МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"

Інститут **КНІТ**
Кафедра **ПЗ**

ЗВІТ

До лабораторної роботи № 1
З дисципліни: *"Моделювання та аналіз програмного забезпечення"*
На тему: *"Класи, інтерфейси і структури у мові програмування C#"*

Лектор:
доц. каф. ПЗ
Сердюк П.В.

Виконав:
ст. гр. ПЗ-22
Чаус Олег

Прийняв:
викл. каф. ПЗ
Микуляк А. В.

« ____ » _____ 2023 р.

∑= ____ _____.

Львів – 2023

Тема роботи: Класи, інтерфейси і структури у мові програмування C#

Мета роботи: Ознайомлення з основами класів, структур, та інших базових елементів мови програмування C#

ТЕОРЕТИЧНІ ВІДОМОСТІ

Поняття класів, структур є базовими для мов програмування ООП. C# як і мова Java, вводить ще одне нове поняття – інтерфейс, який дозволяє розділяти бізнес логіку і є значним посиленням механізму інкапсуляції у C#.

Клас C# може успадковувати властивості лише одного базового класу, обходиться це за допомогою механізму інтерфейсів, які застосовуються для опису дій над об’єктами.

Використання різноманітних властивостей, індексаторів, подій, а також інтерфейсів дозволяє створювати об'єкти, які можна використовувати, не вникаючи в деталі їх реалізації. При цьому об'єкт (клас) може реалізувати набір інтерфейсів, кожен з яких відповідає за виконання над об'єктом певних дій.

Інтерфейси найбільш схожі на віртуальні методи абстрактного класу, які мають бути визначені в базовому класі. Хорошою новиною є те, що кожен клас C# може реалізувати довільну кількість інтерфейсів. Саме це робить неістотним обмеження C#, що стосується можливості множинного наслідування класів.

Інтерфейси оголошуються за допомогою ключового слова *interface*, аналогічно класам. Всередині оголошення інтерфейсу необхідно перерахувати методи, з яких складається інтерфейс. Окрім методів, усередині інтерфейсів можна також оголошувати властивості, індексатори і події. Клас, що реалізує інтерфейс, має містити в собі тіло методів, оголошених у всіх реалізовуваних інтерфейсах.

При оголошенні інтерфейсів ми складаємо погодження, якому повинні задовольняти методи, властивості, індексатори і події, оголошені в інтерфейсі. Конкретна реалізація інтерфейсу покладається на клас, що реалізовує інтерфейс.

Неможливість множинного наслідування класів в C# з успіхом компенсується наявністю потужного механізму інтерфейсів. Похідний клас може бути успадкований лише від одного базового класу, проте при цьому він може реалізувати довільну кількість інтерфейсів. Інтерфейси групують описи наборів методів, що мають схоже призначення і функціональність. При необхідності можна комбінувати інтерфейси, створюючи нові інтерфейси на базі уже існуючих.

Якщо базовий клас реалізує будь-які інтерфейси, то вони успадковуються похідними класами. При необхідності похідний клас може перевизначити всі або деякі методи інтерфейсів базового класу.

Окрім методів в рамках інтерфейсів C# допускається оголошувати властивості, індексатори і події.

Якщо клас створений для представлення об'єкту з масивом, для доступу до нього можна використовувати індексатори. Індексатор може бути включений в оголошення інтерфейсу поряд з методами і іншими членами.

ЗАВДАННЯ

Перша частина

1. Реалізувати ланцюжок наслідування у якому б був звичайний клас, абстрактний клас та інтерфейс. Перелічіть відмінності та подібності у цих структурах у звіті у вигляді таблиці.
2. Реалізувати різні модифікатори доступу. Продемонструвати доступ до цих модифікаторів там де він є, та їх відсутність там, де це заборонено (включити в звіт вирізки з скріншотів Intelisense з VisualStudio).
3. Реалізувати поля та класи без модифікаторів доступу. Дослідити який буде доступ за замовчуванням у класів, структур, інтерфейсів, вкладених класів, полів, і т.д. У звіті має бути відповідна таблиця.
4. Оголосити внутрішній клас з доступом меншим за public. Реалізувати поле цього типу даних. Дослідити обмеження на модифікатор.
5. Реалізувати перелічуваний тип. Продемонструвати різні булівські операції на перелічуваних типах(^,||, &&. &, |,…).
6. Реалізувати множинне наслідування у C#.
7. Реалізувати перевантаження конструкторів базового класу та поточного класу. Показати різні варіанти використання ключових слів base та this. Реалізувати перевантаження функції.
8. Реалізувати різні види ініціалізації полів як статичних так і динамічних: за допомогою статичного та динамічного конструктора, та ін. Дослідити у якій послідовності ініціалізуються поля.
9. Реалізувати функції з параметрами out, ref. Показати відмінності при наявності та без цих параметрів. Показати випадки, коли ці параметри не мають значення.
10. Продемонструвати boxing / unboxing
11. Реалізувати явні та неявні оператори приведення то іншого типу (implicit та explicit)
12. Реалізувати логіку свого завдання у системі класів.
13. Скопіювати проект і перейменувати всі класи у структури. Дослідити відмінності у класах та структурах та записати їх у вигляді таблиці до звіту. Реалізувати наслідування структур через інтерфейси
14. Перевизначити і дослідити методи класу object (у тому числі і protected методи)

