

Технології програмування

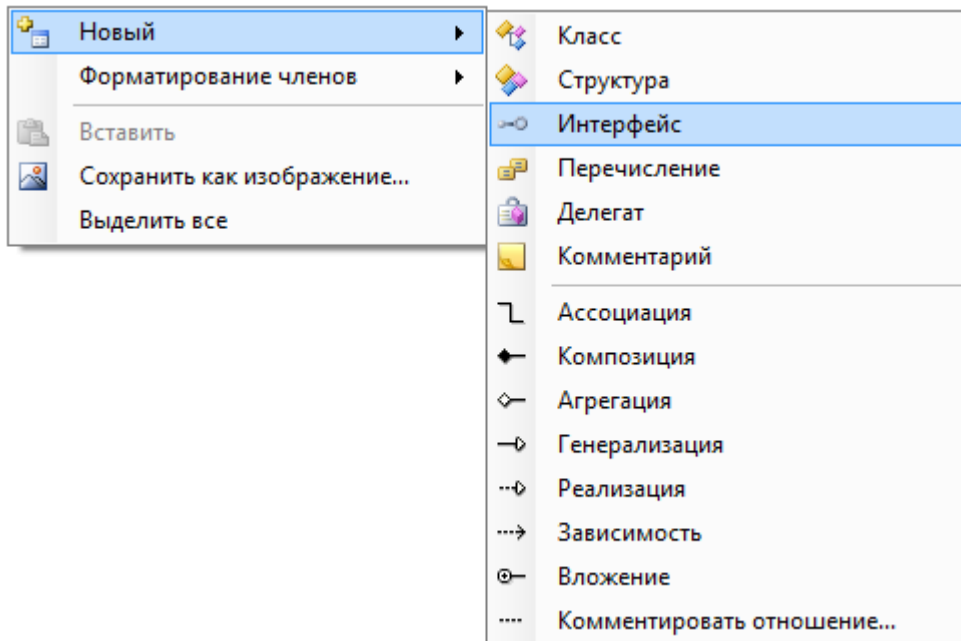
Тема 06. Інтерфейси

Євген БАБЕШКО
E-mail: e.babeshko@csn.khai.edu



Інтерфейси (1)

- Інтерфейс – набір сигнатур методів (властивостей, подій або індексаторів)
- Інтерфейс застосовується для специфікування послуг, які надає клас (== контракт!)
- До версії C# 8.0 у інтерфейсах була заборонена **будь-яка** реалізація



```
interface IEducable  
{  
  
}  
}
```

Ідентифікатори інтерфейсів
зазвичай починаються
з літери I



Інтерфейси (2)

```
interface IInterface {  
    void Method();  
}
```

Клас реалізує інтерфейс

```
{  
class MyClass : IInterface {  
    public void Method() {  
        Console.WriteLine("Реалізація інтерфейсу.");  
    }  
}
```

```
class Program {  
    static void Main() {  
        MyClass my = new MyClass();  
        my.Method();  
    }  
}
```



Інтерфейси (3)

- Клас може реалізовувати декілька інтерфейсів:

```
interface IInterface1 {  
    [void Method1();]  
}  
interface IInterface2 {  
    void Method2();  
}  
class MyClass : IInterface1, IInterface2 {  
    public void Method1() {  
        Console.WriteLine("Реалізація IInterface 1");  
    }  
    public void Method2() {  
        Console.WriteLine("Реалізація IInterface 2");  
    }  
}
```

Усі члени інтерфейсу неявно
public та abstract



Інтерфейси (4)

- Реалізація декількох інтерфейсів з однаковою сигнатурою:

