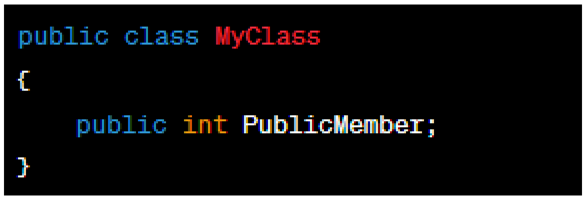
OOP Notes

Access Modifiers

Public

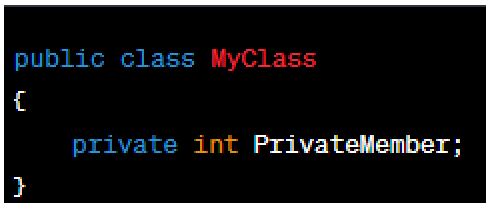
The member is accessible from any other code in the same assembly or a different assembly.



```
public class MyClass
{
    public int PublicMember;
}
```

Private

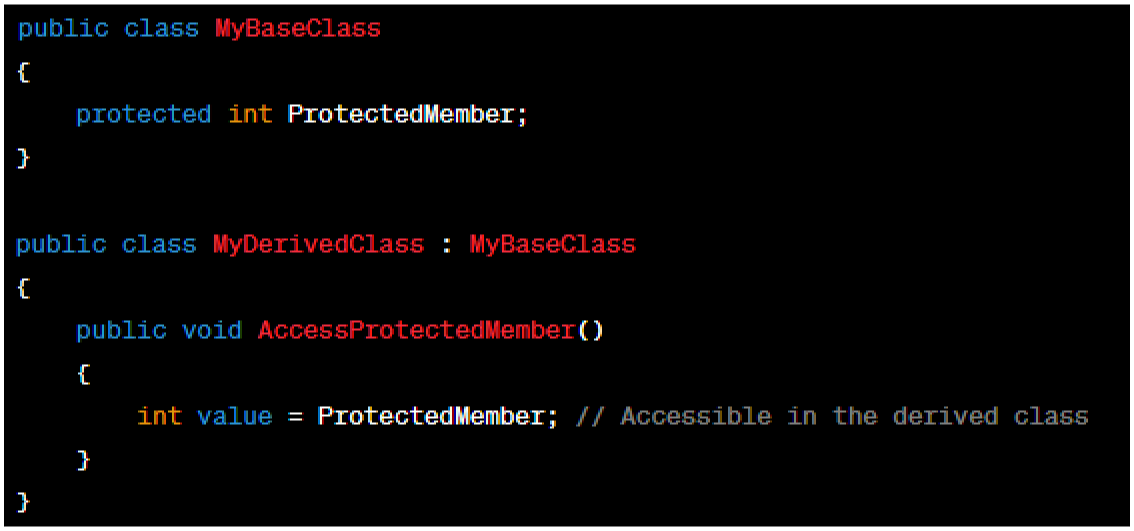
The member is accessible only within the same type or class.



```
public class MyClass
{
    private int PrivateMember;
}
```

Protected

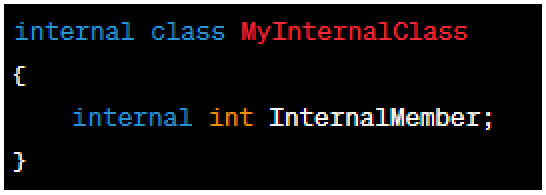
The member is accessible within the same type or class and by derived types or classes.



```
public class MyBaseClass
{
    protected int ProtectedMember;
}

public class MyDerivedClass : MyBaseClass
{
    public void AccessProtectedMember()
    {
        int value = ProtectedMember; // Accessible in the derived class
    }
}
```

Internal

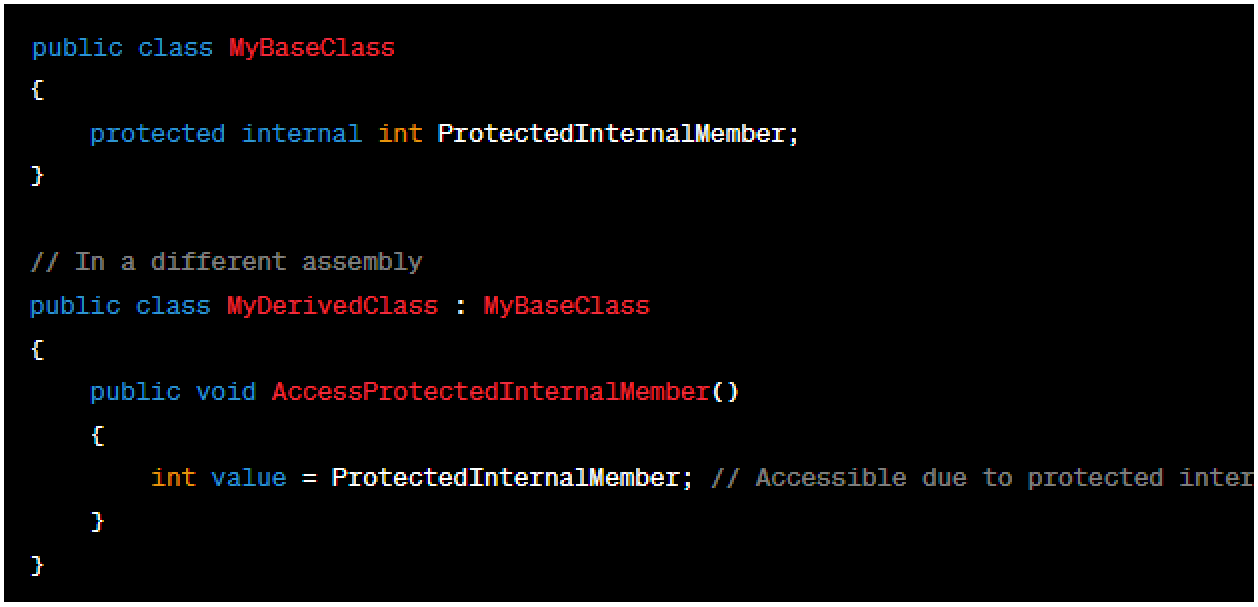
The member is accessible within the same assembly but not from outside the assembly.



```
internal class MyInternalClass
{
    internal int InternalMember;
}
```

Protected Internal

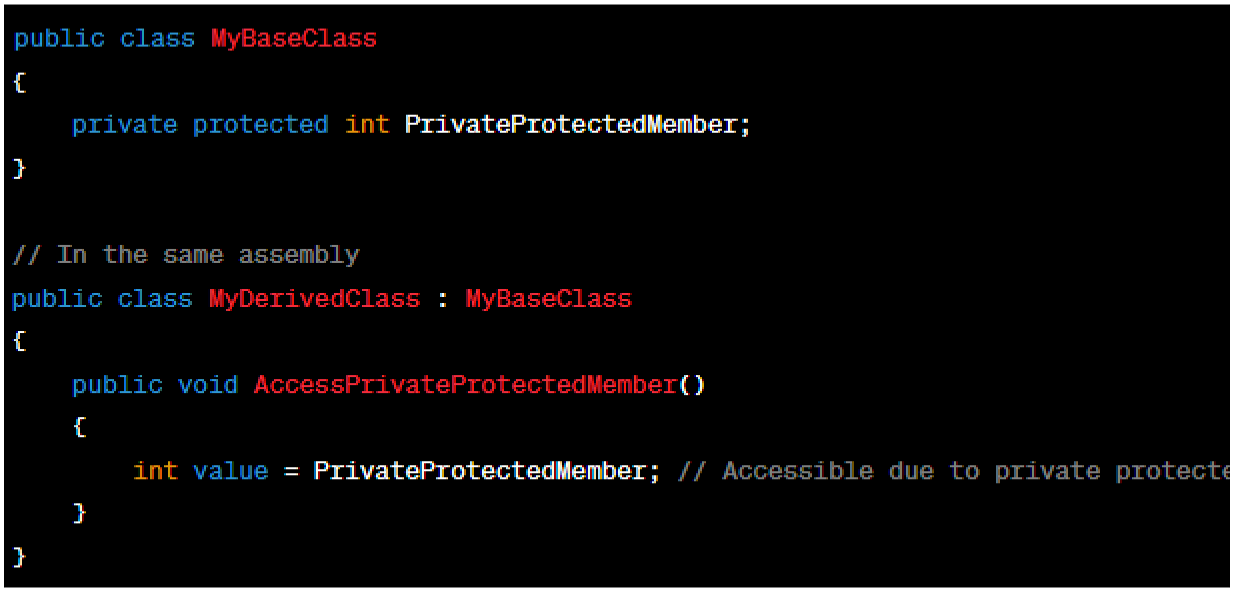
The member is accessible within the same assembly and by derived types or classes, even if they are in a different assembly.



```
public class MyBaseClass
{
    protected internal int ProtectedInternalMember;
}

// In a different assembly
public class MyDerivedClass : MyBaseClass
{
    public void AccessProtectedInternalMember()
    {
        int value = ProtectedInternalMember; // Accessible due to protected inter
    }
}
```

