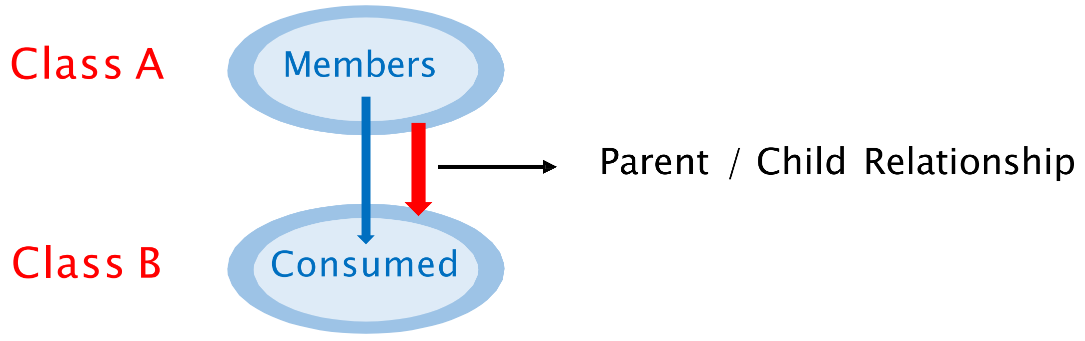
iamneo

Inheritance & Polymorphism

www.iamneo.ai

Inheritance

- Inheritance in C# is a mechanism of consuming the members that are defined in one class from another class.
- A class is a collection of members.
- And the members defined in one class can be consumed from another class by establishing a parent/child relationship between the classes.
- Once established the parent/child relationship, then all the members of the parent class (A) can be consumed under the child class (B).



Implement Inheritance

The syntax

```
[<modifiers>] class <child class> : <parent class>
```

Parent class and Child Class can also be called Base Class (Superclass) and Derived Class (Subclass)

- A => Parent/ Base/ Superclass
- B => Child/ Derived/ Subclass

```
class A
{
        //Members
}
Class B : A  ➡  Establishing Parent/Child Relationship

{
        //Consuming the Members of A from here
}
```

Inheritance

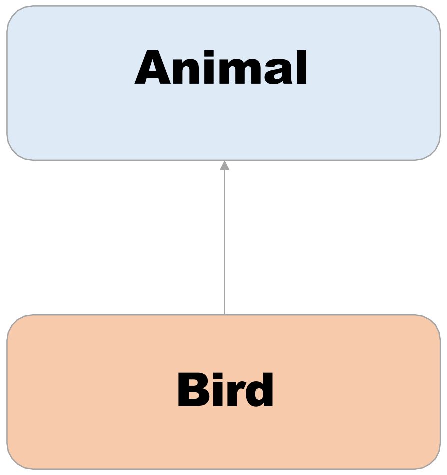
- Inheritance allows a software developer to derive a new class from an existing one.
- The existing class is called the parent, super, or base class.
- The derived class is called a child or subclass.
- The child inherits characteristics of the parent.
- Methods and data defined for the parent class.

Inheritance

- The child has special rights to the parents methods and data.
- Public access like any one else
- Protected access available only to child classes (and their descendants).
- The child has its own unique behaviors and data.
- Private members are not accessed in a derived class when one class is derived from another.

Inheritance

- Inheritance relationships are often shown graphically in a class diagram, with the arrow pointing to the parent class.
- Inheritance should create an is-a relationship, meaning the child is a more specific version of the parent.