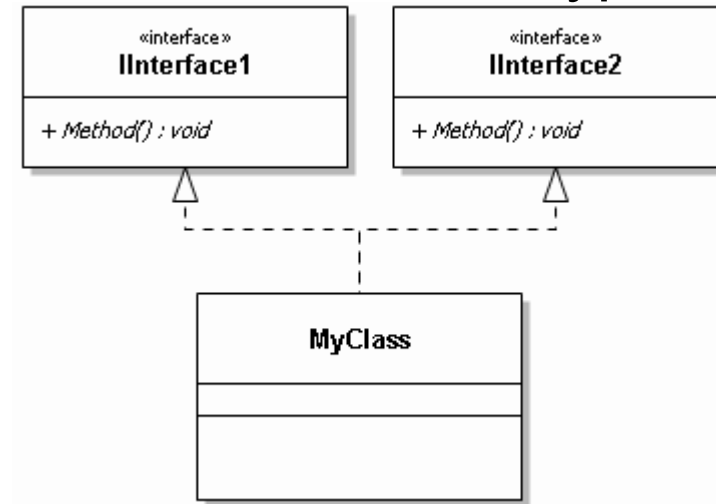
```
interface IInterface1 {  
    void Method();  
}
```

```
interface IInterface2 {  
    void Method();  
}
```

```
class MyClass : IInterface1,  
                IInterface2 {
```

```
    void IInterface1.Method() {  
        Console.WriteLine("Реалізація IInterface 1");  
    }  
    void IInterface2.Method() {  
        Console.WriteLine("Реалізація IInterface 2");  
    }  
}
```

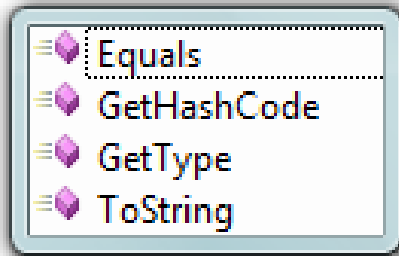
обидві реалізації – private



Інтерфейси (5)

- Реалізація декількох інтерфейсів з однаковою сигнатурою (продовження):

```
class Program {  
    static void Main() {  
        MyClass instance = new MyClass();  
        instance.|
```



UpCast

```
IInterface1 instance1 = instance as IInterface1;  
instance1.Method();
```

```
IInterface2 instance2 = instance as IInterface2;  
instance2.Method();
```

```
}
```



Інтерфейси (6)

- Реалізація декількох інтерфейсів з однаковою сигнатурою (інший приклад):

```
interface IInterface1 {  
    void Method();  
}
```

```
interface IInterface2 {  
    void Method();  
}
```

```
class MyClass : IInterface1, IInterface2 {  
    public void Method() {  
        Console.WriteLine("Реалізація IInterface  
(1+2)");  
    }  
}
```


Об'єднання реалізації однойменних абстрактних членів



Інтерфейси (7)

- Інтерфейси можуть наслідувати інші інтерфейси:

```
interface IUndoable {  
    void Undo();  
}  
interface IRedoable : IUndoable {  
    void Redo();  
}  
class MyClass : IRedoable {  
    public void Undo() {  
        Console.WriteLine("Реалізація IUndoable");  
    }  
    public void Redo() {  
        Console.WriteLine("Реалізація IRedoable");  
    }  
}
```



Інтерфейси (8)

- Інтерфейс може бути реалізований декількома класами:

```
interface IAnnoyable {  
    void Annoy();  
}
```

```
class Fly : IAnnoyable {  
    public void Annoy() {  
        Console.WriteLine("Bzzzzzzzzzzzzzzzzzzzz");  
    }  
}
```

```
class AdvertisingAgent : IAnnoyable {  
    public void Annoy() {  
        Console.WriteLine("Do you need glasses?");  
    }  
}
```



Реалізація інтерфейсів за замовчуванням (1)

- Починаючи з версії C# 8.0 (.NET Core 3), члени інтерфейсів можуть мати реалізацію за замовчуванням:

```
interface IAnnoyable {  
    void Annoy() {  
        Console.WriteLine("Grrrrrr!");  
    }  
}  
  
class Fly : IAnnoyable {  
    public void Annoy() {  
        Console.WriteLine("Bzzzzzzzzzzzzzzzzzzzz");  
    }  
}  
  
class AnnoyingPerson : IAnnoyable {  
}
```

