OBJECT-ORIENTED PROGRAMMING IN C#

Understanding Encapsulation, Inheritance, Polymorphism, and Abstraction with Real-World Scenarios

Abstract

This article provides an in-depth exploration of Object-Oriented Programming (OOP) concepts in C#, focusing on Encapsulation, Inheritance, Polymorphism, and Abstraction. Through real-world examples and code samples, readers will gain a comprehensive understanding of these fundamental principles and their significance in C# programming. From encapsulating data within objects to leveraging inheritance for code reuse, and from implementing polymorphic behaviour to achieving abstraction for better code organization, this article covers essential topics that are essential for building scalable and maintainable applications in C#. Whether you're new to OOP or looking to deepen your understanding, this article aims to provide valuable insights into object-oriented design principles in the context of C# development.

Vikram

https://www.linkedin.com/in/4kvikram

Vikram
https://www.linkedin.com/in/4kvikram

CONTENTS

OBJECT-ORIENTED PROGRAMMING IN C#

Object oriented programming Concepts form the foundation of C# programming. Let's dive into these concepts by discussing each of the five principles: **Classes & Objects , Inheritance, Encapsulation, Abstraction, and Polymorphism.**

1. CLASSES & OBJECTS:

Classes and objects are the building blocks of OOP in C#.

- **Class**: A class is a blueprint or template for creating objects. It defines the structure, behaviour, and state of objects that can be created from it.
- **Object**: An object is an instance of a class, with its own unique set of property values. Objects interact with each other by invoking methods and accessing properties defined in their respective classes.

Understanding and applying these OOP concepts effectively will help you write maintainable, scalable, and reusable code in C#.

Let's dive into classes and objects with a detailed code example.

1. Define a class called "Person":

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

    public void SayHello()
    {
        Console.WriteLine("Hello, my name is " + Name + ".");
    }
}
```

In this class, we have defined two properties (Name and Age) and a method (SayHello()) that prints a greeting message.

1. Create objects (instances) of the Person class:

```
Person person1 = new Person
{
    Name = "Alice",
    Age = 25
};

Person person2 = new Person
{
    Name = "Bob",
    Age = 30
};
```

Here, we have created two objects (person1 and person2) of the Person class, setting their properties using the object initializer syntax.

1. Access and modify object properties and call methods:

```
Console.WriteLine("Person 1:");
person1.SayHello();
Console.WriteLine("Name: " + person1.Name + ", Age: " + person1.Age);

person1.Name = "Alex";
person1.Age = 28;

Console.WriteLine("Person 2:");
person2.SayHello();
Console.WriteLine("Name: " + person2.Name + ", Age: " + person2.Age);
```

This will output:

```
Person 1:
Hello, my name is Alice.
Name: Alice, Age: 25
Person 2:
Hello, my name is Bob.
Name: Bob, Age: 30
```

In this example, we have demonstrated how to create classes and objects in C#. The Person class defines the structure and behavior of objects, while the objects (person1 and person2) are instances of the Person class with their own property values. We can access and modify object properties and call methods specific to each object.

Classes and objects are the building blocks of object-oriented programming, enabling the creation of reusable code, encapsulation of data and functionality, and promoting code organization and maintainability.

QUESTIONS:

✓ **What is the difference between a class and an object in C#?**

A class in C# is a blueprint or template that defines the structure and behaviour of an object. It contains properties, methods, and other members that describe the characteristics and actions of the object. An object, on the other hand, is an instance of a class, with its own unique set of property values and state. Objects are created from classes and can interact with each other through method invocations and property access.

✓ **What is the purpose of constructors in C# classes?**

Constructors in C# classes are special methods that are used to initialize and set up the initial state of an object when it is created. They provide a way to assign initial values to an object's properties and perform any necessary setup tasks. Constructors have the same name as the class they belong to and do not have a return type.

✓ **What is the default constructor in C#?**

The default constructor in C# is a parameter less constructor that is automatically generated by the compiler for a class if no other constructors are defined. It initializes an object of the class with default (usually null or zero) values for its properties and members.

✓ **What is the 'this' keyword used for in C#?**

The 'this' keyword in C# is a reference to the instance of the class that is being accessed or modified within a member (such as a method or property). It can be used to access the class members or disambiguate between similar member names.

✓ **What is encapsulation in C#?**

Encapsulation in C# is the process of hiding the internal details and complexity of a class and exposing only the necessary functionality to the outside world. It is achieved by combining access modifiers (e.g., 'public', 'private', 'protected') with the principles of data hiding and abstraction. Encapsulation helps in maintaining data integrity, controlling access to class members, and promoting code organization and reusability.

✓ **What is the difference between 'public' and 'private' access modifiers in C#?**

In C#, the 'public' access modifier allows members (properties, methods, etc.) to be accessible from anywhere, while the 'private' access modifier restricts access to only within the class itself. 'Public' members are used for exposing functionality to other classes, while 'private' members are used for implementing internal logic and maintaining data integrity.

✓ **What is inheritance in C# classes?**

Inheritance in C# classes is a mechanism that allows a new class (derived class) to be created based on an existing class (base class) to reuse and extend the functionality of the base class. The derived class can access and override the members of the base class, enabling code reuse and promoting polymorphism.

✓ **What is abstraction in C#?**

Abstraction in C# is a programming concept that focuses on presenting only essential information to the user while hiding complex implementation details. It is achieved through abstract classes and interfaces, which define a contract that derived classes or implementing classes must follow. Abstraction helps in achieving better code organization, flexibility, and maintainability.

✓ **What is an abstract class in C#?**

An abstract class in C# is a class that cannot be instantiated directly and is intended to be inherited by other classes. It contains one or more abstract methods (declared without an implementation) that must be overridden by derived classes to provide the actual functionality. Abstract classes are used to define a common structure or behaviour for related classes.

✓ **What is an interface in C#?**

An interface in C# is a reference type that defines a contract, specifying a set of methods, properties, and events that a class must implement to adhere to the interface. Interfaces are used for achieving abstraction, promoting polymorphism, and enforcing a common behaviour among unrelated classes. Classes can implement multiple interfaces, allowing them to support multiple contracts or functionalities.

2. INHERITANCE:

Inheritance is the process by which a class derives properties and methods from another class, forming an "is-a" relationship. In C#, inheritance allows a class to inherit properties, methods, and other characteristics from a parent class. Key aspects of inheritance include:

- **Single Inheritance:** A class can inherit from only one direct parent class.
- **Multilevel Inheritance**: Inheritance can be chained, allowing a class to be derived from another class, which in turn is derived from another class.
- **Hierarchical Inheritance:** Multiple derived classes inherit from a single base class.
- **Multiple Inheritance (through interfaces):** A derived class can implement multiple interfaces, allowing it to inherit behaviour from multiple sources.
- **Hybrid Inheritance (combination of multiple inheritance types):** Combination of any of the above types of inheritance.

  Note: C# does not support multiple inheritance of classes, but it does support multiple inheritance through interfaces.
- **Overriding**: Inherited methods can be overridden in derived classes to provide new implementations.
- **Hiding**: When a derived class and its parent class have members with the same signature, the derived class member hides the parent class member.

Let's dive into inheritance with a detailed code example.

Consider a scenario where we have a base class called "Animal" and a derived class called "Dog". The Dog class inherits properties and behaviours from the Animal class.

❖ Define the base class "Animal":

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

    public virtual void MakeSound()
    {
        Console.WriteLine("The animal is making a sound.");
    }
}
```

In this base class, we have defined two properties (Name and Age) and a virtual method **MakeSound()** that will be overridden in the derived class.

1. Define the derived class "Dog" that inherits from the Animal class:

```
public class Dog : Animal
{
    public string Breed { get; set; }

    public override void MakeSound()
    {
        Console.WriteLine("The dog is barking.");
    }
}
```

In this derived class, we have added a new property called Breed and overridden the **MakeSound()** method from the Animal class.

Now, you can create an instance of the Dog class and use its properties and methods:

```
Dog myDog = new Dog
{
    Name = "Buddy",
    Age = 3,
    Breed = "Labrador Retriever"
};

Console.WriteLine("Dog Name: " + myDog.Name);
Console.WriteLine("Dog Age: " + myDog.Age);
Console.WriteLine("Dog Breed: " + myDog.Breed);

myDog.MakeSound();
```

This will output:

```
Dog Name: Buddy
Dog Age: 3
Dog Breed: Labrador Retriever
The dog is barking.
```

In this example, we have demonstrated how inheritance can be implemented in C# by creating a derived class (Dog) that inherits properties and behaviours from a base class (Animal). The derived class can add its own properties and override the methods from the base class to provide specific implementations.

