[Заняття 1: Введення, Git, класи та структури 2](#_Toc143178569)

[Заняття 2: ООП. Реалізація принципів ООП 7](#_Toc143178570)

[Заняття 3: Абстрактні класи та інтерфейси 12](#_Toc143178571)

[Заняття 4: Перевантаження операторів, приклади 17](#_Toc143178572)

[Заняття 5: Generic. Шаблонні класи та інтерфейси 20](#_Toc143178573)

[Заняття 6: Generic. SOLID, other principles 23](#_Toc143178574)

[Заняття 7: Складальник сміття 27](#_Toc143178575)

[Типові інтерфейси 35](#_Toc143178576)

[Корисні книги 38](#_Toc143178577)

# 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).

struct Person

{

public string name = "no name";

public int age = 18;

public Person() { }

public Person(string name = "Tom", int age = 1)

{

this.name = name;

this.age = age;

}

public void Print()

{

Console.WriteLine($"Name: {name} Age: {age}");

}

}

**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>

# Заняття 3: Абстрактні класи та інтерфейси

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

interface IMovable

{

// реализация метода по замовчуванню

void Move() => Console.WriteLine("Walking");

//Інші поля і методи

}

public class Person : IMovable { }

З інтерфейсами можна зробити таку річ:

IMovable persone = new Persone();

IMovable animal = new Animal();

IMovable robot = new Robot();

Після «=» (new Xxxx) створюються конкретні класи, які реалізовують даний інтерфейс. Тому ми можемо метод з інтерфейсу викликати через змінну інтерфейсу, якій присвоєне посилання на конкретний об’єкт, із реалізацією в класі даного об’єкта. Проте, всі додаткові методи та поля, які створені в цьому класі, не будуть доступними (все обмежується даними та методами з інтерфейсу).

Інтерфейс представляє тип посилання, який може визначати деякий функціонал - набір методів і властивостей без реалізації.

Потім цей функціонал реалізують класи та структури, які застосовують дані інтерфейси.

Для визначення інтерфейсу використається ключове слово interface. Як правило, назви інтерфейсів у C# починаються з великої літери I, наприклад, IComparable, IEnumerable (так звана угорська(венгерская) нотація), проте це не обов'язкова вимога, а більше стиль програмування.

Що може визначити інтерфейс? Загалом інтерфейси можуть визначати такі сутності:

-Методи

- Властивості

- Індексатори

- Події

- Статичні поля та константи (починаючи з версії C# 8.0)

Проте інтерфейси неспроможні визначати нестатичні змінні.

Ще один момент в оголошенні інтерфейсу: якщо його члени - методи та властивості не мають модифікаторів доступу, то фактично за умовчанням доступ public, оскільки мета інтерфейсу - визначення функціоналу для реалізації його класом. Це стосується також констант і статичних змінних, які в класах і структурах за замовчуванням мають модифікатор private.

В інтерфейсах вони мають за замовчуванням модифікатор public. Але, починаючи з версії C# 8.0, ми можемо явно вказувати модифікатори доступу у компонентів інтерфейсу.

Як і класи, стандартні інтерфейси мають рівень доступу internal, тобто такий інтерфейс доступний тільки в рамках поточного проекту. Але за допомогою модифікатора public ми можемо зробити інтерфейс загальнодоступним.

Також, починаючи з версії C# 8.0, інтерфейси підтримують реалізацію методів і властивостей за умовчанням. Це означає, що ми можемо визначити в інтерфейсах повноцінні методи та властивості, які мають реалізацію як у звичайних класах та структурах.

Інтерфейси мають ще одну важливу функцію: в C# не підтримується множинне успадкування, тобто ми можемо успадкувати клас тільки від одного класу, на відміну, скажімо, від мови С++, де успадкування можна використовувати. Інтерфейси дозволяють частково обійти це обмеження, оскільки C# класи і структури можуть реалізувати відразу кілька інтерфейсів. Всі реалізовані інтерфейси вказуються через кому.

interface IMovable

{

// реалізація методу по замовчуванню

void Move() => Console.WriteLine("Walking");

// реалізація властивості по замовчуванню

// int MaxSpeed { get { return 0; } }

// Якщо інтерфейс має приватні методи та властивості (тобто з модифікатором private), то вони повинні мати реалізацію за умовчанням. Те саме стосується статичних методів (не обов'язково приватних)

public const int minSpeed = 0; // мінімальна швидкість

private static int maxSpeed = 60; // максимальна швидкість

// час, за який проходимо distance зі швидкістю speed

static double GetTime(double distance, double speed) => distance / speed;

static int MaxSpeed

{

get => maxSpeed;

set

{

if (value > 0) maxSpeed = value;

}

}

}

**!!!!Наслідування інтерфейсів!!!!**

**Всі об’єкти** в С# є наслідниками базового класу **object**. Від нього передались наступні методи:

* ToString()
* GetHashCode()
* GetType()
* Equals();

**ToString()** - служить для отримання рядкового представлення даного об'єкта.

**GetHashCode() -** дозволяє повернути деяке числове значення, яке буде відповідати даному об'єкту або його хеш-коду. За цим числом, наприклад, можна порівнювати об'єкти. Можна визначати різні алгоритми генерації подібного числа або взяти реалізацію базового типу.