Private Protected

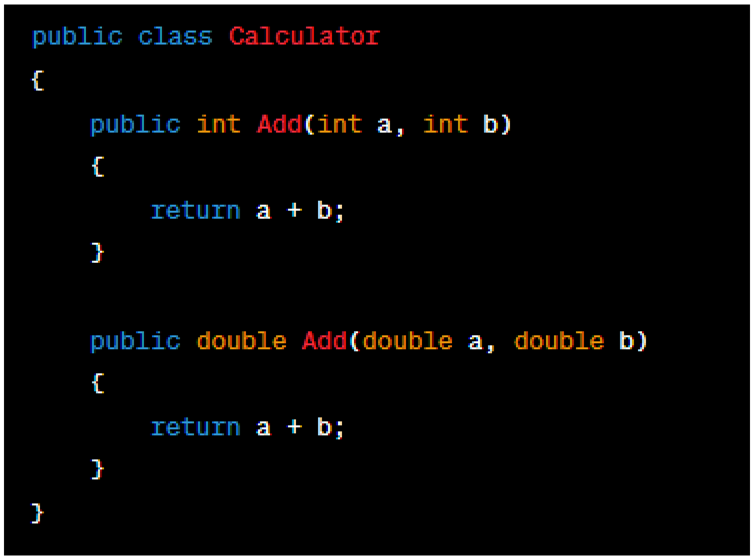
The member is accessible within the same assembly by derived types or classes.



```
public class MyBaseClass
{
    private protected int PrivateProtectedMember;
}

// In the same assembly
public class MyDerivedClass : MyBaseClass
{
    public void AccessPrivateProtectedMember()
    {
        int value = PrivateProtectedMember; // Accessible due to private protecte
    }
}
```

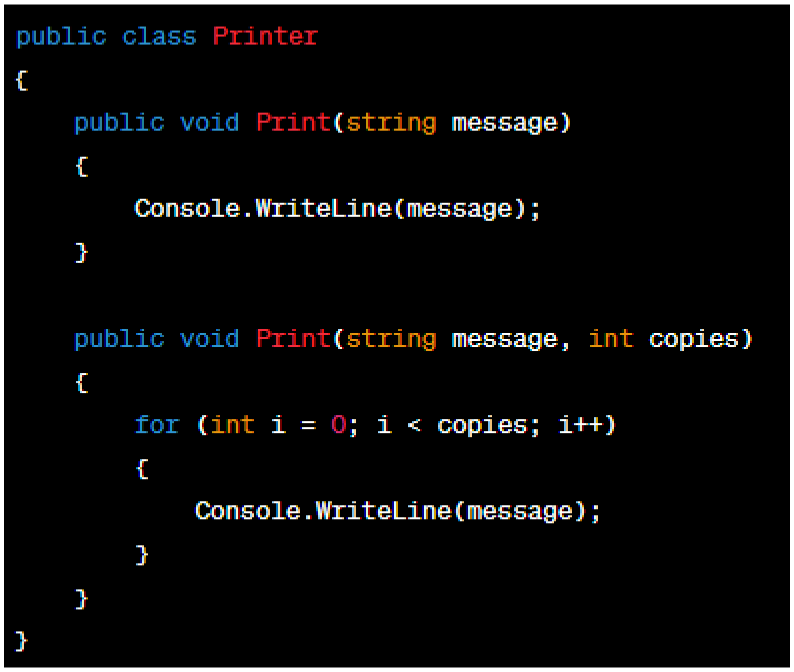
Overloading Types

1. Method Overloading
   a. Overloading By Parameter type



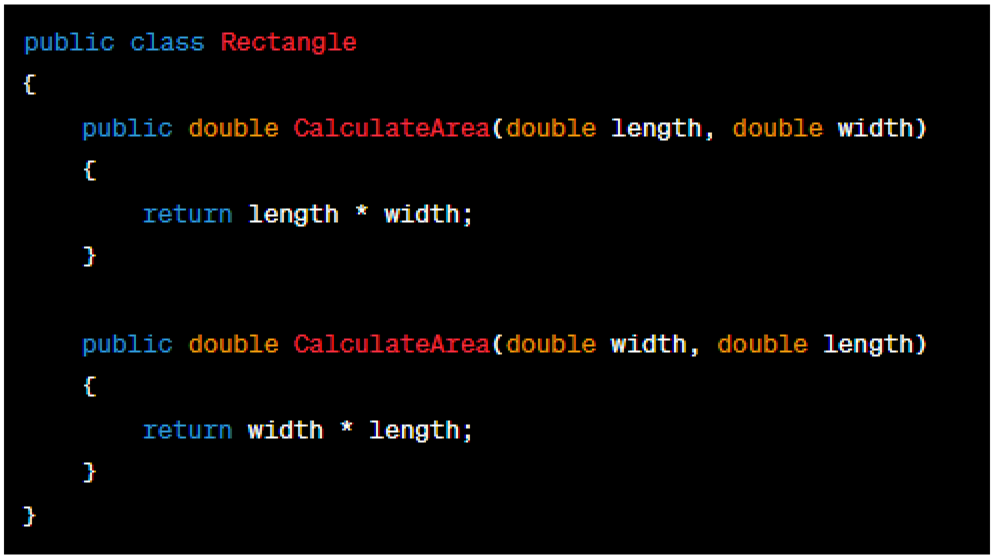
```
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }

    public double Add(double a, double b)
    {
        return a + b;
    }
}
```

   b. Overloading By Parameter Count



```
public class Printer
{
    public void Print(string message)
    {
        Console.WriteLine(message);
    }

    public void Print(string message, int copies)
    {
        for (int i = 0; i < copies; i++)
        {
            Console.WriteLine(message);
        }
    }
}
```

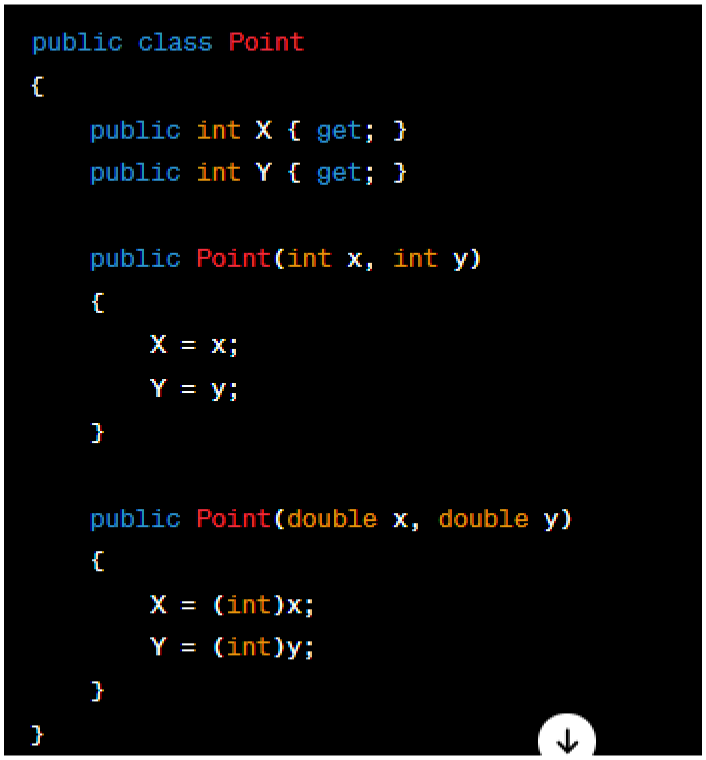
   c. Overloading By Parameter Order



```
public class Rectangle
{
    public double CalculateArea(double length, double width)
    {
        return length * width;
    }

    public double CalculateArea(double width, double length)
    {
        return width * length;
    }
}
```