- Виклик Annoy у екземплярі типу AnnoyingPerson можливий лише з використанням UpCast



Реалізація інтерфейсів за замовчуванням (2)

- Для чого може знадобитися реалізація інтерфейсів за замовчуванням?
- Основний варіант використання – для забезпечення зворотної сумісності.
- Приклад:
 - Ви розробили надзвичайно успішну бібліотеку класів, якою користуються тисячі розробників по всьому світу.
 - Ваша бібліотека містить інтерфейс **I**, і тепер ви вирішили, що у новій версії вам потрібен додатковий метод **M**.
 - Проблема: Ви не можете додати інший метод **M** до **I**, тому що це порушить існуючі класи, які реалізують **I** (оскільки вони не реалізують **M**), і ви не можете замінити **I** на абстрактний базовий клас, тому що це також порушить існуючі класи, які реалізують **I**, і ви втратите можливість множинного наслідування.



Реалізація інтерфейсів за замовчуванням (3)

```
interface I1{
    void M() { Console.WriteLine("I1.M"); } }
interface I2{
    void M() { Console.WriteLine("I2.M"); }
}
class C : I1, I2 { }

class Program{
    static void Main(string[] args) {
        C c = new C();
        I1 i1 = c;
        I2 i2 = c;

        i1.M(); // prints "I1.M"
        i2.M(); // prints "I2.M"
        c.M();  // compile error: class 'C' does
                // not contain a member 'M'
    }
}
```



Інтерфейси IEnumerable та IEnumerator (1)

- Раніше ми розглядали, що у C# є оператор циклу foreach, який дозволяє «перебрати» усі елементи масиву

```
int[] myArrayOfInts = {10, 20, 30, 40};  
foreach(int i in myArrayOfInts)  
{  
    Console.WriteLine(i);  
}
```

- Виглядає так, що foreach може застосовуватися лише з масивами, але це не так
- foreach може бути застосований до будь-якого типу даних, у якому є метод GetEnumerator(), який, власне кажучи, і використовує foreach



Інтерфейси IEnumerable та IEnumerator (2)

■ Розглянемо приклад:

```
class Car {  
    public string Brand {get; set;}  
    public string Model {get; set;}  
    public Car(string b, string m) {  
        Brand = b;  
        Model = m;  
    }  
}
```



Інтерфейси IEnumerable та IEnumerator (3)

■ Розглянемо приклад (продовження):

```
public class Garage {  
    private Car[] carArray = new Car[4];  
    public Garage() {  
        carArray[0] = new Car("Toyota", "C-HR");  
        carArray[1] = new Car("Renault", "Clio");  
        carArray[2] = new Car("Hyundai", "Tucson");  
        carArray[3] = new Car("Skoda", "Octavia");  
    }  
}
```