Метод **Equals()** приймає як параметр об'єкт будь-якого типу, який потім приводити до поточного класу. Якщо переданий об'єкт є типом поточного класу, то повертаємо результат порівняння вказаних полів двох об'єктів поточного класу. Якщо об'єкт представляє інший тип, то повертається false. При необхідності реалізацію методу можна зробити складнішою, наприклад, порівнювати за декількома властивостями за їх наявності. Слід зазначити, що з методом Equals слід реалізувати метод GetHashCode. Якщо порівнювати два складні об'єкти, то краще використовувати метод Equals, а не стандартну операцію ==

**GetType() –** повертає об'єкт Type, тобто тип об'єкта.

**Глибоке порівняння** – порівняння по значенні, а не по посиланню.

**!!!Почитати що в середині словника!!!**

**Абстрактний клас** – клас хоча б з одним абстрактним членом. Позначається (як і абстрактні члени) ключовим словом **abstract**. Особливістю є абстрактність даних сутностей. Тобто, за допомогою цього ми абстрагуємось від конкретної реалізації. Дану реалізацію ми покладаємо на дочірній клас. Якщо не реалізувати абстрактну сутність в дочірньому класі, то і цей дочірній клас стає абстрактним.

**!!!Особливістю** є те, що ми не можемо створити **екземпляр абстрактного класу**.

Абстрактними можуть бути такі члени класу:

- Методи

- Властивості

- Індексатори

- Події

**Абстрактні** члени класів не повинні мати **private** модифікатор.

В **дочірньому класі** реалізація повинна іти з **override** ключовим словом, бо інакше буде **відмова від реалізації** абстрактного члена і дочірній клас стане абстрактним.

Головні відмінності між **абстрактним класом і інтерфейсом** (***додатково уточнити***):

* АК має хоча б 1 абстрактний член;
* Не можна зробити успадкування від кількох АК, а І можна імплементувати скільки завгодно;
* Хоча І можуть мати реалізацію за замовчуванням, але клас, який імплементує І має переозначити функціонал з І, а з АК переозначення не абстрактного методу робити не обов’язково;
* І не може бути sealed або abstract але методи в І можуть юзати new для пере означення методів з однаковими сигнатурами;
* Клас успадковуємо, а інтерфейс реалізовуємо;

|  |  |  |
| --- | --- | --- |
| **Основа для порівняння** | **Інтерфейс** | **Анотація класу** |
| Основний | Коли ви тільки знаєте вимоги не про його реалізацію, ви використовуєте "Інтерфейс". | Коли ви частково знаєте про реалізації, ви використовуєте "Абстрактні класи". |
| Методи | Інтерфейс містить лише абстрактні методи. | Абстрактний клас містить абстрактні методи, а також конкретні методи. |
| Модифікатор методів доступу | Методи інтерфейсу завжди "Public" і "Abstract", навіть якщо ми не оголошуємо. Отже, це можна сказати як 100%, чистий абстрактний клас. | Не обов'язково, що метод в абстрактному класі буде публічним і абстрактним. Він також може мати конкретні методи. |
| Обмежений модифікатор методів | Метод інтерфейсу не може бути оголошений наступними модифікаторами: Загальнодоступний: приватний і захищений Анотація: фінальний, статичний, синхронізований, рідний, рок. | Немає обмежень на модифікатори змінних абстрактного класу. |
| Модифікатор доступу до змінних | Модифікатор Acess, дозволений для змінних інтерфейсу, є загальнодоступними, статичними та остаточними, чи ми декларуємо чи ні. | Змінні в абстрактному класі не повинні бути публічними, статичними, кінцевими. |
| Обмежені модифікатори для змінних | Інтерфейсні змінні не можуть бути оголошені як приватні, захищені, перехідні, мінливі. | Не існує обмежень на модифікатори абстрактних змінних класу. |
| Ініціалізація змінних | Змінні інтерфейси повинні бути ініціалізовані під час його декларування. | Не обов'язково, щоб абстрактні змінні класу повинні бути ініціалізовані під час його декларування. |
| Примірник і статичні блоки | Всередині інтерфейсу не можна оголосити примірник або статичний блок. | Абстрактний клас допускає примірник або статичний блок всередині нього. |
| Конструктори | Не можна оголосити конструктор всередині інтерфейсу. | Ви можете оголосити конструктор всередині абстрактного класу. |

Різниця між **is** і **as:**

* **as** використовується для приведення одного типу до іншого. При цьому при **невдалому** приведенні **не відбудеться викидання ексепшину** (при звичайному касті відбудеться);
* **is** використовується для порівняння двох типів. Якщо один можна привести до іншого, то повертається true, в іншому випадку – false;

Посилання:

<https://www.geeksforgeeks.org/is-vs-as-operator-keyword-in-c-sharp/>

<https://www.c-sharpcorner.com/blogs/is-and-as-keyword-difference-in-c-sharp>

<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>

# Заняття 4: Перевантаження операторів, приклади

**Методи розширення(extension methods)** дозволяють додавати нові методи вже існуючі типи без створення нового похідного класу. Ця функціональність буває особливо корисною, коли нам хочеться додати до певного типу новий метод, але сам тип (клас чи структуру) ми змінити не можемо, оскільки ми не маємо доступу до вихідного коду. Або якщо ми не можемо використати стандартний механізм успадкування, наприклад, якщо класи визначені з модифікатором sealed.