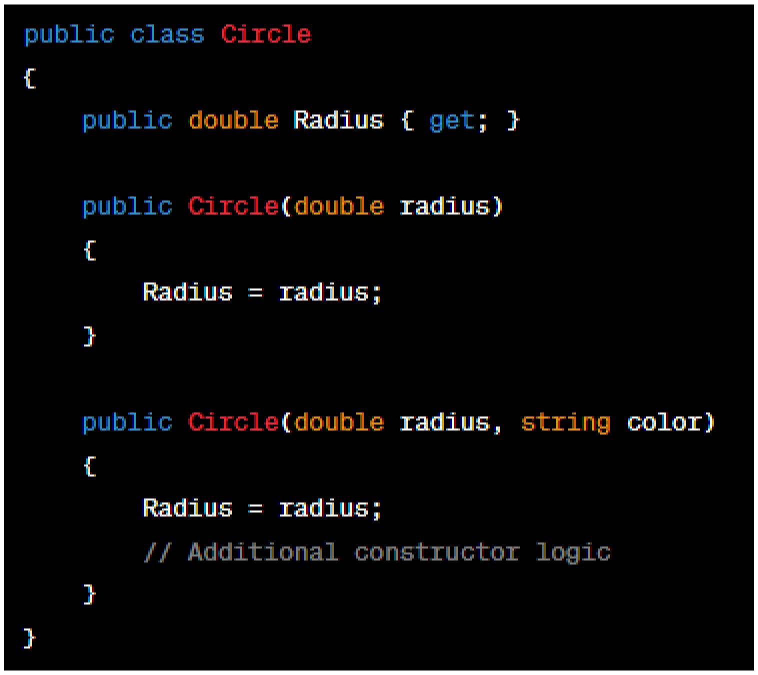
2. Constructor Overloading

a. Overloading By Parameter Type



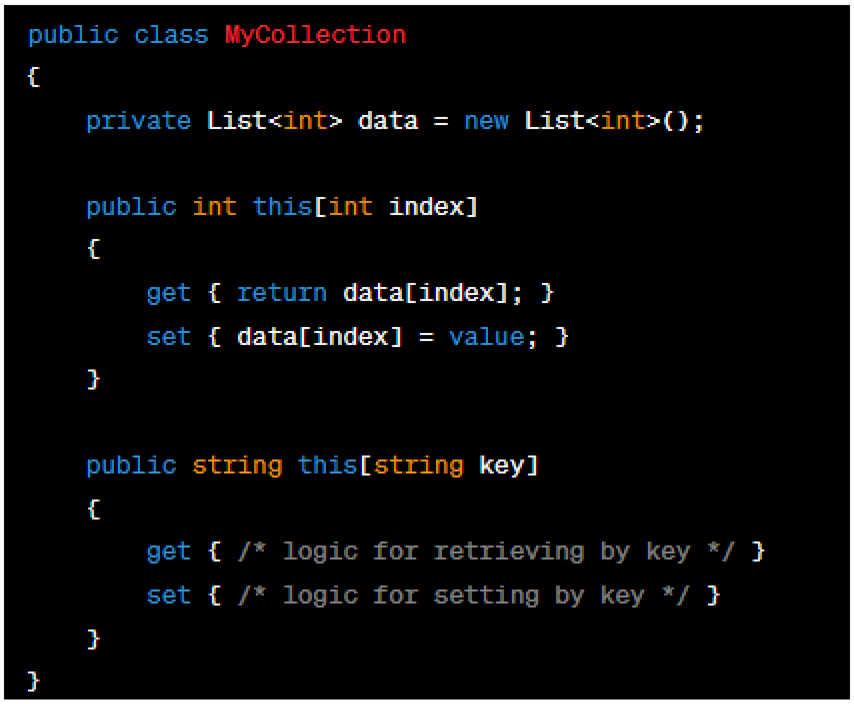
```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public Point(double x, double y)
    {
        X = (int)x;
        Y = (int)y;
    }
}
```

b. Overloading By Parameter Count



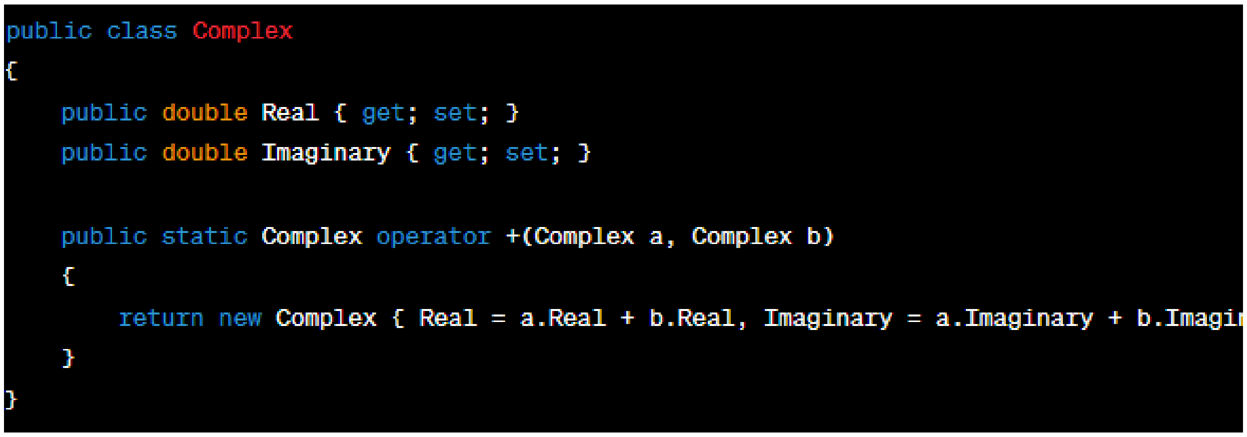
```
public class Circle
{
    public double Radius { get; }

    public Circle(double radius)
    {
        Radius = radius;
    }

    public Circle(double radius, string color)
    {
        Radius = radius;
        // Additional constructor logic
    }
}
```

3. Indexer Overloading
    a. Overloading By Parameter Type



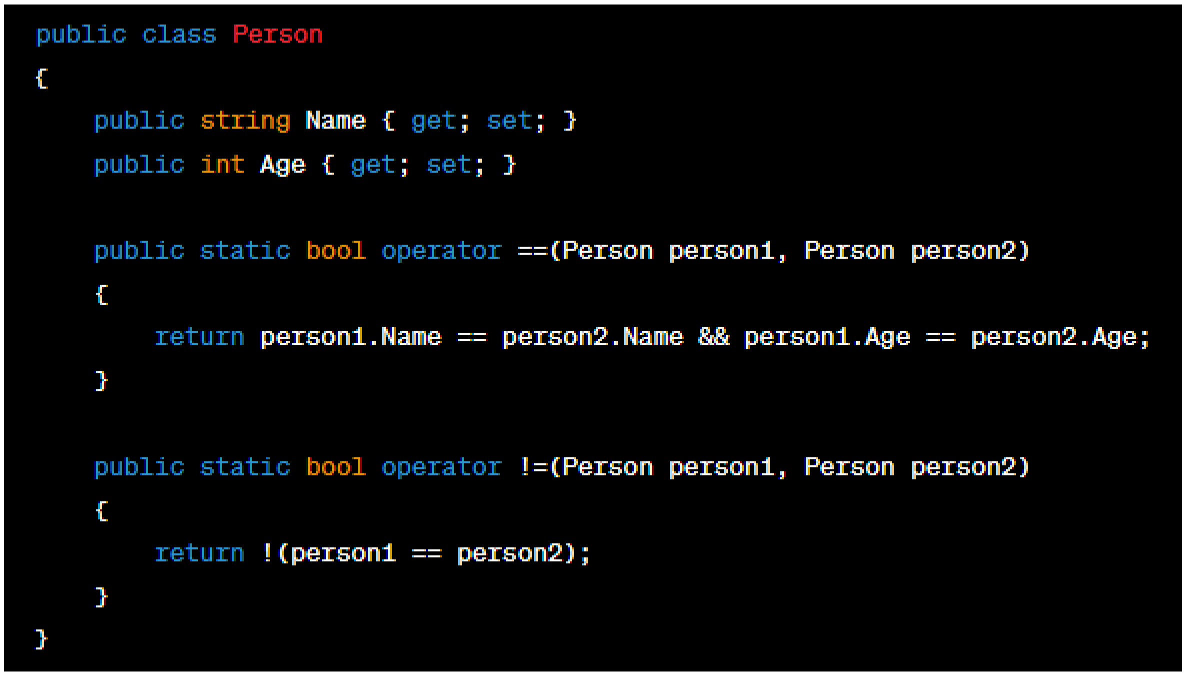
```
public class MyCollection
{
    private List<int> data = new List<int>();

    public int this[int index]
    {
        get { return data[index]; }
        set { data[index] = value; }
    }

    public string this[string key]
    {
        get { /* logic for retrieving by key */ }
        set { /* logic for setting by key */ }
    }
}
```

