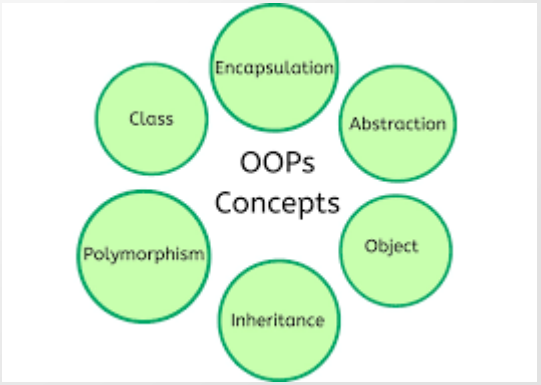
Лекція 3

ОБ'ЄКТНО-ОРІЄНТУВАНЕ ПРОГРАМУВАННЯ МОВОЮ C#



Лекція 3. Об'єктно-орієнтоване програмування мовою C#

План

1. Поняття класу в C #.
2. Інкапсуляція, успадкування та поліморфізм.
3. Інтерфейси.
4. Перевантаження операторів.
5. Перетворення типів.
6. Клас System.Object.

1. Поняття класу в C

Клас в объектно-ориентированном програмуванні – це тип даних, що створюється програмістом. Кожен клас описує деяку сутність певної предметної області (вікно, базу даних, обчислювальний процес тощо). **Атрибутами** класу (його властивостями) є **поля** (змінні). Для реалізації **поведінки** об'єкта використовуються **методи** (функції).

У C# класи визначаються з допомогою ключового слова **class**. Це можна робити всередині та за межами простору імен, усередині іншого класу. Найчастіше визначення класів розміщуються в окремих файлах. Наприклад:

```
class Complex
{
	// Опис полів
	double re;
	double im;
	// Опис методів
	public Complex()
	{
		re = im = 0;
	}
	// ...
}
```

1. Поняття класу в C

Крім звичайних **методів**, що реалізують всю необхідну функціональність класу, використовуються також і спеціальні методи, які називаються **конструкторами**.

Конструктори автоматично викликаються під час створення нового об'єкта даного класу та призначені для його **ініціалізації**. Ім'я конструктора має співпадати з назвою класу. Конструкторів у класу може бути декілька. Вони повинні відрізнятися кількістю аргументів та/або їх типами (тобто мати різні **сигнатури**).

Якщо в класі не визначено жодного конструктора, то для цього класу автоматично створюється **конструктор за замовчанням**, який не має параметрів.

Для створення об'єкта класу використовується оператор **new**, що виділяє для нього пам'ять і викликає відповідний конструктор. Наприклад:

```
Complex val = new Complex();
```

1. Поняття класу в C

Ключове слово **this** представляє посилання на поточний екземпляр класу. Воно може бути використане для уникнення дублювання функціональності конструкторів. Наприклад:

```
class Person
{
	string name;
	int age;
	public Person() : this("Unknown") {}
	public Person(string name) : this(name, 0) {}
	public Person(string name, int age)
	{
		this.name = name;
		this.age = age;
	}
	public void GetInfo()
	{
		Console.WriteLine("Ім'я: {0} Вік: {1}", name, age);
	}
}
```

У цьому прикладі перший конструктор викликає другий, а другий – третій.

2. Інкапсуляція, наслідування та поліморфізм

Класи C# підтримують такі концепції об'єктно-орієнтованого програмування (ООП), як **інкапсуляція**, **успадкування** і **поліморфізм**.

Поняття **інкапсуляції** в ООП має два аспекти:
1) можливість розміщення в одному компоненті даних та функцій, що з ними працюють;
2) приховування внутрішньої реалізації від інших компонентів (у цьому випадку доступ до прихованого атрибуту надаватися не безпосередньо, а за допомогою спеціальних методів читання (**геттер**) і запису (**сеттер**)).

