Object-Oriented Programming (OOP)

Mirolim Majidov

1. What is OOP and its concepts in C#?

What is this?

Object-Oriented Programming (OOP) is a programming paradigm that uses objects to organize code.

These four concepts—**encapsulation, inheritance, polymorphism, and abstraction**—form the foundation of object-oriented programming and are key principles in designing and implementing software systems using languages like C#.

How/where can I use this?

Encapsulation: Bundle the data (attributes) and methods (functions) that operate on the data into a single unit, i.e., a class. Helps in data hiding and organization.

Inheritance: Create a new class by reusing properties and behaviors of an existing class. Promotes code reusability.

Polymorphism: Use a single interface to represent different types. Enables method overloading and method overriding.

Abstraction: Abstraction is the process of hiding the complex implementation details and showing only the essential features of an object. It allows programmers to focus on the relevant aspects of an object and ignore the irrelevant details

2. Inheritance

What is this?

Inheritance is a mechanism where a new class inherits properties and behaviors from an existing class.

How/where can I use this?

It promotes code reuse and allows the creation of a new class that is a modified version of an existing class.

Example:

```
1 reference
class Animal
{
    0 references
    public void Eat()
    {
        Console.WriteLine("Eating...");
    }
}

0 references
class Dog : Animal
{
    0 references
    public void Bark()
    {
        Console.WriteLine("Barking...");
    }
}
```

3. Abstract Classes

What is this?

An abstract class is a class that cannot be instantiated and may contain abstract methods (methods without implementation).

How/where can I use this?

Abstract classes are used when you want to provide a common base for multiple derived classes but want to enforce certain methods to be implemented in each derived class.

Example:

```
1 reference
abstract class Shape
{
    1 reference
    public abstract void Draw();
}

0 references
class Pentagon : Shape
{
    1 reference
    public override void Draw()
    {
    }
}
```

4. Read-only Properties in a Class

What is this?

Read-only properties are properties that can only be assigned a value during initialization or within the constructor of the class

How/where can I use this?

Use read-only properties when you want to ensure that the property value cannot be changed once the object is created.

Example:

```
1 reference
class Circle
{
    1 reference
    public double Radius { get; }

    0 references
    public Circle(double radius)
    {
        Radius = radius;
    }
}
```

5. Virtual Methods and Properties

What is this?

Virtual methods and properties can be overridden in derived classes.

How/where can I use this?

Use virtual methods and properties when you want to provide a default implementation in the base class but allow derived classes to override it.

Example:

```
class Shape
{
    0 references
    public virtual void Draw()
    {
        Console.WriteLine("Drawing a shape");
    }
}
```

6. Hiding, Overriding, and Abstract Methods

What is this?

Hiding allows a derived class to provide a new implementation for a method, overriding replaces the base class method in the derived class, and abstract methods are declared in the base class and must be implemented in derived classes.

How/where can I use this?

Use hiding, overriding, and abstract methods to customize behavior in derived classes.

Example:

```
1 reference
abstract class Base
{
    1 reference
    public abstract void Method();
}

1 reference
class Derived : Base
{
    1 reference
    public override void Method()
    {
        Console.WriteLine("Derived method");
    }
}
```

```
0 references
class Child : Derived
{
    0 references
    new public void Method()
    {
        Console.WriteLine("Child method");
    }
}
```

7. Interfaces

What is this?

Interfaces define a contract for classes, specifying a set of methods and properties that implementing classes must provide.

How/where can I use this?

Interfaces are used to achieve multiple inheritance and provide a way for classes to share a common set of methods.

Example:

```
1 reference
interface ILogger
{
    1 reference
    void Log(string message);
}

0 references
class MyLogger : ILogger
{
    1 reference
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
```

8. Interface Inheritance

What is this?

An interface can inherit from one or more interfaces, allowing a class to implement multiple interfaces.

How/where can I use this?

Interface inheritance helps in building a hierarchy of interfaces, making it easier to organize and manage code.

Example:

```
1 reference
interface IShape
{
    0 references
    void Draw();
}

0 references
interface IResizableShape : IShape
{
    0 references
    void Resize(int factor);
}
```

9. Generic Classes

What is this?

Generic classes allow the definition of classes with placeholders for data types.

How/where can I use this?

Use generic classes when you want to create classes that can work with different data types without sacrificing type safety.

Example:

```
1 reference
class Box<T>
{
    0 references
    public T Data { get; set; }
}


var box = new Box<int>();
```

10. Generic Methods

What is this?

Generic methods allow the definition of methods with placeholders for data types.

How/where can I use this?

Use generic methods when you want to write methods that can work with different data types.

Example:

```
1 reference
class Utilities
{
    1 reference
    public static T Max<T>(T a, T b) where T : IComparable<T>
    {
        return a.CompareTo(b) > 0 ? a : b;
    }
}

var a = 10;
var b = 10;

var maxValue = Utilities.Max<int>(a, b);
```

11. Generic Methods

What is this?

Generic properties allow the definition of properties with placeholders for data types.

How/where can I use this?

Use generic properties when you want a property to work with different data types.

Example:

```
1 reference
class Container<T>
{
    1 reference
    public T Value { get; set; }
}

var container = new Container<string>();
container.Value = "My name";
```