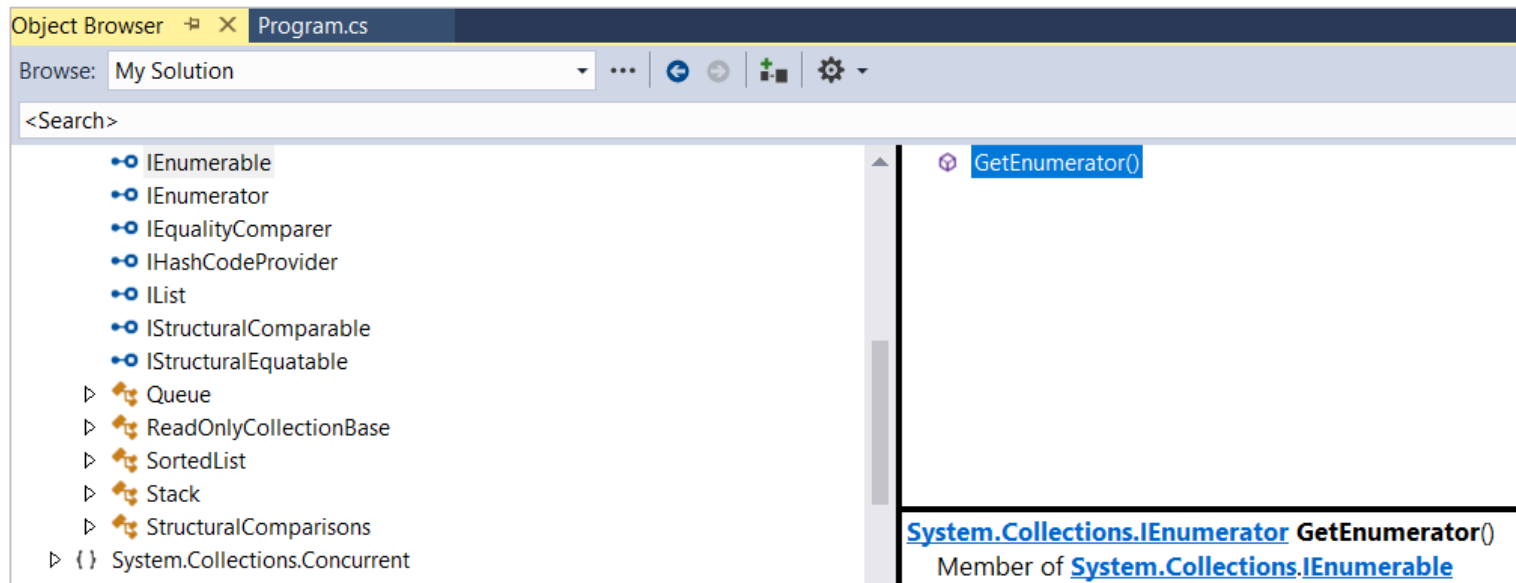
■ Було б дуже зручно обробляти вміст Garage з використанням foreach:

```
Garage KhAICars = new Garage();  
foreach (Car c in KhAICars) {  
    c.PrintState();  
}
```



Інтерфейси IEnumerable та IEnumerator (4)

- Але компілятор повідомить, що у класі Garage не реалізовано метод GetEnumerator(), сигнатуру якого описано у інтерфейсі IEnumerable з простору імен System.Collections
- Тобто, щоб до типу даних можна було застосувати оператор foreach, тип даних має реалізувати інтерфейс IEnumerable. Цей інтерфейс описує лише один метод:



Інтерфейси IEnumerable та IEnumerator (5)

- Опис інтерфейсу IEnumerable виглядає так:

```
public interface IEnumerable {  
    IEnumerator GetEnumerator();  
}
```

- Тобто, метод GetEnumerator() повертає посилання на інший інтерфейс, System.Collections.IEnumerator.
- Даний інтерфейс містить необхідні члени, що дозволяють обробляти «IEnumerable-сумісний» тип даних.
- Опис інтерфейсу IEnumerator виглядає так:

```
public interface IEnumerator  
{  
    bool MoveNext (); // Advance the internal position of the cursor.  
    object Current { get; } // Get the current item (read-only property).  
    void Reset (); // Reset the cursor before the first member.  
}
```



Інтерфейси IEnumerable та IEnumerator (6)

■ Оновимо опис класу Garage:

```
using System.Collections;
namespace CustomEnumerator {
    public class Garage : IEnumerable {
        private Car[] carArray = new Car[4];
        public Garage() {
            carArray[0] = new Car("Toyota", "C-HR");
            carArray[1] = new Car("Renault", "Clio");
            carArray[2] = new Car("Hyundai", "Tucson");
            carArray[3] = new Car("Skoda", "Octavia");
        }
        public IEnumerator GetEnumerator()
            => carArray.GetEnumerator();
    }
}
```

- У даному прикладі ми використовуємо Enumerator, який реалізовано у класі System.Array, щоб не робити власну реалізацію інтерфейсу IEnumerator



Інтерфейс IComparable (1)