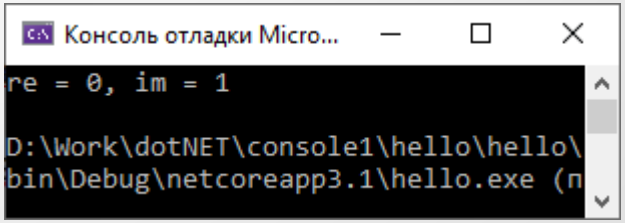
Для завдання режиму доступу до члена класу в C# використовуються такі основні модифікатори:

- **public** – загальнодоступний клас або член класу (доступний з будь-якого місця в коді, а також інших програм та збірок);
- **private** – закритий клас або член класу (доступний тільки з коду в тому класі або контексті);
- **protected** – член класу, доступний з будь-якого місця у поточному класі чи похідних від нього.

2. Інкапсуляція, наслідування та поліморфізм

Наприклад:

```
using System;
namespace hello
{
    public class Complex
    {
        private double re; // Поля
        private double im;
        Complex() // Методи
        {
            re = im = 0;
        }
        public Complex(double r, double i)
        {
            re = r;
            im = i;
        }
        public void set(double r, double i)
        {
            re = r;
            im = i;
        }
        public void get(ref double r, ref double i)
        {
            r = re;
            i = im;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Complex i = new Complex(0, 1);
            double re=0, im=0;

            i.get(ref re, ref im);
            Console.WriteLine("re = {0}, im = {1}", re, im);
        }
    }
}
```



2. Інкапсуляція, наслідування та поліморфізм

Успадкування в ООП – це концепція, згідно з якою клас може успадковувати дані та функціональність деякого вже існуючого класу, що сприяє повторному використанню компонентів програмного забезпечення. Наприклад:

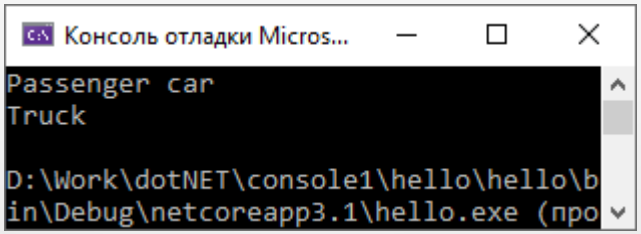
```
abstract class Vehicle // Абстрактний клас «Транспортний засіб»
{
	virtual public void GetInfo() {}
}
class Truck : Vehicle // Клас «Вантажний автомобіль»
{
	public override void GetInfo()
	{
		Console.WriteLine("Truck");
	}
}
class Car : Vehicle // Клас «Легковий автомобіль»
{
	public override void GetInfo()
	{
		Console.WriteLine("Passenger car");
	}
}
```

2. Інкапсуляція, наслідування та поліморфізм

Поліморфізм – це можливість методів різних об'єктів мати однаковий інтерфейс, але відрізнятися за змістом. Ця можливість реалізується шляхом перевизначення методу батьківського класу у класі-спадкоємці.

Розглянемо наступний приклад для наведених раніше класів Vehicle – Truck – Car:

```
// ...
Vehicle[] v = new Vehicle[2];

v[0] = new Car();
v[1] = new Truck();

foreach (Vehicle it in v)
    it.GetInfo();
```



2. Інкапсуляція, наслідування та поліморфізм

Для передачі параметрів у конструктор базового класу використовується ключове слово **base**. Наприклад:

```
abstract class Vehicle // Абстрактний клас «Транспортний засіб»
{
	protected string type;	// Тип транспортного засобу
	protected string model;	// Модель
	public Vehicle(string t, string m)
	{
		type = t; model = m;
	}
	// ...
}
class Truck : Vehicle // Клас «Вантажівка»
{
	private int carrying;
	public Truck(string t, string m, int c) : base(t, m)
	{
		carrying = c;
	}
	// ...
}
```

3. Інтерфейси

У мові C# на відміну від C++ заборонено **множинне успадкування** класів. Для реалізації такого функціоналу використовуються интерфейси.