Examples: Base Classes and Derived Classes

Base class	Derived classes
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
Account	CheckingAccount, SavingsAccount

Declaring a Derived Class

Define a new class DerivedClass which extends BaseClass

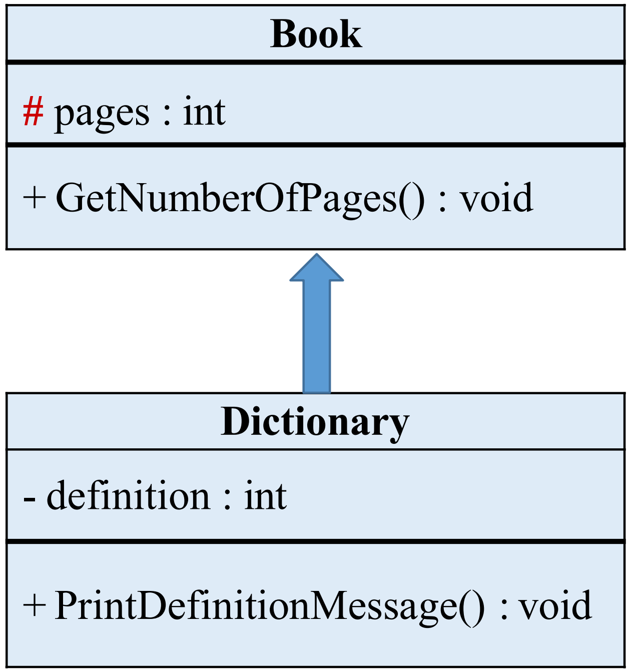
```
class BaseClass
    {
        //class contents
    }
    class DerivedClass : BaseClass
    {
        //class contents
    }
```

Controlling Inheritance

- A child class inherits the methods and data defined for the parent class; however, whether a data or method member of a parent class is accessible in the child class depends on the visibility modifier of a member.
- Variables and methods declared with **private** visibility are not accessible in the child class
- However, a private data member defined in the parent class is still part of the state of a derived class.
- Variables and methods declared with **public** visibility are accessible; but public variables violate our goal of encapsulation
- There is a third visibility modifier that helps in inheritance situations: **protected**.

The Protected Modifier

Variables and methods declared with protected visibility in a parent class are only accessible by a child class or any class derived from that class


Book
# pages : int
+ GetNumberOfPages() : void
Dictionary
- definition : int
+ PrintDefinitionMessage() : void

