Measure();
}
```

За замовчуванням інтерфейси мають рівень доступу internal (доступні в поточній збірці), але можна вказати рівень доступу public.

Починаючи з C# 8, інтерфейси підтримують реалізацію методів і властивостей за замовчуванням, тобто можна вказувати реалізацію методів і властивостей безпосередньо в інтерфейсі:

```
interface IInstrument1
{
    //Властивість - кількість вимірювань
    int MeasCount { get { return 0; } }
    //Метод - вимірювання
    void Measure() => Console.WriteLine("Вимірюємо...");
}
```

Методи з модифікаторами private та static обов'язково повинні мати реалізацію за замовчуванням.

Щоб додати до проекту інтерфейс, в Visual Studio 2022 слід вибрати пункт меню «Project\Add class...», у вікні вибрати пункт «Interface», внизу вікна ввести ім'я інтерфейсу і клацнути по кнопці «Add».

Для застосування інтерфейсу слід визначити клас, і в описі класу після імені і двокрапки вказати ім'я інтерфейсу, який клас реалізує. Клас повинен реалізувати всі методи та властивості інтерфейсу, якщо ці методи та властивості не мають реалізації за замовчуванням:

```
class Voltmeter : IInstrument
{
    protected int fMeasCount;
    public int MeasCount { get => fMeasCount; }
    public string Name { get; set; }

    void Measure()
    {
        fMeasCount++;
        Console.WriteLine("Вимірюємо напругу...");
    }
}
```

Якщо клас не реалізує метод, визначений за замовчуванням, буде використовуватися реалізація за замовчуванням з інтерфейсу. Як наслідок, можна не вносити зміни в класи, що реалізують інтерфейс, якщо виникла потреба змінити сам інтерфейс.

Інтерфейсів, які реалізує клас, може бути декілька. У цьому випадку їх імена перераховуються через кому:

```
class myClass : myInterface1, myInterface2, myInterface3, ...
```

Все сказане щодо **перетворення типів класів** притаманне й інтерфейсам. Так, змінна типу IInstrument може зберігати клас Voltmeter, причому приведення класу до типу інтерфейсу, який він реалізує, виконується автоматично. Зворотне перетворення слід виконувати явно; можна використати оператори as та is.

```
IInstrument Instr1 = new Voltmeter();

if (Instr1 is Voltmeter V1)
```

```
{
    Console.WriteLine(V1.Voltage);
}
```

Існує також так звана явна реалізація інтерфейсу. При явній реалізації разом із назвою інтерфейсу вказується назва використовуваного методу або властивості. При явній реалізації не можна використовувати модифікатори public, protected, … – усі методи закриті (private), і звернутися до них можна лише через інтерфейс.

Необхідність у явній реалізації виникає у випадку, коли клас реалізує кілька інтерфейсів, але вони мають метод із однаковим іменем, тим самим набором параметрів:

```
class Student : IMath, IPhysics
{
    void IMath.Study() => Console.WriteLine("Вчу математику");
    void IPhysics.Study() => Console.WriteLine("Вчу фізику");
}

interface IMath
{
    void Study();
}
interface IPhysics
{
    void Study();
}
```

Виклик обох реалізованих інтерфейсів:

```
Student Student1 = new();
((IMath)Student1).Study();
((IPhysics)Student1).Study();
```

Якщо модифікатор доступу члену класу чи структури ***не public***, то для реалізації такого члену класу теж необхідно використовувати **явну реалізацію інтерфейсу**. Як і у випадку реалізації абстрактних класів, можна не реалізувати методи інтерфейсу, зробивши їх абстрактними і переклавши право їх реалізації на похідні класи. Якщо клас одночасно успадковує інший клас і реалізує інтерфейс, назва базового класу має бути вказана до реалізованих інтерфейсів:

```
class Student : Human, IMath, IPhysics //Спершу клас, потім інтерфейси
{

}
```

Інтерфейси підтримують успадкування:

```
interface IEtaloneInstrument : IInstrument
{
    int PrecisionClass { get; set; }
    void Calibrate();
}
```

На відміну від класів, у інтерфейсах не можна використовувати модифікатор sealed, щоб закрити можливість успадкування. Також в інтерфейсах не можна використовувати модифікатор abstract.

Щоб приховати метод базового інтерфейсу, методи інтерфейсів можуть використовувати ключове слово new:

```
interface IEtaloneInstrument : IInstrument
{
    int PrecisionClass { get; set; }
    void Calibrate();
    new void Measure() => Console.WriteLine("Вимірюємо точно...");
}
```

Як і при успадкуванні класів, при успадкуванні інтерфейсів нащадок повинен мати **той самий рівень доступу або суворіший, ніж базовий інтерфейс.**

7.6. Делегати.

Делегати – об'єкти, які вказують на методи, тобто це – покажчики на методи, і за допомогою делегатів ми можемо викликати ці методи.

Синтаксис оголошення делегата такий

```
delegate <тип результату> <ім'я делегата>([<параметри>])
```

Наприклад, інструкція

```
delegate double CalculateParam();
```

оголошує делегат, який не приймає параметрів і повертає значення типу double. Тому цей делегат може вказувати на будь-який метод, який не приймає параметрів і повертає double.

Делегат може приймати параметри:

```
delegate double DoOperation(double x, double y);
```

Для використання делегата треба присвоїти йому значення метода, на який він вказуватиме. При присвоюванні метода делегату, тип результату, кількість, типи і модифікатори параметрів делегата і методу, покажчик на який присвоюється делегату, повинні співпадати. Виклик делегата здійснюється подібно до виклику методу:

```
CalculateParam calc = Return3PI_2;
Console.WriteLine(calc()); //Виклик делегата

double Return3PI_2()
{
    return 3.0*Math.PI/2.0;
}

delegate double CalculateParam();
```

Делегат не обов’язково повинен вказувати на локальний метод або метод того класу, де визначено делегат. Наприклад, можна присвоїти делегату посилання на метод іншого об'єкта (або структури):

```
Circle C1 = new() { CenX = 0, CenY = 0, Radius = 5 };

CalculateParam calc = C1.Perimeter;
Console.WriteLine(calc()); //Виклик делегата
```

Об’єкт делегата можна створити **за допомогою конструктора**, в який передається потрібний метод:

```
CalculateParam calc1 = new CalculateParam(C1.Perimeter);
```

Обидва способи рівноцінні.

Делегат може вказувати не на один метод, а на кілька. Для цього до делегату додаються методи оператором +=. Виключити метод із делегату можна оператором -= :

```
Circle C1 = new() { CenX = 0, CenY = 0, Radius = 5 };
Circle C2 = new() { CenX = 0, CenY = 0, Radius =10 };

CalculateParam calc = C1.Perimeter;
calc += C2.Perimeter;
calc += Return3PI_2;

calc -= Return3PI_2;
```

Всі методи, присвоєні делегату, потрапляють до списку виклику (invocation list), і виконуються один за одним в порядку додавання. Якщо делегат повертає значення, повертається значення, повернене останнім методом зі списку виклику.

При видаленні слід враховувати, що якщо делегат містить кілька посилань на один і той самий метод, операція -= починає пошук із кінця списку виклику і видаляє лише перше знайдене посилання. Якщо методу у списку виклику делегата немає, то операція -= не має жодного ефекту. Якщо будуть видалені всі методи списку виклику, делегат отримає значення null.

Можливе також злиття делегатів

```
CalculateParam Calc3 = calc1 + calc2;
```

В результаті у список виклику calc3 буде включено всі методи зі списків виклику делегатів calc1 та calc2.

Делегати можуть бути параметрами методів і результатами методів.

Для чого потрібні делегати? Застосування делегату дає змогу делегувати виконання певних дій із класу назовні – в інший клас або в головну програму. Наприклад, у клас Circle можна додати обробник виключної ситуації – спроби задати радіус кола менше нуля:

```
public delegate void DataErrorHanlder(ref double val);

class Circle : Shape
{
```

```
    //делегат обробника
    public DataErrorHanlder? BadRadiusHandler;
    //Поле
    protected double fRadius;
    //Властивості
    public double CenX { get; set; }
    public double CenY { get; set; }
    public double Radius
    {
        get => fRadius;
        set
        {
            if (value < 0)
            { //Обробка ситуації делегована з класу назовні
                if (BadRadiusHandler != null)
                    BadRadiusHandler(ref value);
                else
                    value = 0;
            }
            fRadius = value;

        }
    }

    public override double Perimeter()
    {
        return 2 * Math.PI * Radius;
    }
    public override double Area()
    {
        return 2 * Math.PI * Radius;
    }
}
```

Тепер при спробі задати значення радіусу менше нуля, в класі буде викликано делегат BadRadiusHandler, якщо він не null. Як приклад, нижче наведено код, який при спробі задати радіус кола менше нуля, встановить його значення рівним 10:

```
Circle C1 = new() { CenX = 0, CenY = 0, Radius = 5 };
C1.BadRadiusHandler += BadRadErrorHanlder;
Console.WriteLine(C1.Radius); //буде виведено 5
C1.Radius = -100;
Console.WriteLine(C1.Radius); //буде виведено 10

