Polymorphism in C#

Polymorphism is one of the key concepts of **Object-Oriented Programming (OOP)** in C#. It allows objects of different types to be treated as objects of a common base type. The word polymorphism means "many forms," and in programming, it refers to the ability to present the same interface for different underlying forms (data types).

In C#, polymorphism is mainly achieved through **method overriding**, **method overloading**, and **interfaces**. There are two types of polymorphism:

1. **Compile-time Polymorphism (Static Binding)**
2. **Run-time Polymorphism (Dynamic Binding)**

1. Compile-time Polymorphism (Static Binding)

Compile-time polymorphism occurs when the method that needs to be invoked is determined during compilation. It is achieved through:

- **Method overloading**
- **Operator overloading**

a) Method Overloading

Method overloading occurs when multiple methods have the same name but different signatures (number of parameters or types of parameters). The appropriate method is chosen at compile time based on the arguments passed.

```
class BankAccount
{
  public double balance;
  public void Deposit(double amount)
  {
    balance += amount;
    Console.WriteLine($"{amount:C} cash deposited. New balance: {balance:C}");
  }
```

```
    // Overloaded method to deposit a check with reference number
    public void Deposit(double amount, string checkNumber)
    {
        balance += amount;
        Console.WriteLine($"{amount:C} deposited by check {checkNumber}. New balance:
        {balance:C}");
    }
}

class Program
{
        static void Main()
    {
            BankAccount account = new BankAccount();
            account.Deposit(500); // Calls the cash deposit method
            account.Deposit(1000, "CHK12345"); // Calls the check deposit method
    }
}
```

Key Points of Method Overloading:

- The Deposit method is overloaded, meaning the same method name is used but with different parameters (number or type).
- The appropriate method is chosen at compile time based on the arguments passed.

---

b) Operator Overloading

Operator overloading allows you to redefine how operators like +, -, *, etc., work for user-defined types (like classes or structs).

Although operator overloading is less common in a bank system, it can be useful in certain scenarios, such as merging two bank accounts by adding their balances.

```
class BankAccount
{
    public double balance;
    public BankAccount(double balance)
    {
        this.balance = balance;
    }

    // Overloading the + operator to merge two bank accounts
    public static BankAccount operator + (BankAccount acc1, BankAccount acc2)
    {
        return new BankAccount(acc1.balance + acc2.balance);
    }
    public void ShowBalance()
    {
        Console.WriteLine($"Balance: {balance:C}");
    }
}

class Program
{
    static void Main()
    {
        BankAccount account1 = new BankAccount(1000);
        BankAccount account2 = new BankAccount(1500);
        BankAccount mergedAccount = account1 + account2;
        mergedAccount.ShowBalance(); // Output: Balance: $2,500.00
    }
}
```

Another Example

```
class MyDate
{
   public DateTime date;

   public MyDate(DateTime date)
   {
      this.date = date;
   }
   // Overload the + operator to add days to a date
   public static MyDate operator +  (MyDate myDate, int days)
   {
      return new MyDate(myDate.date.AddDays(days));
   }
   public void ShowDate()
   {
      Console.WriteLine($"Date: {date.ToShortDateString()}");
   }
}

class Program
{
   static void Main()
   {
      MyDate today = new MyDate(DateTime.Now);
      // Add 10 days to today's date using the overloaded + operator
      MyDate futureDate = today + 10;
      // Display the new date
      futureDate.ShowDate(); // Output: Date: (today's date + 10 days)   } }
```

2. Run-time Polymorphism (Dynamic Binding)

Run-time polymorphism occurs when the method that needs to be invoked is determined during runtime rather than compile time. It is achieved through:

- **Method overriding** using virtual and override keywords
- **Interfaces** and their implementations

Run-time polymorphism allows derived classes to provide specific implementations of methods that are defined in a base class.

a) Method Overriding

Method overriding occurs when a derived class has a method with the same name and signature as in the base class. The base class method must be marked with the virtual keyword, and the derived class overrides this method using the override keyword.

```
// Base class representing a general bank account
class BankAccount
{
   protected double balance;
   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }
   // Virtual method that can be overridden in derived classes
   public virtual void Withdraw(double amount)
   {
      balance -= amount;
      Console.WriteLine($"{amount:C} withdrawn. New balance: {balance:C}");
   }
   public void ShowBalance()
   { Console.WriteLine($"Balance: {balance:C}");  }
}
```

```
// Derived class representing a savings account with withdrawal limits
class SavingsAccount : BankAccount
{
    public SavingsAccount(double initialBalance) : base(initialBalance) { }

    // Override the Withdraw method to implement specific logic for savings accounts
    public override void Withdraw(double amount)
    {
        if (amount > balance)
        {
            Console.WriteLine("Insufficient funds in Savings Account.");
        }
        else
        {
            base.Withdraw(amount); // Use the base class Withdraw method
        }
    }
}

// Derived class representing a current account with overdraft limit
class CurrentAccount : BankAccount
{
    private double overdraftLimit;

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

```
    // Override the Withdraw method to allow overdraft
    public override void Withdraw(double amount)
    {
        if (amount > balance + overdraftLimit)
        {
            Console.WriteLine("Withdrawal exceeds overdraft limit.");
        }
        else
        {
            base.Withdraw(amount);
        }
    }
}

class Program
{
    static void Main()
    {
        BankAccount savings = new SavingsAccount(1000);
        BankAccount current = new CurrentAccount(1000, 500);

        savings.Withdraw(1500); // Output: Insufficient funds in Savings Account.
        current.Withdraw(1500); // Output: $1,500.00 withdrawn. New balance: $-500.00
    }
}
```