+ public

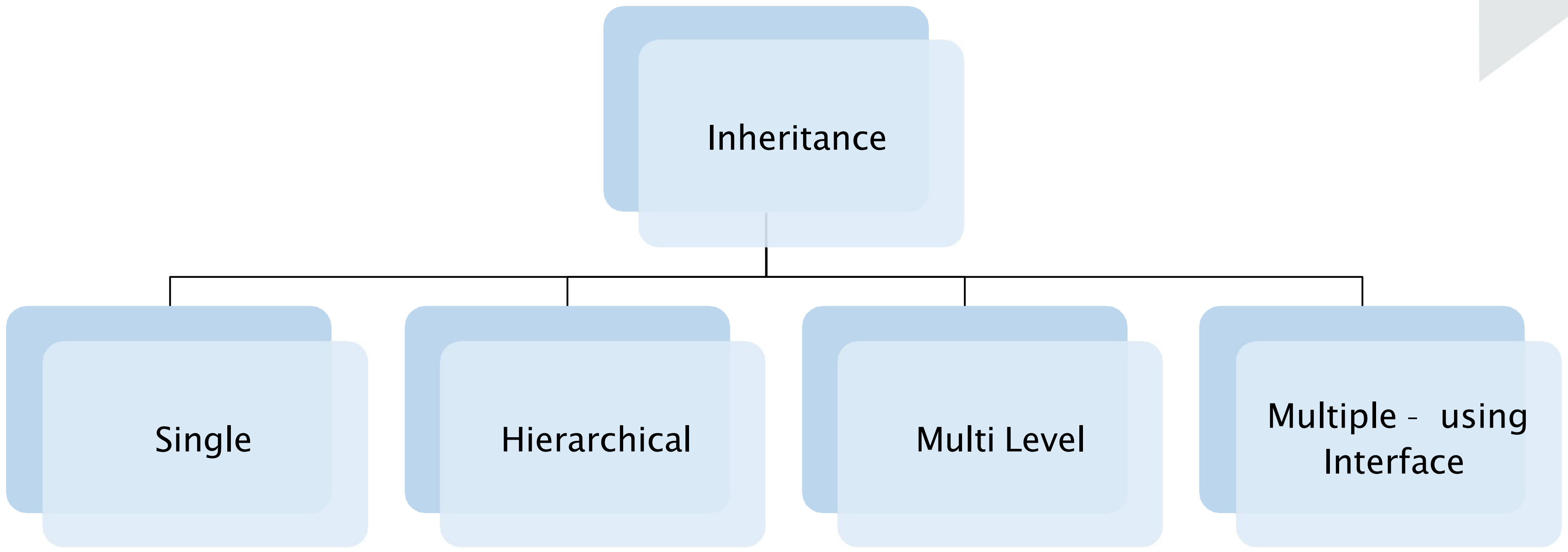
- private

# protected

Inheritance Example

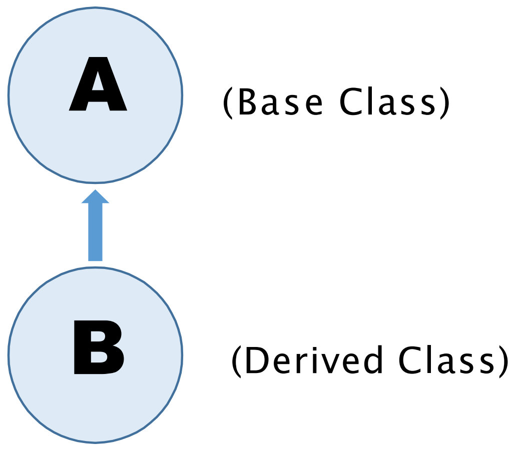
```
using System;
namespace InheritanceDemo
{
    class A
    {
        public void Method1()
        {
            Console.WriteLine("Method 1");
        }
        public void Method2()
        {
            Console.WriteLine("Method 2");
        }
    }
    class B : A
    {
        public void Method3()
        {
            Console.WriteLine("Method 3");
        }
        static void Main()
        {
            B obj = new B();
            obj.Method1();
            obj.Method2();
            obj.Method3();
            Console.ReadKey();
        }
    }
}
```

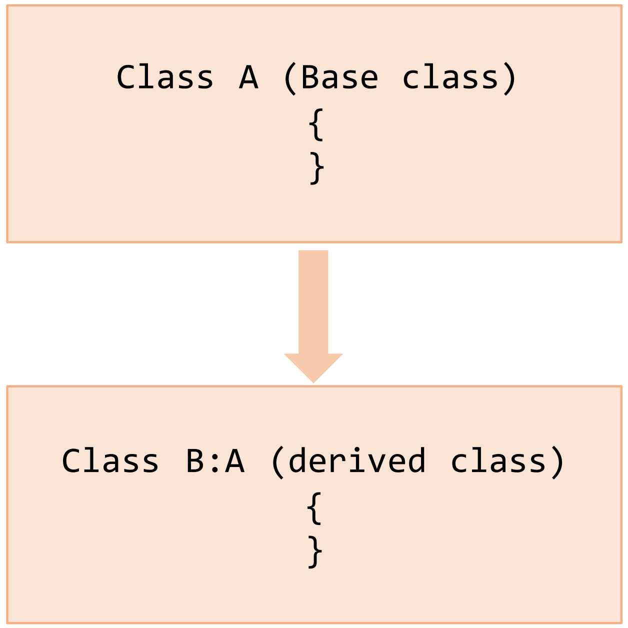
Types of Inheritance



Single Inheritance

When a class is inherited from a single base class then the inheritance is called single inheritance.





```
Class A (Base class)
        {
        }

Class B:A (derived class)
        {
        }
```

