C# Day2 Object-Oriented Programming

1. What are the six combinations of access modifier keywords and what do they do?
   - public: Accessible from any other code in the same assembly or another assembly that references it.
   - private: Accessible only from within the same class or struct.
   - protected: Accessible only within the same class or from derived classes.
   - internal: Accessible only within files in the same assembly.
   - protected internal: Accessible within the same assembly or from derived classes in any assembly.
   - private protected: Accessible only within the same assembly and only by derived classes.

2.What is the difference between the static, const, and readonly keywords when applied to a type member?
   - static: Indicates that the member belongs to the type itself rather than to instances of the type. It's shared across all instances of the class.
   - const: Specifies that the member is a constant whose value is set at compile-time and cannot be changed.
   - readonly: Indicates that the member can only be assigned a value in its declaration or in a constructor. After initialization, its value cannot be changed.

3. What does a constructor do?
   1) Constructor is a special method that has same name as the class name and does not have any return type; not even void
   2) It is used to create an object of the class and initalize the class members.
   3) if there's no constructor in the class, c# compiler will provide a default constructor and is parameterless.
   4) Constructor can be overloaded which means method can have the same name but different parameters.
   5) Constructor cannot be inherited. So a constructor can not be overriden.
   6) By default a derived class constructor will make a call to a base class constructor.

4. Why is the partial keyword useful?
   - The partial keyword allows a class, struct, method, or interface to be defined across multiple files within the same namespace.
   - This is useful in large projects where multiple developers may work on different parts of a class simultaneously or when using code generation tools that produce part of a class.
   - It enables better organization and maintenance of large codebases by logically separating parts of a type into different files.

5. What is a tuple?

   - A tuple is a data structure that allows you to store multiple items in a single variable. It's a lightweight, immutable collection of elements that can be of different types.
   - In C#, tuples are often used to return multiple values from a method.

6. What does the C# record keyword do?

- The `record` keyword in C# is used to define a reference type that provides built-in functionality for encapsulating data. It is a concise way to create immutable data objects.
- Records are value-based, meaning two record instances are considered equal if all their properties are equal.
- Example: `public record Shape(int length, int width);`

7. What does overloading and overriding mean?

Overloading: multiple methods in the same scope with the same name but different signatures (different number or types of parameters).
Overriding: This is when a method in a derived class has the same name, return type, and parameters as a method in its base class. The derived class method replaces the base class method. Usually happens in polymorphism.

8. What is the difference between a field and a property?

Field: A variable that is declared directly in a class or struct. It is used to store data directly.
Property: A member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can have get and set accessors.

Eg:

```
public class MyClass {
   private int myField;
   public int MyProperty {
      get { return myField; }
      set { myField = value; }
   }
}
```

9. How do you make a method parameter optional?

- make a method parameter optional by providing a default value for it in the method signature.
- eg:

```
public void PrintMessage(string message = "Hello, World!") {
    Console.WriteLine(message);
}
// Calling PrintMessage() without an argument will print "Hello, World!"
PrintMessage();
// Calling PrintMessage("Hi!") will print "Hi!"
PrintMessage("Hi!");
```

10. What is an interface and how is it different from abstract class?

Interface is a collection of methods which are by default abstract and public and will be implemented by the derived classes.

- Abstract class will provide base class to its subclasses; is a wise choice when we have class hierarchy.
  --Interface will define common functionalities and behaviors that can be implemented by any class.

- Once class can only inherit from one abstract or concrete class but one class can implement multiple interfaces.

- Methods in abstract class can be abstract method or non-abstract method but methods in an interface is by default public and abstract.

Abstract Class:

- means we cannot create any instance out of this class
- it can both abstract methods (without implementation) and concrete methods (with implementation).
- A class can inherit only one abstract class due to single inheritance in C#, but it can implement multiple interfaces.

11. What accessibility level are members of an interface?

- By default, all members of an interface are `public`, and they cannot have any other access modifiers. Interface members are implicitly public and abstract, and they do not require explicit access modifiers.
- eg:

```
public interface Shape {
    void Draw();  // This is implicitly public
}
```

12. True/False. Polymorphism allows derived classes to provide different implementations of the same method.

13. True/False. The override keyword is used to indicate that a method in a derived class is providing its own implementation of a method.
14. True/False. The new keyword is used to indicate that a method in a derived class is providing its own implementation of a method.