Inheritance promotes code reusability and reduces code duplication by allowing you to create new classes based on existing ones, inheriting their properties and behaviours. This makes the code more modular, maintainable, and extensible.

QUESTIONS

✓ **What is inheritance in C#?**

Inheritance is a programming concept that allows a new class (derived class) to be created based on an existing class (base class) to reuse and extend the functionality of the base class.

✓ **What are the benefits of inheritance in C#?**

Inheritance promotes code reusability, reduces redundancy, simplifies code maintenance, and enables polymorphism.

✓ **What is a derived class in C#?**

A derived class in C# is a class that inherits properties and methods from a base class, extending or customizing the functionality of the base class.

✓ **What is a base class in C#?**

A base class in C# is a class from which other classes inherit properties and methods. It serves as the foundation for creating derived classes that extend or specialize its functionality.

✓ **What is the ':' operator used for in C#?**

The ':' operator in C# is used to specify the base class for a derived class, indicating inheritance.

✓ **What is the difference between inheritance and composition in C#?**

Inheritance establishes a direct relationship between a base class and a derived class, allowing the derived class to inherit and extend the base class's functionality. Composition, on the other hand, creates an association between classes through object composition, where one class holds references to other objects.

Example of Composition:

```
public class Engine
{
    public void Start()
    {
        Console.WriteLine("Engine started.");
    }
}

public class Car
{
    private Engine engine = new Engine();

    public void StartCar()
    {
        engine.Start();
        Console.WriteLine("Car started.");
    }
}
```

✓ **What is method hiding in C#?**

Method hiding in C# occurs when a derived class has a method with the same signature as a method in the base class, and the derived class does not override the base class method using the 'override' keyword. This can lead to ambiguity and is generally discouraged.

✓ **What is the purpose of the 'base' keyword in C#?**

The 'base' keyword in C# is used to access and invoke members of the base class within a derived class, allowing derived classes to extend or override the base class's functionality.

✓ **What is the 'override' keyword used for in C# inheritance?**

The 'override' keyword in C# is used in a derived class to replace or provide an alternative implementation of a method or property that exists in the base class.

✓ **What is the role of polymorphism in C# inheritance?**

Polymorphism in C# inheritance allows objects of different classes related by inheritance to be treated as objects of a common base class or interface. It promotes flexibility, extensibility, and code reusability by enabling a single method or variable to have multiple implementations or forms.

3. ENCAPSULATION:

Encapsulation is the practice of bundling data and methods that operate on that data into a single unit, called a class. It helps in hiding the internal complexity of the class and provides a public interface for interaction. Encapsulation has two main aspects:

- **Data Hiding:** It involves hiding the implementation details of the class and providing only the necessary information to the external world. This is achieved by using access modifiers like public, private, and protected.

- **Information Hiding:** It ensures that the internal workings of a class are not exposed to the outside world. By encapsulating data and methods, we can make changes to the internal implementation without affecting the code that uses the class.

In C#, encapsulation can be achieved by using access modifiers like public, private, and protected. Let's dive into encapsulation with a detailed code example.

Consider a simple class called "Person" that encapsulates the properties and methods related to a person's information:

```
public class Person
{
    private string firstName;
    private string lastName;
    private int age;

    // Constructor
    public Person(string firstName, string lastName, int age)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
        this.Age = age;
    }

    // Encapsulated properties
    public string FirstName
    {
        get { return this.firstName; }
        set { this.firstName = value; }
    }

    public string LastName
    {
        get { return this.lastName; }
        set { this.lastName = value; }
    }

    public int Age
    {
        get { return this.age; }
        set { this.age = value; }
    }

    // Public method to display person's information
    public void DisplayPersonInfo()
    {
        Console.WriteLine("First Name: " + this.FirstName);
        Console.WriteLine("Last Name: " + this.LastName);
        Console.WriteLine("Age: " + this.Age);
    }
}
```

- In this example, we have defined three private fields (firstName, lastName, and age) to store the person's information.
- To access and modify these fields, we have created public properties (FirstName, LastName, and Age).
- By using properties, we can control the access to these private fields, ensuring that the internal implementation details are hidden from the outside world.
- The constructor of the Person class is used to initialize the object's state when it is created. It takes the initial values for the firstName, lastName, and age and assigns them to the respective properties.
- The DisplayPersonInfo method is a public method that can be used to print the person's information. Since it is a public method, it can be accessed from outside the class.