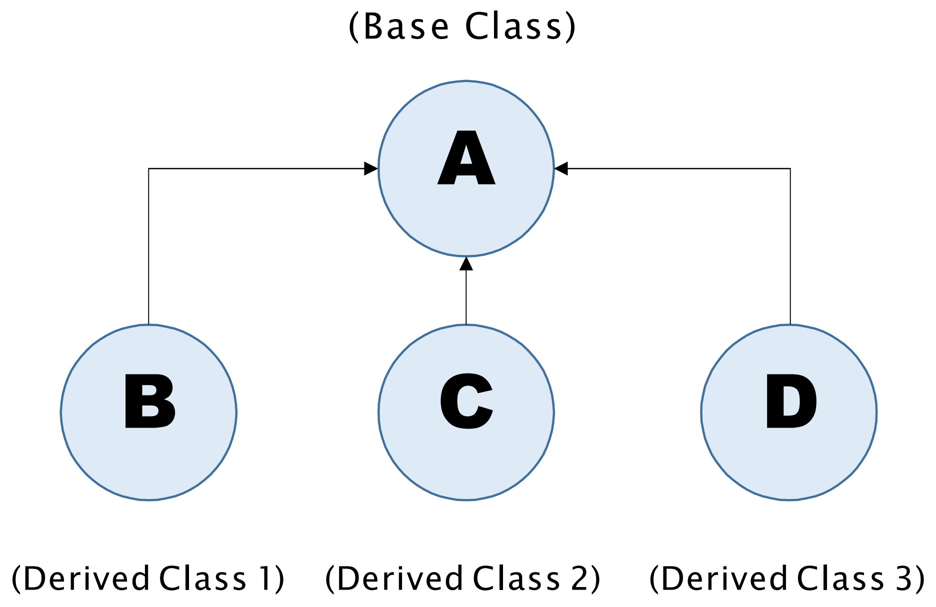
Single Inheritance Example

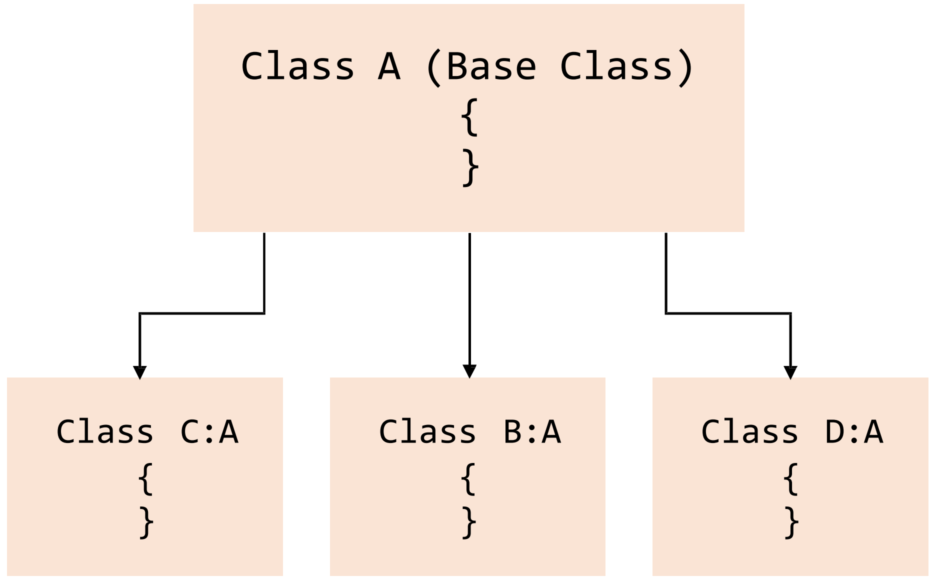
```
public class Accountcreditinfo //base class
{
    public string Credit()
    {
        return "balance is credited";
    }
}
public class debitinfo : Accountcreditinfo //derived class
{
    public string debit()
    {
        Credit();                       ////derived class method
        return "balance is debited";
    }
}
```

