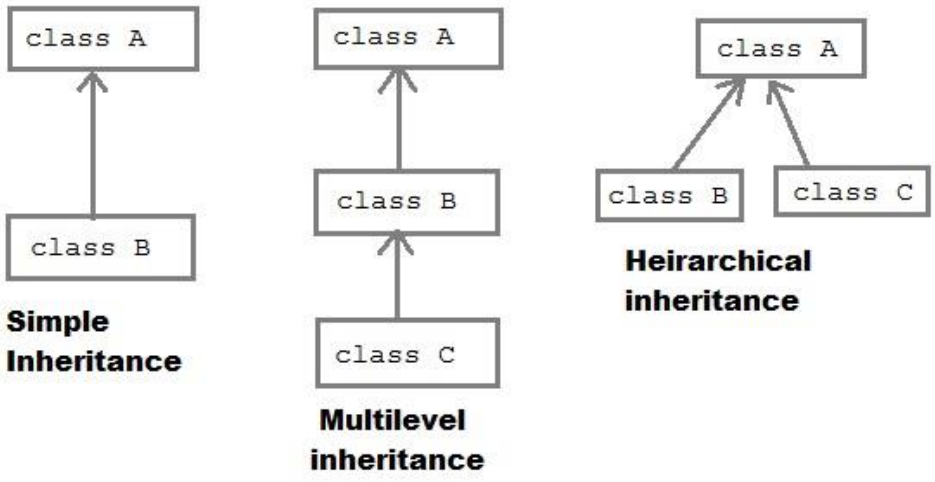
Наслідування реалізації
Implementation Inheritance

Клакович Л. М.

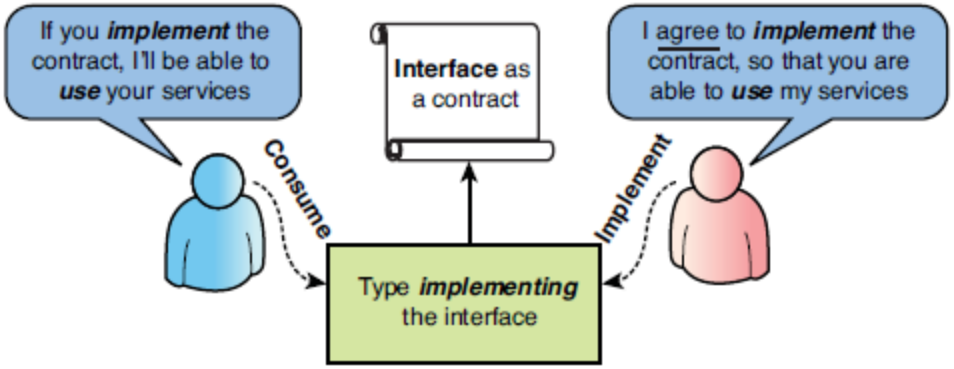
Implementation inheritance

- **Наслідування реалізації (Implementation inheritance)** означає, що тип, наслідується від базового класу, отримуючи всі поля та методи базового типу.
- При цьому, похідний тип отримує реалізацію кожного методу базового типу, крім випадків, коли метод в похідному класі позначений як override.
- Цей тип наслідування найчастіше використовується коли необхідно додати функціональність до існуючого типу, або коли декілька споріднених типів мають однакову загальну функціональність



