# Заняття 1: Введення, Git, класи та структури

**Git-flow** – Організація процесу розділення версій на гілки (branch) та злиття їх (merge) назад у головну гілку (main/master). Перед процесом злиття в головну гілку роблять **pull request** – коли інші перевіряють код в гілці на помилки і дають дозволи на злиття з головною гілкою.

**Клас** – шаблон з даними (філдами, змінними та ін.) та функціями по якому створюються об'єкти. По замовчуванні клас має ідентифікатор дозволу **internal**. Клас є видимим в межах проекту. Ідентифікатори доступу допомагають реалізувати інкапсуляцію. Бажано прописувати ідентифікатори доступу явно (не зловживати ідентифікаторами по замовчуванню).

Якщо в класі немає інших конструкторів, то під час створення об’єкта буде викликатись **конструктор по замовчуванню**. Він зникає, якщо явно вказаний інший конструктор.

***!!! Хороша практика роботи*** – один клас на один файл. Назва файлу та назва класу має бути однаковими.

Ключове слово **this** – представляє на поточний екземпляр об’єкта класу. Для прикладу його використовують коли назва параметру співпадає з назвою поля:

Public int age;

testFunc(int age)

{

this.age = age;

}

**Ланцюжок виклику конструкторів** – це коли один конструктор викликає інший конструктор в межах даного об’єкта. Виклик відбувається через ключове слово **this**.

public Person() : this(“Undefined”)

{

//additional logic

}

public Person(string name) : this(name, 18)

{

//additional logic

}

public Person(string name, int age)

{

this.name = name;

this.age = age;

}

**Ініціалізатор об’єкта** – альтернативний варіант створення та ініціалізації об’єкта:

var user = new Person {name = “John”, age = 55};

За допомогою ініціалізації об'єктів можна надавати значення всім доступним полям і властивостям об'єкта в момент створення.

При використанні ініціалізації слід враховувати наступні моменти:

За допомогою ініціалізатора ми можемо встановити значення лише доступних із позакласу полів та властивостей об'єкта.

Наприклад, у прикладі вище поля name і age мають модифікатор доступу public, тому вони доступні будь-якої частини програми.

Ініціалізатор виконується після конструктора, тому якщо і в конструкторі, і в ініціалізаторі встановлюються значення тих самих полів і властивостей, то значення, що встановлюються в конструкторі, замінюються значеннями з ініціалізатора.

**Типи даних:**

Цілочисленні типи (byte, sbyte, short, ushort, int, uint, long, ulong)

Типи з плаваючою комою (float, double)

Тип decimal

Тип bool

Тип char

Перерахування enum

Структури (struct)

Посилальні типи: (ссылочные типы данных)

Тип object

Тип string

Класи (class)

Інтерфейси (interface)

Делегати (delegate)

**Кортеж** – колекція, яка представляє константні значення. Має вигляд (string name, int age).

**Деконструктор** дозволяє виконати декомпозицію об'єкта на окремі частини.

var person = new PersonV4("Tom", 33);

(string name, int age) = person;

Для реалізації **деконструктора** необхідно, щоб в класі був реалізований метод Deconstruct

public void Deconstruct(out string personName, out int personAge)

{

personName = name;

personAge = age;

}

Різниця між **типами-посиланнями** (об’єктами) та **типами-значннями** (об’єкти структур, базові типи):

* Спосіб збереження в мап’яті: типи-значень розміщуються в стеку повністю, типи-посилання – в стеку лише посилання на об’єкт, а поля у керованій купі;
* Відмінність в базовому типі: для значимих типів System.ValueType, а для посилальних типів може бути будь-який не sealed тип крім System.ValueType;
* Значимі типи не можуть бути успадкованими;
* При передачі у функцію значимі типи передаються за значенням, а посилальні – за посиланням (за значенням копіюється тільки посиання);
* Типи значень не потребують фіналізації (перевизначення методу System.Object.Finalize()), бо вони ніколи не розміщуються в керованій купі;
* Типи-значення припиняють існування, коли виходять за рамки контексту, в якому визначались, а типи-посилання знищуються збирачем сміття