- Інтерфейс System.IComparable визначає поведінку, що дозволяє сортувати об'єкт за якоюсь визначеною ознакою
- Опис даного інтерфейсу виглядає так:

```
public interface IComparable {  
    int CompareTo(object o);  
}
```

- Розглянемо приклад

```
class Car {  
    public string Brand {get; set;}  
    public string Model {get; set;}  
    public double Price {get; set;}  
    public Car(string b, string m, double p) {  
        Brand = b;  
        Model = m;  
        Price = p;  
    }  
}
```



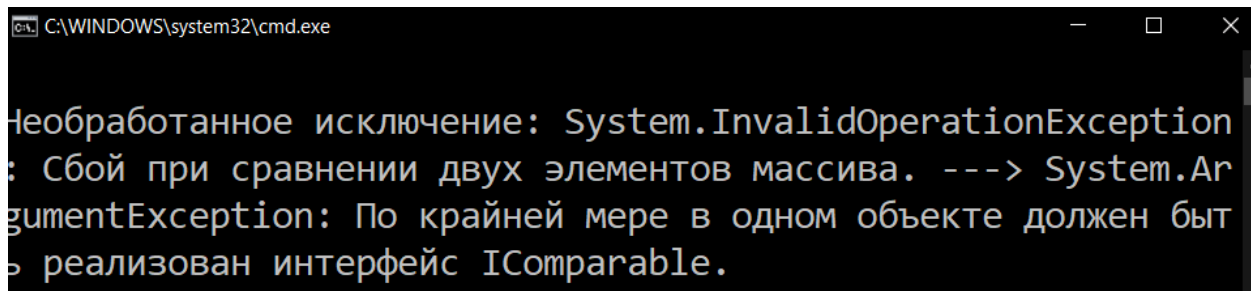
Інтерфейс IComparable (2)

- У основній програмі створимо масив екземплярів типу Car:

```
Car[] myAutos = new Car[5];  
myAutos[0] = new Car("Volvo", "XC90", 71885.0);  
myAutos[1] = new Car("Kia", "Sportage", 20120.0);  
myAutos[2] = new Car("Skoda", "Kodiaq", 38833.0);  
myAutos[3] = new Car("MINI", "Hatch", 41062.0);  
myAutos[4] = new Car("Ravon", "R2", 11369.0);
```

- Спробуємо відсортувати масив:

```
Array.Sort(myAutos);
```



Інтерфейс IComparable (3)

- Метод Sort класу Array не знає, як треба сортувати елементи масиву типу Car
- Щоб сортування запрацювало, необхідно реалізувати у класі Car інтерфейс IComparable:

```
class Car : IComparable{  
    ...  
    int IComparable.CompareTo(object obj) {  
        if (obj is Car temp) {  
            if (this.Price > temp.Price) {  
                return 1;  
            }  
            if (this.Price < temp.Price) {  
                return -1;  
            }  
            return 0;  
        }  
        throw new ArgumentException("Not a Car!");  
    }  
}
```



Інтерфейс IComparable (4)

- Метод CompareTo має повернути ціле число, на базі якого буде прийняте рішення щодо порядку сортування

Повертане значення	Опис
Менше ніж нуль	Даний екземпляр має йти ДО параметру, з яким порівнюємо
Нуль	Даний екземпляр дорівнює параметру, з яким порівнюємо
Більше ніж нуль	Даний екземпляр має йти ПІСЛЯ параметру, з яким порівнюємо



Інтерфейс IComparable (5)

- Завдяки тому, що стандартні класи реалізують інтерфейс IComparable, можна спростити опис реалізації даного інтерфейсу у власному класі:

```
class Car : IComparable{  
    ...  
    int IComparable.CompareTo(object obj) {  
        if (obj is Car temp) {  
            return this.Price.CompareTo(temp.Price);  
        }  
        throw new ArgumentException("Not a Car!");  
    }  
}
```