Друга частина

Дослідити швидкодію структур у відповідності з завданням.

За замовчуванням для експериментів брати 10 млн однакових операцій, але якщо виникають проблеми з швидкодією чи визначенням часу, то орієнтуватись до можливостей власного комп’ютера .

Потрібно провести кілька обчислень у різний час (оскільки інші процеси можуть вплинути на результат) і узяти середнє значення відповідних вихідних даних.

Для вимірювання часу рекомендується використати клас *Stopwatch*.

13) Порівняти час створення для викликів методу звичайним способом та через рефлексію. Порівняти час створення і потреби пам'яті для створення класів звичайним способом та через рефлексію.

ХІД ВИКОНАННЯ

1. Створив ланцюжок наслідування інтерфейсу, абстрактного класу та звичайного класу.

Створив інтерфейс IDrivable. Інтерфейс - це клас, який не може інстанціюватися, натомість він слугує “контрактом” реалізації його методів для класів, що його успадковують.

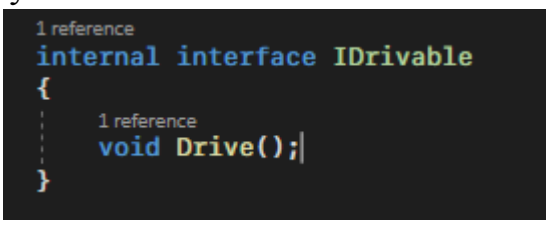

Рис. 1. Інтерфейс IDrivable

Створив абстрактний клас Vehicle. Абстрактний клас, як і інтерфейс, не дозволяє створення об’єктів свого типу, проте, на відміну від інтерфейсів, дозволяє здійснювати імплементацію своїх методів.

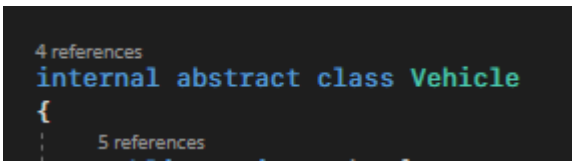

Рис. 2. Абстрактний клас Vehicle

Створив клас Car.

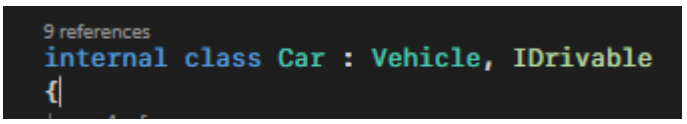

Рис. 3. Клас Car.

Таблиця різниці між абстрактними класами, інтерфейсами та класами:

	Клас	Абстрактний клас	Інтерфейс
Можна інстанціювати	так	ні	ні
Містить методи	так	так	так
Містить поля	так	так	так

Містить абстрактні методи	так	так	фактично так
Має конструктор	так	так	ні
Модифікатори доступу	всі	всі	public
Можна наслідувати декілька	ні	ні	так

2. Реалізував різні модифікатори доступу.

```
4 references
internal abstract class Vehicle
{
    5 references
    public string Make { get; set; }
    5 references
    public string Model { get; set; }
    5 references
    protected int Year { get; set; }
    4 references
    public bool IsElectric { get; set; }
    0 references
    private void Explode()
    {
        Console.WriteLine("It is quite unfortunate to inform you that I have exploded.");
    }
```

Рис. 6. Клас Vehicle з полями та методами з різними модифікаторами доступу.