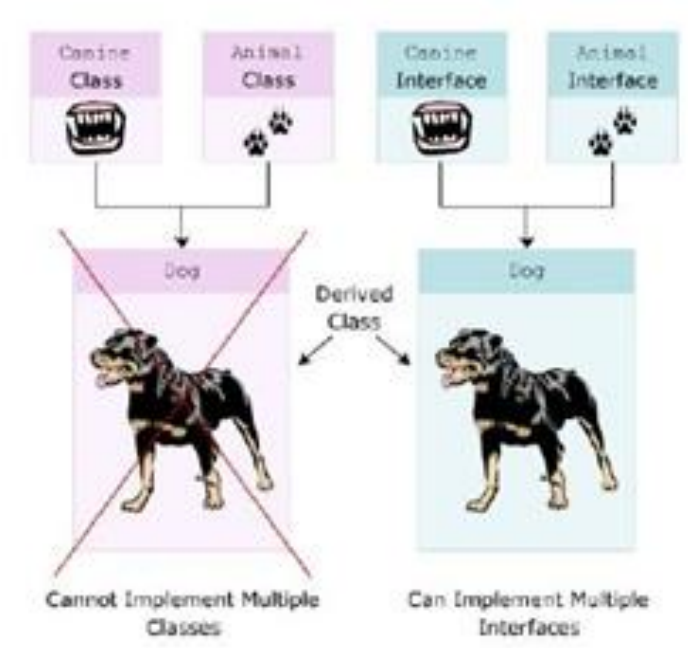
Interface inheritance

- **Наслідування інтерфейсів (Interface inheritance)** означає, що тип наслідує тільки сигнатуру методів, але не наслідує імплементацію
- Наслідування інтерфейсу часто розглядається як гарантія контракту: при наслідуванні інтерфейсу, тип гарантовано забезпечує певну функціональність для клієнтів.



Multiple Inheritance

- C# не підтримує множинного наслідування імплементації (класів).
- Проте C# дозволяє типам наслідувати множину інтерфейсів.
- Тобто, C# class може наслідувати лише один клас і багато інтерфейсів.



Структури і наслідування

- **Структури** не допускають наслідування реалізації, але дозволяють наслідування інтерфейсів
- **Всі структури** наслідуються від типу ***System.ValueType***.
- **Структури** можуть наслідувати довільну кількість інтерфейсів.
- **Всі класи** наслідуються від типу ***System.Object***

Implementation Inheritance

```
class MyDerivedClass : MyBaseClass
{
    // functions and data members here
}
```

- *C# не підтримує приватного наслідування*
- *Тому відсутні акцессори public чи private біля імені базового класу*

Implementation Inheritance

- Якщо клас (або структура) наслідує також ітерфейси, тоді список базового класу повинен передувати списку інтерфейсів:

```
public class DerivedClass : BaseClass, Iface1, Iface2
{
  // etc.
}
```

```
public struct DerivedStruct : Iface1, Iface2
{
        // etc.
}
```

Virtual Methods

- Визначивши метод базового класу як віртуальний (virtual) ми дозволяємо методу бути перевизначеним в будь-якому похідному класі:

```
class MyBaseClass
{
  public virtual string VirtualMethod()
  {
   return “This method defined in MyBaseClass”;
  }
}
```

- Також віртуальною може бути властивість, індексатор чи подія класу

```
class MyBaseClass2
{
   public virtual string ForeName
   {
            get { return foreName;}
            set { foreName = value;}
   }
   private string foreName;
}
```

Virtual and Override

- Для перевизначення віртуального методу (чи властивості) в похідному класі повинен бути специфікатор **override**

```
class MyDerivedClass : MyBaseClass
{
    public override string VirtualMethod()
    {
        return “This is an override defined in MyDerivedClass”;
    }
}
```

- Поля класу та статичні методи не можуть бути визначені як virtual
- Невіртуальні методи не можуть мати специфікатор override
- Перевизначені методи можуть бути повторно перевизначені в наступних похідних класах
- Перевизначення не може міняти видимість члена класу (public члени не можуть бути private після override)

Hiding members and new keyword

• Приховування члена класу відбувається тоді, коли ми визначаємо член з тим самим іменем в похідному типі
• Приховування віртуальних членів руйнує поліморфізм (а компілятор покаже зауваження).
• Для приховування віртульних членів потрібто використовувати спецтфікатор **new:**

```
class MyDerivedClass : MyBaseClass
{
    public new string VirtualMethod() { //…}
}
```

• Приховані члени можуть також бути позначені я віртуальні і розпочати новий ланцюжок поліморфізму:

```
class MyDerivedClass : MyBaseClass
{
    public new virtual string VirtualMethod() { //…}
}
```

```
public class Base
   {
       public virtual void Show()
      {
          Console.WriteLine("Show From Base Class.");
      }
}
```

```
public class Derived : Base
 {
       public new void Show()
       {
           Console.WriteLine("Show From Derived Class.");
       }
}
```

```
public static void Main(string[] args)
       {
          Base objBase;
          objBase = new Base();
          objBase.Show();//Output ----> Show From Base Class.

          objBase = new Derived();
          objBase.Show();//Output--> Show From Base Class.

          Derived objDerived;
          objDerived = new Derived();
          objDerived.Show();//Output ----> Show From Derived Class.
       }
```