Інтерфейсом в C# називається спеціальний тип посилань, що містить тільки абстрактні елементи, які не мають реалізації. Безпосередня їх реалізація повинна бути в класі, похідному від цього інтерфейсу.

Нехай, наприклад, потрібно реалізувати систему класів, що описують різні геометричні фігури. Абстрактний базовий клас CShape (форма) можна реалізувати таким чином:

```
// Абстрактна геометрична фігура
abstract class CShape
{
    public abstract string GetName(); // Назва фігури
}
```

А відповідний інтерфейс, що містить метод обчислення площі фігури, так:

```
interface IShape
{
    double Square();
}
```

3. Інтерфейси

Тоді, наприклад, клас, що описує прямокутник, можна реалізувати таким чином:

```
// Клас, що описує прямокутник
class CRectangle : CShape, IShape
{
	private double width; // Ширина та висота
	private double height;
	public CRectangle(double w, double h)
	{
		width = w;
		height = h;
	}
	public double Square() // Реалізація інтерфейсного методу обчислення площі
	{
		return width * height;
	}
	public override string GetName()
	{
		return "Square";
	}
}
```

3. Інтерфейси

Аналогічним чином можна описати клас для кола:

```
// Клас, що описує коло
class CCircle : CShape, IShape
{
	private double radius;
	public CCircle(double r)
	{
		radius = r;
	}
	public double Square()
	{
		return Math.PI*radius*radius;
	}
	public override string GetName()
	{
		return "Circle";
	}
}
```

Таким чином, класи CRectangle та CCircle є похідними від CShape та реалізують метод Square() інтерфейсу IShape.

4. Перевантаження операторів

Добре спроектований клас повинен бути таким же зручним у використанні, як і вбудований тип даних. Для цього в класах використовується **перевантаження операторів**.

Воно полягає у додаванні до класу, для об'єктів якого потрібно визначити новий оператор, спеціального методу **operator**, синтаксис якого має такий вигляд:

```
public static <тип_що_поветрається> operator <оператор>(<параметри>)
{
    // ...
}
```

(a + b) * (c + d)

Примітка: при реалізації методів, що перевантажують операції, їх параметри не повинні змінюватися!

4. Перевантаження операторів

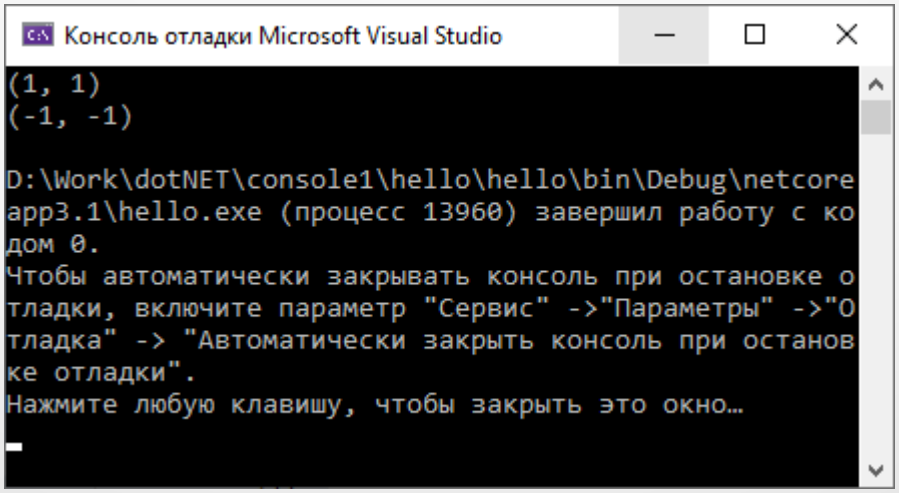
Наприклад, реалізація класу, що інкапсулює поняття двомірного вектора з перевантаженими операторами, може мати такий вигляд:

```
class Vector2D
{
   private double x; // Координати вектора
   private double y;
   public Vector2D(double px, double py)
   {
       x = px;
       y = py;
   }
   public static Vector2D operator + (Vector2D lhs, Vector2D rhs) // Перевантаження бінарної операції
   {
       return new Vector2D(lhs.x + rhs.x, lhs.y + rhs.y); // Повернення нового об'єкта
   }
   public static Vector2D operator - (Vector2D rhs) // Перевантаження унарної операції
   {
       return new Vector2D(-rhs.x, -rhs.y);
   }
   // …
}
```

