Part01

1. Why can't a struct inherit from another struct or class in C#?

- In C#, **structs** are **value types**, while inheritance is primarily designed for **reference types** (classes).
- Structs are meant to be lightweight and small in memory footprint; allowing inheritance would add complexity to memory management.
- However, every struct **implicitly inherits** from System.ValueType (which itself inherits from System.Object).

📌 **Summary:** A struct can't inherit from another struct or class, but it can implement **interfaces**.

---

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

- **private** → Member is accessible only within the same class.
- **internal** → Member is accessible only within the same project (assembly).
- **protected** → Member is accessible within the class and any derived classes.
- **public** → Member is accessible from anywhere in the code.

📌 **Summary:** The access modifier determines **who** can see or use a member.

---

3. Why is encapsulation critical in software design?

- Protects data from unauthorized access or modification.
- Allows controlled access through methods or properties.
- Makes code flexible and easy to maintain without breaking other parts of the program.

📌 **Example:** Instead of making Salary public, make it private and use a **property** to validate changes.

---

4. What are constructors in structs?

- A **constructor** in a struct is a special method that runs when a struct instance is created.
- In structs, **all fields must be initialized** inside the constructor.
- You can create **parameterized constructors**, but you cannot explicitly define a default (parameterless) constructor—it's provided automatically by C#.

📌 **Note:** Struct variables must be fully assigned before use.

---

5. How does overriding ToString() improve code readability?

- By default, ToString() prints the type name (e.g., "Namespace.Point").
- Overriding it lets you display meaningful, human-readable information instead.
- This improves debugging, logging, and user output clarity.

📌 **Example:**

```
Console.WriteLine(new Point(3, 4)); // Instead of "Namespace.Point", prints "(3, 4)"
```

---

6. How does memory allocation differ for structs and classes in C#?

- **Struct (Value Type)** → Stored directly on the **stack** (or inline inside another object), and values are **copied** when assigned or passed to a method.
- **Class (Reference Type)** → Stored on the **heap**, and variables hold a **reference** to the actual object in memory.

📌 **Summary:**

- Struct → Each variable has its own independent copy.
- Class → Variables share the same object reference.

Part02

1. Copy Constructor

A **copy constructor** is a special constructor that creates a new object by copying the values from another existing object of the same type.

In C#

- Unlike C++, C# doesn't provide a default copy constructor.
- You must define it yourself if you need to clone an object manually.

Example:

```
class Employee
{
  public string Name;
  public double Salary;

  // Copy Constructor
  public Employee(Employee other)
  {
    Name = other.Name;
    Salary = other.Salary;
  }
}

class Program
{
  static void Main()
  {
    Employee e1 = new Employee { Name = "Ahmed", Salary = 5000 };
```

```
    Employee e2 = new Employee(e1); // Uses copy constructor

    Console.WriteLine($"{e2.Name}, {e2.Salary}"); // Ahmed, 5000
  }
}
```

📌 **Business usage:**

- Cloning customer profiles.
- Creating a backup snapshot of an object's state.
- Passing copies of sensitive data so the original remains unchanged.

---

2. Indexer

An **indexer** in C# lets you access class or struct members **like an array** using square brackets [].

When to use

You use an indexer when your object logically represents a **collection of items** and you want to provide **array-like access**.

Example:

```
class Department
{
  private string[] employees = new string[5];

  public string this[int index]
  {
    get { return employees[index]; }
    set { employees[index] = value; }
  }
```

```
}

class Program
{
  static void Main()
  {
    Department dept = new Department();
    dept[0] = "Ali";
    dept[1] = "Omar";

    Console.WriteLine(dept[0]); // Ali
    Console.WriteLine(dept[1]); // Omar
  }
}
```

📌 **Business usage cases:**

1. **Employee Directory** → Access employee by ID index.
2. **Inventory Management** → Access products by SKU index.
3. **Financial Systems** → Access transactions by sequential number.
4. **Data Grid UI Components** → Allow [row, column] access to data cells.

Summarize keywords we have learnt last lecture

1. struct

- **Value type**.
- Stored in **stack** (or inline in another object).
- Cannot inherit from another struct or class (but can implement interfaces).
- Can have fields, methods, properties, and constructors (except explicit parameterless constructors).

---

2. class

- **Reference type**.
- Stored in **heap**.
- Supports inheritance, polymorphism, encapsulation.
- Members can have different access modifiers.

---

3. Access Modifiers

- private → Only within the same class.
- internal → Within the same assembly.
- protected → Within class + derived classes.
- public → Anywhere.

---

4. Encapsulation

- Hiding fields (usually with private).
- Controlling access via **methods** or **properties**.
- Improves security and maintainability.

---

5. Constructor

- Special method to initialize objects.
- For struct, must initialize all fields.
- Can overload constructors with different parameters.
- **Copy Constructor** → Special form to copy from another object.

---

6. Overriding Methods

- override keyword lets you change base class methods (e.g., ToString()).
- Improves readability and debugging output.