void BadRadErrorHanlder(ref double val)
{
    val = 10;
}
```

Цей підхід активно використовується в бібліотеках для розробки інтерфейсу користувача (Windows Forms, WPF та інших) для обробки подій, наприклад, натискання на кнопку у вікні або активації пункту меню.

Запитання для самоперевірки

1. Що таке успадкування в об’єктно-орієнтованому програмуванні? Які є обмеження при успадкуванні?
2. До яких членів базового класу має доступ клас-нащадок?

3. Як задати конструктор похідного класу і для чого можна використовувати ключове слово base?

4. Як виконується приведення типів класів?

5. Яке призначення операторів typeof, is, as?

6. Що таке поліморфізм? Що таке віртуальний метод?

7. Як приховати методи, успадковані від базового класу, у похідному класі?

8. Що таке абстрактний клас? Які особливості мають абстрактні методи і властивості?

9. Що таке інтерфейси в C#?

10. Як реалізовують інтерфейси в C#?

11. Що таке делегат? Як створити делегат?

12. Як і навіщо використовують делегати?

Питання з теми, що виносяться на самостійне опрацювання

1. Інтерфейси в C#. Реалізація інтерфейсів.

2. Використання делегатів для ~~забезпечення~~ контролю введених даних, заданого ззовні класу.

Лекція 8. Windows Forms і WPF. Швидке створення інтерфейсу користувача. Властивості і події. Форми. Елементи управління.

8.1 Windows Forms і WPF. Швидке створення інтерфейсу користувача. Властивості і події.

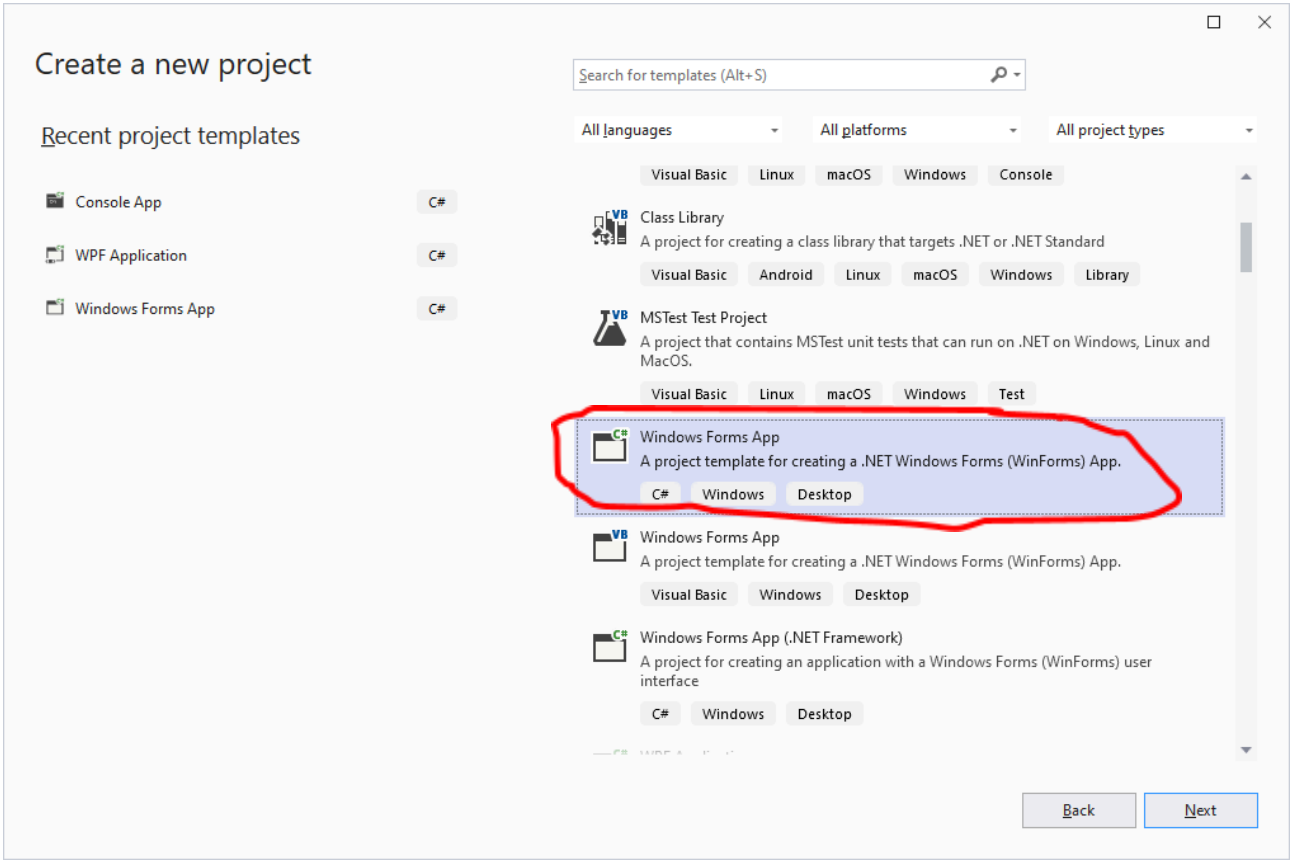
На всіх лекціях до цього введення і виведення даних відбувалось у консоль в текстовому режимі. Сучасні програми, за винятком небагатьох системних утиліт, мають графічний віконний інтерфейс користувача, що набагато зручніше в користуванні за консоль.

Для створення графічного інтерфейсу у Visual Studio можна застосувати різні технології – Windows Forms, WPF, UWP, MAUI. Найпростішою і найзручнішою є Windows Forms, що не дивно: ідеологію цієї бібліотеки класів розроблено Андерсом Гейлсбергом (Anders Hejlsberg), який розробив першу справді зручну бібліотеку візуальних компонентів Visual Component Library для Borland Delphi ще в 90-х. Хоча Microsoft називає Windows Forms застарілою технологією, вона зручна для початківців і при написанні не надто складних програм, оскільки зовнішній вигляд інтерфейсу користувача створюється візуальним редактором без єдиного рядка коду або XML.

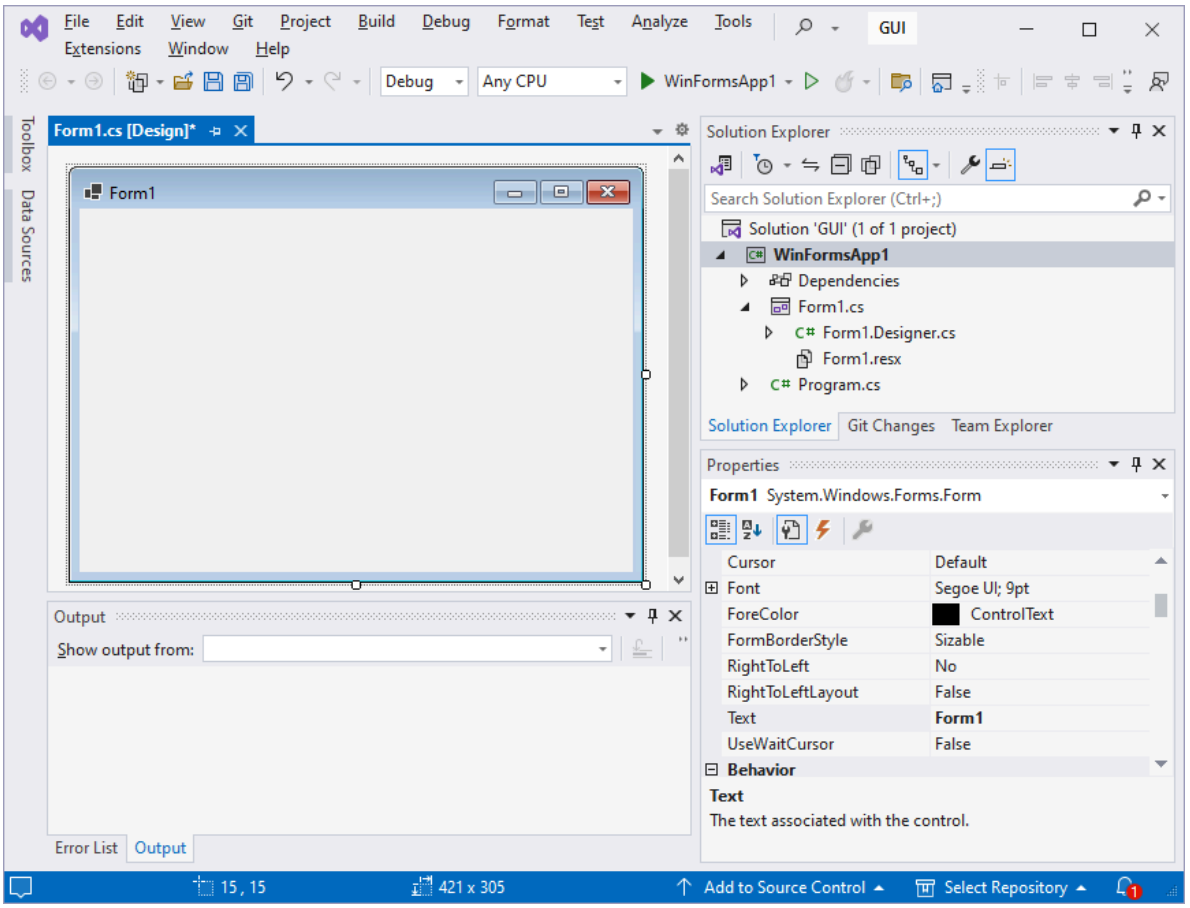
На заміну Windows Forms Microsoft пропонує технологію WPF (Windows Presentation Foundation), засновану на XAML – діалекті XML. Ідеологія побудови WPF увібрала в себе риси Windows Forms і засобів побудови інтерфейсу користувача під Android. При створенні інтерфейсу за допомогою XAML часто доводиться змінювати текст опису форми на XAML вручну.

У цьому курсі ми зосередимо увагу на Windows Forms. Робота з WPF аналогічна Windows Forms; назви та властивості компонентів WPF аналогічні описаним нижче.

Проект на основі Windows Forms створюється аналогічно проекту консольного додатку (див. лекцію 1), але у вікні вибору типу проекту слід вибрати пункт «Windows Forms App» (для WPF слід вибрати «WPF Application»).



Після створення проекту Visual Studio виведе на екран вікно з файлами проекту, створеними за замовчуванням:



Зліва у вікні знаходиться візуальний редактор із графічним поданням головного вікна (головної форми) програми. Справа – Solution Explorer, що показує структуру проекту, а під ним редактор властивостей.

Спершу розглянемо Solution Explorer. У дереві проекту показано проект WinFormApp1, який був щойно створений. Під ним у вузлі Dependencies вказано бібліотеки (збірки .NET), які використовує проект. Далі розміщено файли доданих у проект класів. За замовчуванням створений проект містить одну форму (клас Form1), поміщену в файл Form1.cs, і ресурси форми (файл Form1.resx) – рядки повідомлень, іконки та ін. Сама програма розміщена у файлі Program.cs.

Для відкриття файлу слід клацнути по ньому двічі.

Головна програма додатку Windows Forms містить у статичному методі Main дві інструкції – ініціалізацію додатка і запуск програми зі створенням головної форми (Form1):

```
namespace WinFormsApp1
{
    internal static class Program
    {
        /// <summary>
        ///  The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
```