Single Inheritance – Accountcreditinfo is the base class, and debitinfo is the derived class

Hierarchical Inheritance

When more than one derived class is created from a single base class then it is called Hierarchical inheritance.





```
Class A (Base Class)
{
}
```

```
Class C:A
{
}
```

```
Class B:A
{
}
```

```
Class D:A
{
}
```

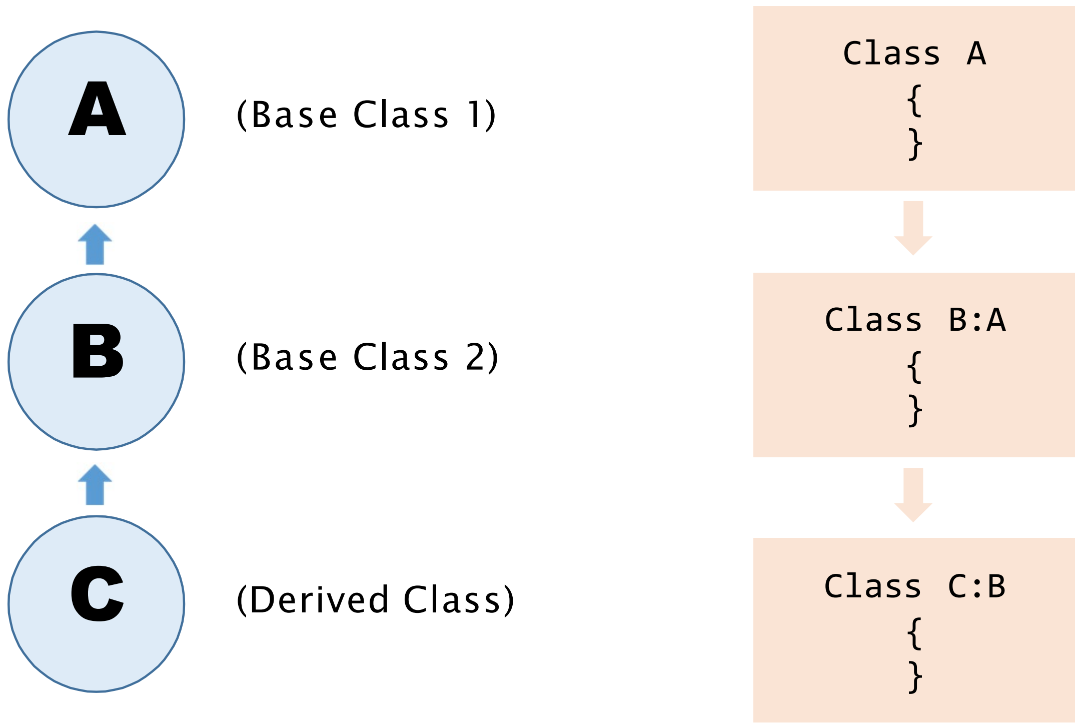
Hierarchical Inheritance Example

```
class A  //base class
{
    public string msg()
    {
        return "this is A class Method";
    }
}
class B : A
{
    public string info()
    {
        msg();
        return "this is B class Method";
    }
}
class C : A
{
    public string getinfo()
    {
        msg();
        return "this is C class Method";
    }
}
```

Hierarchical Inheritance – one base class is derived from many classes

Multilevel Inheritance

When a derived class is created from another derived class, then such a type of inheritance is called Multilevel Inheritance.



```
Class A
{
}
```

```
Class B:A
{
}
```