4. Operator Overloading
    a. Overloading Arithmetic Operator



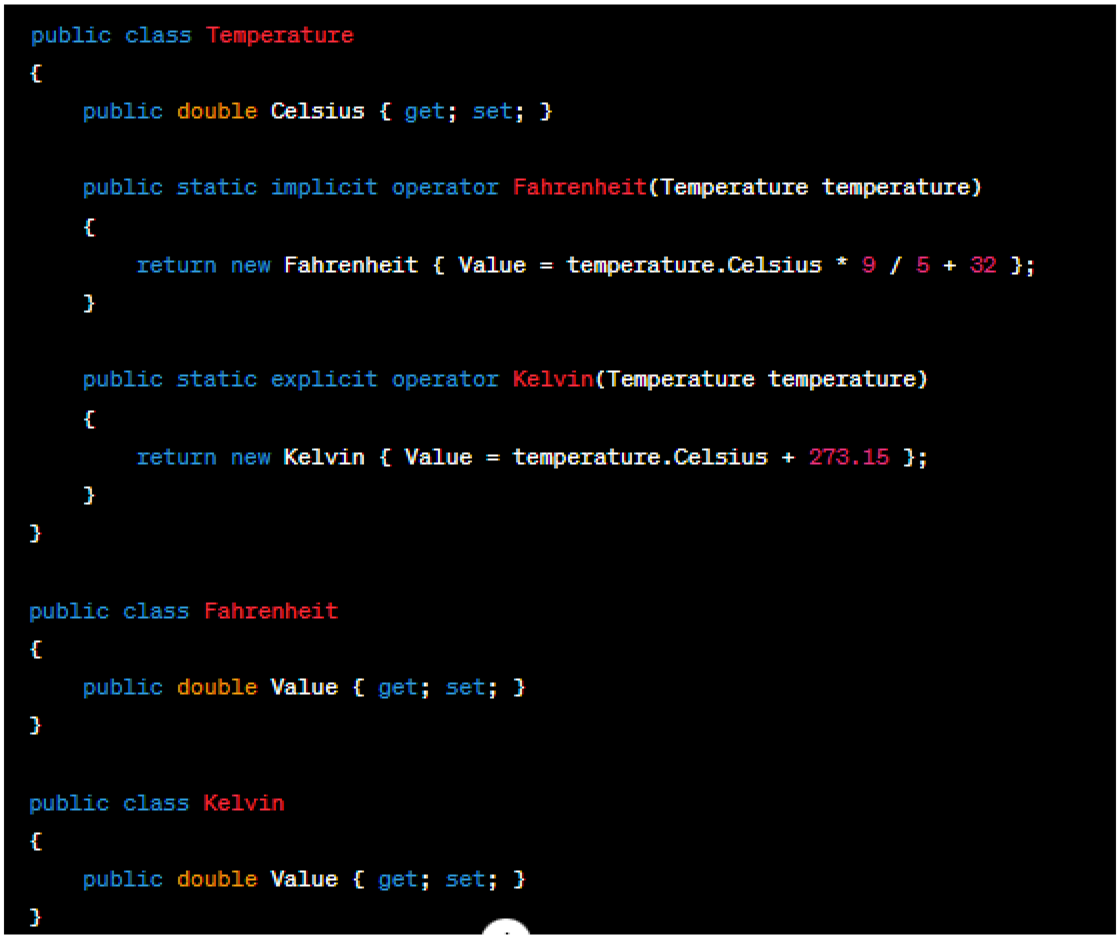
```
public class Complex
{
    public double Real { get; set; }
    public double Imaginary { get; set; }

    public static Complex operator +(Complex a, Complex b)
    {
        return new Complex { Real = a.Real + b.Real, Imaginary = a.Imaginary + b.Imagi
    }
}
```

b. Overloading By Comparison Operator



```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public static bool operator ==(Person person1, Person person2)
    {
        return person1.Name == person2.Name && person1.Age == person2.Age;
    }

    public static bool operator !=(Person person1, Person person2)
    {
        return !(person1 == person2);
    }
}
```

5. Casting Operator Overloading



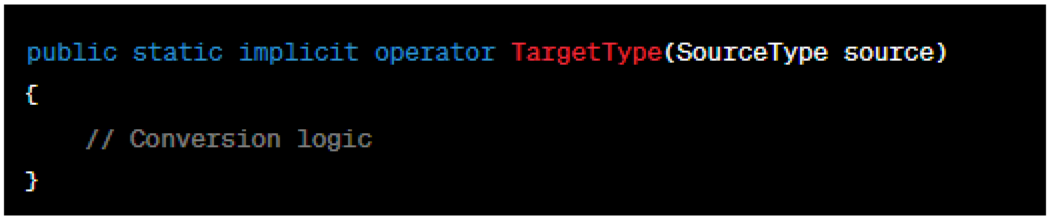
```
public class Temperature
{
    public double Celsius { get; set; }

    public static implicit operator Fahrenheit(Temperature temperature)
    {
        return new Fahrenheit { Value = temperature.Celsius * 9 / 5 + 32 };
    }

    public static explicit operator Kelvin(Temperature temperature)
    {
        return new Kelvin { Value = temperature.Celsius + 273.15 };
    }
}

public class Fahrenheit
{
    public double Value { get; set; }
}

public class Kelvin
{
    public double Value { get; set; }
}
```

<u>**Casting Operator Overloading**</u>

Casting operator overloading in C# allows you to define custom conversion behavior between different types. This enables you to provide meaningful conversions when casting objects from one type to another. There are two types of casting operator overloading: implicit and explicit.

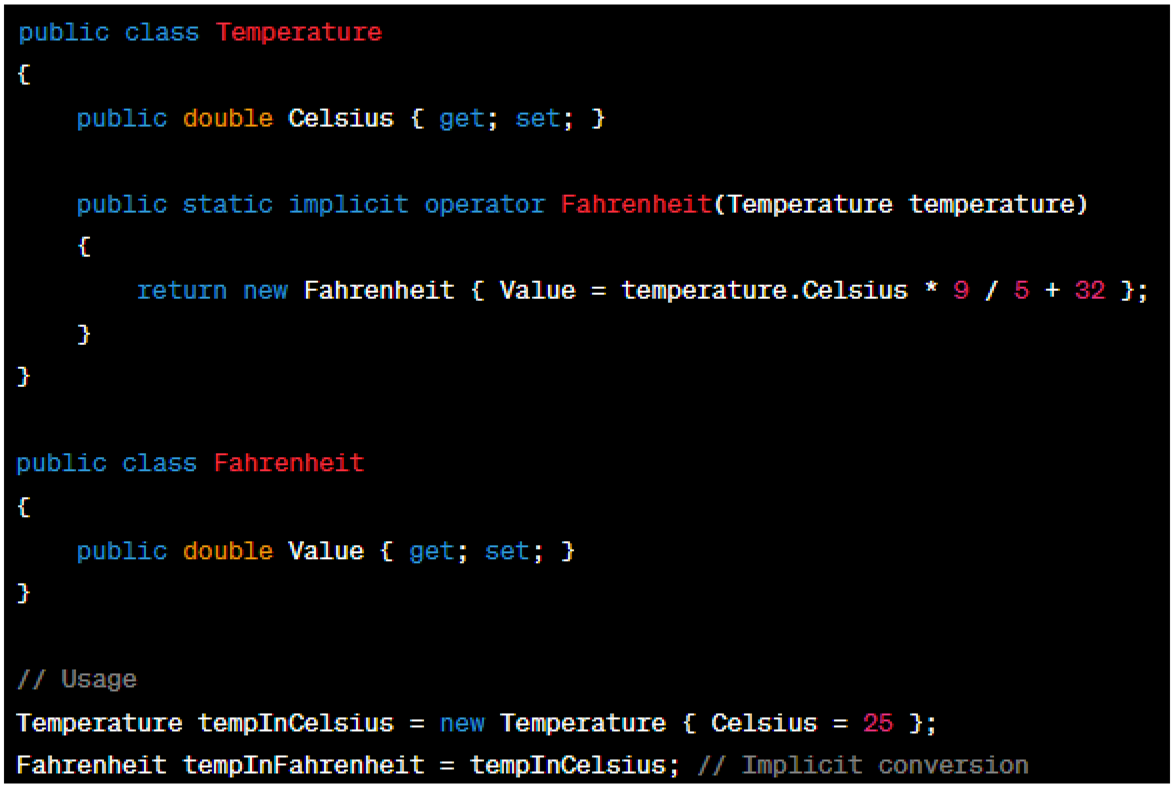
<u>Implicit Casting Operator Overloading</u>

Syntax



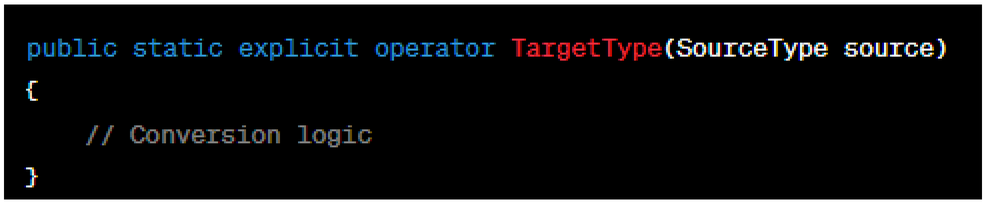
```
public static implicit operator TargetType(SourceType source)
{
    // Conversion logic
}
```

The **implicit** keyword indicates that the conversion is performed implicitly without requiring an explicit cast from the user.
This type of casting is performed automatically by the compiler when a compatible assignment or operation is encountered.



```
public class Temperature
{
    public double Celsius { get; set; }

    public static implicit operator Fahrenheit(Temperature temperature)
    {
        return new Fahrenheit { Value = temperature.Celsius * 9 / 5 + 32 };
    }
}

public class Fahrenheit
{
    public double Value { get; set; }
}

// Usage
Temperature tempInCelsius = new Temperature { Celsius = 25 };
Fahrenheit tempInFahrenheit = tempInCelsius; // Implicit conversion
```