By encapsulating the data and providing a public interface, we ensure that the internal implementation details of the Person class are hidden from the users of the class. They can only interact with the class through the provided properties and methods, promoting data integrity and reducing the chances of unintended modifications.

To use this encapsulated class, you can create an instance of the Person class and call its methods:

```
Person person = new Person("John", "Doe", 25);
person.DisplayPersonInfo();
```

This will output:

```
First Name: John
Last Name: Doe
Age: 25
```

QUESTIONS

✓ **What is encapsulation in C#?**

Encapsulation is the process of hiding the internal details of an object and exposing only the necessary functionality through a public interface.

✓ **What are the benefits of encapsulation in C#?**

Encapsulation promotes data protection, modular design, code reusability, and easier maintenance.

✓ **What are the access modifiers used in C# encapsulation?**

In C#, the access modifiers used for encapsulation are ‘public’, ‘private’, ‘protected’, and ‘internal’.

✓ **What is the purpose of the ‘private’ access modifier in C#?**

The ‘private’ access modifier is used to restrict access to a member (property, field, method, etc.) within the class itself, preventing direct access from outside the class.

✓ **What is the use of the ‘public’ access modifier in C#?**

The ‘public’ access modifier allows members to be accessed from anywhere, promoting the encapsulation principle of providing a public interface for the class.

✓ **What is the difference between ‘private’ and ‘protected’ access modifiers in C#?**

The ‘private’ access modifier restricts access to a member within the class itself, while the ‘protected’ access modifier allows access within the class and its derived classes (subclasses).

✓ **What is the purpose of the 'internal' access modifier in C#?**

The 'internal' access modifier allows members to be accessed from any class within the same assembly (DLL), promoting encapsulation by limiting access to the internal components of the assembly.

✓ **What is the use of properties in C# encapsulation?**

Properties in C# are used for encapsulation to control access to private fields, providing a public interface for getting and setting the field value, allowing validation or additional logic when accessing or modifying the field.

✓ **What is the difference between a field and a property in C#?**

A field is a direct data member of a class, while a property is an accessor to a private field, providing a controlled way to get and set the field value.

✓ **What is the purpose of the 'get' and 'set' accessors in C# properties?**

The 'get' accessor is used to retrieve the value of a property, while the 'set' accessor is used to set the value of a property, allowing validation or additional logic during the assignment process.

✓ **What is the role of the 'protected internal' access modifier in C#?**

The 'protected internal' access modifier allows access to members from within the same class, derived classes, and classes within the same assembly, providing a balance between encapsulation and inheritance.

4. ABSTRACTION:

Abstraction is a technique used to simplify complex systems by focusing on essential features and hiding implementation details. It helps in achieving modularity, maintainability, and extensibility in software development. Abstraction can be implemented in C# using:

- **Abstract Classes:** These are classes that cannot be instantiated and are used as a blueprint for derived classes to implement shared functionality. They provide a template for derived classes to follow.
- **Interfaces:** Interfaces define a contract that a class must implement, specifying a set of methods and properties. By implementing an interface, a class promises to provide the functionality described in the interface.

Let's dive into abstraction with a detailed code example.

Consider a scenario where we have a hierarchy of shapes, such as Rectangle, Circle, and Triangle. We want to create an abstract base class and an interface to define common properties and behaviours for these shapes.

❖ Define an abstract base class called "Shape":

```
public abstract class Shape
{
    public abstract double CalculateArea();
    public abstract double CalculatePerimeter();
}
```

In this abstract class, we have defined two abstract methods, CalculateArea() and CalculatePerimeter(). These methods will be implemented by the derived classes to provide specific calculations for each shape.