```
            // To customize application configuration such as set high DPI settings or
default font,
            // see https://aka.ms/applicationconfiguration.
            ApplicationConfiguration.Initialize(); //Ініціалізація
            Application.Run(new Form1());          //Запуск зі створенням форми
        }
    }
}
```

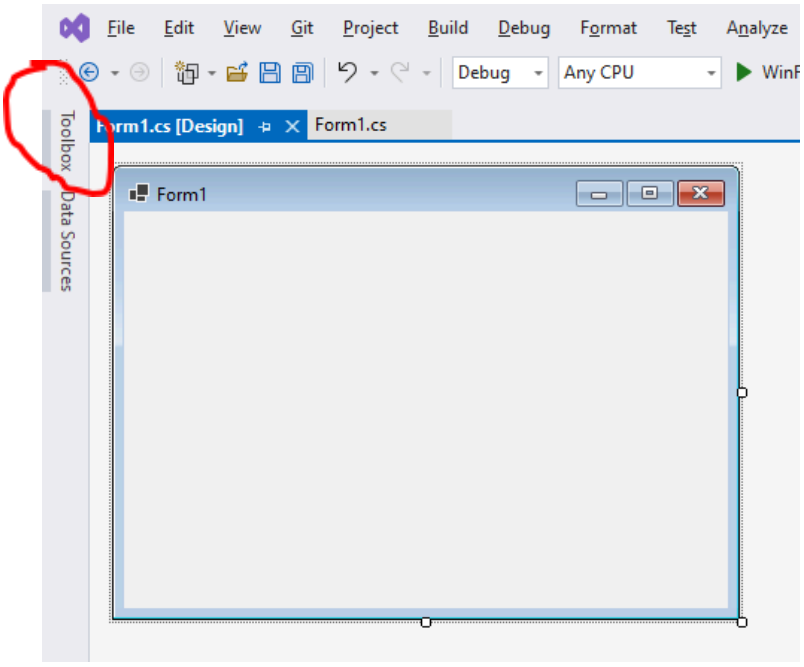
У більшості випадків жодних змін у програму вносити не доведеться. Файл Form1.cs містить опис класу головної форми

```
namespace WinFormsApp1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Він породжений від стандартного класу Form1 і описаний як частковий (**partial**). Це означає, що в файлі Form1.cs реалізована лише частина функціоналу головної форми; решта реалізується у файлі Form1.Designer.cs, який обслуговується Visual Studio і містить опис інтерфейсу користувача.

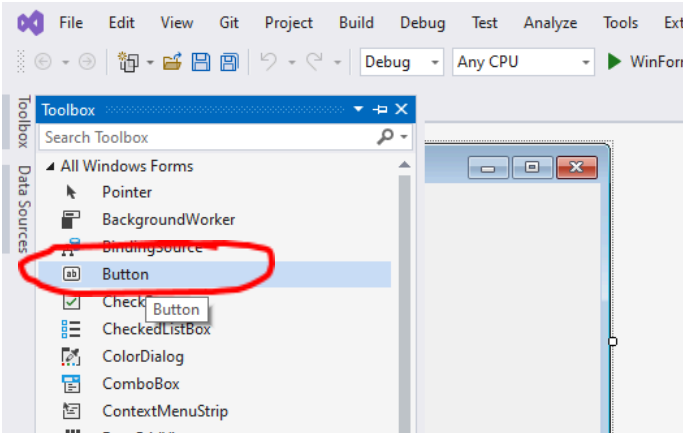
Створений проект можна запустити на виконання; на екран буде виведено пусте вікно, яке можна переміщувати по екрану, змінювати його розмір і закрити. Для того, щоб користувач міг робити щось справді корисне, у проект слід додати елементи керування (controls).

Для цього слід відкрити візуальний редактор. При відкритому на екрані файлі Form1.cs слід натиснути Shift+F7; на екран буде виведено вікно візуального редактора з пустою формою:

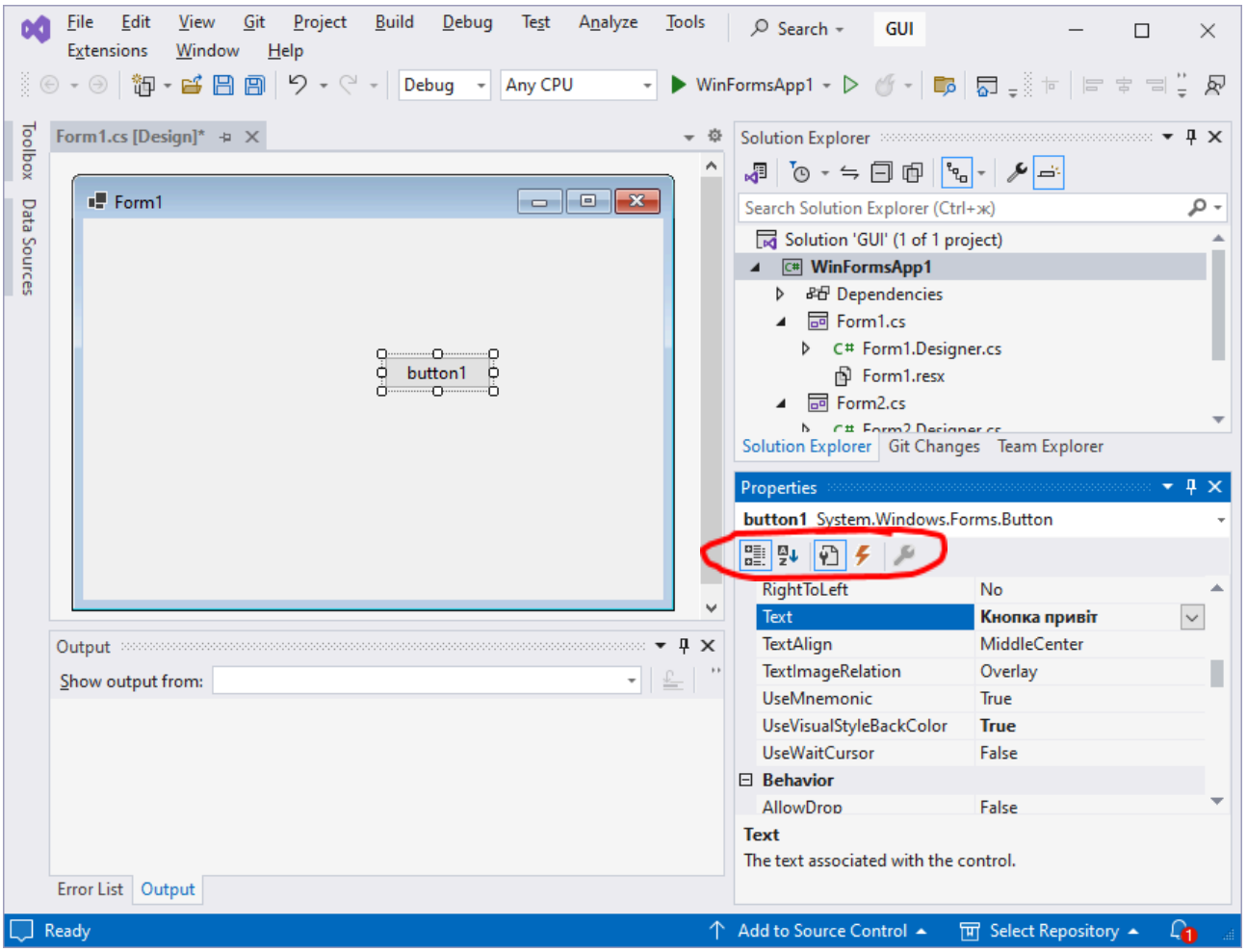


Всі доступні компоненти Windows Forms знаходяться в меню Toolbox, для його виведення слід вибрати пункт меню «View\Toolbox» або натиснути Ctrl+Alt+X. Для того, щоб додати на форму, наприклад, кнопку, необхідно:

1) клацнути по меню Toolbox;
2) в меню вибрати компонент «Button»;
3) перетягнути його на форму.



В результаті на формі з’явиться кнопка з написом button1:



Кнопка, як і будь-який інший компонент, має ряд властивостей. Для того, щоб змінити ці властивості, слід:

1) якщо кнопку не вибрано, клацнути по кнопці;
2) у вікні Properties вибрати властивість і змінити її, ввівши значення з клавіатури або вибравши зі списку.

Наприклад, можна змінити поле Text кнопки, вказавши як назву «Кнопка привіт», а ім'я кнопки з імені по замовчуванню button1 – на більш змістовне, наприклад BtnHello. Оскільки напис на кнопку не помістився, потрібно змінити її розмір. Для цього слід клацнути по «ручці» (квадратику) в нижньому правому кутку кнопки, і, утримуючи ліву кнопку миші, розтягнути кнопку до бажаного розміру, після чого відпустити кнопку миші. Клацнувши по самому тілу кнопки, і перетягнувши кнопку не відпускаючи ліву кнопку миші, можна розмістити її на формі в бажаному місці.

Звернімо тепер увагу на кнопки під заголовком вікна Properties:

 Categorized – розміщення властивостей у вікні по категоріям.

 Alphabetical – розміщення властивостей у вікні за алфавітом.

 Properties – показ властивостей.

 Events – показ подій та їх обробників.

Подія Windows Forms – це певна подія, яка відбувається з компонентом: клацання по кнопці, відкриття діалогу вибору файлу та ін. Реакцію на подію задає обробник події – метод (як правило, це метод класу форми); сам механізм реакції на подію реалізується через делегат у об'єкті, який реагує на подію. Код метода – реакція на подію – задається у класі форми. Наприклад, для того, щоб задати реакцію на клацання по кнопці мишею, слід у вікні «Properties» клацнути по кнопці «Events», і вибравши рядок «Mouse click», двічі клацнути справа від нього.

Visual Studio додає в Form1.cs метод

```
private void BtnHello_MouseClick(object sender, MouseEventArgs e)
{

}
```

Цей метод буде викликатися кожен раз, коли користувач клацне мишею по кнопці BtnHello. Як найпростішу реакцію, можна задати виведення на екран повідомлення «Привіт!»:

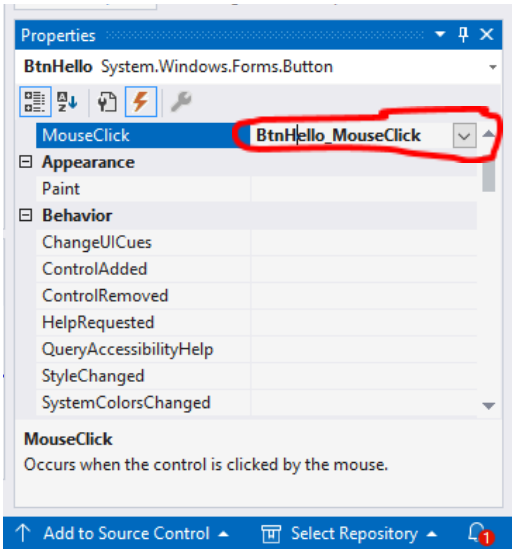
```
private void BtnHello_MouseClick(object sender, MouseEventArgs e)
{
    MessageBox.Show("Привіт!");
}
```

На відміну від консольних додатків, які ми писали вище, програма у Windows forms переважну частину часу не працює, а очікує певних дій користувача або зовнішніх сигналів (наприклад, приходу повідомлень через мережу). Такий спосіб побудови додатків застосовується в усіх сучасних операційних системах; інтерфейс, побудований на взаємодії з користувачем та зовнішнім середовищем шляхом реакції на події, називається керованим подіями (event-driven).