<u>Explicit Casting Overloading</u>
Syntax



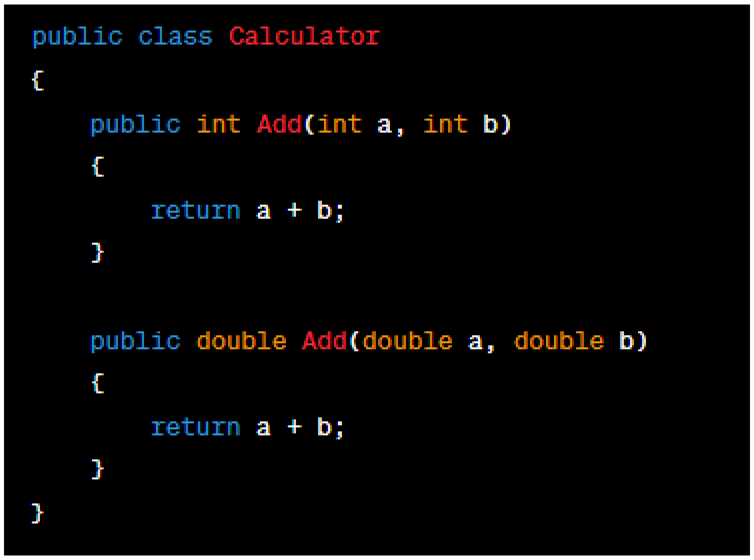
```
public static explicit operator TargetType(SourceType source)
{
    // Conversion logic
}
```

The **explicit** keyword indicates that the conversion must be explicitly specified by the user through casting.
This type of casting is useful when there might be loss of data or when the conversion is not straightforward.

```
public class Temperature
{
    public double Celsius { get; set; }

    public static explicit operator Kelvin(Temperature temperature)
    {
        return new Kelvin { Value = temperature.Celsius + 273.15 };
    }
}

public class Kelvin
{
    public double Value { get; set; }
}

// Usage
Temperature tempInCelsius = new Temperature { Celsius = 25 };
Kelvin tempInKelvin = (Kelvin)tempInCelsius; // Explicit conversion
```

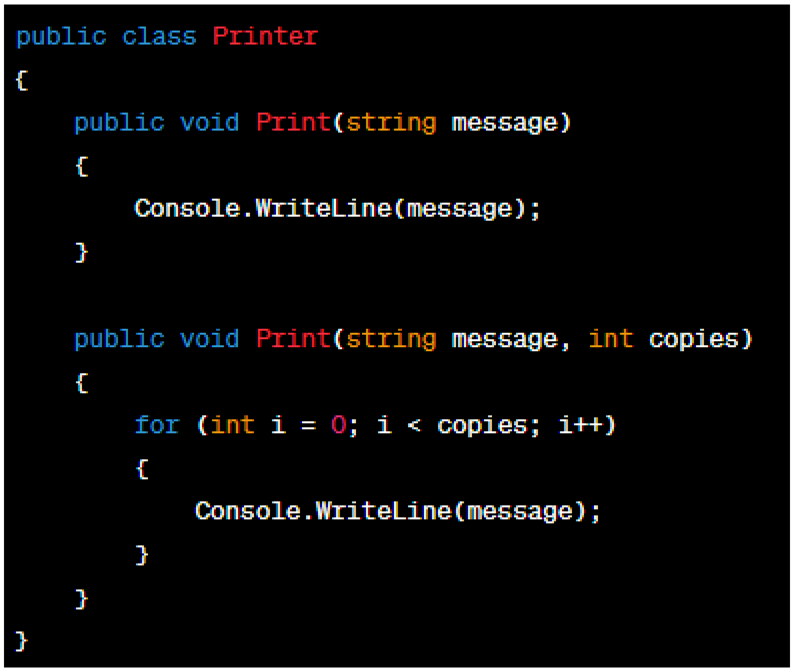
Overloading Types

6. Method Overloading
    d. Overloading By Parameter type



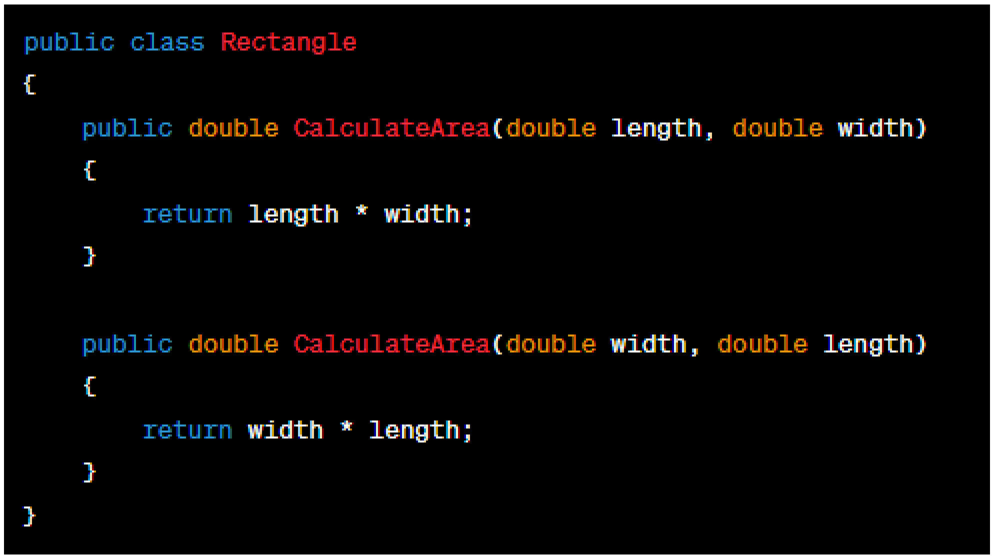
```
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }

    public double Add(double a, double b)
    {
        return a + b;
    }
}
```

    e. Overloading By Parameter Count



```
public class Printer
{
    public void Print(string message)
    {
        Console.WriteLine(message);
    }

    public void Print(string message, int copies)
    {
        for (int i = 0; i < copies; i++)
        {
            Console.WriteLine(message);
        }
    }
}
```

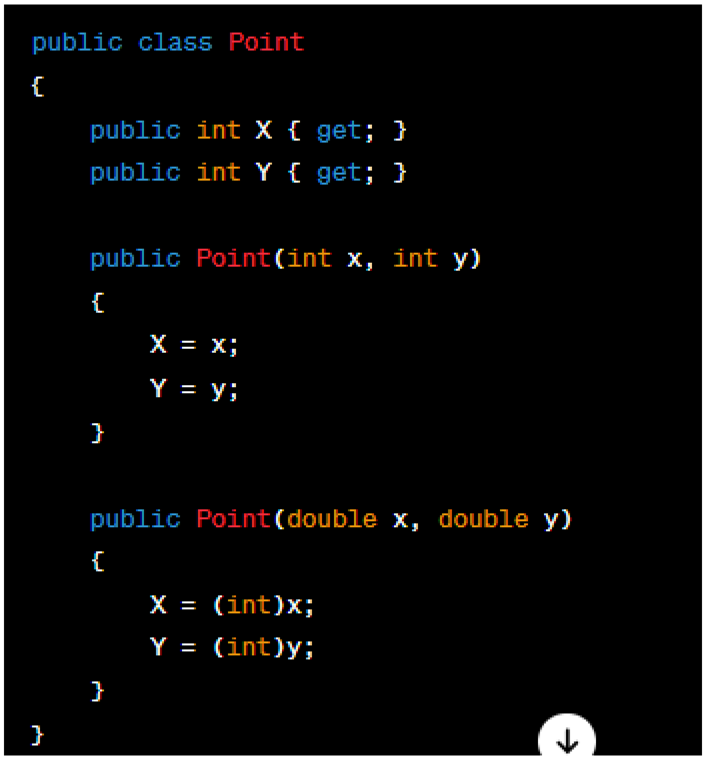
    f. Overloading By Parameter Order



```
public class Rectangle
{
    public double CalculateArea(double length, double width)
    {
        return length * width;
    }

    public double CalculateArea(double width, double length)
    {
        return width * length;
    }
}
```

7. Constructor Overloading

c. Overloading By Parameter Type



```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public Point(double x, double y)
    {
        X = (int)x;
        Y = (int)y;
    }
}
```