❖ Implement the abstract class for specific shapes:

Rectangle class:

```
public class Rectangle : Shape
{
    public double Length { get; set; }
    public double Breadth { get; set; }

    public override double CalculateArea()
    {
        return this.Length * this.Breadth;
    }

    public override double CalculatePerimeter()
    {
        return 2 * (this.Length + this.Breadth);
    }
}
```

Circle class:

```
public class Circle : Shape
{
    public double Radius { get; set; }

    public override double CalculateArea()
    {
        return Math.PI * Math.Pow(this.Radius, 2);
    }

    public override double CalculatePerimeter()
    {
        return 2 * Math.PI * this.Radius;
    }
}
```

Triangle class:

```
public class Triangle : Shape
{
    public double Base { get; set; }
    public double Height { get; set; }
    public double Side1 { get; set; }
    public double Side2 { get; set; }

    public override double CalculateArea()
    {
        return 0.5 * this.Base * this.Height;
    }

    public override double CalculatePerimeter()
    {
        return this.Base + this.Side1 + this.Side2;
    }
}
```

In these derived classes, we have implemented the abstract methods from the Shape class to provide specific calculations for each shape.

❖ Define an interface called "IDrawable" to represent the behavior of drawing shapes:

```
public interface IDrawable
{
    void Draw();
}
```

This interface defines a single method, Draw(), that will be implemented by the classes that implement the interface.

❖ Implement the IDrawable interface for the Shape classes:

Rectangle class:

```
public class Rectangle : Shape, IDrawable
{
    // ...

    public void Draw()
    {
        Console.WriteLine("Drawing a Rectangle with Length: " + this.Length + " and Breadth: " + this.Breadth);
    }
}
```

Circle class:

```
public class Circle : Shape, IDrawable
{
    // ...

    public void Draw()
    {
        Console.WriteLine("Drawing a Circle with Radius: " + this.Radius);
    }
}
```

Triangle class:

```
public class Triangle : Shape, IDrawable
{
    // ...

    public void Draw()
    {
        Console.WriteLine("Drawing a Triangle with Base: " + this.Base + ",
Height: " + this.Height + " and Side1: " + this.Side1 + " Side2: " + this.Side2);
    }
}
```

In these implementations, we have provided the logic for drawing each shape in the Draw() method.

Now, you can use these abstracted classes and interfaces to work with shapes in a more organized and flexible manner:

```
Shape shape1 = new Rectangle { Length = 5, Breadth = 4 };
Shape shape2 = new Circle { Radius = 3 };
Shape shape3 = new Triangle { Base = 6, Height = 5, Side1 = 7, Side2 = 8 };

Console.WriteLine("Area of Rectangle: " + shape1.CalculateArea());
Console.WriteLine("Area of Circle: " + shape2.CalculateArea());
Console.WriteLine("Area of Triangle: " + shape3.CalculateArea());

shape1.Draw();
shape2.Draw();
shape3.Draw();
```

This will output:

```
Area of Rectangle: 20
Area of Circle: 28.27431376860352
Area of Triangle: 24
Drawing a Rectangle with Length: 5 and Breadth: 4
Drawing a Circle with Radius: 3
Drawing a Triangle with Base: 6, Height: 5 and Side1: 7 Side2: 8
```

In this example, we have demonstrated how abstraction can be implemented in C# using abstract classes and interfaces. By defining common properties and behaviours in an abstract class and an interface, we can create a hierarchy of classes that share similarities while maintaining their unique implementations. This promotes code reusability, modularity, and extensibility.

QUESTIONS:

- ✓ **What is abstraction in C#?**

  Abstraction is a programming concept that allows creating a high-level representation of an object or system by hiding its complex implementation details and exposing only essential functionality through abstract methods and properties.

- ✓ **What are the benefits of abstraction in C#?**

  Abstraction promotes code reusability, simplifies complex systems, reduces coupling between classes, and enables polymorphism.

- ✓ **What is an abstract class in C#?**

  An abstract class in C# is a base class that cannot be instantiated directly. It is used to define a common structure and behaviour for derived classes, containing abstract methods that must be implemented by the derived classes.

✓ **What is an interface in C#?**

An interface in C# is a collection of abstract methods, properties, and indexers that define a contract for a class or struct to implement. It allows multiple classes to share a common set of behaviours.

✓ **What is the difference between an abstract class and an interface in C#?**

An abstract class can have implementations for some methods, while an interface only contains method signatures without any implementation. Additionally, a class can inherit from only one abstract class, but it can implement multiple interfaces.