Поглянемо тепер на параметри обробника. Перший параметр object sender передає в обробник об'єкт, який власне й викликав виникнення події. Завдяки цьому один обробник може застосовуватися для кількох подій або для однотипної події, генерованої різними компонентами. Наприклад, при клацанні по кнопці «Зберегти», кнопці з дискетою в панелі швидкого доступу та пункту меню «Файл\Зберегти» повинна виконуватися одна дія – збереження файлу. Для визначення, який саме об'єкт викликає обробник – кнопка чи пункт меню – та роботи із відповідним об'єктом, можна скористатися операторами is та as.

Другий параметр обробника MouseEventArgs е містить параметри миші – які кнопки натиснуті, координати курсора та ін. Для вивчення параметрів методів можна скористатися онлайн-системою допомоги Visual Studio.

Зверніть увагу на те, що у вікні «Properties» внизу є короткий опис події або властивості, вибраної у вікні. У випадку, якщо для події вже створено сумісні за сигнатурою обробники, можна вибрати потрібний обробник із випадаючого списку (на рисунку внизу виділений червоним). Якщо обробник події більше не потрібен, виберіть його назву і натисніть кнопку Del. Текст обробника в .cs файлі доведеться видаляти вручну.



8.2. Форми.

Форма (вікно) також має властивості, які можна змінювати через вікно «Properties». Основні властивості перераховані нижче.

Name встановлює ім'я класу форми (успадкованого від класу Form).

BackColor встановлює колір фону форми.

BackgroundImage встановлює зображення фону форми.

BackgroundImageLayout визначає розташування на формі зображення BackgroundImage.

ControlBox дає змогу задати, чи слід показувати в заголовку форми іконку програми, кнопки мінімізації форми та хрестик. Якщо ControlBox==false, іконки і кнопки показано не буде.

Cursor встановлює тип курсору форми (можна замінити традиційну стрілочку, наприклад, на чарівну паличку або руку).

Enabled – якщо Enabled==false, форма не зможе отримувати введення від користувача, і ні на що не реагуватиме. Така властивість є у всіх компонентів Windows Forms.

Font встановлює шрифт для заголовку форми та (за замовчуванням) шрифт усіх розміщених на ній компонентів. При цьому, використовуючи властивість Font кожного компонента форми, для нього можна вибрати свій шрифт.

ForeColor – задає колір шрифту форми.

FormBorderStyle дає змогу визначити, як будуть відображатися рядок заголовка та границя форми.

HelpButton дає змогу відобразити (true) або приховати (false) кнопку довідки форми.

Icon задає іконку форми.

Location визначає координати форми відносно лівого верхнього кута екрана, якщо для властивості StartPosition встановлено значення Manual.

MaximizeBox вказує, чи буде доступна в заголовку форми кнопка максимізації вікна.

MinimizeBox вказує, чи буде доступна в заголовку форми кнопка мінімізації вікна.

MaximumSize задає максимальний розмір форми.

MinimumSize задає мінімальний розмір форми.

Opacity визначає прозорість форми.

Size задає початковий розмір форми.

StartPosition – початкова позиція форми на екрані при її створенні та показі:

- Manual – положення форми визначається властивістю Location.
- CenterScreen – у центрі екрану.
- WindowsDefaultLocation – позиція форми на екрані визначається Windows, а розмір - властивістю Size.
- WindowsDefaultBounds – початкова позиція та розмір форми на екрані визначається Windows.
- CenterParent – у центрі батьківського вікна.

Text задає заголовок форми.

TopMost. Якщо TopMost true, форма завжди буде знаходитися поверх інших вікон.

Visible дає змогу показати чи приховати форму.

WindowState дає змогу вказати, в якому стані форма перебуватиме при запуску додатку: у нормальному, максимізованому (форма розгорнута на весь екран) або мінімізованому (форма згорнута в панель завдань).

Кожну із цих властивостей можна змінити також динамічно (в коді обробника події), наприклад як реакцію на клацання по кнопці:

```
private void button1_Click(object sender, EventArgs e)
{
    ForeColor = Color.Blue;
    BackColor = Color.Yellow;
    Font = new Font(Font, FontStyle.Italic);
    Text = "Перефарбовано!";
}
```

Так само для форми можна задати обробники подій.

Проект може містити кілька форм. Для того, щоб додати нову форму, слід вибрати пункт меню «Project\Add Form (Windows Forms)», і у вікні ввести назву класу форми. Нова форма буде додана до проекту, але не буде ніде задіяною.

Для того, щоб вивести форму на екран, наприклад, при клацанні по кнопці, в обробник події слід додати такий код:

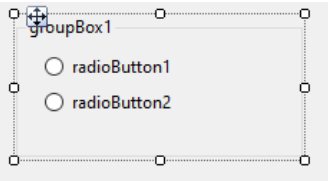
```
private void BtnForm2_Click(object sender, EventArgs e)
{
    Form2 form2 = new Form2(); //створити форму
    form2.Show(); //показати
}
```

Якщо застосовано виклик form2.Show(), на екран буде виведено форму Form2, при цьому вікно батьківської форми Form1 залишиться доступним для введення. Якщо потрібно, щоб користувач при роботі з формою Form2 не міг взаємодіяти з головним вікном Form1 (наприклад, при виклику вікна конфігурації програми), слід застосувати метод form2.ShowDialog().

8.3. Контейнери.

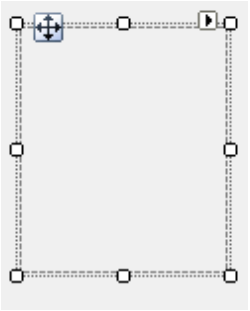
Контейнерами називають компоненти, які можуть містити інші компоненти. Крім, власне, форми, у Windows Forms існують такі компоненти-контейнери:

Group Box – відділена від решти форми область із заголовком.



GroupBox часто використовується для групування перемикачів – елементів **RadioButton**, оскільки дає змогу розмежувати їхні групи.

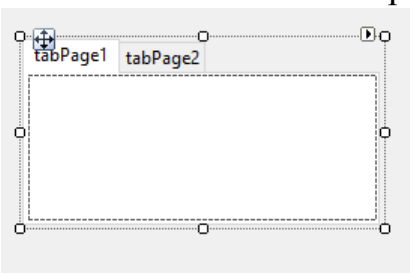
Panel – аналогічний за функціоналом GroupBox, але не має межі і зливається з формою.



FlowLayoutPanel успадкована від класу Panel. Цей компонент може змінювати позиціонування та компонування дочірніх елементів під час зміни розмірів форми при виконанні програми.

TableLayoutPanel також успадкована від панелі і автоматично розміщує дочірні елементи у комірках таблиці – кожен у своїй. Щоб у такій комірці розмістити більше одного елемента управління, до неї слід додати ще один компонент TableLayoutPanel, в який вкладаються інші елементи.

TabControl визначає компонент, що містить багато сторінок із заголовками. Вміст кожної сторінки задається індивідуально.



8.4 Елементи управління.

Всі елементи управління Windows Forms – візуальні класи, які отримують введені користувачем дані і можуть ініціювати події. Всі елементи управління походять від класу Control і мають низку спільних властивостей:

Anchor задає, як при зміні розміру форми буде розтягуватись цей елемент управління, шляхом можливості фіксації його верхнього лівого та нижнього правого кутів.

BackColor визначає фоновий колір елемента.

BackgroundImage визначає фонове зображення елемента.

ContextMenu – контекстне меню, яке відкривається при натисканні на елемент правою кнопкою миші. Задається за допомогою компонента ContextMenu.

Cursor визначає, як відображатиметься курсор миші при наведенні на елемент.

Dock задає розташування елемента на формі.

Enabled визначає, чи буде доступний елемент для використання. Якщо ця властивість має значення False, елемент блокується (не реагує на введення).

Font встановлює шрифт тексту для елемента.

ForeColor визначає колір шрифту.

Location визначає координати верхнього лівого кута елемента керування.

Name – задає ім'я елемента керування.

Size – визначає розмір елемента.

Width – ширина елемента.

Height – висота елемента.

TabIndex – визначає порядок обходу елемента натисканням клавіші Tab.

Tag – дає змогу зберігати цілочисельне значення, асоційоване із цим елементом управління.

Далі перелічено елементи управління, які найчастіше використовуються, та їх ключові властивості. Розширений список можна знайти тут: https://abitap.com/4-1-knopka/

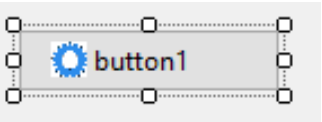
8.4.1. Label – мітка

Мітка – це текст, напис у вікні. Основна її властивість– Text (текст мітки).

8.4.2. Button – кнопка

FlatStyle – керує зовнішнім видом кнопки.

TextImageRelation – вказує взаємне розташування зображення та тексту на кнопці. Це дає змогу, наприклад, виставивши в кнопці зображення Image та встановивши властивість TextImageRelation у значення ImageBeforeText, отримати таку кнопку:



8.4.3. TextBox – текстове поле

Слугує для введення тексту. Властивість Text містить введений текст. Крім того, у поля є властивості:

Multiline – введення/виведення кількох рядків тексту (як у редакторі «Блокнот» (Notepad));

Scroll Bars – смуги прокрутки;

WordWrap – автоматичний перенос за словами.

Головна подія текстового поля – **TextChanged**, викликається при зміні тексту користувачем.