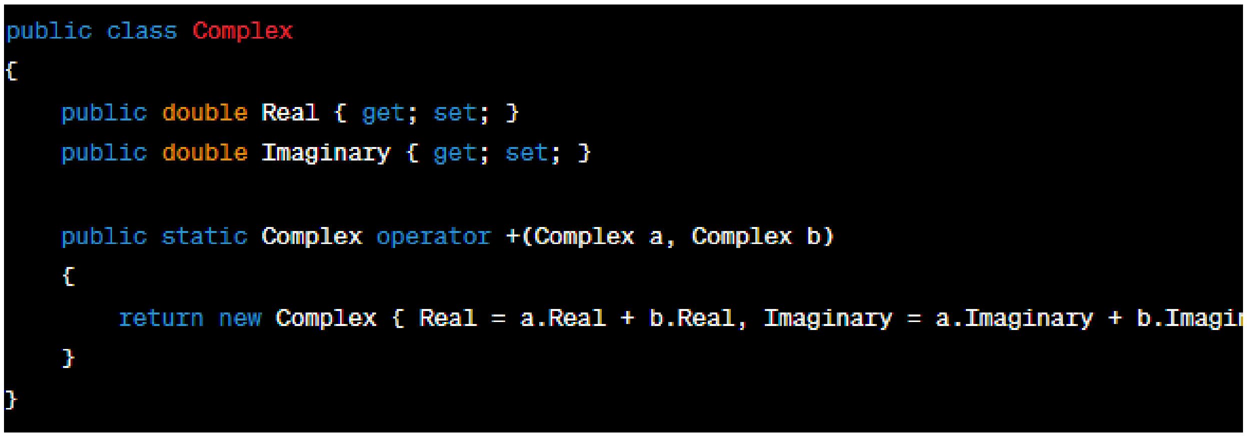
d. Overloading By Parameter Count

```
public class Circle
{
    public double Radius { get; }

    public Circle(double radius)
    {
        Radius = radius;
    }

    public Circle(double radius, string color)
    {
        Radius = radius;
        // Additional constructor logic
    }
}
```

8. Indexer Overloading
    b. Overloading By Parameter Type

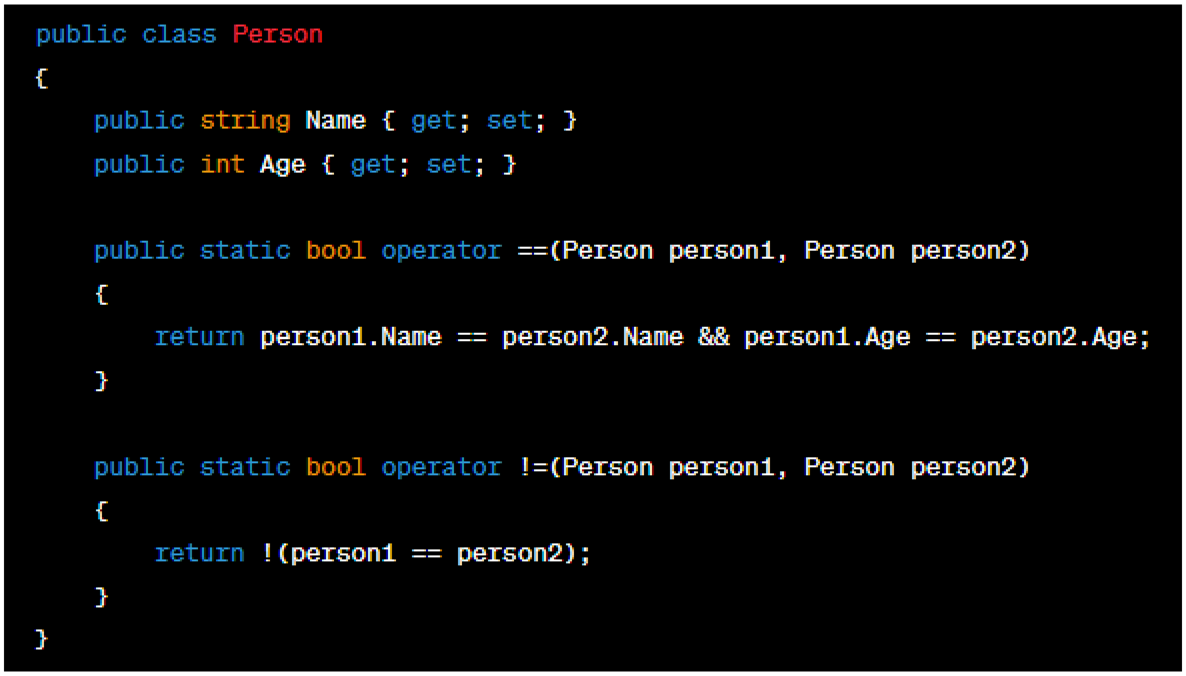
```
public class MyCollection
{
    private List<int> data = new List<int>();

    public int this[int index]
    {
        get { return data[index]; }
        set { data[index] = value; }
    }

    public string this[string key]
    {
        get { /* logic for retrieving by key */ }
        set { /* logic for setting by key */ }
    }
}
```

9. Operator Overloading
    c. Overloading Arithmetic Operator



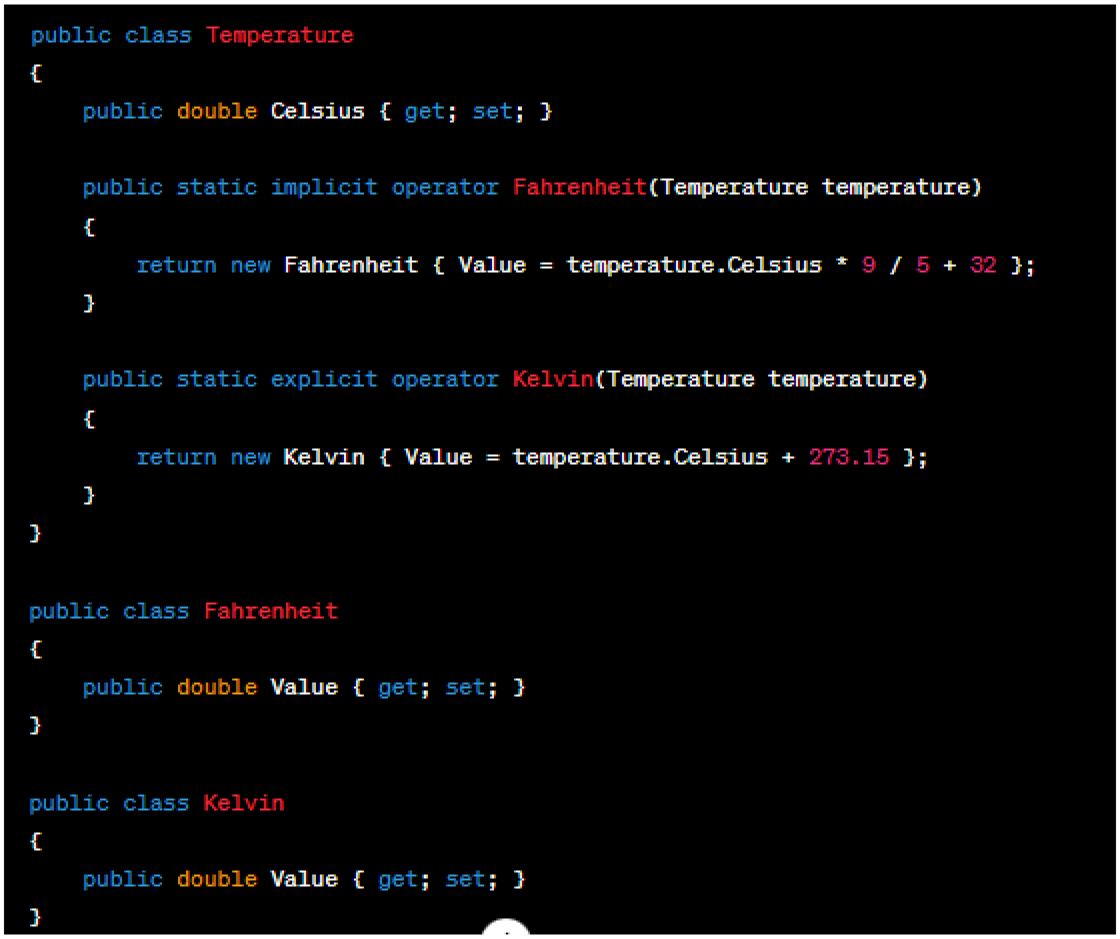
```
public class Complex
{
    public double Real { get; set; }
    public double Imaginary { get; set; }

    public static Complex operator +(Complex a, Complex b)
    {
        return new Complex { Real = a.Real + b.Real, Imaginary = a.Imaginary + b.Imagi
    }
}
```

d. Overloading By Comparison Operator



```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public static bool operator ==(Person person1, Person person2)
    {
        return person1.Name == person2.Name && person1.Age == person2.Age;
    }

    public static bool operator !=(Person person1, Person person2)
    {
        return !(person1 == person2);
    }
}
```

10. Casting Operator Overloading



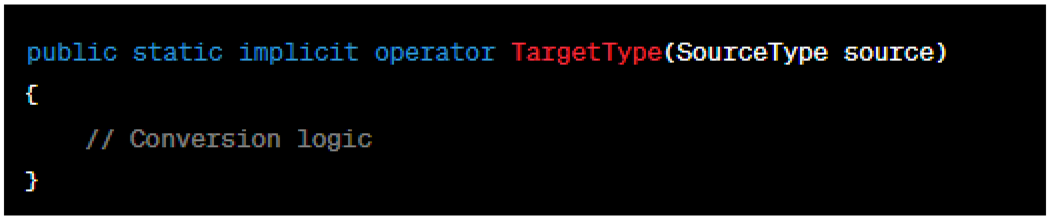
```
public class Temperature
{
    public double Celsius { get; set; }

    public static implicit operator Fahrenheit(Temperature temperature)
    {
        return new Fahrenheit { Value = temperature.Celsius * 9 / 5 + 32 };
    }

    public static explicit operator Kelvin(Temperature temperature)
    {
        return new Kelvin { Value = temperature.Celsius + 273.15 };
    }
}

public class Fahrenheit
{
    public double Value { get; set; }
}

public class Kelvin
{
    public double Value { get; set; }
}
```

Casting Operator Overloading

Casting operator overloading in C# allows you to define custom conversion behavior between different types. This enables you to provide meaningful conversions when casting objects from one type to another. There are two types of casting operator overloading: implicit and explicit.

