Inheritance in C#

Inheritance in C# is a fundamental concept in object-oriented programming (OOP) that allows a class to inherit properties and behaviors (fields, methods) from another class. This promotes code reusability, better organization, and structure.

1. Definition of Inheritance

Inheritance allows you to create a new class (derived class) based on an existing class (base class). The derived class inherits all the non-private members (fields, properties, methods) of the base class, meaning it can use those members as if they were defined in the derived class itself.

```
// Base class representing a general bank account

abstract class BankAccount

{

   public double balance; // Field that holds the balance

   public void Deposit(double amount)

   {

      balance += amount;

      Console.WriteLine($"{amount:C} deposited. Current balance: {balance:C}");

   }

}


// Derived class representing a savings account

class SavingsAccount : BankAccount

{


}


// Derived class representing a current account

class CurrentAccount : BankAccount

{

   private double overdraftLimit;

}
```

2. Types of Inheritance in C#

C# supports the following types of inheritance:

- **Single Inheritance**: One class derives from a single base class.
  - Example: Dog inherits from Animal.
- **Multilevel Inheritance**: A class derives from a derived class, creating a chain.

```
class Animal { }

class Dog : Animal { }

class Bulldog : Dog { } // Bulldog derives from Dog, which derives from Animal
```

- **Hierarchical Inheritance**: Multiple classes derive from a single base class.

```
class Animal { }

class Dog : Animal { }

class Cat : Animal { }
```

C# does **not support multiple inheritance** (one class deriving from more than one base class) to avoid ambiguity. However, multiple inheritance is achieved through interfaces.

---

3. Access modifiers control which class members are inherited:

- **public**: Members are accessible in derived classes.
- **protected**: Members are accessible in derived classes, but not outside the class hierarchy.
- **private**: Members are not inherited by the derived class.
- **internal**: Members are accessible only within the same assembly.

```
class Animal

{

  private int age; // Not inherited

  protected string name; // Inherited, but not accessible outside the class hierarchy

  public void Communication() { Console.WriteLine("Comm..."); } // Inherited and accessible

}
```

4. base Keyword

The base keyword is used in a derived class to:

- Access members of the base class that are hidden by new members in the derived class.
- Call the base class constructor.

```
class BankAccount

{

    Protected string accountNumber;

    protected double balance; // Field that holds the balance

  // Constructor to initialize the account balance

     public BankAccount(string accountNumber ,double initialBalance)

     {

          This. accountNumber = accountNumber;

         balance = initialBalance;

     }

      public void ShowBalance()

      {

         Console.WriteLine($"Account {accountNumber} balance: {balance:C}");

      }

}

class SavingsAccount : BankAccount

{

   public SavingsAccount(double initialBalance) : base(initialBalance)

   {

      // Call the base class constructor to set the initial balance

   }

      public new void ShowBalance()

      {

      Console.WriteLine("This is derived class")

        base. ShowBalance();     }   }
```

6. **Method Overriding and virtual/override Keywords**

When a method in the base class is marked with the virtual keyword, it can be overridden in the derived class using the override keyword. This allows the derived class to provide its own implementation of the method.

```
class Animal
{
  public virtual void Speak()
  {
    Console.WriteLine("Animal is speaking...");
  }
}
class Dog : Animal
{
  public override void Speak()
  {
    Console.WriteLine("Dog is barking...");
  }
}
```

----

7. **Sealing Classes and Methods**

Sealed Classes: A class can be marked as sealed, preventing it from being inherited.

```
sealed class Cat { }
// class Lion : Cat { } // Error: Cannot inherit from sealed class
```

Sealed Methods: You can also seal methods in derived classes to prevent further overriding.

```
class Dog : Animal
{
  public sealed override void Speak()
  {
    Console.WriteLine("Dog is barking...");
  } }
```

```
class Bulldog : Dog
{
    // public override void Speak() { } // Error: Cannot override sealed method
}
```

---

8. Hiding Base Class Members

Sometimes, you may want to provide a new implementation in the derived class but not override the base class method. In this case, you can use the new keyword to hide the base class member.

```
class Animal
{
    public void Speak()
    {
        Console.WriteLine("Animal is speaking...");
    }
}

class Dog : Animal
{
    public new void Speak()
    {
        Console.WriteLine("Dog is barking...");
    }
}
```