- Public

Доступ до типу або елемента можна отримати за допомогою будь-якого іншого коду в тій самій збірці або іншій збірці, яка посилається на нього. Рівень доступності відкритих членів типу контролюється рівнем доступності самого типу.

```
0 references
static void Main(string[] args)
{
    var newCar = new Car();
    newCar.Make = "asddasd";
```

Рис. 7. Доступ до публічного поля поза межами класу.

```
1 reference
public void Drive()
{
    Console.WriteLine($"Driving my {Color} {Year} {Make} {Model}");
}
```

string Vehicle.Make { get; set; }

Рис. 8. Доступ до публічного поля в тілі дочірнього класу.

- Private

Доступ до типу або члена можна отримати лише за допомогою коду в тому самому класі чи структурі.

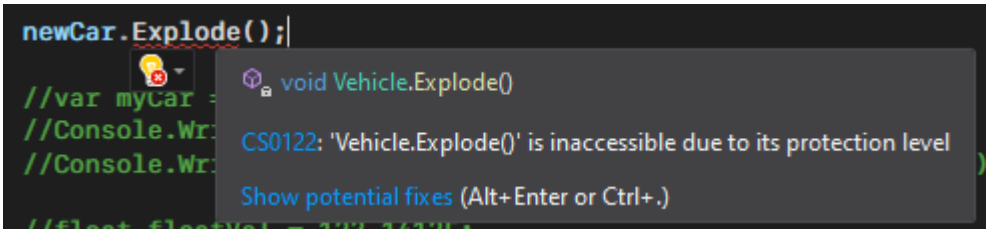

Рис. 10. Доступ до приватного поля поза межами класу.

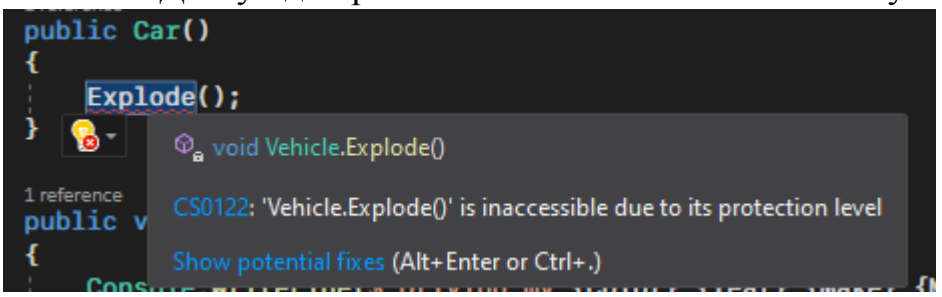

Рис. 12. Доступ до приватного поля в тілі дочірнього класу.

- Protected

До типу або члена можна отримати доступ тільки з коду в тому ж класі або в класі, який є похідним від цього класу.

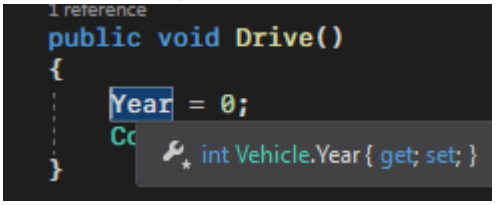

Рис. 14. Доступ до захищеного поля в тілі дочірнього класу.

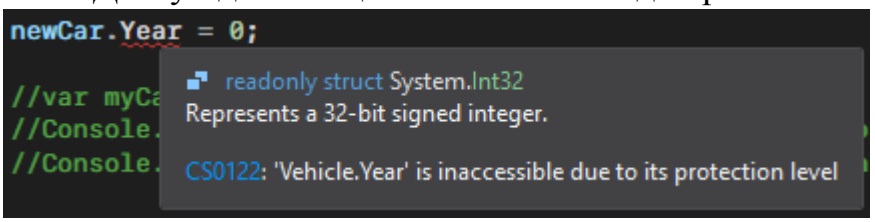

Рис. 15. Доступ до захищеного поля поза межами класу.

3. Реалізував поля та класи без модифікаторів доступу.

Створив клас без модифікаторів доступу, також для нього створив поле, метод, та вкладений клас.