Для зчитування введених даних слід використовувати такий код:

```
//Вичитка введеного рядка (імені і т.п.)
string UserName = textBox1.Text;

// Читання чисел
// Якщо введено не числа – помилка виконання
int age = Convert.ToInt32(textBox1.Text);
double number1 = double.Parse(textBox1.Text);

// Безпечне читання даних тексту в змінну цілого типу
int.TryParse(textBox1.Text, out int number4);
```

Для виведення даних у поле тексту можна застосувати такий код:

```
// для запису тексту в поле
textBox1.Text = UserName;

// для додавання тексту до змісту поля
textBox1.AppendText(Convert.ToString(age));
textBox1.Text += Convert.ToString(height);

//додавання нового рядка в багаторядкове поле
MultilineTextBox2.Text += UserName+"\r\n";
```

8.4.4. RadioButton та CheckBox

Ці компоненти – прапорець та радіо кнопка (кнопка із залежною фіксацією). Обидва мають ключове поле **Checked**, яке вказує, вибрано компонент чи ні, і генерують подію **CheckedChanged**, якщо змінюється їх стан.

Різниця між компонентами полягає в тому, що Radio Button розміщують групами (як правило, в компонентах GroupBox), і якщо вибрано одну таку кнопку в групі, то вибір з решти знімається (одночасно може бути вибрана одна кнопка RadioButton в групі).

8.4.5. ListBox – список

Ключова властивість списку – Items – містить список елементів.

Елементи до списку можуть бути додані як під час розробки, так і програмним способом:

```
listBox1.Items.Add("Новий елемент");
```

Очищення списку:

```
listBox1.Items.Clear();
```

Видалення елемента:

```
listBox1.Items.Remove("Top"); //За текстом
listBox1.Items.RemoveAt(1); //Позиція в списку
```

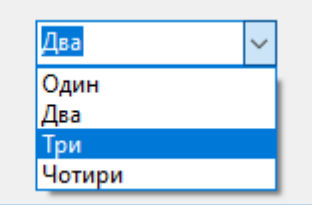
Отримання кількості елементів і першого елемента списку (індекс першого елемента 0):

```
cnt = listBox1.Items.Count; //кількість елементів
string firstElement = listBox1.Items[0]; //перший елемент
```

Властивість SelectedIndex містить індекс вибраного елемента або -1, якщо жоден елемент не вибраний.

8.4.6. ComboBox – випадаючий список

Аналогічний ListBox, задає випадаючий список:



Для написання простих програм з обробки даних досить перерахованих вище компонентів. Вивчення згаданих нижче елементів винесене на самостійне опрацювання.

8.4.7. MessageBox

Слугує для виведення на екран діалогового вікна, при цьому використовується метод:

```
public static DialogResult Show(
   string text,
   string caption,
   MessageBoxButtons buttons,
   MessageBoxIcon icon,
   MessageBoxDefaultButton defaultButton,
   MessageBoxOptions options
)
```

Параметри методу:

text – текст повідомлення;

caption – текст заголовка вікна повідомлення;

buttons – кнопки у вікні повідомлення. Може бути одним зі значень переліку MessageBoxButtons:

- AbortRetryIgnore: кнопки Abort (Скасувати), Retry (Повторення), Ignore (Пропустити).
- OK: кнопка OK.
- OKCancel: кнопки OK і Cancel (Скасувати).
- RetryCancel: кнопки Retry (Повтор) і Cancel (Скасувати).

- YesNo: кнопки Yes та No.
- YesNoCancel: кнопки Yes, No і Cancel (Скасувати).

Icon – значок вікна повідомлення. Може бути одним зі значень переліку MessageBoxIcon:

- Asterisk, Information: значок, що складається з літери “i” у нижньому регістрі, поміщеної в коло.
- Error, Hand, Stop: значок, що складається із білого знаку “X” в колі червоного кольору.
- Exclamation, Warning: значок, що складається із знаку оклику в жовтому трикутнику.
- Question: значок, що складається із знаку питання в колі.
- None: піктограми повідомлення немає.

defaultButton – кнопка, на яку за замовчуванням встановлюється фокус. Може бути одним зі значень переліку MessageBoxDefaultButton:

- Button1: перша кнопка із тих, що задаються перерахуванням MessageBoxButtons.
- Button2: друга кнопка.
- Button3: третя кнопка.

options – параметри вікна повідомлення. Може бути одним зі значень переліку MessageBoxOptions:

- DefaultDesktopOnly: вікно повідомлення відображається на активному робочому столі.
- RightAlign: текст вікна повідомлення вирівнюється праворуч.
- RtlReading: всі елементи вікна розташовуються у зворотному порядку (справа наліво).
- ServiceNotification: вікно повідомлення відображається на активному робочому столі, навіть якщо в системі не зареєстровано жодного користувача.

Детальний опис наведено тут: https://abitap.com/4-16-element-messagebox/

8.4.8. OpenFileDialog, SaveFileDialog

Діалоги для відкриття та збереження файлів.

Детальний опис наведено тут: https://abitap.com/4-17-openfiledialog-ta-savefiledialog/

8.4.9. Меню і панелі інструментів

Детальний опис меню і панелей інструментів наведено тут: https://abitap.com/5-1-panel-instrumentiv-toolstrip/

Запитання для самоперевірки

1. Які технології можна використовувати для створення інтерфейсу під Windows?
2. Як створити форму і швидко створити інтерфейс за допомогою компонентів?

3. Що таке властивості компонентів і події? Як реалізується реакція на події в C#?

4. Які основні властивості мають екранні форми?

5. Що таке компоненти –контейнери?

6. Які основні елементи управління застосовуються для створення інтерфейсу?

Питання з теми, що виносяться на самостійне опрацювання

1. Компоненти Windows Forms і їх використання для створення інтерфейсу.

2. Компоненти WPF і їх використання для створення інтерфейсу.

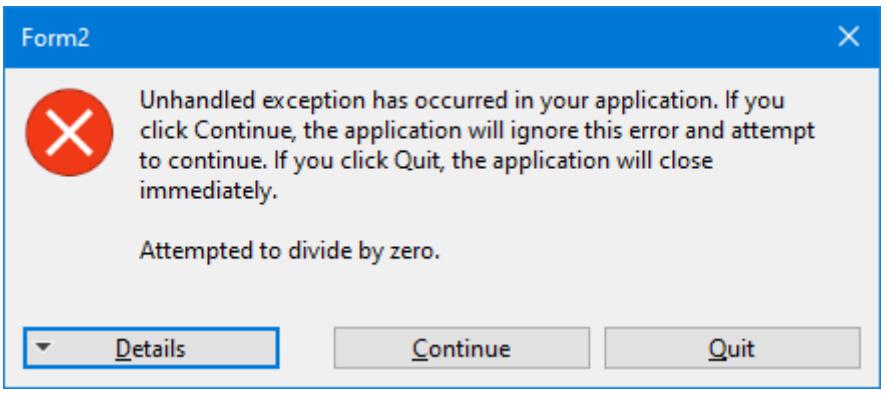
3. Використання OpenFileDialog, SaveFileDialog і меню та панелей інструментів.

Лекція 9. Обробка виняткових ситуацій. Фільтри винятків. Створення винятків. Робота з файлами в C#. Узагальнення (generics).

9.1 Обробка виняткових ситуацій. Конструкції try-catch-finally, try-catch, try-finally.

При виконанні програми можливе виникнення помилок часу виконання (run-time errors). Як правило, причинами таких помилок є некоректне введення даних (наприклад, рядка замість числа), помилки роботи з файлами (спроба відкрити неіснуючий файл, прочитати дані після кінця файлу, апаратні збої), ділення на нуль та подібні дії. У ранніх мовах програмування контроль виникнення помилок покладався на програміста; у більшості випадків програма просто аварійно завершувалася.

У C# наслідком помилок часу виконання є виникнення **виняткових ситуацій** (винятків, exception). Якщо такий виняток залишити не обробленим, консольна програма або програма на основі WPF припинить виконання без будь-яких повідомлень про помилку, а програма з графічним інтерфейсом на основі Windows Forms виведе на екран повідомлення про помилку на зразок такого:



При цьому, якщо клацнути по кнопці «Quit», програма припинить виконання, по кнопці «Continue» – припинить виконання обробника, в якому виник виняток. При цьому усі відкриті програмою файли залишаться відкритими, а виділені ресурси системи не звільненими.

Для того, щоб дати можливість програмісту коректно обробити виникнення виняткових ситуацій (закрити відкриті файли, повідомити користувачу про помилку), і забезпечити подальше виконання програми замість аварійного завершення, у C# введено конструкцію:

```
try
{
  //Оператори
}
catch
{
  //Оператори

}
finally
```

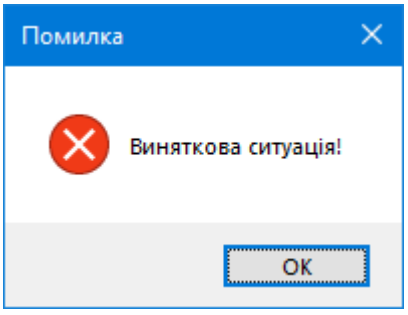
```
{
   //Оператори

}
```

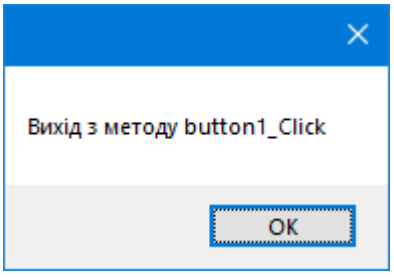
При виконанні такої конструкції спершу виконуються оператори в блоці після **try**. Якщо при виконанні виникне виключна ситуація, виконуються оператори в блоці **catch** (тут можна вивести повідомлення користувачу про помилку). Останніми виконуються оператори в блоці **finally** – незалежно від того, був виняток чи ні. У блоці finally слід виконати операції, необхідні для коректного завершення роботи методу – наприклад, закрити файли і порти, послати сигнал завершення роботи на апаратуру та ін. Приклад:

```
private void button1_Click(object sender, EventArgs e)
{
    try
    {
        int i = 1, j = 0, k;
        k = i / j;
        MessageBox.Show(k.ToString());
    }
    catch
    {
        MessageBox.Show("Виняткова ситуація!", "Помилка",
                        MessageBoxButtons.OK,MessageBoxIcon.Error);
    }
    finally
    {
        MessageBox.Show("Вихід з методу button1_Click");
    }
}
```

У даному методі – обробнику клацання по кнопці button1 – спеціально введено ділення на нуль. Як наслідок, значення k ніколи не буде виведено на екран, замість цього спершу з’явиться повідомлення:



(відпрацює блок catch) а потім – повідомлення,



як наслідок роботи блока finally.