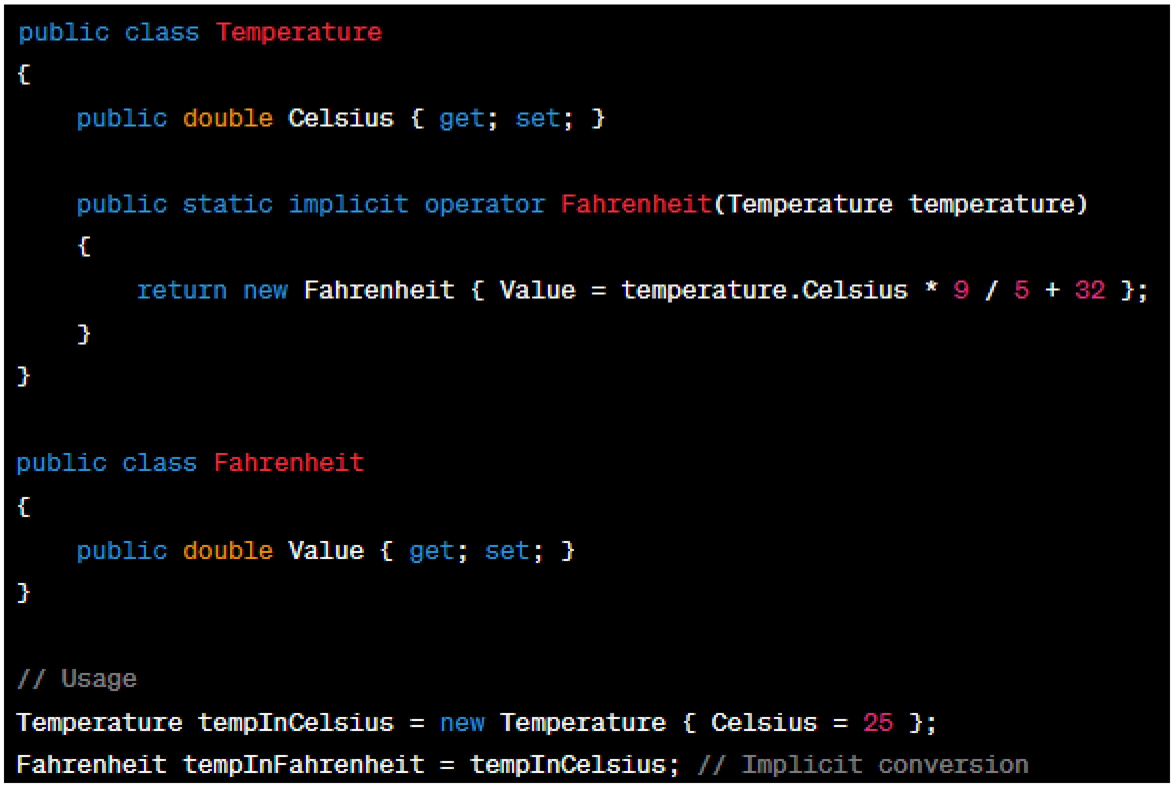
Implicit Casting Operator Overloading

Syntax



```
public static implicit operator TargetType(SourceType source)
{
    // Conversion logic
}
```

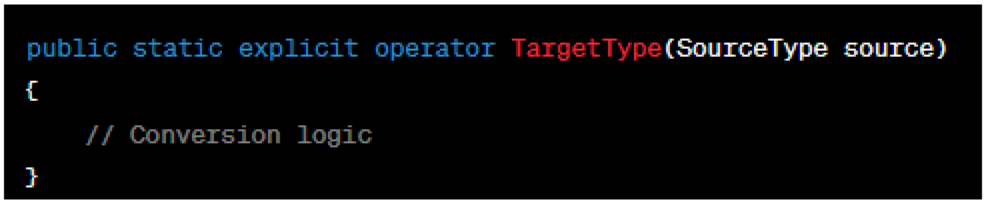
The **implicit** keyword indicates that the conversion is performed implicitly without requiring an explicit cast from the user.
This type of casting is performed automatically by the compiler when a compatible assignment or operation is encountered.



```csharp
public class Temperature
{
    public double Celsius { get; set; }

    public static implicit operator Fahrenheit(Temperature temperature)
    {
        return new Fahrenheit { Value = temperature.Celsius * 9 / 5 + 32 };
    }
}

public class Fahrenheit
{
    public double Value { get; set; }
}

// Usage
Temperature tempInCelsius = new Temperature { Celsius = 25 };
Fahrenheit tempInFahrenheit = tempInCelsius; // Implicit conversion
```