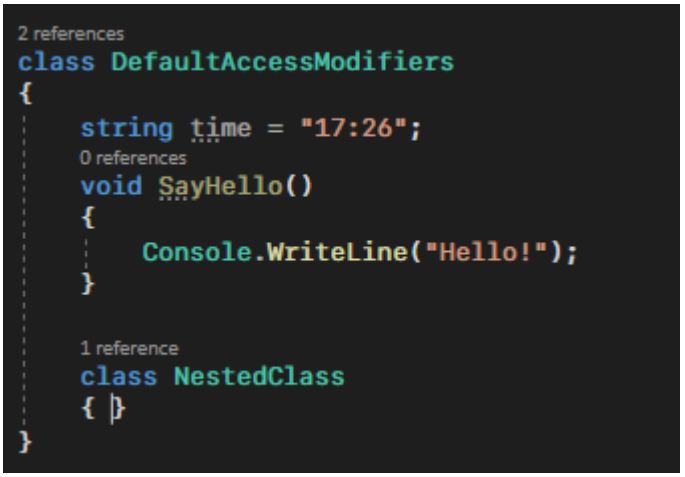

Рис. 16. Клас DefaultAccessModifiers.

Спробував створити клас та доступитися до його полів та вкладеного класу з головного класу

```
var TestClass = new DefaultAccessModifiers();
TestClass.SayHello();
Console.WriteLine(TestClass.time);
var TestClassNested = new DefaultAccessModifiers.NestedClass();
```

Рис. 17. Інстанціювання класу, виклик методу, доступ до поля, та інстанціювання вкладеного класу.

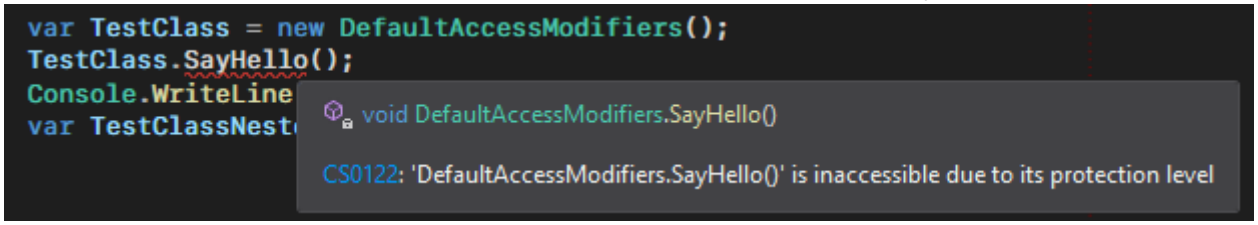

Рис. 18. Виклик методу.

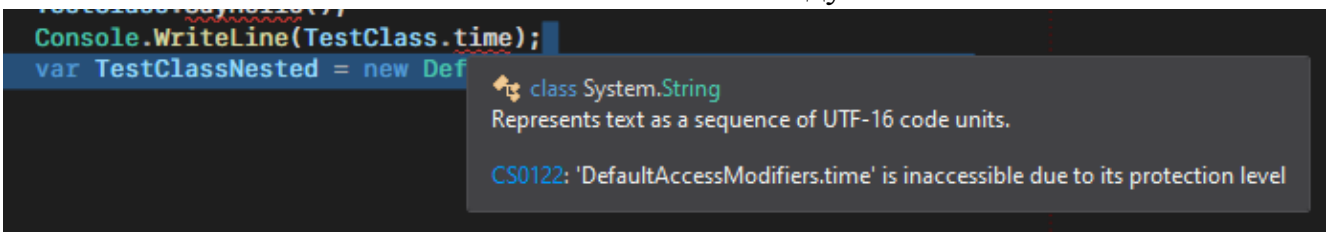

Рис. 19. Доступ до поля.

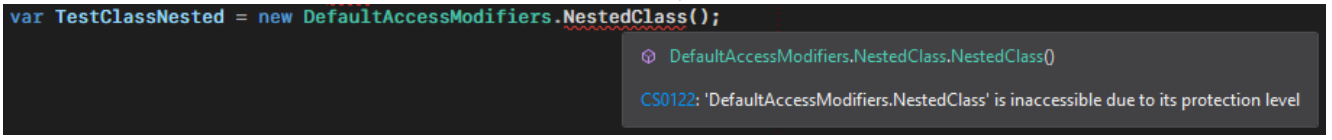

Рис. 20. Інстанціювання вкладеного класу.

З цього можна прийти до висновку, що модифікатор доступу для класів за замовчуванням – public. Для його ж полів, методів та вкладених класів, модифікатор доступу – private.

Дослідив доступ до інтерфейсу без оголошеного модифікатора доступу.

```
1 reference
interface IDrivable
{
    1 reference
    void Drive();
}
```

Рис. 20. Інтерфейс та його метод без модифікаторів доступу.

```
10 references
class Car : Vehicle, IDrivable
{
    1 reference
    public void Drive()
    {
        Console.WriteLine($"Driving my {Color} {Year} {Make} {Model}");
    }
```

Рис. 21. Клас, що утилізує цей інтерфейс.