Спільним для **типів-значень** та **типів-посилань** є те, що для тих і тих можна визначати конструктори.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Характеристика** | **сlass** | **record (он же record class)** | **record struct** | **struct** |
| Ссылочный/значимый | ссылочный | ссылочный | значимый | значимый |
| Есть ли позиционная запись | нет | да | да | нет |
| Неизменяемость (иммутабельность) по умолчанию | нет | с позиционной записью — да | нет | нет |
| Наследование | да | только от record | нет | нет |
| Реализация интерфейсов | да | да | да | да |
| Автоматически генерируются операторы сравнения | сравнение по ссылке | сравнение по значению | сравнение по значению | нет |
| Создание через [инициализатор объектов](https://docs.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/object-and-collection-initializers#object-initializers) | да | с позиционной записью — нет | да | да |
| Автоматически генерируется конструктор без параметров | да | с позиционной записью — нет | да | да |

**Статичний клас** – це клас, який оголошується з ключовим словом static. У порівнянні з нестатичним класом, статичний клас має наступні властивості (відмінності):

* неможна створювати об’єкти статичного класу;
* статичний клас повинен містити тільки статичні члени.

**Упаковка/розпаковка** – приведення типів-значень до типів-посилань та навпаки.

int number = 10;

object my\_number = number; // упаковка

int number\_from\_object = (int)my\_number; // распаковка

Можемо розпакувати лише у вихідний тип даних. В іншому разі буде помилка рантайму.

Посилання:

Завантажити та встановити:

<https://git-scm.com/downloads>

Windows: <http://gitextensions.github.io/>

Windows/Mac: <https://www.sourcetreeapp.com/> чи <https://desktop.github.com/> чи <https://www.gitkraken.com/jc>

Туторіали:

<https://www.w3schools.com/git/>

<https://www.atlassian.com/git/tutorials>

<https://githowto.com/setup>

<https://www.tutorialspoint.com/git/index.htm>

<https://opensource.com/article/18/1/step-step-guide-git>

Static classes:

<https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static>

<https://www.tutorialsteacher.com/csharp/csharp-static>

<https://henriquesd.medium.com/singleton-vs-static-class-e6b2b32ec331#:~:text=Differences%20between%20Singleton%20and%20Static,and%20therefore%20cannot%20be%20inherited.>

<https://dotnethow.net/static-vs-non-static-classes-in-c-understanding-the-differences-and-use-cases/>

<https://debug.to/3494/abstract-class-vs-partial-class-vs-static-class-vs-sealed-class>

Кориснi посилання:

<https://www.geeksforgeeks.org/introduction-to-visual-studio/>

<https://www.edureka.co/blog/visual-studio-tutorial/>

<https://learn.microsoft.com/en-us/visualstudio/version-control/git-with-visual-studio?view=vs-2022>

<https://www.bestprog.net/uk/2018/12/05/static-classes-methods-variables-static-constructors-keyword-static_ua/>

<https://habr.com/ru/articles/675560/>

<https://www.bestprog.net/uk/2019/10/10/c-categories-groups-of-data-types-in-c-value-types-reference-types-basic-data-types-overview-ua/#q03>

# Заняття 2: ООП. Реалізація принципів ООП

**Кортеж** (tuple) – колекція різнотипних значень, але типи відомі наперед.

public static (int sum, double average) GetArrCalculations(int[] numbers)

{

var result = (sum: 0, average: .0);

foreach (var number in numbers)

{

result.sum += number;

}

result.average = Convert.ToDouble(result.sum) / numbers.Length;

return result;

}

var tupleResult = GetArrCalculations(new int[] { 1, 2, 3, 4, 6 });

Console.WriteLine(tupleResult);

Console.WriteLine($"Sum: {tupleResult.sum} Avg: {tupleResult.average}");

**Структура** – значимий тип даних, значення яких записується в стеках. Зараз використовується рідко і у випадках коли є дуже простий набір значимих типів даних. Переважно зараз використовуються **рекорди** та **класи**. Для використання структури її необхідно ініціалізувати. Для ініціалізації створення об'єктів структури, як і у випадку класів, застосовується виклик конструктура з оператором new. Навіть якщо в коді структури не визначено жодного конструктора, проте є, як мінімум, один конструктор - конструктор за замовчуванням, який генерується компілятором (конструктор не приймає параметрів і створює об'єкт структури зі значеннями за замовчуванням). Можна викликати **конструктори ланцюжком**. Також зручно користуватись ключовим словом **with** для копіювання значень одної структури в іншу (юзається і в рекордах).