4. Перевантаження операторів

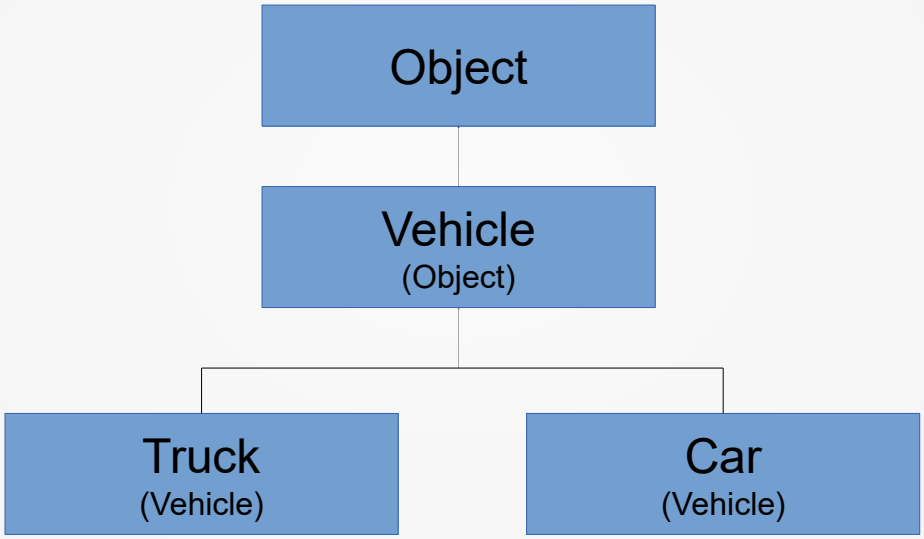
Приклад використання перевантажених операцій може мати такий вигляд:

```
// ...
Vector2D i = new Vector2D(1, 0),
         j = new Vector2D(0, 1),
         sum = i + j;

sum.print();
sum = -sum;
sum.print();
// ...
```



5. Перетворення типів

На практиці часто виникає потреба у перетворенні типів об'єктів. Розглянемо ієрархію раніше розглянутих класів Vehicle – Truck – Car:



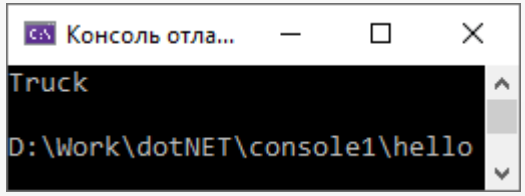
Примітка: всі класи в .NET (в т.ч. і ті, які створюються програмістами) є неявно похідними від System.Object, який знаходиться на вершині ієрархії успадкування. Тому всі типи та класи можуть реалізувати ті методи, які визначені у класі Object.

5. Перетворення типів

У C# реалізовані висхідні та низхідні перетворення типів.

Висхідні перетворення (upcasting)
Об'єкти дочірнього типу також відносяться до базового типу. Наприклад, об'єкт Truck (вантажівка) в той же час є об'єктом класу Vehicle (транспортний засіб). Отже, можна написати, наприклад, такий код:

```
Truck truck = new Truck();
Vehicle vechile = truck; // Неявне висхідне перетворення

vechile.GetInfo();
```