Стандартний модифікатор доступу для інтерфейсів, як і для класів – internal. Методи та поля інтерфейсів публічні.

Створив структуру.

```
2 references
struct Speaker
{
    string Color;
    int Loudness;
    0 references
    Speaker(string color, int loudness)
    {
        Color = color;
        Loudness = loudness;
    }

    1 reference
    void Play()
    {
        Console.WriteLine($"*{Color} speaker noises ({Loudness} dB)*");
    }
}
```

Рис. 22. Структура Speaker.

```
var speaker = new Speaker();
Console.WriteLine(speaker.Color);
Console.WriteLine(speaker.Loudness);
speaker.Play();
```

Рис. 23. Інстанціювання структури, доступ до полів та виклик методу.

За замовчуванням, всі структури internal, їхні поля та методи – private, як і в класів.

Таблиця доступів за замовчуванням:

	Доступ за замовчуванням
Інтерфейс	internal
Клас, структура	internal
Поле, метод класу або структури	private
Поле, метод інтерфейсу	public
Вкладений клас	private

4. Оголосив клас з доступом internal.

```
10 references
internal class Car : Vehicle, IDrivable
{
```

Рис. 23. Клас Car, модифікатор доступу якого був змінений на internal.

```
0 references
static void Main(string[] args)
{
    var car = new Car();
}
```
Lab1.Car.Car(string make, string model, int year, string color) (+ 2 overloads)

CS0122: 'Car' is inaccessible due to its protection level

Show potential fixes (Alt+Enter or Ctrl+.)

Рис. 24. Доступ до класу з іншої збірки.

```
0 references
static void Main(string[] args)
{
    var newCar = new Car();
    newCar.Make = "as
```
Car.Car() (+ 2 overloads)

Рис. 25. Доступ до класу з поточної збірки.

Змінив модифікатор доступу класу на public, та створив internal поле.

```
0 references
static void Main(string[] ar
{
    var car = new Car();
    car.Make = "dasd";
}
```
CS1061: 'Car' does not contain a definition for 'Make' and no accessible extension method 'Make' accepting a first argument of type 'Car' could be found (are you missing a using directive or an assembly reference?)

Рис. 26. Доступ до поля Make з зовнішньої збірки.

Модифікатор доступу internal забезпечує доступ лише з поточної збірки.

Реалізувати перелічуваний тип. Продемонструвати різні булівські операції на перелічуваних типах(^,||, &&. &, |,…).

5. Реалізував перелічуваний тип з атрибутом Flags

```
[Flags]
0 references
enum IceCreamFlavor
{
    Vanilla = 1,          //000001
    Chocolate = 2,        //000010
    Strawberry = 4,       //000100
    Mint = 8,             //001000
    CookieDough = 16,     //010000
    RockyRoad = 32,       //100000
}
```

Рис. 27. Перелічуваний тип IceCreamFlavor з атрибутом [Flags].

```
Console.WriteLine(IceCreamFlavor.Chocolate & IceCreamFlavor.Vanilla);
Console.WriteLine(IceCreamFlavor.CookieDough | IceCreamFlavor.Mint);
Console.WriteLine(IceCreamFlavor.Strawberry ^ IceCreamFlavor.RockyRoad);
```

Рис. 28. Побітові операції над перелічуваним типом з атрибутом [Flags].

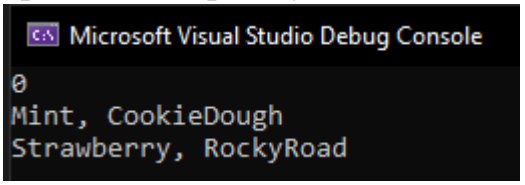

Рис. 29. Вивід побітових операцій.

Також перетворив його у звичайний перелічуваний тип.

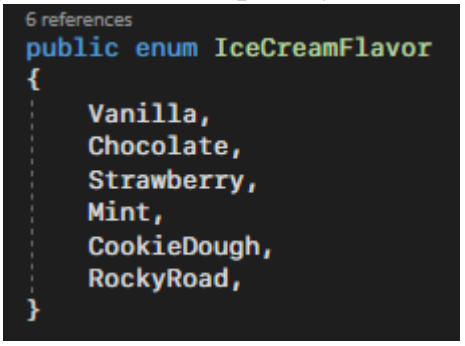

Рис. 30. Перелічуваний тип IceCreamFlavor.

