Part 01

◆ Problem 1: Car Class with Constructors

```
using System;

class Car
{
    public int Id { get; set; }
    public string Brand { get; set; }
    public double Price { get; set; }

    // 1. Default constructor
    public Car()
    {
        Id = 0;
        Brand = "Unknown";
        Price = 0.0;
    }

    // 2. Constructor with one parameter
    public Car(int id)
    {
        Id = id;
        Brand = "Unknown";
        Price = 0.0;
    }

    // 3. Constructor with two parameters
    public Car(int id, string brand)
    {
        Id = id;
        Brand = brand;
        Price = 0.0;
    }

    // 4. Constructor with all three parameters
    public Car(int id, string brand, double price)
    {
        Id = id;
        Brand = brand;
        Price = price;
    }

    public override string ToString()
    {
        return $"Car[Id={Id}, Brand={Brand}, Price={Price}]";
    }
}

class Program
{
    static void Main()
    {
        Car c1 = new Car();
        Car c2 = new Car(1);
```

```
        Car c3 = new Car(2, "Toyota");
        Car c4 = new Car(3, "BMW", 50000);

        Console.WriteLine(c1);
        Console.WriteLine(c2);
        Console.WriteLine(c3);
        Console.WriteLine(c4);
    }
}
```

✅ **Question:** Why does defining a custom constructor suppress the default constructor in C#?
👉 Because C# does **not** automatically generate a parameterless constructor if **any** constructor is explicitly defined. You must define it manually if you still want one.

🔹 Problem 2: Calculator with Overloaded Methods

```
class Calculator
{
    public int Sum(int a, int b) => a + b;
    public int Sum(int a, int b, int c) => a + b + c;
    public double Sum(double a, double b) => a + b;
}

class Program
{
    static void Main()
    {
        Calculator calc = new Calculator();
        Console.WriteLine(calc.Sum(2, 3));
        Console.WriteLine(calc.Sum(1, 2, 3));
        Console.WriteLine(calc.Sum(2.5, 3.5));
    }
}
```

✅ **Question:** How does method overloading improve code readability and reusability?
👉 It allows using the **same method name** with different signatures, so you don't need different names (`SumInt`, `SumDouble`). This makes the code cleaner and easier to understand.

🔹 Problem 3: Constructor Chaining with Inheritance

```
class Parent
{
    public int X { get; set; }
    public int Y { get; set; }

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

```
}

class Child : Parent
{
    public int Z { get; set; }

    public Child(int x, int y, int z) : base(x, y)
    {
        Z = z;
    }
}
```

✅ **Question:** What is the purpose of constructor chaining in inheritance?
👉 It ensures that the **base class fields are initialized properly** before executing the child's constructor.

🔹 Problem 4: Method Overriding (`new` vs `override`)

```
class Parent
{
    public int X { get; set; }
    public int Y { get; set; }

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

    public virtual int Product() => X * Y;
}

class Child : Parent
{
    public int Z { get; set; }

    public Child(int x, int y, int z) : base(x, y) => Z = z;

    // Hides the parent method
    public new int Product() => X * Y * Z;

    // Proper override
    public override int Product() => X * Y * Z;
}
```

✅ **Question:** How does `new` differ from `override`?

- `new` → hides the base method (method chosen depends on reference type).
- `override` → replaces the base method (method chosen depends on actual object type).

🔹 Problem 5: Overriding `ToString()`

```
class Parent
{
    public int X { get; set; }
    public int Y { get; set; }

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

class Child : Parent
{
    public int Z { get; set; }

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

✅ **Question:** Why is `ToString()` often overridden?
👉 To provide a **meaningful string representation** of an object instead of the default class name.

🔹 Problem 6: Interface `IShape`

```
interface IShape
{
    double Area { get; }
    void Draw();
}

class Rectangle : IShape
{
    public double Width { get; set; }
    public double Height { get; set; }

    public Rectangle(double w, double h) { Width = w; Height = h; }

    public double Area => Width * Height;

    public void Draw() => Console.WriteLine("Drawing Rectangle");
}
```

✅ **Question:** Why can't you create an instance of an interface?
👉 Because an interface only defines **contracts** (methods, properties), not implementation.

🔹 Problem 7: Default Interface Methods (C# 8.0)

```
interface IShape
{
    double Area { get; }
    void Draw();

    // Default implementation
    void PrintDetails() => Console.WriteLine($"Area = {Area}");
}
```

```
class Circle : IShape
{
    public double Radius { get; set; }

    public Circle(double r) { Radius = r; }

    public double Area => Math.PI * Radius * Radius;
    public void Draw() => Console.WriteLine("Drawing Circle");
}
```

✅ **Question:** Benefits of default interface implementations?
👉 Allows **backward compatibility** and code reuse without breaking existing implementations.