Якщо у програміста немає необхідності використовувати блок catch або блок finally, він може опустити (не використовувати) його – але не обидва одночасно, оскільки це призведе до помилки компіляції. За відсутності блоку finally, метод не передбачатиме дій, які необхідно обов'язково виконати при завершенні його роботи – нормальному чи аварійному. Із блоком catch не все так просто.

Нехай у нас метод Method1 викликав Method2. За відсутності в методі Method2 блока catch, при виникненні виключення буде аварійно зупинене виконання методу Method2, в якому це виключення виникло. Виконання програми продовжиться в методі Method1, який викликав Method2; якщо в ньому є блок catch, то виключення буде оброблено в методі Method1. У випадку, коли в Method1 немає блоку catch, він також аварійно завершиться, і керування потрапить у метод, який викликав Method1. Такий пошук продовжиться вгору по всьому ланцюжку (стеку) викликів. Якщо в жодному із методів виключну ситуацію оброблено не буде, програма аварійно завершиться.

9.2 Форми блока catch. Фільтри винятків. Створення винятків. Ключове слово throw.

Блок catch має кілька форм. Перша

```
catch
{
  //Оператори
}
```

спрацьовує, якщо в блоці try виникла будь-яка виняткова ситуація.

Друга

```
catch (<тип виключення>)
{
  //Оператори
}
```

обробляє **лише винятки заданого типу,** наприклад:

```
catch (DivideByZeroException)
{
    MessageBox.Show("Діленя на нуль!", "Помилка",
                    MessageBoxButtons.OK,MessageBoxIcon.Error);
}
```

Третя форма

```
catch (<тип виключення> <змінна>)
{
  //Оператори
}
```

обробляє винятки лише заданого типу, причому в <змінну> буде поміщено інформацію про виняток. Наприклад, блок

```
catch (Exception ex)
{
```

```
        MessageBox.Show("Виняткова ситуація: "+ex.Message, "Помилка",
                        MessageBoxButtons.OK,MessageBoxIcon.Error);
    }
```

виведе повідомлення про виняткову ситуацію із системним повідомленням (ex.Message) про причини її виникнення.

Фільтри винятків дають змогу обробляти винятки залежно від умов:

```
catch (<тип виключення> [змінна]) when (<умова>)
{
  //Оператори
}
```

У цьому випадку виняток буде оброблятися, лише коли умова істинна, наприклад:

```
catch (HttpRequestException ex) when (ex.Message.Contains("301"))
{
    return "Сайт переїхав";
}
```

Всі винятки в C# – це класи, які породжено від загального класу винятків **Exception**. Цей клас містить такі властивості:

InnerException зберігає інформацію про виняток, що спричинив поточний виняток.

Message - повідомлення про виняток.

Source - ім’я об’єкта (або збірки), який викликав виняток.

StackTrace повертає рядкове представлення стеку викликів, які призвели до винятку.

TargetSite повертає метод, у якому було викликано виняток.

Оскільки клас Exception є предком всіх винятків, то блок catch (Exception e) оброблятиме всі винятки, які можуть виникнути.

Типів винятків у C# багато, ось лише деякі з них:

DivideByZeroException – ділення на нуль.

ArgumentOutOfRangeException – аргумент поза діапазоном допустимих значень.

ArgumentException – у метод для параметра передається некоректне значення.

IndexOutOfRangeException – індекс елемента масиву або колекції поза діапазоном допустимих значень.

InvalidCastException – неприпустиме перетворення типів.

NullReferenceException – звернення до об’єкта, який дорівнює null (невизначений).

IOException – помилка введення-виведення, і його потомки:

DirectoryNotFoundException – каталог (теку) не знайдено.

EndOfStreamException – досягнуто кінець потоку введення-виведення (при спробі прочитати щось за кінцем файлу).

FileNotFoundException – файл не знайдено.

FileLoadException – помилка при завантаженні файлу (збірка .NET знайдена, але завантажити її не вдається).

PathTooLongException – шлях занадто довгий (довший, ніж максимальна дозволена системою довжина шляху файлу).

У одного блока try може бути декілька блоків catch:

```
try
{
}
catch (DivideByZeroException)
{
    MessageBox.Show("Ділення на нуль!", "Помилка",
                    MessageBoxButtons.OK, MessageBoxIcon.Error);
}
catch (IndexOutOfRangeException)
{
    MessageBox.Show("Вихід за діапазон!", "Помилка",
                    MessageBoxButtons.OK, MessageBoxIcon.Error);
}
catch (Exception ex)
{
    MessageBox.Show("Виняткова ситуація: " + ex.Message, "Помилка",
                    MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

однак, з точки зору раціональності, краще використовувати один універсальний блок, що оброблятиме всі типи винятків.

Користувач може створити свій тип винятку. Наприклад, якщо було введено рядок із некоректними даними, користувач може створити і згенерувати свій виняток:

```
class NameException : Exception
{
    public NameException(string message) : base(message) { }
}

        ...

    private void button1_Click_1(object sender, EventArgs e)
    {
        try
        {
            //Вичитка введеного імені
            string UserName = textBox1.Text;
            if (UserName.Length == 0)
                throw new NameException("Пусте ім'я користувача!");
        }
        catch (Exception ex)
        {
            MessageBox.Show("Виняткова ситуація: " + ex.Message, "Помилка",
                            MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
    }
```

У наведеному прикладі створюється новий клас винятку, і за умови, що введене ім’я користувача пусте, генерується виняток цього класу оператором **throw**. За допомогою throw можна генерувати також і винятки стандартних типів, наприклад того ж класу Exception:

```
throw new Exception("Виняток!");
```

9.3 Робота з файлами в C#.

Для того, щоб програма могла зберігати оброблені дані, вона повинна мати можливість роботи з файлами: створювати файли, читати та змінювати їх зміст. Для роботи з файлами (як текстовими, так і бінарними) в C# застосовується клас FileStream (https://learn.microsoft.com/en-us/dotnet/api/system.io.filestream?view=net-8.0; https://abitap.com/29-4-filestream-chytannya-ta-zapys-fajlu/), визначений в просторі імен System.IO.

Найпростіший (і найуживаніший) конструктор FileStream виглядає так:

```
FileStream(string filename, FileMode mode)
```

де filename – ім'я файлу, а mode – режим відкриття файлу:

FileMode.Append – якщо файл існує, дані додаються в кінець файлу. Якщо файлу немає, він створюється. Файл відкривається лише для запису.

FileMode.Create – створюється новий файл. Якщо такий файл вже існує, він перезаписується.

FileMode.CreateNew – створюється новий файл. Якщо такий файл вже існує, то виникне виняток.

FileMode.Open – відкриває файл. Якщо файл не існує, виникне виняток.

FileMode.OpenOrCreate – якщо файл існує, він відкривається, якщо ні – створюється новий.

FileMode.Truncate – якщо файл існує, він перезаписується. Файл відкривається лише для запису.

Для звільнення ресурсів операційної системи, пов'язаних із файлом, клас FileSteam реалізує інтерфейс IDisposable. Для звільнення задіяних ресурсів слід скористатися методом Close:

```
FileStream? fs = null;
try
{
    fs = new FileStream("filename.dat", FileMode.OpenOrCreate);
    // робота з файлом через fs
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
finally
{
    fs?.Close();
}
```

Найважливіші *властивості* об'єкту FileStream:

Length – довжина потоку в байтах;

Position – поточна позиція у потоці;

Name – абсолютний шлях до відкритого файлу.

Клас FileStream надає для роботи з файлами такі *методи*:

CopyTo(Stream destination) – копіює дані з поточного потоку в потік destination.

void Flush() – скидає вміст буфера потоку у файл.

int Read(byte[] array, int offset, int count) – зчитує дані з файлу масив байтів і повертає кількість успішно зчитаних байтів. Приймає три параметри:

array – масив байтів, куди будуть поміщені дані;

offset – зміщення в байтах у масиві array, в який зчитані з файла байти будуть записані;

count – максимальна кількість байтів, призначених для читання. Якщо у файлі знаходиться менша кількість байтів, то всі вони будуть прочитані.

long Seek(long offset, SeekOrigin origin) – встановлює позицію в потоці зі зміщенням на кількість байт, зазначених у параметрі offset. Параметр origin вказує, звідки повинно бути відраховане зміщення: SeekOrigin.Begin – від початку потоку, SeekOrigin.Current – від поточної позиції, SeekOrigin.End – із кінця потоку.

void Write(byte[] array, int offset, int count) – записує в файл дані з масиву байтів. Приймає такі параметри:

array – масив байтів, звідки дані записуватимуться у файл;

offset – зміщення в байтах у масиві array, звідки починається запис байтів у файл;

count – максимальна кількість байтів, призначених для запису.

Для того, щоб записати в файл дані, відмінні від масиву байт, слід записати зміст відповідних змінних у масив байт (і навпаки), для чого слід скористатися класом BitConverter. BitConverter має методи для перетворення даних на масив байт (GetBytes) та для перетворення масиву байт на прості типи, наприклад ToInt32, ToDouble і т.д. Опис цих методів є в системі допомоги Visual Studio (https://learn.microsoft.com/en-us/dotnet/api/system.bitconverter?view=net-8.0). Основні методи BitConverter наведені нижче в таблиці:

Тип	Перетворення в масив байт	Перетворення з масиву байт
Boolean	GetBytes(Boolean)	ToBoolean
Char	GetBytes(Char)	ToChar
Double	GetBytes(Double) або DoubleToInt64Bits(Double) або DoubleToUInt64Bits(Double)	ToDouble або Int64BitsToDouble або UInt64BitsToDouble
Int16	GetBytes(Int16)	ToInt16
Int32	GetBytes(Int32)	ToInt32
Int64	GetBytes(Int64)	ToInt64
Single	GetBytes(Single) або SingleToInt32Bits(Single)	ToSingle або Int32BitsToSingle

Тип	Перетворення в масив байт	Перетворення з масиву байт
	або SingleToUInt32Bits(Single)	або UInt32BitsToSingle
UInt16	GetBytes(UInt16)	ToUInt16
UInt32	GetBytes(UInt32)	ToUInt32
UInt64	GetBytes(UInt64)	ToUInt64

Нижче, як приклад, наведено код, який записує в файл число double і масив int, а потім вичитує ці дані з файлу і виводить їх на екран:

```
using System.ComponentModel;
using System.Text;

FileStream? fs = null;
try
{
    fs = new FileStream("filename.dat", FileMode.OpenOrCreate);

    //Double
    double x = 5.01;
    //Перетворення double в array of byte
    byte[] bt = BitConverter.GetBytes(x);
    // запис у файл
    fs.Write(bt, 0, bt.Length);