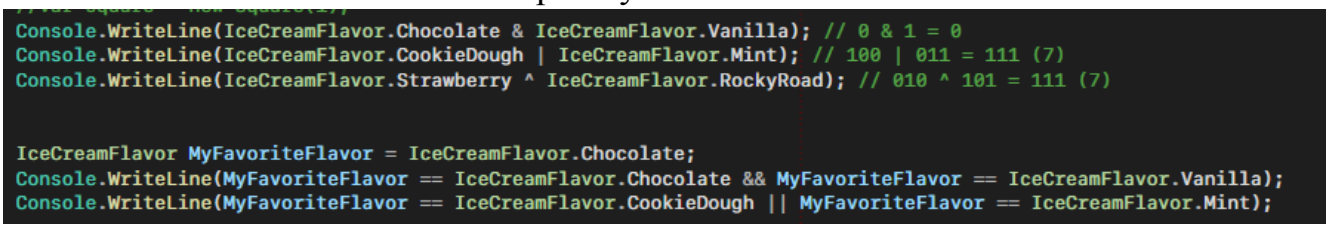

Рис. 31. Побітові булеві операції над перелічуваним типом.

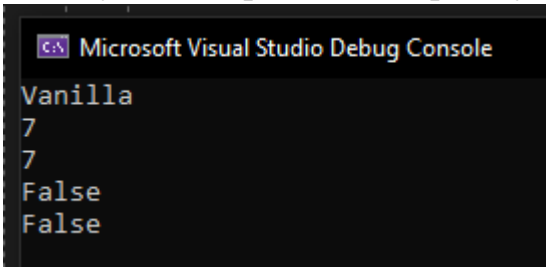

Рис. 32. Вивід побітових та булевих операцій над перелічуваним типом.

6. Реалізував множинне успадкування, використовуючи інтерфейси

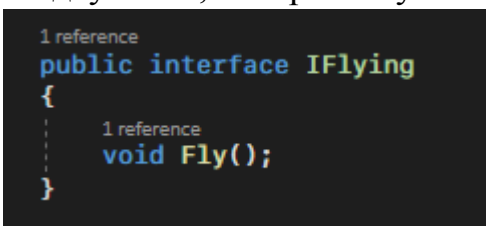

Рис. 33. Інтерфейс IFlying.

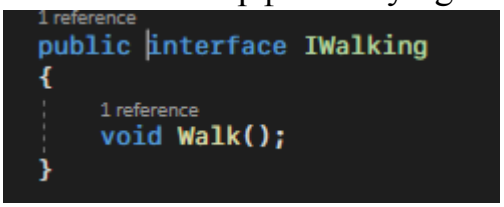

Рис. 34. Інтерфейс IWalking.

```
public interface ISwimming
{
    1 reference
    void Swim();
}
```

Рис. 34. Інтерфейс ISwimming.

```
0 references
public class Duck : IFlying, IWalking, ISwimming
{
    1 reference
    public void Fly()
    {
        Console.WriteLine("The duck is flying.");
    }
    1 reference
    public void Walk()
    {
        Console.WriteLine("The duck is walking.");
    }
    1 reference
    public void Swim()
    {
        Console.WriteLine("The duck is swimming.");
    }
}
```

Рис. 35. Клас Duck, що успадковується від декількох інтерфейсів.

7. Реалізував перевантаження конструкторів базового класу.

```
0 references
public Vehicle()
{
    Make = "";
    Model = "";
    Year = 0;
    IsElectric = false;
}
1 reference
public Vehicle(string make, string model, int year)
{
    Make = make;
    Model = model;
    Year = year;
    IsElectric = false;
}
```

Рис. 36. Конструктори класу Vehicle.

Реалізував перевантаження конструктору дочірнього класу з використанням ключових слів this і base у різних контекстах.

```
1 reference
public Car(string make, string model, int year, string color) : base(make, model, year) //`base` used to access base constructor
{
    this.Color = color; //`this` used to access the object that called the constructor
}
0 references
public Car(string make, string model, int year, bool isElectric) : this(make, model, year, "white") // `this` used to call another constr.
{
    base.IsElectric = isElectric; //`base` used to access base class property
}
```

Рис. 37. Перевантаження конструктору дочірнього класу.

Реалізував перевантаження функцій.

```
public void Drive()
{
    Console.WriteLine($"Driving my {Color} {Year} {Make} {Model}");
}

0 references
public void Drive(int miles)
{
    Console.WriteLine($"Driving my {Color} {Year} {Make} {Model} for {miles} miles");
}
```

Рис. 38. Перевантаження функцій.