Для того, щоб створити метод розширення, спочатку треба створити статичний клас, який і міститиме цей метод. Потім оголошуємо статичний метод. Власне метод розширення - це звичайний статичний метод, який як перший параметр завжди приймає таку конструкцію: this имя\_типа назва\_параметра, (this string str).

public static class StringExtension

{

public static int CharCount(this string str, char c)

{

int counter = 0;

for (int i = 0; i < str.Length; i++)

{

if (str[i] == c)

counter++;

}

return counter;

}

}

string s = "Hello world";

char c = 'l';

int i = s.CharCount(c);

**Перевантаження операторів** призначене для додавання логіки роботи над кастомними об’єктами через оператори +, -, ++, -- і т.д.

можна перевантажувати

унарні оператори +x, -x, !x, ~x, ++, --, true, false

бінарні оператори +, -, \*, /, %

оператори порівняння ==, !=, <, >, <=, >=

порозрядні оператори &, |, ^, <<, >>

логічні оператори &&, ||

Необхідно перевантажувати парами:

== та !=

< та >

<= та >=

class Counter

{

public int Value { get; set; }

public static Counter operator +(Counter counter1, Counter counter2)

{

return new Counter { Value = counter1.Value + counter2.Value };

}

public static bool operator >(Counter counter1, Counter counter2)

{

return counter1.Value > counter2.Value;

}

}

**Явне та неявне перетворення типів:** необхідне для перетворення двох несумісних типів один в інший.

**explicit convert type** – явне приведення типів. Для того, щоб у своєму класі організувати явне приведення до певного типу необхідно зробити наступне:

public static explicit operator int(CounterV2 counter)

{

return counter.Seconds;

}

або

public static explicit operator CounterV2(Timer timer)

{

int h = timer.Hours \* 3600;

int m = timer.Minutes \* 60;

return new CounterV2 { Seconds = h + m + timer.Seconds };

}

**implicit convert type** – неявне перетворення типів:

public static implicit operator Timer(CounterV2 counter)

{

int h = counter.Seconds / 3600;

int m = (counter.Seconds % 3600) / 60;

int s = counter.Seconds % 60;

return new Timer { Hours = h, Minutes = m, Seconds = s };

}

**Перевантаження індексаторів**: реалізація логіки отримання елементів внутрішнього масиву об’єкта по індексу або по іншому полю (ім’я, адрес та ін..) без виклику внутрішнього масиву.

class Organisation

{

User[] users;

public Organisation(User[] users)

{

this.users = users;

}

public User this[int index]

{

get => users[index];

set => users[index] = value;

}

public User this[string userName]

{

get

{

foreach (var user in users)

{

if (user.UserName == userName)

{

return user;

}

}

throw new InvalidOperationException("Unknown name");

}

}

}

Console.WriteLine(myOrg[1]);

Console.WriteLine(myOrg["Anton"]);

Посилання:

<https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/operator-overloading>

<https://www.bestprog.net/uk/2022/09/07/c-overloading-of-operators-general-information-overloading-of-unary-operators-ua/>

# Заняття 5: Generic. Шаблонні класи та інтерфейси

**Generic –** узагальнення. Засіб мови C#, що дозволяє створювати програмний код, що містить єдине (типізоване) рішення задачі для різних типів, з його подальшим застосуванням для будь-якого конкретного типу (int, float, char тощо).

Переваги використання generic:

* спрощення програмного коду;
* забезпечення типової безпеки – тип з яким працюватиме сутність вказується при її оголошенні, тому подальше відслідковування коректності типів бере на себе компілятор;
* виключена необхідність явного приведення типу;
* підвищується продуктивність бо немає необхідності використовувати boxing і unboxing.

Як мовний засіб, узагальнення можуть бути застосовані до:

* класів;
* структур;
* інтерфейсів;
* методів;
* делегатів.

Синтаксис для класів:

class ClassName<T1, T2, ..., TN>

{

// Реалізація методів та полів класу,

// які працюють з узагальненими типами T1, T2, ..., TN

// ...

}

ClassName<t1, t2, ..., tN> obj = new ClassName<t1, t2, ..., tN>(arguments);

Синтаксис для структур:

struct StructName<T1, T2, ..., TN>

{

// ...

}

StructName<t1, t2, ..., tN> val = new StructName<t1, t2, ..., tN>(arguments);

В програмах на узагальнення накладаються наступні **обмеження**:

* властивості не можуть бути узагальненими але можуть використовуватись в узагальненому класі (структурі);
* індексатори не можуть бути узагальненими але можуть використовуватись в узагальненому класі (структурі);
* перевантажені оператори (operator) не можуть бути узагальненими. Однак використання параметру типу T тут допускається;
* події (event) не можуть бути узагальненими але можуть використовуватись в узагальнених класах (структурах);
* до узагальненого класу не можна застосовувати модифікатор extern;
* типи покажчиків не можна використовувати в аргументах типу;
* якщо в узагальненому класі (структурі) використовується статичне поле (static), то в об’єкті кожного конкретного типу (int, double тощо) створюється унікальна копія цього поля. Тобто, немає єдиного статичного поля для всіх об’єктів різних типів, що конструюються;
* до узагальненого типу T не можуть бути застосовані арифметичні операції (+, –, \* та інші) а також операції порівняння. Це пов’язано з тим, що при створенні екземпляру з типом-заповнювачем (інстанціюванні) замість параметру типу може бути використаний тип даних, який не підтримує ці операції.