---

9. Constructor Inheritance

When a derived class object is created, the base class constructor is called first, followed by the derived class constructor. If the base class has a parameterized constructor, the derived class must use the base keyword to pass arguments to it.

```
class Animal
{
  public Animal(string name)
  {
    Console.WriteLine("Animal constructor called with name: " + name);
  }
}
class Dog : Animal
{
  public Dog(string name) : base(name)
  {
    Console.WriteLine("Dog constructor called.");
  }
}
```

----

10. Abstract Classes and Inheritance

An **abstract class** can provide some method implementations and force derived classes to implement specific methods using **abstract methods**. Abstract methods do not have an implementation in the base class and must be implemented in derived classes.

```
abstract class Animal
{
  public abstract void Speak(); // No implementation
}
class Dog : Animal
{
  public override void Speak()
  {
    Console.WriteLine("Dog is barking...");
  }
}
```

Full Example:

```
using System;

namespace BankSystem
{
    // Abstract class representing a general bank account
    abstract class BankAccount
    {
        // Encapsulated fields with private access
        private string accountNumber;
        protected double balance=0;

        // Constructor to initialize bank account
        public BankAccount(string accountNumber, double initialBalance)
        {
            this.accountNumber = accountNumber;
            balance += initialBalance;
        }

        // Abstract method to be implemented by derived classes
        public abstract void Withdraw(double amount);

        // Non-abstract method with implementation
        public void Deposit(double amount)
        {
            balance += amount;
            Console.WriteLine($"{amount} deposited successfully. New balance: {balance:C}");
        }
        // Virtual method that can be overridden in derived classes
        public virtual void ShowBalance()
        {
            Console.WriteLine($"Account {accountNumber} balance: {balance:C}");
        }
    }

    // Derived class representing a Savings Account
    class SavingsAccount : BankAccount
    {
        private const double InterestRate = 0.03; // 3% interest rate

        public SavingsAccount(string accountNumber, double initialBalance)
            : base(accountNumber, initialBalance)
        {
        }
        // Override the abstract method for withdrawing money
        public override void Withdraw(double amount)
        {
            if (amount > balance)
            {
                Console.WriteLine("Insufficient funds. Withdrawal failed.");
            }
            else
            {
                balance -= amount;
                Console.WriteLine($"{amount:C} withdrawn successfully. New balance: {balance:C}");
            }
        }
        // Additional method specific to SavingsAccount
        public void ApplyInterest()
        {
            double interest = balance * InterestRate;
            balance += interest;
            Console.WriteLine($"Interest applied: {interest:C}. New balance: {balance:C}");
        }

        // Optionally override the virtual method
        public override void ShowBalance()
        {
            base.ShowBalance();
            Console.WriteLine($"(Savings account - includes interest)");
        }
    }

    // Derived class representing a Current Account
    class CurrentAccount : BankAccount
    {
        private double overdraftLimit;

        public CurrentAccount(string accountNumber, double initialBalance, double overdraftLimit)
            : base(accountNumber, initialBalance)
        {
            this.overdraftLimit = overdraftLimit;
        }

        // Override the abstract method for withdrawing money
        public override void Withdraw(double amount)
        {
            if (amount > balance + overdraftLimit)
            {
                Console.WriteLine("Withdrawal exceeds overdraft limit. Withdrawal failed.");
            }
            else
            {
                balance -= amount;
                Console.WriteLine($"{amount:C} withdrawn successfully. New balance: {balance:C}");
            }
        }

        // Optionally override the virtual method
        public override void ShowBalance()
        {
            base.ShowBalance();
            Console.WriteLine($"(Current account - Overdraft limit: {overdraftLimit:C})");
        }
    }

    // Bank system for testing accounts
    class Program
    {
        static void Main(string[] args)
        {
            // Create a Savings Account
            BankAccount savings = new SavingsAccount("SAV123", 1000);
            savings.ShowBalance();
```

```
            savings.Deposit(500);
            savings.Withdraw(200);
            ((SavingsAccount)savings).ApplyInterest();
            savings.ShowBalance();

            Console.WriteLine();

            // Create a Current Account
            BankAccount current = new CurrentAccount("CUR456", 2000, 500);
            current.ShowBalance();
            current.Deposit(300);
            current.Withdraw(2500);
            current.ShowBalance();
        }
    }
}
```