**Копіювання по посиланню та по значенні** – по посиланні ми працюємо з адресом пам’яті і зміни змінної через будь яке скопійоване посилання впливають на вихідне значення; по значенні ми працюємо з копією значення і зміни в будь якій скопійованій змінній не впливають на вихідне значення.

**readonly** vs **const**: readonly можна змінити в конструкторі, а константу не можна змінити після оголошення (оголошення та ініціалізація відбуваються одночасно). **readonly** юзають найчастіше при **dependency injection** - коли є змінна (поле) для зберігання, для прикладу, мікросервісу. Тоді ми лише раз, через конструктор, ініціалізуємо цю змінну відповідним мікросервісом.

**Статичний клас** – це клас, який оголошується з ключовим словом static. У порівнянні з нестатичним класом, статичний клас має наступні властивості (відмінності):

* неможна створювати об’єкти статичного класу;
* статичний клас повинен містити тільки статичні члени.

**Статичні класи, методи та змінні** ефективні у наступних випадках:

* коли потрібно створити так звані методи розширення. Методи розширення використовуються для розширення функцій класу. Ці методи є статичними;
* коли у програмі є деякий спільний ресурс, до якого можуть мати звертання методи різних класів які обробляють даний ресурс (читають або змінюють його значення). Цей спільний ресурс оголошується як статична змінна. Наприклад, таким ресурсом може бути деякий лічильник викликів, метод що реалізує унікальну обробку, унікальна файлова змінна (ресурс) тощо;
* коли потрібно об’єднати між собою групи статичних методів;
* коли потрібно використовувати спільні приховані (private) дані класу та організовувати доступ до цих даних зі статичних та нестатичних методів.

Властивості статичних полів:

* Статичні поля зберігають стан всього класу/структури.
* Статичне поле визначається як і просте, лише перед типом поля вказується ключове слово static.
* На рівні пам'яті для статичних полів створюватиметься ділянка в пам'яті, яка буде спільною для всіх об'єктів класу.

Крім звичайних конструкторів, у класу також можуть бути **статичні конструктори**. **Статичні конструктори** мають такі відмінні риси:

* Статичні конструктори не повинні мати модифікатор доступу та не приймають параметрів
* Як і в статичних методах, в статичних конструкторах не можна використовувати ключове слово для посилання на поточний об'єкт класу і можна звертатися тільки до статичних членів класу
* 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** зазвичай використовуються для ініціалізації статичних даних або виконують дії, які потрібно виконати лише один раз.

**Функціональне програмування** базується на чисто-функції. Коли функція робить одну конкретну дію і не залежить від оточення чи глобальних змінних.

**Агрегація** менше жорсткий зв’язок між вкладеним об’єктом та об’єктом в який вкладено. Без вкладеного зовнішній може функціонувати.

**Композиція** – жорсткий зв’язок між вкладеним об’єктом та об’єктом в який вкладено. Без вкладеного об’єкта зовнішній не може функціонувати.

**Процедурне програмування** базується на процедурах. Декомпозиція на менші функціональні частини, які можна використовувати окремо в різних частинах програми.

**ООП** – парадигма програмування, який базується на інкапсуляції, наслідуванні та поліморфізмі.

**Інкапсуляція** – приховування внутрішньої реалізації і надання санкціонованого доступу до інтерфейсу класу. Клас стає, як чорний ящик. Абстрагує код клієнта від внутрішньої реалізації класу.