При оголошенні ми можемо зробити обмеження, на тип даних, який підставлятиметься замість узагальнення:

* Classes (об’єкт якогось конкретного класу)
* Interfaces (об’єкт якогось конкретного інтерфейсу)
* class - the generic parameter must represent a class
* struct - the generic parameter must represent a structure
* new () – the generic parameter must represent a type that has a public parameterless constructor

**examples:**

class Messenger<T> where T : struct

{ }

class Messenger<T> where T : class

{ }

class Messenger<T> where T : new()

{ }

class Smartphone<T> where T : Messenger, new()

{ }

При наслідуванні обмеження з базового класу переходять на дочірні класи і їх необхідно явно прописувати.

Стосовно інтерфейсів є поняття коваріантність, контрваріантність та інваріантність:

**Коваріантність**: дозволяє використовувати більш детальний тип, ніж був використаний при оголошенні.

interface IMessenger<out T>

{

T WriteMessage(string text);

}

**Контрваріантність**: дозволяє використовувати більш універсальний (абстрактний) тип, ніж був використаний при оголошенні.

interface IMessengerV2<in T>

{

void SendMessage(T message);

}

**Інваріантність**: дозволяє використовувати тільки заданий тип.

interface IMessengerV3<T>

{

void SendMessage(T message);

}

Посилання:

<https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/generics>

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

<https://www.programiz.com/csharp-programming/generics>

<https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-interfaces>

<https://www.c-sharpcorner.com/UploadFile/Ashush/generics-in-C-Sharp-part-i/>

<https://www.c-sharpcorner.com/uploadfile/Ashush/generics-in-C-Sharp-part-ii/>

# Заняття 6: Generic. SOLID, other principles

Термін "**SOLID**" являє собою акронім для набору практик проектування програмного коду та побудови гнучкої та адаптивної програми. Цей термін був введений відомим американським фахівцем у галузі програмування Робертом Мартіном (Robert Martin).

Сам акронім утворений за першими буквами назв SOLID-принципів:

* **Single Responsibility Principle** (Принцип єдиного обов'язку)
* **Open/Closed Principle** (Принцип відкритості/закритості)
* **Liskov Substitution Principle** (Принцип підстановки Лісков)
* **Interface Segregation Principle** (Принцип розподілу інтерфейсів)
* **Dependency Inversion Principle** (Принцип інверсії залежностей)

Принципи SOLID - це не патерни, їх не можна назвати певними догмами, які треба обов'язково застосовувати при розробці, проте їх використання дозволить поліпшити код програми, спростити можливі його зміни та підтримку.

**Single Responsibility Principle** (Принцип єдиного обов'язку):

Кожен компонент повинен мати одну і тільки одну причину зміни. У C# як компонент може виступати клас, структура, метод. А під обов'язком тут розуміється набір дій, що виконують єдине завдання. Тобто суть принципу полягає в тому, що клас/структура/метод повинні виконувати одне завдання. Весь функціонал компонента повинен бути цілісним, мати високу зв'язність (high cohesion) та низьке зчеплення (low coupling).

Часто виникають випадки порушення принципу SRP. Не рідко даний принцип порушується при змішуванні в одному класі функціональності різних рівнів за стосунку. Для прикладу: клас виконує розрахунки та виводить їх результати користувачу, тобто, поєднує в собі бізнес-логіку та роботу з інтерфейсом користувача. Або клас керує збереженням/отриманням даних та виконуванням над ними розрахунків, що також не бажано. Клас необхідно використовувати тільки для одного завдання – або бізнес-логіки, або розрахунків, або робота з даними. Іншим поширеним випадком є наявність в класі або його методах повністю не пов'язаного між собою функціоналу.

Поширена наступна схема розподілу функціоналу між компонентами:

* Логіка збереження даних;
* Валідація;
* Механізм інформування користувача;
* Обробка помилок;
* Логування;
* Вибір класу або створення його об’єкту;
* Форматування;
* Парсинг;
* Маппінг даних.

**Open/Closed Principle** (Принцип відкритості/закритості) можна сформулювати так:

Сутність програми повинна бути відкрита для розширення, але закрита для зміни. Наступний приклад ПОГАНИЙ:

public class EmployeeReport

{

public string TypeReport { get; set; }

public void GenerateReport(Employee em)

{

if (TypeReport == "CSV")

{

// Генерация отчета в формате CSV

}

if (TypeReport == "PDF")

{

// Генерация отчета в формате PDF

}

}

}

Вирішенням проблеми є створення інтерфейсу та реалізація його для кожного типу звіту (CSV, PDF т.д.)

public interface IEmployeeReport

{

void GenerateReport(Employee em);

}

public class EmployeeCsvReport : IEmployeeReport

{

public void GenerateReport(Employee em)

{

// реалізація для CSV

}

}

public class EmployeePdfReport : IEmployeeReport

{

public void GenerateReport(Employee em)

{

// реалізація для PDF

}

}

В такому разі при додаванні нового типу звіту ми просто створюємо новий клас, який реалізовуватиме логіку для даного типу і не зачіпатимемо логіку звітування для вже існуючих типів звітів.

