Part 1 — Problems & Questions

1. Struct Point with X and Y + Constructors + ToString()

```
using System;

struct Point
{
    public int X;
    public int Y;

    // Default constructor — structs get an implicit one, but we can
initialize in parameterized constructors
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public override string ToString()
    {
        return $"({X}, {Y})";
    }
}

class Program
{
    static void Main()
    {
        Point p1 = new Point(3, 4);
        Console.WriteLine(p1); // (3, 4)
    }
}
```

Q: Why can't a struct inherit from another struct or class in C#?
A: In C#, all structs **implicitly inherit from `System.ValueType`** and are *value types*. Multiple inheritance would break the simple memory layout of value types and affect performance. Structs can **implement interfaces** but **cannot inherit from other structs or classes**.

---

2. Class TypeA with F, G, H + Access Test

```
// File1.cs
public class TypeA
{
    private int F = 1;
    internal int G = 2;
    public int H = 3;

    public void ShowValues()
    {
        Console.WriteLine($"F: {F}, G: {G}, H: {H}");
    }
}
```

```
// File2.cs
class Program
{
    static void Main()
    {
        TypeA obj = new TypeA();
        // obj.F = 10; // ❌ private - not accessible
        obj.G = 20;    // ✅ internal - same assembly
        obj.H = 30;    // ✅ public - accessible everywhere
        obj.ShowValues();
    }
}
```

Q: How do access modifiers impact scope & visibility?
A:

- **private** → Only inside the class/struct.
- **internal** → Anywhere in the same assembly/project.
- **public** → Accessible from anywhere.
- **protected** → Inside the class and derived classes.
- **protected internal** → Protected OR Internal access.
- **private protected** → Protected AND Internal access.

---

3. Struct Employee + Encapsulation

```
using System;

struct Employee
{
    private int EmpId;
    private string Name;
    private double Salary;

    public string GetName() => Name;
    public void SetName(string name) => Name = name;

    public double SalaryProperty
    {
        get => Salary;
        set => Salary = value;
    }
}

class Test
{
    static void Main()
    {
        Employee e = new Employee();
        e.SetName("John");
        e.SalaryProperty = 50000;

        Console.WriteLine($"Employee: {e.GetName()}, Salary:
{e.SalaryProperty}");
    }
}
```

Q: Why is encapsulation critical?
A: It protects the **integrity of data** by controlling access, hides internal implementation, improves maintainability, and prevents unintended changes.

---

4. Struct Point — Constructor Overloading

```
struct Point
{
    public int X, Y;

    public Point(int x)
    {
        X = x;
        Y = 0;
    }

    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public override string ToString() => $"({X}, {Y})";
}

class Test
{
    static void Main()
    {
        Point p1 = new Point(5);
        Point p2 = new Point(5, 10);
        Console.WriteLine(p1);
        Console.WriteLine(p2);
    }
}
```

Q: What are constructors in structs?
A: Special methods used to initialize data when a struct is created. In structs, you must **initialize all fields** in a parameterized constructor, and they **cannot have a custom default constructor**.

---

5. Custom ToString() Formatting

```
struct Point
{
    public int X, Y;

    public Point(int x, int y) { X = x; Y = y; }

    public override string ToString()
    {
        return $"Point => X: {X}, Y: {Y}";
    }
```

```
}

class Test
{
    static void Main()
    {
        Point[] points = { new Point(1, 2), new Point(3, 4), new Point(5,
6) };
        foreach (var p in points)
            Console.WriteLine(p);
    }
}
```

Q: How does overriding ToString() help?
A: It gives **meaningful, human-readable output** instead of the default type name, making debugging and logging easier.