🔹 Problem 8: Interfaces with Polymorphism

```
interface IMovable
{
    void Move();
}

class Car : IMovable
{
    public void Move() => Console.WriteLine("Car is moving");
}

class Program
{
    static void Main()
    {
        IMovable obj = new Car();
        obj.Move();
    }
}
```

✅ **Question:** Why use an interface reference?
👉 To achieve **polymorphism**—you can switch implementations (Car, Bike, etc.) without changing client code.

🔹 Problem 9: Multiple Interfaces

```
interface IReadable { void Read(); }
interface IWritable { void Write(); }

class File : IReadable, IWritable
{
    public void Read() => Console.WriteLine("Reading file...");
    public void Write() => Console.WriteLine("Writing file...");
}
```

✅ **Question:** How does C# overcome single inheritance limitation?
👉 By allowing **multiple interface implementations** (but only one base class).

🔹 Problem 10: Virtual vs Abstract

```
abstract class Shape
{
    public virtual void Draw() => Console.WriteLine("Drawing Shape");
    public abstract double CalculateArea();
}

class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }

    public Rectangle(double w, double h) { Width = w; Height = h; }

    public override void Draw() => Console.WriteLine("Drawing Rectangle");
    public override double CalculateArea() => Width * Height;
}
```

✅ **Question:** Difference between virtual and abstract?

- `virtual` → provides a **default implementation**, can be overridden.
- `abstract` → has **no implementation**, must be implemented by subclasses.

📌 Part 02

🔹 Difference between Class and Struct in C#

Feature	Class (Reference Type)	Struct (Value Type)
Memory location	Heap	Stack (mostly)
Default ctor	Allowed	Not allowed
Inheritance	Supports	Doesn’t support
Null assignment	Can be `null`	Cannot be `null` (unless `Nullable<T>`)
Performance	Slightly slower	Faster (no heap allocation)

🔹 Relations Between Classes (other than inheritance)

1. **Association** → General relationship (e.g., Teacher ↔ Student).
2. **Aggregation** → "Has-a" weaker relationship (e.g., Library has Books).
3. **Composition** → "Has-a" stronger relationship (e.g., Car has Engine).
4. **Dependency** → "Uses-a" relationship (e.g., Printer depends on Document).

Static Binding (Early Binding / Compile-Time Binding)

- The decision about **which method or property to call** is made by the **compiler at compile time**.
- Happens when the method is **non-virtual** or when **method overloading** is used.
- It is **faster** since no runtime lookup is needed.

✅ **Example (Static Binding via Overloading):**

```
class Calculator
{
    public void Print(int x) => Console.WriteLine("Integer: " + x);
    public void Print(string s) => Console.WriteLine("String: " + s);
}

class Program
{
    static void Main()
    {
        Calculator c = new Calculator();
        c.Print(10);       // Compiler binds to Print(int)
        c.Print("Hello");  // Compiler binds to Print(string)
    }
}
```

Here, the compiler **already knows** which method to call → **Static Binding**.

🔹 Dynamic Binding (Late Binding / Runtime Binding)

- The decision about **which method to call** is made by the **CLR at runtime**.
- Happens when a method is marked as **virtual** and **overridden** in a derived class.
- Enables **runtime polymorphism**.

✅ **Example (Dynamic Binding via Overriding):**

```
class Parent
{
    public virtual void Show() => Console.WriteLine("Parent Show");
}

class Child : Parent
{
    public override void Show() => Console.WriteLine("Child Show");
}

class Program
{
    static void Main()
    {
        Parent obj = new Child();
        obj.Show();  // Resolved at runtime → calls Child.Show()
    }
```

```
}
```

Here, the **reference type is `Parent`**, but the **actual object is `Child`**, so runtime decides which method to call → **Dynamic Binding**.

📌 Key Differences

Feature	Static Binding (Early)	Dynamic Binding (Late)
Decision made at	Compile-time	Runtime
Used with	Overloading, non-virtual methods	Virtual & overridden methods
Speed	Faster	Slightly slower
Supports polymorphism	❌ No	✅ Yes

👉 In short:

- **Static Binding** = Compiler decides method at compile time.
- **Dynamic Binding** = CLR decides method at runtime (polymorphism).