**Liskov Substitution Principle** (Принцип підстановки Лісков) є деяким посібником зі створення ієрархій успадкування. Початкове визначення даного принципу, яке було дано Барбарою Лисков у 1988 році, виглядало так:

Якщо кожного об'єкта o1 типу S існує об'єкт o2 типу T, такий, що з будь-якої програми P, визначеної термінах T, поведінка P не змінюється при заміні o2 на o1, то є підтипом T.

Тобто іншими словами, клас S може вважатися підкласом T, якщо заміна об'єктів T на об'єкти S не призведе до зміни роботи програми. Загалом цей принцип можна сформулювати так:

**Повинна бути можливість замість базового типу підставити будь-який підтип*.***

Фактично принцип підстановки Лиск допомагає чіткіше сформулювати ієрархію класів, визначити функціонал для базових і похідних класів і уникнути можливих проблем при застосуванні поліморфізму.

**Interface Segregation Principle** (Принцип розподілу інтерфейсів) сформулювати так: клієнти не повинні залежати від методів, якими не користуються. Тобто, якщо в ієрархії класів є різний функціонал, специфічний для даного класу, то цей функціонал можна винести в окремий інтерфейс.

Техніки виявлення порушення цього принципу:

* Занадто великі інтерфейси;
* Компоненти в інтерфейсах слабо узгоджені(перегукується із принципом єдиної відповідальності);
* Методи без реалізації(перегукується з принципом Лісків;

**Dependency Inversion Principle** (Принцип інверсії залежностей) служить для створення слабко пов'язаних сутностей, які легко тестувати, модифікувати та оновлювати. Цей принцип можна сформулювати так:

* Модулі верхнього рівня не повинні залежати від модулів нижнього рівня.
* І ті, й інші повинні залежати від абстракцій.
* Абстракції не повинні залежати від деталей.
* Деталі мають залежати від абстракцій.

public interface IMessanger

{

void Send();

}

public class Email: IMessanger

{

public void Send() { }

}

public class Sms : IMessanger

{

public void Send() { }

}

public class Notification

{

private IMessanger \_messanger;

public Notification(IMessanger messanger)

{

\_messanger = messanger;

}

public void Send()

{

\_messanger.Send();

}

}

public static class DependencyInversionPrincipleDemo

{

public static void Demo()

{

IMessanger emailMessanger = new Email();

IMessanger smsMessanger = new Sms();

Notification notification1 = new Notification(emailMessanger);

notification1.Send();

Notification notification2 = new Notification(smsMessanger);

notification2.Send();

}

}

Окрім SOLID є ще ряд принципів, які покликані покращити якість розробленого коду:

DRY(don’t repeat yourself) – не потрібно повторювати код в кількох місцях. Краще виділяти у сутності – методи, класи, інтерфейси і т.д. Це веде до спрощення керуванням кодом (при необхідності зміни потрібно буде змінити лише в одному місці програми), а, відповідно, і зменшення кількості помилок, які виникли б при копі/пастах та їх виправленнях.

KISS(keep it super simple) – необхідно намагатись писати код найбільш простіший.

YANGI(you ain’t gonna need it) – необхідно реалізовувати лише той функціонал, який необхідний.

Посилання:

<https://dou.ua/lenta/articles/solid-principles/>

<https://highload.today/uk/solid/>

<https://www.infoworld.com/article/3693755/applying-the-dry-kiss-and-yagni-principles-in-c-sharp.html#:~:text=The%20DRY%20(%E2%80%9CDon't,in%20mind%20when%20we%20code.>

<https://senior.ua/articles/kiss-dry-solid-yagni--navscho-dotrimuvatis-principv-programuvannya>

<https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29>

# Заняття 7: Складальник сміття

При використанні типів посилань, наприклад, об'єктів класів, для них буде відводитися місце в стеку, тільки там буде зберігатися не значення, а адреса на ділянку пам'яті в хіпі або купі, в якому і будуть знаходитися самі значення даного об'єкта. І якщо об'єкт класу перестає використовуватися, то при очищенні стека посилання на ділянку пам'яті також очищається, проте це не призводить до негайного очищення ділянки пам'яті в купі. Згодом збирач сміття (garbage collector) побачить, що на цю ділянку пам'яті більше немає посилань і очистить її.

Наприклад:

Test();

void Test()

{

Person tom = new Person("Tom");

Console.WriteLine(tom.Name);

}

record class Person(string Name);

У методі Test створюється об'єкт Person. За допомогою оператора new у купі для зберігання об'єкта CLR виділяє ділянку пам'яті. А в стек додає адресу на цю ділянку пам'яті. У неявно визначеному методі Main ми викликаємо метод Test. І після того, як Test відпрацює, місце в стеку очищається, а збирач сміття очищає раніше виділену під зберігання об'єкта Person ділянку пам'яті.

Складальник сміття не запускається відразу після видалення стека посилання на об'єкт, розміщений у купі. Він запускається в той час, коли середовище CLR виявить потребу, наприклад, коли програмі потрібна додаткова пам'ять.

Як правило, об'єкти в купі розташовуються невпорядковано, між ними можуть бути порожнечі. Купа досить сильно фрагментована. Тому після очищення пам'яті в результаті чергового складання сміття об'єкти, що залишилися, переміщаються в один безперервний блок пам'яті. Разом з цим відбувається оновлення посилань, щоб правильно вказувати на нові адреси об'єктів.