---

6. Struct vs Class — Passing to Methods

```
using System;

struct Point { public int X; }
class Employee { public string Name; }

class Test
{
    static void ChangePoint(Point p) { p.X = 100; }
    static void ChangeEmployee(Employee e) { e.Name = "Changed"; }

    static void Main()
    {
        Point p = new Point { X = 1 };
        Employee emp = new Employee { Name = "Original" };

        ChangePoint(p);
        ChangeEmployee(emp);

        Console.WriteLine($"Point X: {p.X}"); // Still 1 - value type copy
        Console.WriteLine($"Employee Name: {emp.Name}"); // Changed -
reference type
    }
}
```

Q: Memory allocation difference?
A:

- **Structs (value types)** → Stored on **stack** (or inline in containing type), copied on assignment.
- **Classes (reference types)** → Stored on **heap**, variables hold references.

Part 2 — Theory

Copy Constructor: A constructor that creates a new object as a copy of an existing one.
Example:

```
public Point(Point other)
{
    X = other.X;
    Y = other.Y;
}
```

LinkedIn Article on Constructors:
Constructor and Its Types in C# — LinkedIn

Indexer:
A special property that allows accessing elements in an object like an array.

```
class Sample
{
    private string[] data = new string[10];
    public string this[int index]
    {
        get => data[index];
        set => data[index] = value;
    }
}
```

Business case: Used in collections, database row objects, configuration lookups.

Keywords learned last lecture (likely list):

- `struct`, `class`, `public`, `private`, `internal`, `protected`, `readonly`, `override`, `this`, `new`, `value type`, `reference type`, `constructor`, `method`, `property`, `ToString()`, `interface`, `encapsulation`.

Self

1) Can a Constructor Be `private`? (Rare Business Case)

Yes, a constructor can be `private` in both classes and structs, but the implications differ:

In a Class

A private constructor **prevents external instantiation**.
Typical uses:

- **Singleton Pattern** → Only one instance allowed.
- **Factory Pattern** → Creation is controlled via static methods.
- **Static Classes** → C# automatically makes constructors private so they can't be instantiated.

Example (Singleton):

```
public class Logger
{
    private static readonly Logger _instance = new Logger();

    private Logger() { } // Private to prevent new Logger()

    public static Logger Instance => _instance;
}
```

In a Struct

- In **.NET Framework / .NET 5 and earlier**, **structs cannot have an explicit parameterless constructor** (private or public). They always have an **implicit public default constructor** that sets fields to default values.
- You *can* have a private parameterized constructor in a struct, which is useful when you want controlled initialization.

Rare business case for **private struct constructor**:

- Enforcing certain initialization rules internally in a library (e.g., avoid partially initialized structs when working with performance-critical code like `System.DateTime`).

---

2) BCL → Override `ToString()`

BCL (Base Class Library) includes many overridden `ToString()` implementations. Examples:

- `System.DateTime.ToString()` → returns a formatted date/time string.
- `System.Decimal.ToString()` → converts number to string in current culture format.
- `System.Guid.ToString()` → returns canonical GUID string.

Why BCL overrides `ToString()`:

- Default `ToString()` in `object` just returns type name (e.g., `"Namespace.Type"`), which is useless in most scenarios.
- Overriding gives **readable, domain-specific output**.

Example:

```
DateTime now = DateTime.Now;
Console.WriteLine(now); // "8/14/2025 10:15:42 AM" (formatted)
```

---

3) Struct Constructor Restrictions (.NET 5 vs .NET 6)

.NET 5 and earlier

- You **cannot** define your own **parameterless constructor** in a struct.
- Reason: CLR (runtime) always generates a public parameterless constructor for structs, zero-initializing all fields.
- This was done for **performance** and **predictability** — structs are value types, and the runtime could allocate them without calling user code.

.NET 6 and later

- **You can** define **parameterless constructors** in structs, and also **field initializers**.
- Reason for change: Developers wanted **immutable structs** with controlled default values without requiring a static factory method.
- CLR and JIT were updated to support calling a user-defined parameterless constructor.

Example (.NET 6+):

```
public struct Point
{
    public int X { get; set; } = 10; // field initializer allowed
    public int Y { get; set; } = 20;

    public Point() // now legal
    {
        // Still must initialize all fields if no default values
    }
}
```

Now:

```
Point p = new(); // Calls your constructor, not just zero-init
```