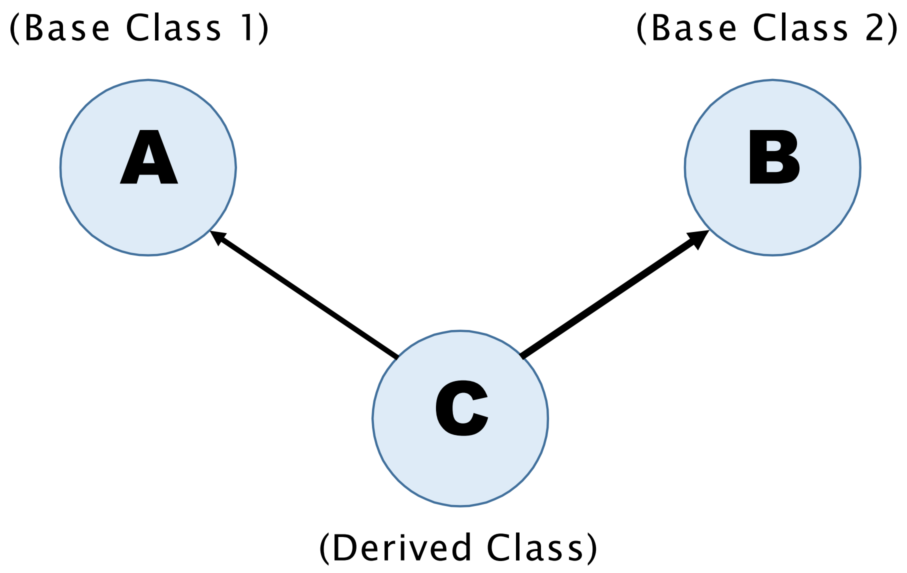
```
Class C:B
{
}
```

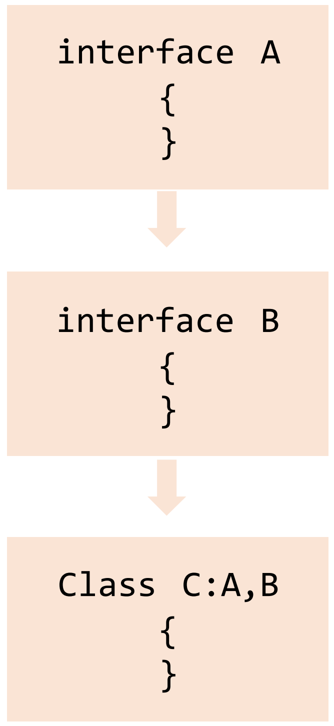
Multilevel Inheritance Example

```
public class Person
{
    public string persondet()
    {
        return "this is the person class";
    }
}
public class Bird : Person
{
    public string birddet()
    {
        persondet();
        return "this is the  birddet Class";
    }
}
public class Animal : Bird
{
    public string animaldet()
    {
        persondet();
        birddet();
        return "this is the Animal Class";
    }
}
```

Multiple Inheritance

- When a derived class is created from more than one base class then such type of inheritance is called multiple inheritances
- C# does not support multiple inheritances of classes.
- Use interface to support multiple inheritance.





```
interface A
    {
    }
```

```
interface B
    {
    }
```

```
Class C:A,B
    {
    }
```

Multiple Inheritance Example

```
public interface IA //ineterface  1
{
    string setImgs(string a);
}
public interface IB  //Interface 2
{
    int getAmount(int Amt);
}
public class ICar : IA, IB //implementatin
{
    public int getAmount(int Amt)
    {
        return 100;
    }
    public string setImgs(string a)
    {
        return "this is the car";
    }
}
```

iamneo

Polymorphism

Polymorphism

- Polymorphism is one of the primary pillars of object-oriented programming
- Polymorphism is one of the fundamental OOPs concepts
- It used to describe situations where something takes various roles or forms. In the programming world, these things can be operators or functions.
- The word polymorphism is derived from two Greek words: poly and morphs.
- The word “Poly” means many and “morphs” means forms.
- Therefore, polymorphism means “many forms” or we can say that the word polymorphism means the ability to take more than one form. That is one thing that can take many forms.

Polymorphism

- Polymorphism is a concept by which we can perform a single task in different ways.
- That is, when a single entity behaves differently in different cases, it is called polymorphism in C#.
- The term polymorphism is an object-oriented programming term that means a function, or an operator behaves differently in different scenarios.