✓ **What is the use of the 'abstract' keyword in C#?**

The 'abstract' keyword is used to define abstract classes and abstract methods. An abstract class cannot be instantiated, and an abstract method must be implemented by any class that inherits from the abstract class.

✓ **What is the role of interfaces in C# abstraction?**

Interfaces in C# provide a way to define a contract or a set of methods, properties, and indexers that a class or struct must implement. It promotes abstraction by allowing multiple, unrelated classes to share a common behaviour without sharing a common inheritance hierarchy.

5. POLYMORPHISM:

Polymorphism allows objects of different classes to be treated as if they were of a common superclass, enabling code to work with multiple related classes using a single interface. Polymorphism in C# can be achieved through:

- **Method Overriding:** When a derived class provides its own implementation of a method that is already present in the parent class.
- **Method Overloading:** Defining multiple methods with the same name but different parameters, allowing the use of a single method name for different scenarios.
- **Late Binding and Dynamic Dispatch:** At runtime, the appropriate method or function is determined and executed based on the actual type of the object, not its reference type.

Let's dive into polymorphism with a detailed code example.

Consider a scenario where we have a base class called "Animal" and derived classes called "Dog" and "Cat". We want to demonstrate polymorphism by creating a method that can work with different types of animals.

❖ Define the base class "Animal" with a virtual method "**MakeSound**()":

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

    public virtual void MakeSound()
    {
        Console.WriteLine("The animal is making a sound.");
    }
}
```

In this base class, we have defined two properties (Name and Age) and a virtual method (**MakeSound**()) that will be overridden in the derived classes.

❖ Define the derived classes "Dog" and "Cat" that inherit from the Animal class and override the MakeSound() method:

```
public class Dog : Animal
{
    public string Breed { get; set; }

    public override void MakeSound()
    {
        Console.WriteLine("The dog is barking.");
    }
}

public class Cat : Animal
{
    public string Breed { get; set; }

    public override void MakeSound()
    {
        Console.WriteLine("The cat is meowing.");
    }
}
```

In these derived classes, we have added a new property called Breed and overridden the MakeSound() method from the Animal class to provide specific implementations.

❖ Create a method called "DisplayAnimalDetails" that takes an Animal object and calls the MakeSound() method:

```
public static void DisplayAnimalDetails(Animal animal)
{
    Console.WriteLine("Animal Name: " + animal.Name);
    Console.WriteLine("Animal Age: " + animal.Age);
    animal.MakeSound();
}
```

This method accepts an Animal object as a parameter and prints the animal's name, age, and makes the animal sound by calling the **MakeSound()** method.

Now, you can create instances of the Dog and Cat classes and pass them to the **DisplayAnimalDetails()** method:

```
Dog myDog = new Dog
{
    Name = "Buddy",
    Age = 3,
    Breed = "Labrador Retriever"
};

Cat myCat = new Cat
{
    Name = "Whiskers",
    Age = 5,
    Breed = "Siamese"
};

DisplayAnimalDetails(myDog);
DisplayAnimalDetails(myCat);
```

This will output:

```
Animal Name: Buddy
Animal Age: 3
The dog is barking.
Animal Name: Whiskers
Animal Age: 5
The cat is meowing.
```

In this example, we have demonstrated how polymorphism can be implemented in C# through method overriding. The DisplayAnimalDetails() method can work with different types of animals (Dog and Cat) because they both inherit from the Animal class and override the MakeSound() method. This showcases the flexibility and extensibility of polymorphism in object-oriented programming.

QUESTIONS:

- **What is polymorphism in C#?**

  Polymorphism in C# is the ability of objects of different classes related by inheritance or implementing a common interface to be treated as objects of a common base class or interface. It allows a single method or variable to have multiple forms or implementations, promoting code flexibility and extensibility.

- **What are the types of polymorphism in C#?**

  In C#, there are two main types of polymorphism: Compile-time polymorphism (also called static binding) and Runtime polymorphism (also called dynamic binding).

❖ What is compile-time polymorphism in C#?

Compile-time polymorphism in C# is achieved through static methods, which are resolved at compile-time. It allows a single method to be implemented in multiple ways based on the types of its arguments or return types.

❖ What is runtime polymorphism in C#?

Runtime polymorphism in C# is achieved through virtual methods, which are resolved at runtime based on the actual object type. It allows a single method to have multiple implementations, depending on the object's class at runtime.

❖ What is method overloading in C#?

Method overloading in C# is a form of compile-time polymorphism that allows creating multiple methods with the same name in the same class, as long as their parameters' types or numbers differ. It provides the flexibility to call a method with different sets of arguments.

❖ What is method overriding in C#?

Method overriding in C# is a form of runtime polymorphism that occurs when a derived class provides its implementation of a method defined in its base class. The derived class's method is marked with the ‘override’ keyword and can have a different or similar signature compared to the base class method.

❖ What is the use of the ‘virtual’ keyword in C#?

The ‘virtual’ keyword in C# allows a base class method to be overridden by derived classes, enabling runtime polymorphism. It indicates that a method can have different implementations depending on the actual object type at runtime.

❖ What is the use of the ‘override’ keyword in C#?

The ‘override’ keyword in C# is used in a derived class to replace or provide an alternative implementation of a method or property that exists in the base class. It enables runtime polymorphism and allows derived classes to customize the behaviour of the base class method.

❖ **What is the purpose of interfaces in C# polymorphism?**

Interfaces in C# promote polymorphism by allowing multiple, unrelated classes to share a common behaviour without sharing a common inheritance hierarchy. Implementing an interface forces classes to provide implementations for the interface's methods, enabling dynamic behaviour at runtime.

❖ **How does the 'base' keyword contribute to polymorphism in C#?**

The 'base' keyword in C# is used to access and invoke methods or properties of the base class within a derived class. It allows derived classes to extend or override the base class's functionality, promoting polymorphism by enabling a single method or variable to have multiple forms or implementations.

MISCELLANEOUS QUESTION

1. **WHAT IS A CLASS IN C#?**
   A class is a blueprint for creating objects that defines its attributes and behaviours.

2. **WHAT IS AN OBJECT IN C#?**
   An object is an instance of a class that encapsulates data and behaviour.

3. **WHAT IS ENCAPSULATION IN C#?**
   Encapsulation is the bundling of data and methods that operate on the data within a single unit (object), hiding the internal state from the outside world.

4. **HOW IS ENCAPSULATION ACHIEVED IN C#?**
   Encapsulation is achieved by using access modifiers (public, private, protected) to control the access to the members of a class.

5. **WHAT IS INHERITANCE IN C#?**
   Inheritance is a mechanism where a new class (derived class) is created from an existing class (base class), inheriting its attributes and behaviours.

6. **HOW IS INHERITANCE IMPLEMENTED IN C#?**
   Inheritance is implemented using the colon (:) syntax to indicate the base class from which the derived class inherits.

7. **WHAT IS POLYMORPHISM IN C#?**
   Polymorphism is the ability of objects to take on multiple forms, allowing methods to behave differently based on the object they are acting upon.

8. **WHAT ARE THE TWO TYPES OF POLYMORPHISM IN C#?**
   Compile-time (method overloading) and runtime (method overriding) polymorphism.

9. **WHAT IS METHOD OVERLOADING IN C#?**
   Method overloading is the ability to define multiple methods in the same class with the same name but different parameters.

10. WHAT IS METHOD OVERRIDING IN C#?