Abstract Classes and Methods

- Абстрактні класи дозволяють утворити базові класи з частковою імплементацією функціональності
- Не можна утворити інстанси (екземпляри) абстрактного класу
- Абстрактні методи не містять імплементації і повинні бути перевизначені в неабстрактному похідному класі
- Абстрактні методи є за замовчуванням virtual
- Якщо клас містить абстрактний метод, або наслідується від абстрактного класу і не перевизначає абстрактний метод, він стає абстрактним і повинен бути позначений специфікатором abstract

```
abstract class Building
{
    public abstract decimal CalculateHeatingCost(); // abstract method
}
```

```
public abstract class Base
    {
        public void Walk()
        {
            Console.WriteLine("This is Base Class walk.");
        }
        public virtual void Run()
        {
            Console.WriteLine("This is Base Class run.");
        }
        public abstract void Swim(); // no implementation yet - just abstraction
    }
```

```
public class Derived : Base
    {
        public override void Run()
        {
            Console.WriteLine("This is Derived Class run.");
        }
        public override void Swim()
        {
            Console.WriteLine("This is implementation of Swim in derived class");
        }
    }
```

```
public static void Main(string[] args)
        {
            Base objBase = new Derived();
            objBase.Walk();
            objBase.Run();
            objBase.Swim();
            Base objBase1 = new Base();
}
```

ABSTRACT CLASSES VS INTERFACES

• Інтерфейс – абсслютно абстрактний клас із спеціальною підтримкою на C#
• В абстрактних класах члени можуть змінювати свої рівні доступу (в інтерфейсі - завжди public)
• Абстрактні класи можуть містити деякі імплементації методів
• Абстрактні методи повинні бути перевизначені у похідних класах, а віртуальні методи – можуть бути перевизначеними
• Абстрактні класи можуть містити довільні члени (інтерфейси тільки обмежену множину)
• Інтерфейси – це контракт (домовленість) між системами, а абстрактні класи – це корінь ієрархії класів

Sealed Classes and Methods

- C# дозволяє класам і методам бути визначеними як **sealed**.
- У випадку sealed класу – ми не можемо цей клас наслідувати.
- У випадку sealed методу – ми не можемо перевизначати (override) цей метод.

```
sealed class FinalClass
{
  …..
}
class DerivedClass : FinalClass // wrong. Will give compilation error
{
  …….
}
```

```
class MyClass
{
    public sealed override void FinalMethod()
    {// etc.}
}
class DerivedClass : MyClass
{
    public override void FinalMethod() // compilation error
    {       }
}
```

Constructors of Derived Classes

- Кожен клас в ієрархії може мати власні конструктори. За замовчування для класу надається дефолтний конструктор
- Інстанс класу утворюється після виклику конструктора класу. Якщо не вказано, дефолтний конструктор буде викликано.
- Конструктори викликаються в порядку від System.Object класу, далі по ієрархії конструктори з базових класів і тоді з похідного.
- У цьому процесі кожен конструктор ініціалізує поля власного класу

```
public class A
{
}
public class B : A
{
}
   public void Main()
   {
        var b = new B();
   }
```

Constructors of Derived Classes

```
public class A
{
    protected A(){// init class }
}

public class B : A
{
    public B() : base(){ // do some another init stuff }
}

public class Program {
    public void Main()
    {
                var b = new B();
    }
}
```

```
public class A
{
		private int _a;
		protected A(int a){_a = a; }
		public void printA()
		{
			Console.WriteLine("_a = {0}", _a);
		}
}
public class B : A
{
		private int _b;
		public B(int a, int b) : base(a)
		{
			_b = b;
		}
		public void printB()
		{
			Console.WriteLine("_b = {0}", _b);
		}
}

public class Program {
		public void Main(){
			var b = new B(10, 20);
			b.printA();
			b.printB();
		}
}
```

class Pixel

```
class Pixel:Point
    {
        private Color c;

        public Pixel() : base() { c = Color.Black; }
        public Pixel(Point p, Color cl):base(p){ c = cl; }
        public Pixel(Point p) : this(p,Color.Black) { }
        public override string ToString()
        {
            return base.ToString() + c;
        }
       ...
}
```

Pixel.Equals()

```
class Pixel : Point { ....
    public override bool Equals(Object obj)  {
        if (!base.Equals(obj)) return false;

        if (this.GetType()!= obj.GetType()) return false;

        Pixel p = (Pixel)obj;
        if (c.Equals(p.c)) return true;
        return false;
    }
    public static bool operator ==(Pixel p1, Pixel p2)  {
        return Object.Equals(p1, p2);
    }
    public static bool operator !=(Pixel p1, Pixel p2)  {
        return !(p1 == p2);
    }
}
```