Polymorphism

Different Behaviors:

1. In Shopping Mall, Behave like a Customer
2. In Class Room, Behave like a student
3. In Bus, Behave like a Passenger
4. In Home, Behave like a Son or Daughter

Polymorphism

There are two types of polymorphism in C#:

- ➢ Static Polymorphism / Compile-Time Polymorphism / Early Binding
- ➢ Dynamic Polymorphism / Run-Time Polymorphism / Late Binding

Types of Polymorphism in C#

Static Polymorphism /
Compile - Time Polymorphism /
Early Binding

Dynamic Polymorphism /
Run - Time Polymorphism /
Late Binding

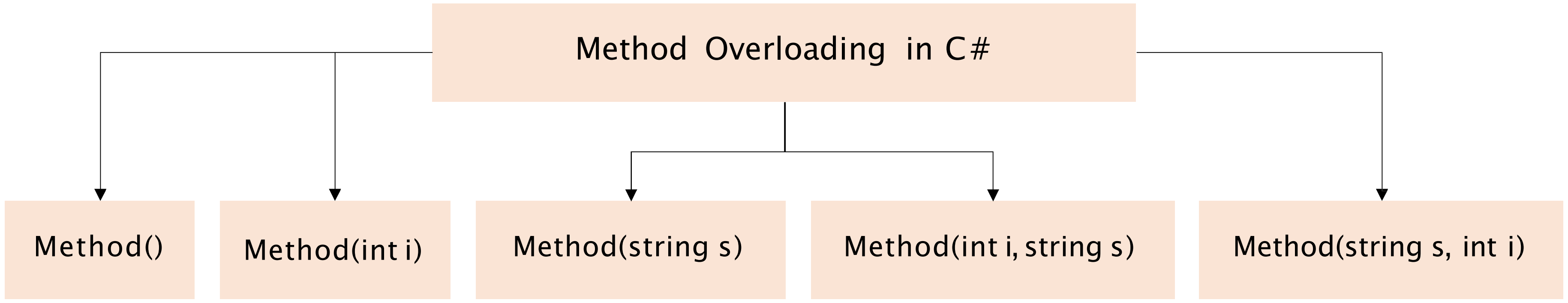
Examples :

- Method Overloading
- Operator Overloading
- Method Hiding

Example :
Method Overriding

Method Overloading

- Method Overloading means it is an approach to defining multiple methods under the class with a single name.
- Define more than one method with the same name in a class.
- The parameters of all those methods should be different (different in terms of number, type, and order of the parameters).



- Method Name is going to be the same
- Number, type and Order of Parameters is going to be different

Method Overloading Example

```
class Program
    {
        static void Main(string[] args)
        {
            Program obj = new Program();
            obj.Method(); //Invoke the 1st Method
            obj.Method(10); //Invoke the 2nd Method
            obj.Method("Hello"); //Invoke the 3rd Method
            obj.Method(10, "Hello"); //Invoke the 4th Method
            obj.Method("Hello", 10); //Invoke the 5th Method

            Console.ReadKey();
        }
```

Method Overloading Example

```
public void Method()
        {
            Console.WriteLine("1st Method");
        }
        public void Method(int i)
        {
            Console.WriteLine("2nd Method");
        }
        public void Method(string s)
        {
            Console.WriteLine("3rd Method");
        }
        public void Method(int i, string s)
        {
            Console.WriteLine("4th Method");
        }
        public void Method(string s, int i)
        {
            Console.WriteLine("5th Method");
        }
    }
```

Operator Overloading

- In C#, it is possible to make operators work with user-defined data types like classes.
- That means C# has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.
- For example, we can overload the + operator in a class like String so that we can concatenate two strings by just using +.
- Using operator overloading in C# we can specify more than one meaning for an operator in one scope.
- The purpose of operator overloading is to provide a special meaning of an operator for a user-defined data type.