8. Створив статичний та конструктор для ініціалізації статичного поля.

```
public static List<Car> cars;
0 references
static Car()
{
    cars = new List<Car>();
}
1 reference
```

Рис. 39. Статичний конструктор

Провівши відладку програми, побачив, що перед першим створення об’єкту викликається статичний конструктор, який ініціалізує статичні змінні.

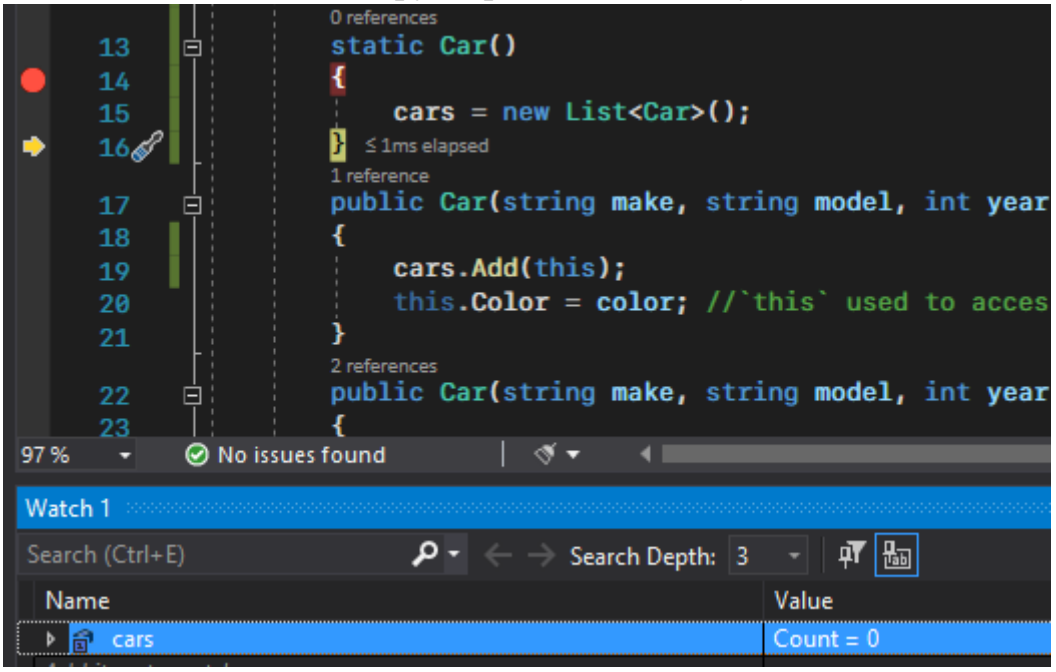

Рис. 40. Виклик статичного конструктора

Після цього викликається динамічний конструктор, який ініціалізує поля у порядку, вказаному програмістом.

9. Створив функції з параметрами out і ref.

```
1 reference
static void refChangeString(ref string par)
{
    par = "Hello";
}

1 reference
static void outChangeString(out string par)
{
    par = "Goodbye";
}
```

Рис. 41. Функції з параметрами ref та out.

Запустивши програму, отримаємо очікуваний вивід:

```
string testString = "ababab";
Console.WriteLine(testString);

refChangeString(ref testString);
Console.WriteLine(testString);

outChangeString(out testString);
Console.WriteLine(testString);
```

Рис. 42. Виклик функцій з параметрами ref і out.

```
Microsoft Visual Studio Debug Console
ababab
Hello
Goodbye
```

Рис. 43. Вивід програми.

І параметр ref, і параметр out дозволяють функції змінювати значення змінних, що їм передаються. Проте, параметр out більш строгий, та очікує, що змінна не була ініціалізована до виклику функції. Тому функція з параметром out не може читати змінну до записування значення, та вимагає обов'язкового записування.

Ці параметри не мають різниці, коли функція лише записує нове значення в змінну та не читає її.

10. Продемонстрував boxing/unboxing.

```
int i = 10;
object o = i; //boxing
i = 100;
i = (int)o; //unboxing
Console.WriteLine(i);
```

Рис. 44. Boxing та Unboxing.

Boxing та unboxing перетворювати типи даних зі value типу в reference тип і навпаки. Перетворення типу value-reference називається boxing, а перетворення reference-value називається unboxing.

```
Microsoft Visual Studio Debug Console
10
```

Рис. 45. Вивід програми.

