**Task06 C#**

**Problem 1**

a **struct** cannot inherit from another struct or class because it is a **value type**, while classes are **reference types**. All structs implicitly inherit from System.ValueType (which itself inherits from System.Object), and C# does not allow value types to have any other base type.

The main reasons are:

1. **Memory Model** – Structs store their data directly (usually on the stack or inline in other objects) and are copied by value. Inheritance would require reference-based behavior, which conflicts with value semantics and could cause unpredictable memory layouts.
2. **Sealed by Design** – Structs are implicitly sealed, meaning they cannot be extended. Allowing inheritance would break this rule and complicate the runtime’s handling of value types.
3. **Boxing/Unboxing Complexity** – Value types can be boxed into objects. Supporting inheritance would make boxing/unboxing more complex and less efficient, undermining one of the key performance benefits of structs.

Instead of inheritance, structs can **implement interfaces**, allowing code reuse and polymorphism without altering the value-type behavior.

**Problem 2**

**How do access modifiers impact the scope and visibility of a class member?**

**Access modifiers in C# define the scope and visibility of class members**, determining where they can be accessed from. They are key to implementing **encapsulation** and controlling how a class’s internal data and behavior are exposed.

* **public** – Accessible from anywhere, inside or outside the assembly.
* **private** – Accessible only within the same class or struct.
* **protected** – Accessible within the same class and its derived classes.
* **internal** – Accessible from any code in the same assembly but not from other assemblies.
* **protected internal** – Accessible from the same assembly and from derived classes in other assemblies.
* **private protected** – Accessible only within the same assembly and only from derived classes.

By using access modifiers, developers can **restrict access to sensitive data**, expose only what is necessary, and maintain a clear separation between a class’s internal implementation and its public interface.

**Problem 3**

**Why is encapsulation critical in software design?**

**Encapsulation** is critical in software design because it **bundles data and the methods that operate on that data into a single unit** (e.g., a class) and **restricts direct access** to the internal state. This principle is a cornerstone of **Object-Oriented Programming (OOP)** and provides several key benefits:

1. **Data Protection** – By hiding internal fields and exposing them only through controlled methods or properties, encapsulation prevents unauthorized or accidental modification of data.
2. **Modularity** – Encapsulation keeps related data and behavior together, making the code more organized and easier to maintain.
3. **Maintainability** – Internal implementation can change without affecting external code that relies on the class’s public interface.
4. **Reusability** – Well-encapsulated components can be reused in different parts of a system or in other projects with minimal modification.
5. **Reduced Complexity** – External code does not need to understand the internal workings of a class, only its public interface, which simplifies usage.

**In short:** Encapsulation supports **security, clarity, and maintainability** in software design, making systems more robust and easier to evolve over time.

**Problem 4**

**what is constructors in structs?**

**constructor** in a struct is a special method used to initialize the struct’s fields when a new instance is created. Like class constructors, they have the same name as the struct and do not have a return type.

**Key Points:**

1. **Default Constructor**
   * Every struct automatically has a parameterless *default constructor* provided by the compiler.
   * It initializes all fields to their default values (e.g., 0 for numeric types, false for bool, null for reference types).
   * You cannot explicitly define your own parameterless constructor in C# versions before 10.0.
2. **Parameterized Constructors**
   * You can define constructors that take parameters to set specific values for the fields.
   * All fields of the struct must be fully assigned before the constructor ends.
3. **No Inheritance**
   * Structs cannot inherit from other structs or classes, so constructors do not involve base constructor calls (except System.ValueType implicitly).

Example :

struct Point

{

public int X;

public int Y;

// Parameterized constructor

public Point(int x, int y)

{

X = x;

Y = y;

}

}

class Program

{

static void Main()

{

Point p1 = new Point(); // Uses default constructor → X=0, Y=0

Point p2 = new Point(5, 10); // Uses parameterized constructor

}

}

**Problem 5**

**How does overriding methods like ToString() improve code readability?**

the ToString() method (inherited from System.Object) returns a string representation of an object. By default, it returns the fully qualified type name, which is often not meaningful for understanding the object’s state. **Overriding** this method allows developers to provide a custom, human-readable description of the object.

**How It Improves Readability:**

1. **Clearer Output** – Instead of generic type names, overridden ToString() methods can display key property values that describe the object in a useful way.
2. **Better Debugging** – When inspecting objects in the debugger or printing them to the console/log, the output is immediately informative without extra formatting code.
3. **Reduced Boilerplate** – You can directly pass the object to methods like Console.WriteLine() without manually building strings each time.
4. **Consistent Representation** – Ensures all parts of the application display the object’s data in the same, consistent format.

Example :

struct Point

{

public int X;

public int Y;

public Point(int x, int y)

{

X = x;

Y = y;

}

// Override ToString for custom formatting

public override string ToString()

{

return $"Point: X={X}, Y={Y}";

}

}

class Program

{

static void Main()

{

Point p = new Point(5, 10);

Console.WriteLine(p); // Output: Point: X=5, Y=10

}}