C# Inheritance

In this tutorial, we will learn about C# inheritance and its types with the help of examples.

In C#, inheritance allows us to create a new class from an existing class. It is a key feature of Object-Oriented Programming (OOP).

The class from which a new class is created is known as the base class (parent or superclass). And, the new class is called derived class (child or subclass)

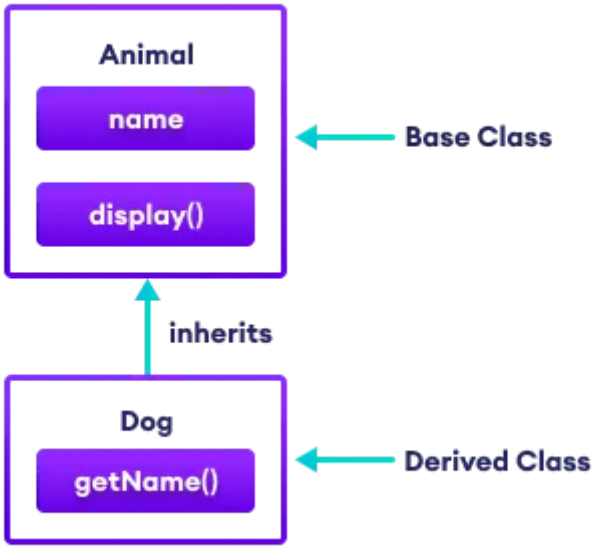
The derived class inherits the fields and methods of the base class. This helps with the code reusability in C#.

How to perform inheritance in C#?

In C#, we use the `:` symbol to perform inheritance. For example,

```
class Animal {
  // fields and methods
}

// Dog inherits from Animal
class Dog : Animal {
  // fields and methods of Animal
  // fields and methods of Dog
}
```



C# Inheritance

Example: C# Inheritance

```
// base class
class Animal {

  public string name;

  public void display() {
    Console.WriteLine("I am an animal");
  }

}

// derived class of Animal
class Dog : Animal {

  public void getName() {
    Console.WriteLine("My name is " + name);
  }
}

class Program {

  static void Main(string[] args) {

    // object of derived class
```

Output

```
I am an animal
My name is Rohu
```

In the above example, we have derived a subclass `Dog` from the superclass `Animal`. Notice the statements,

```
labrador.name = "Rohu";

labrador.getName();
```

Also, we have accessed the `name` field inside the method of the `Dog` class.

---

is-a relationship

In C#, inheritance is an is-a relationship. We use inheritance only if there is an is-a relationship between two classes. For example,

- **Dog** is an **Animal**
- **Apple** is a **Fruit**
- **Car** is a **Vehicle**

We can derive **Dog** from **Animal** class. Similarly, **Apple** from **Fruit** class and **Car** from **Vehicle** class.

---

protected Members in C# Inheritance

When we declare a field or method as `protected`, it can only be accessed from the same class and its derived classes.

Example: protected Members in Inheritance

```
  // base class
  class Animal {
    protected void eat() {
      Console.WriteLine("I can eat");
    }
  }

  // derived class of Animal
  class Dog : Animal {
     static void Main(string[] args) {

      Dog labrador = new Dog();

      // access protected method from base class
      labrador.eat();

      Console.ReadLine();
    }
  }
}
```

Output

```
I can eat
```

In the above example, we have created a class named `Animal`. The class includes a protected method `eat()`.

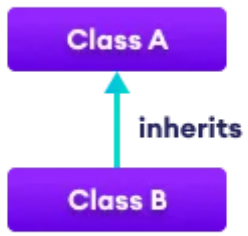
We have derived the `Dog` class from the `Animal` class. Notice the statement,

```
labrador.eat();
```

Since the `protected` method can be accessed from derived classes, we are able to access the `eat()` method from the `Dog` class.

1. Single Inheritance

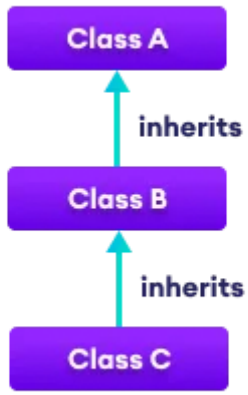
In single inheritance, a single derived class inherits from a single base class.



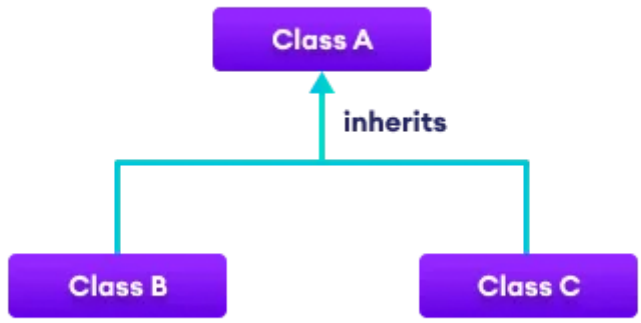
C# Single Inheritance

2. Multilevel Inheritance

In multilevel inheritance, a derived class inherits from a base and then the same derived class acts as a base class for another class.