Також слід зазначити, що з великих об'єктів існує своя купа - Large Object Heap. У цю купу розміщуються об'єкти, розмір яких більше 85 000 байт. Особливість цієї купи полягає в тому, що при складанні сміття стиснення пам'яті не проводиться через великі витрати, пов'язані з розміром об'єктів.

Незважаючи на те, що на стиснення зайнятого простору потрібен час, та й додаток не зможе продовжувати роботу, поки не відпрацює збирач сміття, проте завдяки подібному підходу також відбувається оптимізація програми.

Тепер, щоб знайти вільне місце в купі середовищі CLR не потрібно шукати острівці порожнього простору серед зайнятих блоків. Їй достатньо звернутися до покажчика купи, який вказує на вільну ділянку пам'яті, що зменшує кількість звернень до пам'яті.

Крім того, щоб знизити витрати від роботи збирача сміття, всі об'єкти в купі поділяються на покоління. Усього існує три **покоління об'єктів: 0, 1 та 2-ге.**

**До покоління 0** належать нові об'єкти, які ще жодного разу не піддавалися збиранню сміття.

**До покоління 1** відносяться об'єкти, які пережили одну збірку, а до **покоління 2** - об'єкти, що пройшли більше одного складання сміття.

Коли збирач сміття починає роботу, він спочатку аналізує об'єкти з покоління 0. Ті об'єкти, які залишаються актуальними після очищення, підвищуються до покоління 1. Якщо після обробки об'єктів покоління 0 досі необхідна додаткова пам'ять, то збирач сміття приступає до об'єктів з покоління 1. Ті об'єкти, на які вже немає посилань, знищуються, а ті, які, як і раніше, актуальні, підвищуються до покоління 2.

Оскільки об'єкти з покоління 0 є молодшими і часто перебувають у адресному просторі пам'яті поруч друг з одним, їх видалення проходить із найменшими витратами.

**/// Клас System.GC**

Функціонал збирача сміття у бібліотеці класів .NET представляє клас System.GC. Через статичні методи цей клас дозволяє звертатися до збирача сміття. Як правило, потреба у застосуванні цього класу відсутня.

**Найбільш поширеним випадком його використання** є складання сміття при роботі з некерованими ресурсами, при інтенсивному виділенні великих обсягів пам'яті, при яких необхідне таке ж швидке звільнення.

Розглянемо деякі **методи та властивості класу System.GC**:

Метод **AddMemoryPressure** інформує середовище CLR про виділення великого обсягу некерованої пам'яті, яку слід врахувати під час планування складання сміття. У зв'язку з цим методом використовується метод **RemoveMemoryPressure**, який вказує на CLR, що раніше виділена пам'ять звільнена, і її не треба враховувати при складання сміття.

Метод **Collect** приводить у дію механізм складання сміття. Перевантажені версії методу дозволяють вказати покоління об'єктів, аж до якого треба зібрати сміття.

Метод **GetGeneration(Object)** дозволяє визначити номер покоління, до якого належить переданий як параметр об'єкт.

Метод **GetTotalMemory** повертає обсяг пам'яті в байтах, яке зайняте в купі, що керується.

Метод **WaitForPendingFinalizers** припиняє роботу поточного потоку до звільнення всіх об'єктів, для яких здійснюється складання сміття.

**Приклад:**

long totalMemory = GC.GetTotalMemory(false);

Console.WriteLine(totalMemory);

int number = 1000;

totalMemory = GC.GetTotalMemory(false);

Console.WriteLine(totalMemory);

GC.Collect();

GC.WaitForPendingFinalizers();

За допомогою перевантажених версій методу **GC.Collect** можна виконати більш точне налаштування складання сміття. Так, його перевантажена версія приймає як параметр число - номер покоління, до якого треба виконати очищення. Наприклад, GC.Collect(0) -видаляються лише об'єкти покоління 0.

Ще одна перевантажена версія приймає ще й другий параметр – перерахування **GCCollectionMode**. Цей перелік може набувати трьох значень:

**Default**: значення за промовчанням для цього переліку (Forced)

**Forced**: викликає негайне виконання складання сміття

**Optimized**: дозволяє збирачеві сміття визначити, чи є поточний момент оптимальним для збирання сміття

Наприклад, негайне складання сміття до першого покоління об'єктів: GC.Collect(1, GCCollectionMode.Forced);

Більшість об'єктів, що використовуються в програмах на C#, відносяться до керованого або managed-коду. Такі об'єкти управляються CLR і легко очищаються збирачем сміття. Однак разом з тим зустрічаються також такі об'єкти, які задіяють некеровані об'єкти (підключення до файлів, баз даних, мережеві підключення і т.д.). Такі некеровані об'єкти звертаються до API операційної системи.