- Сортуюмо та перевіряємо:

```
Array.Sort(myAutos);  
Console.WriteLine();  
foreach(Car c in myAutos) {  
    Console.WriteLine("{0} {1} {2:0.00}",  
        c.Brand, c.Model, c.Price);  
}
```



Інтерфейс IComparer (1)

- У вищенаведеному прикладі ми використовували властивість Price класу Car як критерій сортування
- Для іншої задачі може знадобитися сортувати, наприклад, за властивістю Brand
- Як можна реалізувати підтримку можливості сортування і за Price, і за Brand?
- Для цього можна використати стандартний інтерфейс IComparer, визначений у просторі імен System.Collections:

```
public interface IComparer
{
    int Compare(object o1, object o2);
}
```



Інтерфейс IComparer (2)

- На відміну від інтерфейсу IComparable, інтерфейс IComparer зазвичай реалізується не у типі даних, який будемо сортувати.
- Цей інтерфейс реалізується у спеціальних допоміжних класах (helper classes), окремий клас для кожного типу сортування (за Brand, Model і т.д.).
- Приклад:

```
using System;
using System.Collections;
public class BrandComparer : IComparer{
    int IComparer.Compare(object o1, object o2) {
        if (o1 is Car t1 && o2 is Car t2) {
            return string.Compare(t1.Brand, t2.Brand,
                                   StringComparison.OrdinalIgnoreCase);
        }
        else{
            throw new ArgumentException("Not a Car!");
        }
    }
}
```



Інтерфейс IComparer (3)

- Тепер можемо відсортувати масив за Brand'ом:

```
Array.Sort()
```

```
▲ 3 of 17 ▼ void Array.Sort(Array array, IComparer comparer)
```

```
Array.Sort(myAutos, new BrandComparer());  
Console.WriteLine("Відсортовано за брендом:");  
foreach(Car c in myAutos) {  
    Console.WriteLine("{0,-8} {1,-8} {2:0.00}",  
                        c.Brand, c.Model, c.Price);  
}
```

```
Відсортовано за брендом:  
Kia      Sportage 20120.00  
MINI     Hatch     41062.00  
Ravon    R2        11369.00  
Skoda    Kodiaq    38833.00  
Volvo    XC90      71885.00
```



Інтерфейс IComparer (4)

- Для більшої зручності можна додати до класу спеціальну статичну властивість, що буде використовуватися для завдання порядку сортування
- Розглянемо приклад:

```
class Car : IComparable{  
    ...  
    public static IComparer SortByBrand  
        => (IComparer)new BrandComparer();  
    ...  
}
```

- Тепер можна викликати метод сортування так:

```
Array.Sort(myAutos, Car.SortByBrand);
```



Інтерфейси: Висновки (1)

- Інтерфейс можна визначити як іменований набір абстрактних членів
- Також інтерфейс часто розглядають як специфікацію поведінки, що може підтримуватися типом даних
- Коли два або більше класів реалізують один і той самий інтерфейс, ви можете обробляти кожен тип даних однаково (фактично це поліморфізм на основі інтерфейсу), навіть якщо ці типи даних визначені у межах різних ієрархій класів
- Для створення нового інтерфейсу у C# використовується ключове слово `interface`
- Тип даних може реалізовувати стільки інтерфейсів, скільки необхідно



Інтерфейси: Висновки (2)

- Можна створювати інтерфейси, що є похідними від кількох базових інтерфейсів
- Крім можливості створення власних користувацьких інтерфейсів, стандартна бібліотека .NET Core надає для використання стандартні інтерфейси (наприклад, IEnumerable, IComparable і т.д.)
- Ви можете створювати власні типи даних, що реалізують ці стандартні інтерфейси, для отримання уніфікованої функціональності з сортування, перерахування і т.д.
- Більш коректною назвою сутності «інтерфейс» була би «контракт» (щоб не плутати з графічним інтерфейсом), адже дана сутність специфікує набір послуг, який буде виконувати тип даних, що реалізує інтерфейс



Питання?