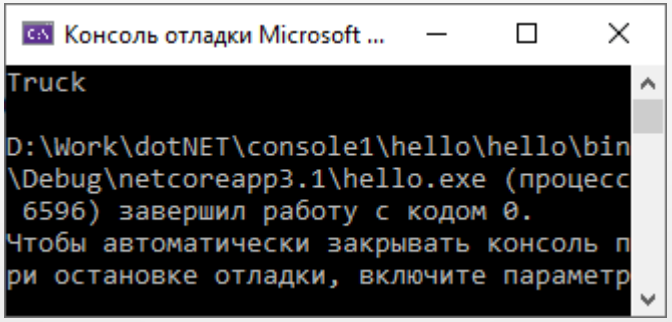
Низхідні перетворення (downcasting)
Іншим типом перетворення є низхідне – від базового типу до похідного. Слід зазначити, що такий вид перетворення не завжди можливий, як, наприклад, у цьому випадку, оскільки початково створити об'єкт абстрактного класу Vehicle не можна.

5. Перетворення типів

Явні способи перетворень

1. Використовуючи ключове слово **as,** можна виконати спробу перетворення виразу до певного типу. У разі невдалого перетворення вираз міститиме значення null (виняткова ситуація не генерується). Наприклад:

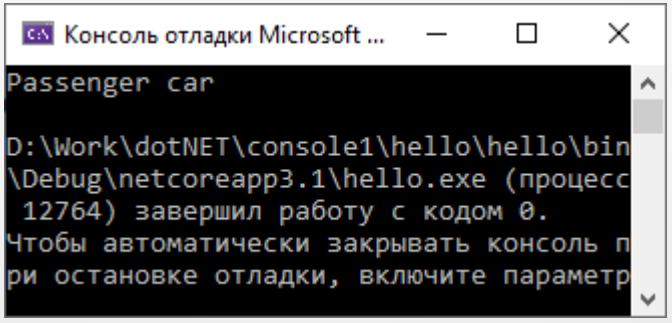
```
Truck truck = new Truck();
Vehicle car = truck as Vehicle;

if (car != null)
    car.GetInfo();
else
    Console.WriteLine("Error!");
```



5. Перетворення типів

2. Перевірка допустимості перетворення за допомогою ключового слова **is** :

```
Car car = new Car();
Vehicle vehicle;

if (car is Vehicle)
{
	vehicle = car;
	vehicle.GetInfo();
}
else
	Console.WriteLine("Error!");
```

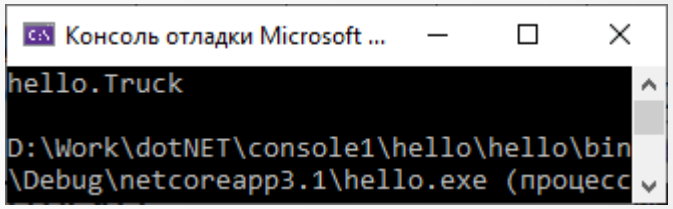


6. Клас System.Object

Як зазначалося, всі класи в .NET є спадкоємцями від базового класу **System.Object**. До його основних методів належать:

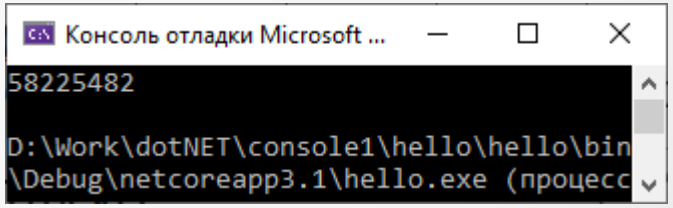
1. Метод **ToString()** дозволяє отримати рядкове представлення поточного об'єкта. Для базових типів просто виводитиметься їх рядкове значення:

```
Truck truck = new Truck();

Console.WriteLine(truck.ToString());
```



2. Метод **GetHashCode()** дозволяє повернути деяке числове значення (хеш-код), що відповідає даному об'єкту:

```
Truck truck = new Truck();

Console.WriteLine(truck.GetHashCode());
```



3. Метод **Equals()** дозволяє порівняти два об'єкти на рівність:

```
Truck truck1 = New Truck(),
truck2 = new Truck();

Console.WriteLine("{0}", truck1.Equals(truck2));
```