Складальник сміття може впоратися з керованими об'єктами, однак він не знає, як видаляти некеровані об'єкти. І тут розробник повинен сам реалізовувати механізми очищення лише на рівні програмного коду. **Звільнення некерованих ресурсів** передбачає реалізацію одного з двох механізмів:

**Створення деструктора**

**Реалізація класом інтерфейсу System.IDisposable**

**Створення деструкторів:**

Якщо ви раптом програмували мовою C++, то, напевно, вже знайомі з концепцією деструкторів. Метод деструктора має ім'я класу (як і конструктор), перед яким стоїть знак тильди (~). Деструктори можна визначити лише у класах. Деструктор на відміну від конструктора не може мати модифікаторів доступу та параметри. При цьому кожен клас може мати лише один деструктор.

class Person

{

public string Name { get; }

public Person(string name) => Name = name;

~Person()

{

Console.WriteLine($"{Name} has deleted");

}

}

В даному випадку в деструкторі, з метою демонстрації, просто виводиться рядок на консоль, що повідомляє, що об'єкт видалено. Але в реальних програмах в деструктор вкладається логіка звільнення некерованих ресурсів. Однак насправді при очищенні збирач сміття викликає не деструктор, а метод Finalize. Все тому, що компілятор C# компілює деструктор у конструкцію, яка еквівалентна наступній:

protected override void Finalize()

{

try

{

// тут інструкції деструктора

}

finally

{

base.Finalize();

}

}

Метод Finalize вже визначений у базовому для всіх типів класі Object, проте цей метод не можна просто перевизначити. І фактична його реалізація відбувається через створення деструктора.

Test();

GC.Collect(); // очищення пам'яті під об'єкт tom

Console.Read();

void Test()

{

Person tom = new Person("Tom");

}

public class Person

{

public string Name { get; }

public Person(string name) => Name = name;

~Person()

{

Console.WriteLine($"{Name} has been deleted");

}

}

Зверніть увагу, що навіть після завершення методу Test і відповідно видалення з стека посилання на об'єкт Person у купі, може не наслідувати негайний виклик деструктора. Тільки після завершення всієї програми гарантовано відбудеться очищення пам'яті.

Однак з .NET 5 та в наступних версіях при завершенні програми деструктори не викликаються. Тому в програмі вище для швидшого очищення пам'яті застосовується метод **GC.Collect** і для гарантованого виклику деструктора встановлюється затримка за допомогою виклику **Console.Read(),** який очікує від користувача введення. На рівні пам'яті це виглядає так: збирач сміття при розміщенні об'єкта в купі визначає, чи підтримує даний об'єкт метод Finalize. І якщо об'єкт має метод Finalize, то покажчик на нього зберігається у спеціальній таблиці, що називається черга фіналізації. Коли настає момент складання сміття, збирач бачить, що даний об'єкт повинен бути знищений, і якщо він має метод Finalize, він копіюється в ще одну таблицю і остаточно знищується лише при наступному проході збирача сміття.

Точний час виклику деструктора не визначено. Крім того, під час фіналізації двох пов'язаних об'єктів порядок виклику деструкторів не гарантується. Тобто якщо об'єкт A зберігає посилання на об'єкт B, і при цьому обидва ці об'єкти мають деструктори, то для об'єкта B деструктор може вже відпрацювати в той час, як для об'єкта A деструктор тільки почне роботу. І тут ми можемо зіткнутися з наступною проблемою: а якщо нам негайно треба викликати деструктор і звільнити всі пов'язані з об'єктом некеровані ресурси? У цьому випадку ми можемо використовувати другий підхід - реалізацію інтерфейсу **IDisposable**.

**Інтерфейс IDisposable**

Test();

void Test()

{

Person? tom = null;

try

{

tom = new Person("Tom");

}

finally

{

tom?.Dispose();

}

}

public class Person : IDisposable

{

public string Name { get; }

public Person(string name) => Name = name;

public void Dispose()

{

Console.WriteLine($"{Name} has been disposed");

}

}

Ми розглянули два підходи. Який із них кращий? З одного боку, метод Dispose дозволяє будь-якої миті часу викликати звільнення пов'язаних ресурсів, з другого - програміст, використовує наш клас, може забути поставити в коді виклик методу Dispose. Загалом бувають різні ситуації. І щоб поєднувати плюси обох підходів, ми можемо використовувати **комбінований підхід**.

Microsoft пропонує нам використовувати наступний формалізований шаблон:

public class SomeClass : IDisposable

{

private bool disposed = false;

// реалізація інтерфейсу IDisposable.

public void Dispose()

{

// звільняємо некеровані ресурси

Dispose(true);

// пригнічуємо фіналізацію

GC.SuppressFinalize(this);

}

protected virtual void Dispose(bool disposing)

{

if (disposed) return;

if (disposing)

{

// Звільняємо керовані ресурси

}

// звільняємо некеровані об'єкти

disposed = true;

}

// Деструктор

~SomeClass()

{

Dispose(false);

}

}

Логіка очищення реалізується перевантаженою версією методу Dispose (bool disposing). Якщо параметр disposing має значення true, цей метод викликається з публічного методу Dispose, якщо false - то з деструктора. При виклику деструктора як параметр disposing передається значення false, щоб уникнути очищення керованих ресурсів, оскільки ми можемо бути впевненими у тому стані, що вони досі перебувають у пам'яті. І тут залишається покладатися на деструктори цих ресурсів. Та й в обох випадках звільняються некеровані ресурси.

Ще один важливий момент - виклик у методі Dispose методу GC.SuppressFinalize(this).GC.SuppressFinalize не дозволяє системі виконати метод Finalize для цього об'єкта. Якщо ж у класі деструктор не визначено, то виклик цього методу не матиме жодного ефекту. Таким чином, навіть якщо розробник не використовує у програмі метод Dispose, все одно відбудеться очищення та звільнення ресурсів.