Operator Overloading

- The return type is the return type of the function.
- the operator is a keyword.
- Op is the symbol of the operator that we want to overload.
- Like: +, <, −, ++, etc.
- The type must be a class or struct. It can also have more parameters.
- It should be a static function.

```
public static return_type operators op (Type t)
{
        //Statements
}
Where Type must be a class or struct.
```

Operator Overloading

Operators	Description
`+, -, !, ~, ++, --`	unary operators take one operand and can be overloaded.
`+, -, *, /, %`	Binary operators take two operands and can be overloaded.
`==, !=, =`	Comparison operators can be overloaded.
`&&, ||`	Conditional logical operators cannot be overloaded directly.
`+=, -+, *=, /=, %=, =`	Assignment operators cannot be overloaded.

Add Function

```
public static Complex Add(Complex cl, Complex c2)
{
    Complex temp = new Complex();

    temp.real = cl.real + c2.real;

    temp.img = cl.img + c2.img;

    return temp;
}
```

+ Operator Overloading

```
public static Complex operator +(Complex cl, Complex c2)
{
    Complex temp = new Complex();

    temp.real = cl.real + c2.real;

    temp.img = cl.img + c2.img;

    return temp;
}
```

Method Overriding

- The process of re-implementing the superclass non-static, non-private, and non-sealed method in the subclass with the same signature is called Method Overriding in C#.
- The same signature means the name and the parameters (type, number, and order of the parameters) should be the same.
- If the Super Class or Parent Class method logic is not fulfilling the Sub Class or Child Class business requirements, then the Sub Class or Child Class needs to override the superclass method with the required business logic.
- Usually, in most real-time applications, the Parent Class methods are implemented with generic logic which is common for all the next-level sub-classes.
- If a method in the sub-class or child class contains the same signature as the superclass non-private, non-static, and non-sealed method, then the subclass method is treated as the overriding method and the superclass method is treated as the overridden method.

Method Overriding

```
class Class1
 {
     //Virtual Function (Overridable Method)
     public virtual void Show()
     {
         //Parent Class Logic Same for All Child Classes
         Console.WriteLine("Parent Class Show Method");
     }
 }
 class Class2 : Class1
 {
     //Overriding Method
     public override void Show()
     {
         //Child Class Reimplementing the Logic
         Console.WriteLine("Child Class Show Method");
     }
 }
 class Program
 {
     static void Main(string[] args)
     {
         Class1 obj1 = new Class2();
         obj1.Show();

         Class2 obj2 = new Class2();
         obj2.Show();
         Console.ReadKey();
     }
 }
```

Method Hiding

- Method Overriding is an approach of re-implementing the parent class methods under the child class exactly with the same signature (same name and same parameters).

- Method Hiding/Shadowing is also an approach of re-implementing the parent class methods under the child class exactly with the same signature (same name and same parameters).

Method Hiding

When we use the new keyword to hide a Parent Class Methods under the child class, then it is called Method Hiding/Shadowing in C#.

```
public class Parent
{
    public virtual void Show() { }
    public void Display() { }
}

public class Child : Parent
{
    // Method Hiding using new keyword
    public new void Show() { }

    // Method Hiding without using new keyword
    // Getting one warning
    public void Display() { }
}
```

Method Hiding

```
public class Parent
    {
        public virtual void Show()
        {
            Console.WriteLine("Parent Class Show Method");
        }
        public void Display()
        {
            Console.WriteLine("Parent Class Display Method");
        }
    }
```

Method Hiding

```
public class Child : Parent
    {
        //Method Overriding
        public override void Show()
        {
            Console.WriteLine("Child Class Show Method");
        }
        //Method Hiding/Shadowing
        public new void Display()
        {
            Console.WriteLine("Child Class Display Method");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Child obj = new Child();
            obj.Show();
            obj.Display();
            Console.ReadKey();
        }
    }
```

Thank You