Як можна побачити, хоч і значення `i` було змінене після боксингу, значення, яке було розпаковане з об’єкту, йому не дорівнює. Це означає, що boxing копіює значення value-type змінної.

11. Реалізував неявні та явні оператори перетворення типів.

```
public static implicit operator string(Car c) => $"{c.Color} {c.Year} {c.Make} {c.Model}";
public static explicit operator bool(Car c) => c.IsElectric;
```

Рис. 46. Оператори перетворення типів.

```
tic void Main(string[] args)

var myCar = new Car("Obey", "9F", 2013, "white");
Console.WriteLine(myCar);       // implicit conversion
Console.WriteLine((bool)myCar); //explicit conversion (cast)
```

```
Microsoft Visual Studio Debug Console
white 2013 Obey 9F
False
```

Рис. 47. Вивід програми.

Перетворив всі класи на структури.

```
10 references
public struct Car : Vehicle, IDrivable
{
    5 references
```

Рис. 48. Клас Car, що змінений на структуру.

Проект не скомпілюється, це пов’язано з тим, що структури не підтримують успадкування та лише можуть імплементовувати інтерфейси.

Структрура	Клас
Value-тип	Reference-тип
Не підтримує успадкування	Пітримує успадкування

Перевизначив методи класу object.

```
1 reference
public override bool Equals(object obj)
{
    return obj is Car car &&
           car.Color == Color;
}

1 reference
public override string ToString()
{
    return $"{Color} {Year} {Make} {Model}";
}
```

Рис. 49. Перевизначені методи класу object.

Частина 2.

Дані комп'ютера:

Процесор – AMD Ryzen 5 3.6 GHz, ОЗП 16 Гб.

Порівняв час створення для викликів методу звичайним способом та через рефлексію.

Таблиця 1

	Час виклику 10 млн. методів звичайним способом, мс	Час виклику 10 млн. методів через рефлексію, мс
Експеримент 1	21	642
Експеримент 2	21	650
Експеримент 3	21	664
Експеримент 4	21	660
Експеримент 5	21	650
Середнє значення	21	653.2

Порівняти час створення і потреби пам'яті для створення класів звичайним способом та через рефлексію.

Таблиця 2

	Час створення класів 10 млн. звичайним способом, мс	Час створення 10 млн. класів за допомогою рефлексії, мс
Експеримент 1	51	1749
Експеримент 2	49	1728
Експеримент 3	49	1727
Експеримент 4	49	1730
Експеримент 5	49	1746
Середнє значення	49.4	1736

За результатами можна зробити висновок, що використання рефлексії в C# може значно збільшити час, необхідний для виклику методів та створення класів, порівняно зі звичайним способом робити це.

Різниця в часі між звичайним способом та використанням рефлексії є досить значною. Виклик методів за допомогою рефлексії зайняв приблизно 31 раз довше, тоді як створення класів за допомогою рефлексії зайняло приблизно 35 разів довше.

Причина цієї різниці полягає в тому, що рефлексії включає більш складний процес аналізу та маніпулювання метаданими в режимі виконання, тоді як звичайний спосіб виклику методів та створення класів є простішим та прямішим процесом.

Тому рекомендується уникати використання рефлексії в сценаріях, де час, необхідний для виклику методів та створення класів, є значним фактором.

Рис. 50. Загальне виділення пам’яті звичайним способом

Рис. 50. Загальне виділення пам’яті з використанням рефлексії.

ВИСНОВКИ

Під час виконання моєї лабораторної роботи я знайомився з мовою програмування C# та вивчав основи класів, структур та інших базових елементів у C#. Я також використовував різні модифікатори доступу, такі як public, internal, protected та private, або без модифікатора доступу, який означає приватний доступ до полів та методів класів та структур або ж internal для класів та структур. Для інтерфейсів цей доступ публічний. Я реалізував множинне наслідування через використання інтерфейсів, яке єдине унікальне для мови C#. Я також реалізував перелічувані типи та продемонстрував побітові операції з ними, а також перевантажував конструктори та методи. Я досліджував різні види ініціалізації статичних та динамічних полів, зокрема через статичні конструктори, та дізнався про різницю між out та ref параметрами. Я також вивчив процеси boxing та unboxing, а також явні та неявні приведення типів. Я досліджував різні види поліморфізму через інтерфейси, віртуальні методи та рефлексію, порівнюючи їх швидкість. Це був мій перший досвід програмування в C# та я з захопленням вивчав нові технології.