**Загальні рекомендації щодо використання Finalize та Dispose**

* Деструктор слід реалізовувати тільки в тих об'єктів, яким він дійсно необхідний, оскільки метод Finalize дуже впливає на продуктивність.
* Після виклику методу Dispose необхідно блокувати у об'єкта виклик методу Finalize за допомогою GC.SuppressFinalize
* При створенні похідних класів від базових, які реалізують інтерфейс IDisposable, слід також викликати метод Dispose базового класу:

public class Base : IDisposable

{

public virtual void Dispose()

{

//

}

}

public class Derived : Base

{

private bool IsDisposed = false;

public override void Dispose()

{

if (IsDisposed) return;

IsDisposed = true;

// Звернення до методу Dispose базового класу

base.Dispose();

}

}

Віддавайте перевагу комбінованому шаблону, що реалізує як метод Dispose, так і деструктор.

Для автоматичного виклику методу Dispose() використовують конструкцію **using:**

using (Person tom = new Person("Tom"))

{

}

Конструкція using сформовує блок коду та створює об’єкт певного типу, який реалізовує інтерфейс IDisposable та його метод Dispose. Після виходу з блоку коду, в об’єкта викликається метод Dispose. Дана конструкція можлива лише для типів, які реалізовують інтерфейс IDisposable.

Починаючи з версії 8.0 через using можемо задати область дії область видимості зовнішньої функції:

void Test()

{

using Person tom = new Person("Tom");

// змінна tom доступна в блоці Test()

Console.WriteLine($"Name: {tom.Name}");

Console.WriteLine("Конец метода Test");

}

В даному випадку метод Dispose викличеться після виконання методу Test().

Посилання:

<https://www.loginworks.com/blogs/garbage-collector-work-net-c/>

<https://ironpdf.com/blog/net-help/csharp-using/>

<https://dou.ua/lenta/articles/principles-of-garbage-collection/>

# Типові інтерфейси

**IClonable**: призначений для глибокого копіювання об’єктів. У цьому випадку створюється новий об’єкт та в нього перекопійовуються усі значення полів старого. Необхідно, бо об’єкти – це значення-посилання, а при використанні звичайного = для присвоєння одного посилання іншому буде присвоюватись якраз саме посилання. Для поверхневого копіювання можна також використати **MemberwiseClone(),** але тоді вкладені об’єкти копіюватимуться по посиланню (лише типи-значення копіюватимуться по значенні).

class Person : ICloneable

{

public string Name { get; set; }

public int Age {get; set; }

public Person(string name, int age)

{

Name = name;

Age = age;

}

public object Clone() => MemberwiseClone();

}

**IComparable**: призначений для порівняння поточного об’єкта з об’єктом, який передається як параметр object? o. На виході він повертає ціле число, яке може мати одне із трьох значень:

* Меньше нуля. Отже, поточний об’єкт повинен перебувати перед об’єктом, який передається як параметр;
* Рівне нулю. Отже, обидва об’єкти рівні;
* Більше нуля. Отже, поточний об’єкт повинен перебувати після об’єкта, що передається як параметр.

Використовується (між іншим) для сортування об’єктів (за допомогою метода **Sort**()).

class Person : IComparable

{

public string Name { get;}

public int Age { get; set; }

public Person(string name, int age)

{

Name = name; Age = age;

}

public int CompareTo(object? o)

{

if(o is Person person) return Name.CompareTo(person.Name);

else throw new ArgumentException("Некоректне значення параметра");

}

}

**IEnumerable та IEnumerator**: призначені для реалізації перебирання колекції об’єктів у циклі (зокрема у foreach).

Інтерфейс IEnumerable має метод, який повертає посилання на інший інтерфейс - перечислювач

public interface IEnumerable

{

IEnumerator GetEnumerator();

}

А інтерфейс IEnumerator визначає функціонал для перебору внутрішніх об'єктів у контейнері

public interface IEnumerator

{

bool MoveNext(); // переміщення однією позицію вперед у контейнері елементів

object Current { get; } // поточний елемент у контейнері

void Reset(); // Переміщення на початок контейнера

}

Метод MoveNext() переміщує покажчик на поточний елемент наступну позицію в послідовності. Якщо послідовність ще закінчилася, то повертає true. Якщо послідовність закінчилася, то повертається false.

Властивість Current повертає об'єкт у послідовності, який вказує покажчик.

Метод Reset() скидає покажчик позиції початкове положення.

**Ітератор** по суті є блоком коду, який використовує оператор yield для перебору набору значень. Цей блок коду може представляти тіло методу, оператора або блок get у властивостях.

Ітератор використовує дві спеціальні інструкції:

**yield return** – визначає елемент, що повертається (повертає значення та запам’ятовує стан)

**yield break** – вказує, що послідовність більше не має елементів

class Numbers

{

public IEnumerator<int> GetEnumerator()

{

for (int i = 0; i < 6; i++)

{

yield return i \* i;

}

}

}

static class Int32Extension

{

public static IEnumerator<int> GetEnumerator(this int number)

{

int k = (number > 0) ? number : 0;

for (int i = number - k; i <= k; i++) yield return i;

}

}

<https://www.programiz.com/csharp-programming/yield-keyword#:~:text=yield%20return%20%2D%20returns%20an%20expression,yield%20break%20%2D%20terminates%20the%20iteration>

# 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»