15. True/False. Abstract methods can be used in a normal (non-abstract) class.

Abstract methods must be declared in an abstract class. The class that inherits from the abstract class must provide an implementation for the abstract methods. Abstract methods define a signature but do not provide an implementation, and their implementation must be provided by derived classes.

```
// Abstract class
public abstract class Animal
{
    // Abstract method, no implementation
    public abstract void MakeSound();

    // Concrete method, has implementation
    public void Sleep()
    {
        Console.WriteLine("Sleeping...");
    }
}

// Derived class
public class Dog : Animal
{
    // Implementing the abstract method
    public override void MakeSound()
    {
        Console.WriteLine("Bark");
    }
}

// Using the derived class
public class Program
{
    public static void Main()
    {
        Dog myDog = new Dog();
        myDog.MakeSound(); // Output: Bark
        myDog.Sleep();     // Output: Sleeping...
    }
}
```

1. `Animal` is an abstract class containing an abstract method `MakeSound()` and a concrete method `Sleep()`.
2. `Dog` is a derived class that must implement the `MakeSound()` method.
3. In the `Program` class's `Main` method, we create a `Dog` object and call its `MakeSound()` and `Sleep()` methods.

The abstract method `MakeSound()` must be declared in an abstract class (`Animal`) and cannot be declared in a non-abstract (normal) class. The derived class `Dog` is required to provide an implementation for the `MakeSound()` method.

16.True/False. Normal (non-abstract) methods can be used in an abstract class.

17. True/False.Derived classes can override methods that were virtual in the base class.

18. True/False. Derived classes can override methods that were abstract in the base class.

Derived classes can (and must) override methods that were marked as `abstract` in the base class. Abstract methods do not have an implementation in the base class and must be implemented in any derived non-abstract class.

19. True/False. In a derived class, you can override a method that was neither virtual non abstract in the base class.

Only methods that are marked with the `virtual` or `abstract` keyword in the base class can be overridden in derived classes. If a method is not marked as `virtual` or `abstract`, it cannot be overridden, but it can be hidden using the `new` keyword.
Eg:

```
public class BaseClass
{
    public void Display()
    {
        Console.WriteLine("BaseClass Display");
    }
}

public class DerivedClass : BaseClass
{
    // This will result in a compile-time error because Display() in BaseClass is not virtual or
abstract
    // public override void Display()
    // {
    //     Console.WriteLine("DerivedClass Display");
    // }

    // Hiding the base class method using 'new' keyword
    public new void Display()
    {
        Console.WriteLine("DerivedClass Display");
    }
}

public class Program
{
    public static void Main()
    {
        DerivedClass obj = new DerivedClass();
```

```
        obj.Display(); // Output: DerivedClass Display

        BaseClass baseObj = obj;
        baseObj.Display(); // Output: BaseClass Display
    }
}
```

20. True/False. A class that implements an interface does not have to provide an implementation for all of the members of the interface.

A class that implements an interface must provide implementations for all of the members defined in the interface. If a class does not implement all interface members, it must be declared as `abstract`.

Eg:

```
public interface IAnimal
{
    void MakeSound();
    void Eat();
}

public class Dog : IAnimal
{
    public void MakeSound()
    {
        Console.WriteLine("Bark");
    }

    public void Eat()
    {
        Console.WriteLine("Dog is eating");
    }
}

public class Program
{
    public static void Main()
    {
        Dog myDog = new Dog();
        myDog.MakeSound(); // Output: Bark
        myDog.Eat();       // Output: Dog is eating
    }
}
```

21. True/False. A class that implements an interface is allowed to have other members that aren't defined in the interface.

```
public interface IAnimal
{
    void MakeSound();
    void Eat();
}

public class Dog : IAnimal
{
    // Implementation of IAnimal members
    public void MakeSound()
    {
        Console.WriteLine("Bark");
    }

    public void Eat()
    {
        Console.WriteLine("Dog is eating");
    }

    // Additional members not defined in the interface
    public void Sleep()
    {
        Console.WriteLine("Dog is sleeping");
    }
}

public class Program
{
    public static void Main()
    {
        Dog myDog = new Dog();
        myDog.MakeSound(); // Output: Bark
        myDog.Eat();       // Output: Dog is eating
        myDog.Sleep();     // Output: Dog is sleeping
    }
}
```

22. True/False. A class can have more than one base class.

23. True/False. A class can implement more than one interface.

Eg:

```
// Define two interfaces
interface IShape
{
    void Draw();
```

```
}

interface IMovable
{
    void Move();
}

// Implementing class that implements both interfaces
class Circle : IShape, IMovable
{
    public void Draw()
    {
        Console.WriteLine("Circle is drawn");
    }

    public void Move()
    {
        Console.WriteLine("Circle is moved");
    }
}

// Usage
class Program
{
    static void Main()
    {
        Circle circle = new Circle();
        circle.Draw();  // Output: Circle is drawn
        circle.Move();  // Output: Circle is moved
    }
}
```