**Наслідування** – створення нового класу на основі вже створеного. За допомогою наслідування ми можемо розширяти функціонал базового класу в дочірньому класі без копі/пасту. Абстрагуємось від базового класу, використовуючи дочірній. В С# відсутнє множинне наслідування для класів, проте дозволено реалізовувати безліч **інтерфейсів**.

**Інтерфейс** – контракт, тобто опис полів та методів, які клас зобов’язується виконати у разі імплементації інтерфейсу.

**Поліморфізм** – один інтерфейс і багато реалізацій. Абстрагуємось від конкретної реалізації.

**Модифікатори доступу** допомагають реалізувати інкапсуляцію. По замовчуванню для класу – internal, а для полів та функцій – private. ***!!!Хороша практика*** – завжди явно прописувати модифікатори доступу. Є наступні типи модифікаторів доступу:

* private – поле/функція доступні лише всередині поточного класу;
* protected private – поле/функція доступні в поточному класі або класах, які були наслідувані від поточного в даному проекті;
* protected – поле/функція доступні в поточному класі або класах, які були наслідувані від поточного;
* internal – поле/функція доступні в будь якому місці поточного проекту;
* public – поле/функція доступні в будь якому місці поточного проекту, а також для інших програм та збірок;

Для спрощення доступу до полів класу використовують **властивості** та **автовластивості**. Це автоматичні замінники гетерів та сетерів. **Автовластивості** використовують у випадках, коли гетери та сетери не мають додаткової логіки.

**Властивості**:

private string \_hobby;

public string Hobby

{

get

{

return \_hobby;

}

set

{

\_hobby = value;

}

}

**Автовластивості**:

private string \_firstName;

public string Name { get; set; } = "noname"; //Значення по замовчуванні

**readonly властивість** (можна тільки прочитати):

private string \_companyName = "Google";

public string CompanyName

{

// v1

// get { return \_companyName; }

// v2

get;

}

**writeonly властивість** (можна тільки записати):

private string \_firstName;

public string FirstName

{

set { \_firstName = value; }

}

***!!!Хороший стиль програмування:*** назву поля починати з нижнього підкреслення (\_ххх), а назви властивостей з великої літери Хххх

**virtual** – ключове слово, яке означає, що метод ми можемо явно переозначити в насліднику.

public virtual void Print()

{

Console.WriteLine($"Name: {Name}\nCost: {Cost}");

}

**override** – явне переозначення методу з базового класу (найкраще робити так).

public override void Print()

{

base.Print(); // вызываем реализацию из базового класса

Console.WriteLine($"Pixels: {Pixels}");

}

При використанні **override** переписується функціональність базового класу з дочірнього. При цьому, якщо зробити приведення до базового типу, все рівно буде викликатись пере означений метод (при інших пере означеннях такого не відбуватиметься).

Якщо зробити public **override sealed** void Print() – означає, що заборонено подальше пере означення методу.

**Стилі переозначення**, які працюватимуть, але є поганими з точки зору втрати методу базового класу:

**Неявне приховування методу** – втрачаємо реалізацію методу з такою назвою з базового класу

public void Print()

{

Console.WriteLine($"Pixels: {Pixels}");

}

**Явне приховування методу** - втрачаємо реалізацію методу з такою назвою з базового класу

public new void Print()

{

Console.WriteLine($"Pixels: {Pixels}");

}

Посилання:

<https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop>

<https://www.c-sharpcorner.com/UploadFile/mkagrahari/introduction-to-object-oriented-programming-concepts-in-C-Sharp/>

<https://www.tutorialsteacher.com/csharp/oop>

<https://medium.com/codex/c-object-oriented-programming-oop-2d92a5cd336f>

<https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers>

<https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-tuples>

# Корисні книги

1. А. Васильєв. «Програмування на С# для початківців. Основні відомості"

2. Джеффрі Ріхтер. CLR via C#. Програмування на платформі Microsoft .NET Framework мовою C#

3. Філіп Джепікс, Ендрю Троєлсен. «Мова програмування C# 7 та платформи .NET та .NET Core»

4. Марк Дж. Прайс. C# 7 і .NET Core. Крос-платформна технологія для професіоналів»