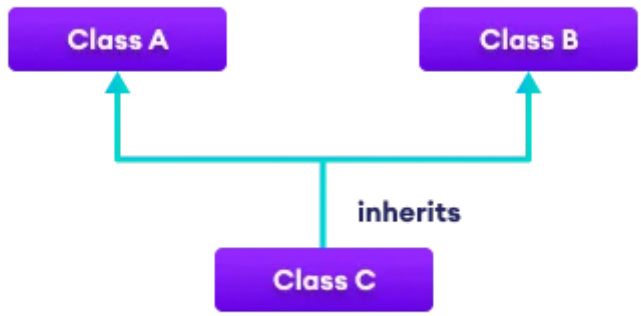
C# Multilevel Inheritance



C# Hierarchical Inheritance

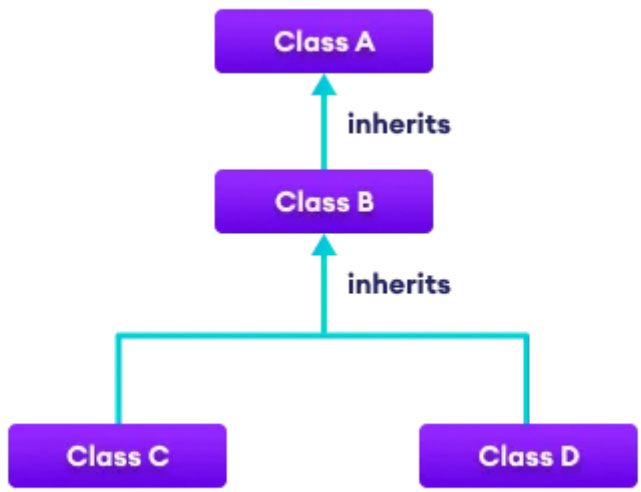
4. Multiple Inheritance

In multiple inheritance, a single derived class inherits from multiple base classes. **C# doesn't support multiple inheritance.** However, we can achieve multiple inheritance through interfaces.



Multiple Inheritance

5. Hybrid Inheritance



C# Hybrid Inheritance

Method Overriding in C# Inheritance

If the same method is present in both the base class and the derived class, the method in the derived class overrides the method in the base class. This is called method overriding in C#. For example,

```
// base class
class Animal {
  public virtual void eat() {

    Console.WriteLine("I eat food");
  }
}

// derived class of Animal
class Dog : Animal {

  // overriding method from Animal
  public override void eat() {

    Console.WriteLine("I eat Dog food");
  }
}
class Program {

  static void Main(string[] args) {
    // object of derived class
    Dog labrador = new Dog();
```

Output

```
I eat Dog food
```

In the above example, the `eat()` method is present in both the base class and derived class.

When we call `eat()` using the `Dog` object `labrador`,

```
labrador.eat();
```

the method inside `Dog` is called. This is because the method inside `Dog` overrides the same method inside `Animal`.

`virtual` - allows the method to be overridden by the derived class

- `override` - indicates the method is overriding the method from the base class

---

base Keyword in C# Inheritance

In the previous example, we saw that the method in the derived class overrides the method in the base class.

However, what if we want to call the method of the base class as well?

In that case, we use the `base` keyword to call the method of the base class from the derived class.

Example: base keyword in C# inheritance

```
// base class
class Animal {
  public virtual void eat() {

    Console.WriteLine("Animals eat food.");
  }
}

// derived class of Animal
class Dog : Animal {

  // overriding method from Animal
  public override void eat() {

    // call method from Animal class
    base.eat();

    Console.WriteLine("Dogs eat Dog food.");
  }
}
class Program {

  static void Main(string[] args) {
```

Output

```
Animals eat food.
Dogs eat Dog food.
```

In the above example, the `eat()` method is present in both the base class `Animal` and the derived class `Dog`. Notice the statement,

```
base.eat();
```

Here, we have used the `base` keyword to access the method of `Animal` class from the `Dog` class.

consider a situation.

Suppose we are working with regular polygons such as squares, rectangles, and so on. And, we have to find the perimeter of these polygons based on the input.

1. Since the formula to calculate perimeter is common for all regular polygons, we can create a `RegularPolygon` class and a method `calculatePerimeter()` to calculate perimeter.

```
class RegularPolygon {

  calculatePerimeter() {
    // code to compute perimeter
  }
}
```

2. And inherit `Square` and `Rectangle` classes from the `RegularPolygon` class. Each of these classes will have properties to store the length and number of sides because they are different for all polygons.

```
class Square : RegularPolygon {

  int  length = 0;
  int sides = 0;
}
```

We pass the value of the `length` and `sides` to `calculateperimeter()` to compute the perimeter.

This is how inheritance makes our code reusable and more intuitive.

```
namespace Inheritance {

  class RegularPolygon {

      public void calculatePerimeter(int length, int si

        int result = length * sides;
        Console.WriteLine("Perimeter: " + result);
      }
  }

  class Square : RegularPolygon {

    public int  length = 200;
    public int sides = 4;
    public void calculateArea() {

      int area = length * length;
      Console.WriteLine("Area of Square: " + area);
    }
  }

 class Rectangle : RegularPolygon {

    public int length = 100;
```

Output

```
Area of Square: 40000
Perimeter: 800
Area of Rectangle: 20000
Perimeter: 400
```

In the above example, we have created a `RegularPolygon` class that has a method to calculate the perimeter of the regular polygon.

Here, the `Square` and `Rectangle` inherit from `RegularPolygon`.

And since the formula to calculate the area is different for different shapes, we have created a separate method inside the derived class to calculate the area.

Next Tutorial:
C# Abstract Class & Methods (/csharp-programming/abstract-class)

Previous Tutorial:
C# Strings (/csharp-programming/string)

Related Tutorials

C# Tutorial

C# abstract class and method

C# sealed class and method

(/csharp-programming/sealed-class)

C# Tutorial

C# interface

(/csharp-programming/interface)

C# Tutorial

C# Polymorphism

(/csharp-programming/polymorphism)