    Method overriding is the ability to provide a new implementation of a method in the derived class that is already defined in the base class.

11. WHAT IS ABSTRACTION IN C#?

Abstraction is the process of hiding the complex implementation details and showing only the essential features of an object.

12. HOW IS ABSTRACTION ACHIEVED IN C#?

Abstraction is achieved using abstract classes and interfaces to define a contract for classes that implement them.

13. WHAT IS AN ABSTRACT CLASS IN C#?

An abstract class is a class that cannot be instantiated and may contain abstract methods that must be implemented by its derived classes.

14. WHAT IS AN INTERFACE IN C#?

An interface is a contract that defines a set of methods and properties that a class must implement.

15. CAN A CLASS IMPLEMENT MULTIPLE INTERFACES IN C#?

Yes, a class can implement multiple interfaces in C#.

16. WHAT IS A CONSTRUCTOR IN C#?

A constructor is a special method in a class that is called automatically when an instance of the class is created.

17. CAN A CLASS HAVE MULTIPLE CONSTRUCTORS IN C#?

Yes, a class can have multiple constructors in C#, known as constructor overloading.

18. WHAT IS A DESTRUCTOR IN C#?

A destructor is a special method in a class that is called automatically when an object is destroyed or goes out of scope.

19. DOES C# SUPPORT MULTIPLE INHERITANCE?

No, C# does not support multiple inheritance for classes, but it supports multiple inheritance through interfaces.

20. WHAT IS THE BASE KEYWORD USED FOR IN C#?

The base keyword is used to access members of the base class from within a derived class.

21. WHAT IS THE DIFFERENCE BETWEEN VALUE TYPES AND REFERENCE TYPES IN C#?

Value types store their data directly, while reference types store a reference to the data's location in memory.

22. WHAT ARE EXAMPLES OF VALUE TYPES IN C#?

Examples of value types in C# include int, float, double, struct, enum, etc.

23. WHAT ARE EXAMPLES OF REFERENCE TYPES IN C#?

Examples of reference types in C# include classes, interfaces, arrays, delegates, strings, etc.

24. WHAT IS A NAMESPACE IN C#?

A namespace is a way to organize and group related classes and other types.

25. HOW ARE NAMESPACES USED IN C#?

Namespaces are used to avoid naming conflicts and to organize code into logical groups for better maintainability.

26. WHAT IS THE DIFFERENCE BETWEEN AN ABSTRACT CLASS AND AN INTERFACE IN C#?

An abstract class can have both abstract and non-abstract members, while an interface can only have abstract members.

27. CAN AN ABSTRACT CLASS HAVE CONSTRUCTORS IN C#?

Yes, an abstract class can have constructors in C#, but they cannot be used to instantiate the abstract class directly.

28. CAN AN INTERFACE HAVE FIELDS IN C#?

No, an interface cannot have fields in C#; it can only have properties, methods, events, and indexers.

29. WHAT IS METHOD HIDING IN C#?

Method hiding is a mechanism where a method in a derived class hides a method with the same signature in the base class.

30. WHAT IS THE PURPOSE OF THE SEALED KEYWORD IN C#?

The sealed keyword is used to prevent a class from being inherited and to prevent method overriding in derived classes.

31. WHAT IS A CONSTRUCTOR IN C#?

A special method used to initialize objects of a class.

32. WHAT IS METHOD OVERLOADING?

Defining multiple methods in the same class with the same name but different parameters.

33. WHAT IS METHOD OVERRIDING?

Providing a new implementation for a method in a subclass that is already defined in its superclass.

34. WHAT IS THE DIFFERENCE BETWEEN ABSTRACT CLASSES AND INTERFACES?

Abstract classes can contain method implementations, while interfaces cannot.

35. WHAT IS A STATIC CLASS IN C#?

A class that cannot be instantiated and can only contain static members.

36. WHAT IS THE PURPOSE OF THE 'BASE' KEYWORD IN C#?

It is used to access members of the base class from within a derived class.

37. WHAT IS A SEALED CLASS IN C#?

A class that cannot be inherited by other classes.

38. WHAT IS A DELEGATE IN C#?

A type that represents references to methods with a specific signature.

39. WHAT IS AN EVENT IN C#?

A mechanism for enabling a class to notify other classes or objects when something of interest occurs.

40. WHAT IS THE 'THIS' KEYWORD USED FOR IN C#?

It refers to the current instance of the class and is used to access members of the current object.

41. WHAT IS A NAMESPACE IN C#?

A way to organize code by grouping related classes, interfaces, structs, enums, and delegates.

42. WHAT IS THE PURPOSE OF ACCESS MODIFIERS IN C#?

They control the visibility of classes, methods, and other members within a program.

43. WHAT IS POLYMORPHISM IN C#?

The ability of objects of different types to be treated as objects of a common base type.

44. WHAT IS THE DIFFERENCE BETWEEN 'REF' AND 'OUT' PARAMETERS IN C#?

'ref' parameters must be initialized before being passed to a method, while 'out' parameters do not need to be initialized.

45. WHAT IS THE PURPOSE OF THE 'OVERRIDE' KEYWORD IN C#?

It is used to override a virtual method in a derived class, providing a new implementation.