    //Масив int
    int[] arr = { 1, 2, 3, 4, 5 };
    //Спочатку пишемо довжину масиву
    bt = BitConverter.GetBytes(arr.Length);
    fs.Write(bt, 0, bt.Length);

    //Потім весь масив в циклі
    for (int i=0; i<arr.Length; i++) {
        bt = BitConverter.GetBytes(arr[i]);
        fs.Write(bt, 0, bt.Length);
    }
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
finally
{
    fs?.Close();
}
//============================================================================
try
{
    fs = new FileStream("filename.dat", FileMode.Open);

    //Double
    double x1;

    //Виділяємо пам'ять під буфер
    byte[] bt = new byte[sizeof(double)];
    //Читаємо в буфер
    fs.Read(bt, 0, bt.Length);
    //Перетворення array of byte в double
    x1=BitConverter.ToDouble(bt);

    //Вивод прочитаного
```

```
    Console.WriteLine($"Прочитано x={x1}");

    //Масив int
    int[] arr;
    //Спочатку читаємо довжину масиву
    bt = new byte[sizeof(int)];
    //Читаємо в буфер довжину масиву
    fs.Read(bt, 0, bt.Length);
    int len=BitConverter.ToInt32(bt, 0);

    arr = new int[len];
    //Потім читаємо весь масив
    bt = new byte[sizeof(int)];
    for (int i = 0; i < arr.Length; i++)
    {
        //Читаємо в буфер елементи масиву
        fs.Read(bt, 0, bt.Length);
        arr[i] = BitConverter.ToInt32(bt, 0);
    }

    //Вивод прочитаного
    Console.WriteLine("Прочитаний масив:");
    foreach (int arr_item in arr)
        Console.WriteLine(arr_item);
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
finally
{
    fs?.Close();
}
```

Для збереження таким чином класу/структури слід послідовно записувати у файл значення всіх його/її полів. Вичитувати значення полів класу/структури слід в порядку запису.

При відкритті файлу FileStream встановлює на початок файлу так званий курсор потоку (його поточне положення – властивість Position). Курсор потоку вказує, з якої саме позиції в файлі буде вестись читання або запис; кожна операція читання/запису переміщує курсор потоку на задану кількість байт вперед. Встановити курсор на потрібну позицію (байт) у файлі можна присвоївши властивості Position відповідний номер байту. Для встановлення курсору на перший байт файлу слід встановити значення Position рівним 0.

Метод **Seek** дає змогу перемістити курсор потоку на задане зміщення offset. Параметр origin вказує, звідки повинно бути відраховане зміщення: SeekOrigin.Begin – від початку потоку, SeekOrigin.Current – від поточної позиції, SeekOrigin.End – із кінця потоку. Якщо зміщення більше нуля, курсор рухається вперед, якщо менше – назад. При спробі встановити курсор за межі потоку виникне виключення. Приклад (робота ведеться з файлом із переднього прикладу):

```
//Виставити курсор потоку на початок файлу
fs.Seek(0, SeekOrigin.Begin);
//Встановити курсор в файлі на перший записаний елемент масиву
fs.Seek(sizeof(double) + sizeof(int), SeekOrigin.Current);
//Встановити курсор в файлі на третій записаний елемент масиву
```

```
fs.Seek(2*sizeof(int), SeekOrigin.Current);
//Повернтуи курсор на перший елемент масиву
fs.Seek(-2 * sizeof(int), SeekOrigin.Current);
//Виставити курсор потоку на кінець файлу
fs.Seek(0, SeekOrigin.End);
```

Зауважимо, що FileStream орієнтовано на роботу з бінарними файлами з довільним доступом, тому застосовувати його для роботи з текстовими файлами незручно. Для роботи з текстовими файлами в просторі імен System.IO визначено класи StreamReader та StreamWriter.

Клас StreamWriter призначений для запису тексту в текстовий файл і має такі конструктори:

StreamWriter(string path) – відкриває файл з іменем path для запису. Якщо файл існує, він буде перезаписаний.

StreamWriter (string path, bool append) – те саме, але якщо параметр append дорівнює true, нові дані додаються в кінець файлу. Якщо append == false, то файл перезаписується наново.

StreamWriter (string path, bool append, System.Text.Encoding encoding) - параметр encoding вказує на кодування, яке застосовуватиметься під час запису: System.Text.ASCIIEncoding, System.Text.UnicodeEncoding, System.Text.UTF32Encoding, System.Text.UTF7Encoding, System.Text.UTF8Encoding. За замовчуванням – UTF-8.

Основні методи класу:

Close() – закриває записуваний файл і звільняє всі ресурси.

void Flush() – записує в файл дані, що залишилися в буфері, і очищає буфер.

void Write(string value) – записує в файл прості типи (int, double, char, string тощо). Відповідно має ряд перевантажених версій для запису даних елементарних типів, наприклад Write(char value), Write(int value), Write(double value) і т.д.

WriteLine(string value) – також записує дані, тільки після запису додає у файл символ закінчення рядка .

Клас StreamReader призначений для читання тексту з текстового файлу і має такі конструктори:

StreamReader(string path) – відкриває файл з іменем path для читання. Якщо файлу не існує, виникне виняток.

StreamReader (string path, System.Text.Encoding encoding) – параметр encoding задає кодування для читання файлу.

Основні методи класу:

void Close() – закриває файл і звільняє всі ресурси.

int Peek() – повертає наступний доступний символ, якщо символів більше немає, то повертає -1.

int Read() – зчитує та повертає наступний символ у чисельному поданні. Має перевантажену версію: Read(char[] array, int index, int count), де array – масив, куди зчитуються символи, index – індекс в масиві array, починаючи з

якого записуються символи, що зчитуються, і count – максимальна кількість символів, що зчитуються.

string ReadLine() – зчитує один рядок у файлі. Якщо досягнуто кінець файлу, ReadLine повертає null.

string ReadToEnd() – зчитує весь текст із файлу.

Приклад:

```
//Запис
StreamWriter? sw = null;
try
{
    sw = new StreamWriter("text.txt");

    sw.WriteLine("Клас StreamWriter призначений для запису тексту в текстовий
файл.");
    sw.WriteLine("Клас StreamReader призначений для читання тексту з текстового
файлу.");
}
finally
{
    sw?.Close();
}
//Читання
StreamReader? sr = null;
try
{
    sr = new StreamReader("text.txt");

    string? str;
    do
    {
        str = sr.ReadLine();
        if (str != null)
            Console.WriteLine(str);
    } while (str != null);
}
finally
{
    sr?.Close();
}
```

9.4. Узагальнення (generics)

Середовище .NET підтримує узагальнені типи і створення узагальнених методів. Концепцію узагальнених типів взято з C++; засіб мови C++ із подібним функціоналом називається шаблонами (templates).

Почнемо з **узагальнених методів**. Узагальнений метод в C# оголошується так:

```
[модифікатори] <тип результату> <ім'я>< <тип1>, <тип2>, …>([формальні
параметри]) [where <тип1>:<обмеження> where <тип2>:<обмеження> …]
{
  //Оператори
}
```

Оголошення узагальненого методу відрізняється від звичайного тим, що в дужках < > вказують назви типів (так званих універсальних параметрів); при

використанні узагальненого методу замість цих типів буде підставлено конкретні типи, з якими має працювати метод.

Застосування узагальнених методів має сенс тоді, коли є типовий алгоритм, який ***однаково*** може обробляти дані різних типів. Як приклад, візьмемо просту задачу: поміняти місцями значення двох змінних. Звичайно, можна оголосити кілька перезавантажених методів – наприклад, один для типу int, другий для типу double, але тоді доведеться для кожного типу писати свій метод, що незручно. Використання узагальнених типів дає змогу розв'язати цю задачу так:

```
static void SwapVals<T>(ref T a, ref T b)
{
    T x = b;
    b = a;
    a = x;
}
```

Оголошення узагальненого методу в дужках <> містить тип (універсальний параметр) T; обидва параметри методу мають цей тип. Для того, щоб скористатися цим методом аби поміняти місцями значення конкретних змінних, при виклику методу після ідентифікатора метода можна вказати їх тип, хоча компілятор може визначити тип змінних за їх іменами. Наприклад:

```
//Робота з double
double x=1, y=2;
SwapVals<double>(ref x,ref y); //Явно вказано тип
SwapVals(ref x,ref y); //Тип не вказано

//Робота з масивом чисел типу int
int[] a1 = { 1, 2, 3 };
int[] a2 = { 4, 5, 6 };
SwapVals<int[]>(ref a1, ref a2);
```

Така універсальність узагальненого методу досягається тим, що всі типи C# сумісні з класом Object; як наслідок, вказаний у методі тип T за замовчуванням сумісний з класом Object, і ми можемо в узагальненому методі використовувати, наприклад, метод ToString().

Якщо в узагальненому методі необхідно використати певне поле, властивість або метод узагальненого параметра (наприклад, поле Host класу MailAddress), то просто написати щось на зразок

```
a.Host = "127.0.0.1";
```

не вдасться – оскільки тип T за замовчуванням те саме що Object, який не містить такого поля. Щоб запобігти подібним ситуаціям, слід визначити, які саме типи допустимо підставляти в узагальнений метод, вказавши після **where** умову допустимості підстановки, наприклад тип класу:

```
static void DoExchange<T>(ref T a, ref T b) where T:MailAddress
```

Таке оголошення означає, що тип T може бути лише класом MailAddress або його нащадком, і всередині класу Account з полями і змінними типу T можна працювати як з об'єктами класу MailAddress.

Крім обмеження у вигляді імені типу (класу або інтерфейсу), в C# допустимі також такі обмеження:

class – тип (універсальний параметр) обов'язково повинен бути класом (типом-посиланням);

struct – тип (універсальний параметр) обов'язково повинен бути структурою;

new() – тип (універсальний параметр) повинен мати загальнодоступний (public) конструктор без параметрів.

Для використання як обмеження стандартних типів (int, float, double…) слід використовувати обмеження srtuct:

```
static void DoExchange<T>(ref T a, ref T b) where T:struct
```

При цьому використовувати конкретні імена типів (як-от where T:int) заборонено.

Якщо для універсального параметра слід встановити кілька обмежень, їх слід перерахувати через кому, причому:

- першою повинна йти назва конкретного класу/class/struct (щось одне з трьох);
- далі – назва інтерфейсу;
- останнім обмеженням може бути new().

Обмеження для кількох типів (універсальних параметрів) записують послідовно через пробіл, для кожного ключове слово where своє:

```
static void Test<N, K>(N name, K key)
            where N : struct where K : class, IComparable, new();
```

Крім узагальнених методів, у C# можна також використовувати **узагальнені класи**, тобто класи, які використовують узагальнені параметри. Оголошення узагальненого класу можна зробити так:

```
[модифікатори] class <ім'я>< <тип1>, <тип2>, …> [where <тип1>:<обмеження> where
<тип2>:<обмеження> …]
{
  //Поля, властивості, методи
}
```

Узагальнений клас для облікового запису можна представити так:

```
class Account<Tnam, Tpwd>
{
    public Tnam Name { get; set; }
    public Tpwd Password { get; set;}

    //Конструктор
    public Account()
    {
        Name = default(Tnam);
        Password = default(Tpwd);
```

```
        }
        public Account(Tnam name, Tpwd pwd)
        {
            Name = name;
            Password = pwd;
        }
        //Друк
        public void Print()
        {
            Console.WriteLine($"Name={Name} Password={Password}");
        }
    }
```

Цей клас має два узагальнених параметри типів – Tnam і Tpwd для імені користувача і пароля відповідно. Для типового застосування в системі контролю доступу до e-mail можливе таке використання цього класу:

```
  Account<string, string> a = new Account<string, string>("Іван Сила", "QWERTY");
  Account<string, string> b = new Account<string, string> { Name = "Аліна
Заборовська",
                                                            Password = "12345"  };
```

У наведеному прикладі створено два об’єкти узагальненого класу – один з ініціалізатором, інший - за допомогою конструктора. Оголошення об’єктів узагальненого класу та їх використання нічим не відрізняються від звичайних об’єктів – крім того, що слід вказати типи узагальнених параметрів.

Якщо замість імені користувача (тип string) використовується e-mail (клас MailAddress з простору імен System.Net.Mail), а замість рядка-пароля користувачу потрібно ввести число, об’єкт узагальненого класу Account слід створювати так:

```
Account<MailAddress, int> c = new Account<MailAddress, int>(new
                                    MailAddress("foo@foo.com"), 123);
```

Обмеження в узагальнених класах застосовують так само, як і в узагальнених методах.

Клас у C# може бути успадкований від узагальненого класу. При цьому є три варіанти:

1) Узагальнений клас – нащадок типізується тим же набором типів, що й базовий:

```
class AccountWithRights<Tnam, Tpwd> : Account<Tnam, Tpwd>
{
    public bool[] Rights { get; set;}
    public AccountWithRights(Tnam name, Tpwd pwd, bool[] r) : base(name, pwd)
    {
        Rights = r;
    }
}
```

Використання такого класу-нащадку не відрізняється від використання базового класу.

2) Створюється звичайний клас-нащадок на основі узагальненого із заданими типами:

```
        class AccountForMail : Account<MailAddress, int>
        {
            public string ServerURL { get; set; }
            public AccountForMail(MailAddress name, int pwd, string URL) : base(name,
pwd)
            {
                ServerURL = URL;
            }
        }
```

Використання такого класу таке ж саме, як і інших звичайних класів. Зверніть увагу на те, що змінні типів AccountForMail і Account<MailAddress, int> сумісні за присвоюванням (бо AccountForMail – похідний клас):

```
Account<MailAddress, int> ss=new AccountForMail(new MailAddress("foo@foo.com"),
123,"dd");
```

3) Створюється похідний узагальнений клас із зовсім іншим набором універсальних параметрів (типів) і своїми обмеженнями:

```
        class SuperAccount<T, K> : Account<MailAddress, int> where K : struct
        {
            public T Time { get; set; };
            public K Key { get; set; };
            public SuperAccount(MailAddress name, int pwd, T tm, K key) : base(name,
pwd)
            {
                Time = tm;
                Key = key;
            }
        }
```

Якщо в базовому класі було використано обмеження, то в похідних класах повинно бути визначене подібне обмеження. Як приклад, якщо в базовому класі тип Tnam має обмеження class, у похідних повинно бути застосовано обмеження class або ім’я конкретного класу.

Узагальнені типи широко використовуються в бібліотеках C#. Такі класи як List<T> – список, Queue<T> – черга, Stack<T> – стек, Dictionary<TKey, TValue> – словник «ключ-значення», а також ряд інших є узагальненим класами. Усі ці класи є колекціями (простір імен System.Collections.Generic), і їх члени можна перебирати циклом foreach; основні властивості та методи згаданих класів наведено нижче.

List<T>:

List<T>.Count – кількість елементів списку;

List<T>.Add(T) – додає значення у список;

List<T>.IndexOf(T) – шукає значення в списку, і повертає його індекс (починається з 0) або -1, якщо значення не знайдено;

List<T>.Clear() – очищує список;

List<T>.Remove(T) – видаляє зі списку вказаний елемент;

List<T>.RemoveAt(Int32) – видаляє зі списку елемент із заданим індексом;

List<T>.Sort(Comparison<T>) – сортує список, використовуючи узагальнений делегат Comparison<T>, заданий програмістом.

Також клас реалізує методи інтерфейсу IList, зокрема:

IList.Item[Int32] – повертає елемент списку з заданим індексом;

IList.Insert(Int32, Object) – вставляє елемент на задану позицію списку;

Queue<T>:

Queue<T>.Count – кількість елементів черги;

Queue<T>.Enqueue(T) – додає значення у кінець черги;

Queue<T>.Dequeue() – видаляє з черги і повертає перший елемент;

Queue<T>.Peek() – повертає перший елемент черги, не видаляючи його з черги;

Queue <T>.Contains(T) – перевіряє, чи є в черзі заданий елемент.

Dictionary<TKey,TValue>:

Dictionary<TKey,TValue>.Count – кількість елементів словника;

Dictionary<TKey,TValue>.Comparer – клас для порівняння елементів словника;

Dictionary<TKey,TValue>.Item[TKey] – елемент словника із заданим значенням ключа;

Dictionary<TKey,TValue>.Add(TKey, TValue) – додає пару «ключ-значення» в словник;

Dictionary<TKey,TValue>.Clear() – очищує словник;

Dictionary<TKey,TValue>.ContainsKey(TKey) – перевіряє наявність ключа в словнику;

Dictionary<TKey,TValue>.ContainsValue(TValue) – перевіряє наявність значення в словнику;

Dictionary<TKey,TValue>.Remove(TKey, TValue) – видалення пари «ключ-значення» зі словника.

Детальне вивчення цих класів виноситься на самостійне опрацювання; матеріали доступні тут: https://learn.microsoft.com/en-us/dotnet/standard/collections/ .

На цьому курс з основ C# завершено. Через обмежений обсяг курсу не охопленими залишилося багато питань, зокрема LINQ, робота з JSON, підтримка роботи із базами даних, створення багато-потокових програм. Однак, базового курсу досить для того, щоб на його основі розпочати розробку власних програм для вирішення практичних задач.

Якщо функціоналу, запропонованого стандартними бібліотеками C# недостатньо – використовуйте пакети NuGet. За потреби, свій функціонал ви тепер зможете дописати самі.

Запитання для самоперевірки

1. Що таке виключна ситуація (виключення, exception)?

2. Як обробляється виключення в конструкції try-catch-finally?

3. Що відбудеться, якщо виключення не буде оброблено жодним блоком catch?

4. Які є форми блоку catch? Що таке фільтр винятків?

5. Як створити свій виняток?

6. Як в C# працювати з текстовими та бінарними файлами?

7. Що таке узагальнений метод і коли він використовується?

8. Що таке обмеження на узагальнені параметри, і навіщо вони потрібні?

9. Як оголосити узагальнений клас і як можна успадкувати від нього новий клас?

10. Які узагальнені класи-колекції містить стандартна бібліотека C#?

Питання з теми, що виносяться на самостійне опрацювання

1. Робота з бінарними файлами: як зберігати і зчитувати об’єкти.
2. Класи List<T> – список змінних, Queue<T> – черга, Stack<T> – стек, Dictionary<TKey, TValue> – словник «ключ-значення» і робота з ними.

Список рекомендованої літератури

1. Коноваленко І.В. Програмування мовою C# 7.0 : навчальний посібник / Коноваленко І.В., Марущак П.О., Савків В.Б. – Т.: ТНТУ, 2017. – 300 с.
2. Сучасні технології програмування: Частина І. / В. І. Бендюг, Б. М. Комариста. – К.: КПІ ім. Ігоря Сікорського, 2019. – 269 с.
3. Мова програмування C# 12 (.NET 8) [Електронний ресурс] / Режим доступу: https://abitap.com/category/c/ – Заголовок з екрану.
4. Windows Forms [Електронний ресурс] / Режим доступу: https://abitap.com/category/35-grafichni-windows-zastosunky/ – Заголовок з екрану.
5. Stellman, Andrew. Head First C#: A Learner's Guide to Real-World Programming with C# and .Net Core, Fourth Edition / Andrew Stellman, Jennifer Greene.– S.: O'Reilly, 2021. – 800 c.
6. Hejlsberg, Anders. The C# Programming Language, Fourth Edition. / Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, Peter Golde. – B.: Pearson, 2010. – 864 p.
7. Nagel, Christian. Professional C# 2012 and . NET 4.5. / Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, Morgan Skinner. – H.: Wiley, 2012. – 1584 p.
8. C# documentation [Електронний ресурс] / Режим доступу: https://learn.microsoft.com/en-us/dotnet/csharp/ – Заголовок з екрану.
9. The complete WPF tutorial [Електронний ресурс] / Режим доступу: https://wpf-tutorial.com – Заголовок з екрану.
10. Івохін, Євген Вікторович. Розроблення додатків засобами мови програмування C# : посібник / Є.В. Івохін, М.Ф. Махно, О.Г. Піскунов ; Міністерство освіти і науки України, Київський національний університет імені Тараса Шевченка. - Київ : ВПЦ "Київський університет", 2021. – 134 с.
11. Соловей, Людмила Валентинівна. Основи програмування мовою C# : навчальний посібник для студентів хімічних спеціальностей, у тому числі для іноземних студентів / Л.В. Соловей, Н.М. Мірошніченко, Т.Г. Бабак, О.О. Голубкіна, Є.Д. Пономаренко ; Міністерство освіти і науки України, Національний технічний університет "Харківський політехнічний інститут". – Харків : НТУ "ХПІ", 2019. – 487 с.