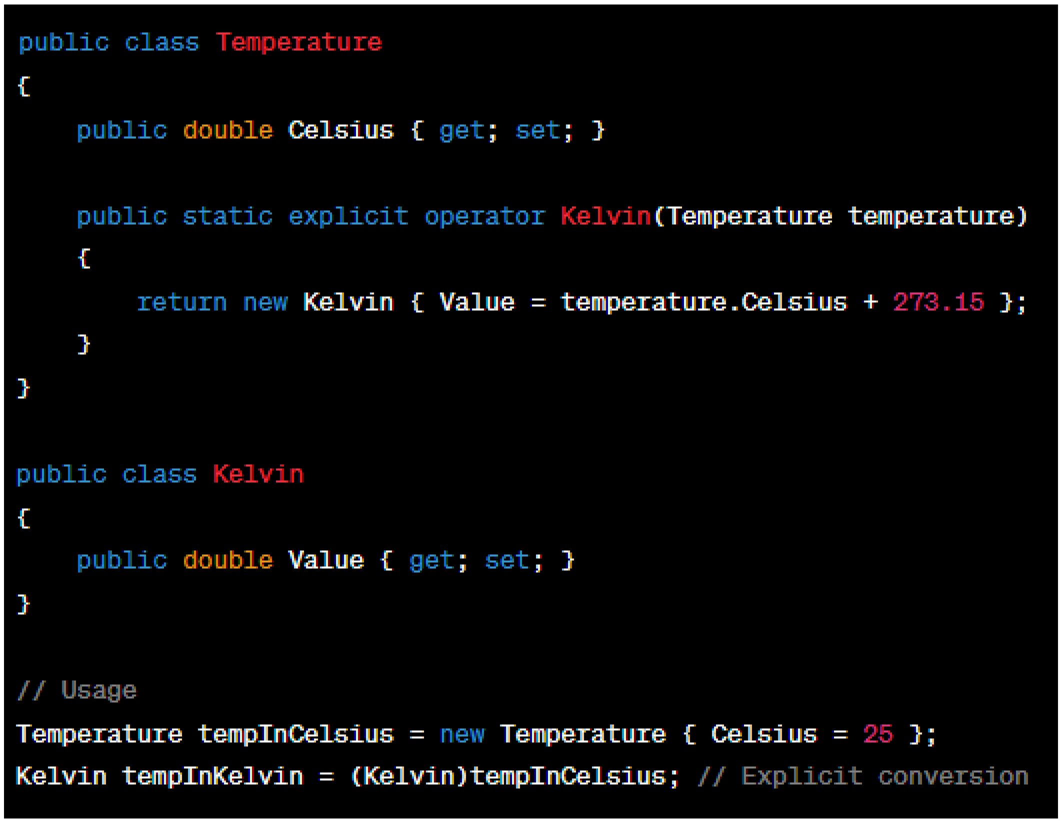
Explicit Casting Overloading

Syntax

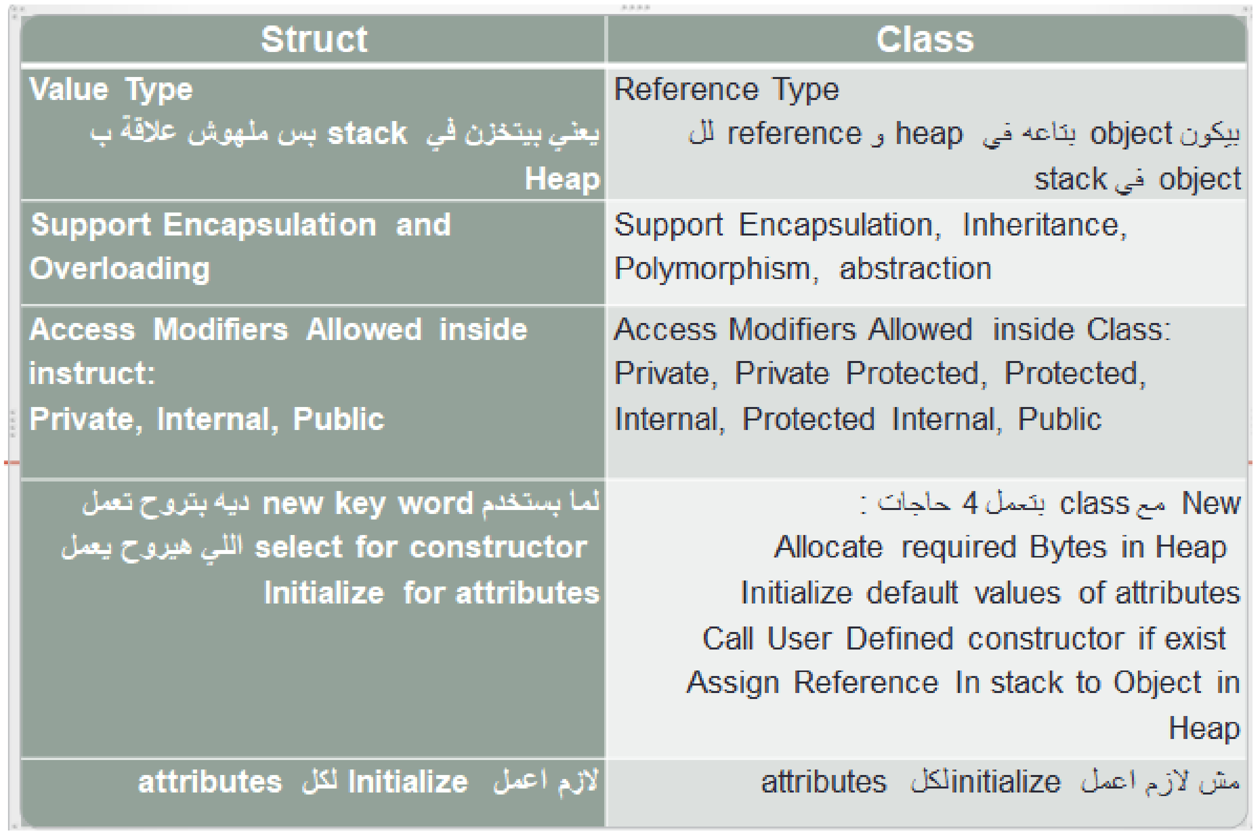


```csharp
public static explicit operator TargetType(SourceType source)
{
    // Conversion logic
}
```

The **explicit** keyword indicates that the conversion must be explicitly specified by the user through casting.
This type of casting is useful when there might be loss of data or when the conversion is not straightforward.



```
public class Temperature
{
    public double Celsius { get; set; }

    public static explicit operator Kelvin(Temperature temperature)
    {
        return new Kelvin { Value = temperature.Celsius + 273.15 };
    }
}

public class Kelvin
{
    public double Value { get; set; }
}

// Usage
Temperature tempInCelsius = new Temperature { Celsius = 25 };
Kelvin tempInKelvin = (Kelvin)tempInCelsius; // Explicit conversion
```



Struct	Class
Value Type يعني بيتخزن في stack بس ملهوش علاقة ب Heap	Reference Type بيكون object بتاعه في heap و reference لل object في stack
Support Encapsulation and Overloading	Support Encapsulation, Inheritance, Polymorphism, abstraction
Access Modifiers Allowed inside instruct: Private, Internal, Public	Access Modifiers Allowed inside Class: Private, Private Protected, Protected, Internal, Protected Internal, Public
لما بستخدم new key word ديه بتروح تعمل select for constructor اللي هيروح يعمل Initialize for attributes	New مع class بتعمل 4 حاجات : Allocate required Bytes in Heap Initialize default values of attributes Call User Defined constructor if exist Assign Reference In stack to Object in Heap
لازم اعمل Initialize لكل attributes	مش لازم اعمل initializeلكل attributes

امتي استخدم Structوامتي Class ؟

لما بستخدم class فهو بيعملي object في heap لما main function بتخلص شغلها هيمسحلي reference بتاع object اللي في Heap فهيكون unreachable object وده هيعملي مشكلة لانه اول ما يتملي هيبتدي garbish collector تشتغل و هيوقف Normal path execution

طب لو عملت struct

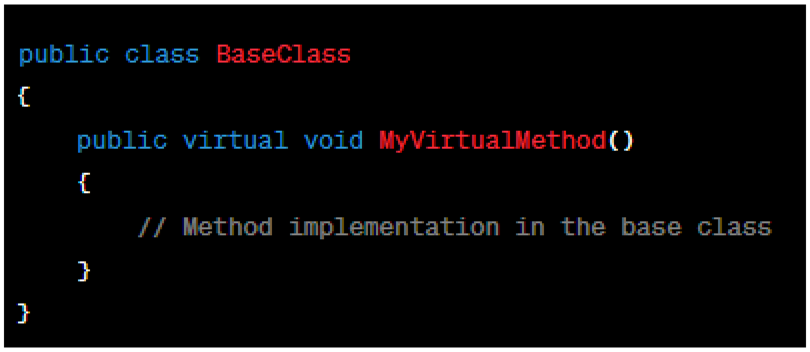
Object موجود اصلا في main stack frame فلما يخلص شغله هيتمسح و الميموري مضيعتهاش

فلو اقدر استخدم struct يبقي احسن اني استخدمه ، طب امتي مقدرش استخدمه ؟

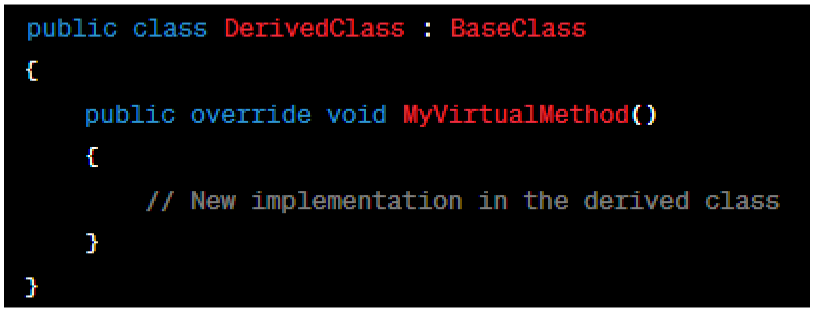
لو معظم business موجود في الواقع فلازم oop فهستخدم class لاني محتاج اعمل Inheritance

<u>Virtual Keyword</u>

Use it to override method in child to parent method.
Achieve dynamic behavior at runtime.



```
public class BaseClass
{
    public virtual void MyVirtualMethod()
    {
        // Method implementation in the base class
    }
}
```



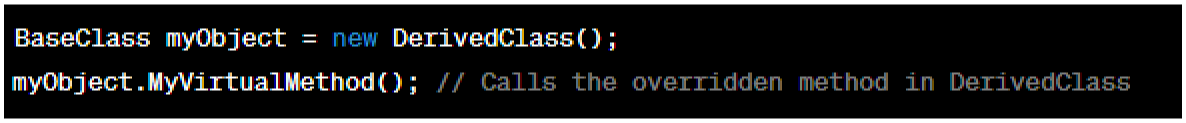
```
public class DerivedClass : BaseClass
{
    public override void MyVirtualMethod()
    {
        // New implementation in the derived class
    }
}
```

Derived classes can provide their own implementation of a virtual method by using the **override** keyword. This allows them to replace the behavior defined in the base class.

<u>Dynamic Binding (Late Binding)</u>

Virtual methods support dynamic binding, meaning that the decision about **which method implementation to call is made at runtime based on the actual type of the object**.

When a virtual method is called on a base class reference that points to an object of a derived class, the overridden method in the derived class is invoked.



```
BaseClass myObject = new DerivedClass();
myObject.MyVirtualMethod(); // Calls the overridden method in DerivedClass
```

Applying override by new keyword called static binding and called Early binding because compiler will bind method based on reference type not object type at compilation time

Applying override by override keyword called dynamic binding and called Late bind because CLR will bind method call based on object type not reference type at run time

OOP Pillars

Encapsulation

الهدف منها انك تعمل encapsulate للداتا عندك وتبدا تتعامل معاها عن طريق getter, setters او Properties (Automatic property, full property, indexers)

Inheritance

عن طريق انك inherit من class تاني attributes, methods

Polymorphism

بينقسم

Overloading

انك تعمل نفس function في class باختلاف No of parameters او types of parameters او order

Overriding

تكون inherit من class تاني method وتعمل عليها override باستخدام Override او New keyword

Abstraction

بيكون class مش كل methods اللي فيه معملوها implementation وبالتالي اي class تاني يعمله Implementation لل methods ديه وبيكون اسمه Concrete class