1) Why does defining a custom constructor suppress the default constructor in C#?

In C#, the compiler automatically generates a **default constructor** (parameterless) if you don't define any constructor. Once you define a **custom constructor**, the compiler assumes you want full control over object initialization, so it no longer generates the default constructor automatically. If needed, you must define it explicitly.

---

2) How does method overloading improve code readability and reusability?

Method overloading improves readability by allowing multiple versions of a method with the same name but different parameter lists, instead of creating separate confusing method names. It also enhances reusability since you can apply the same concept (like Sum) to different data types or numbers of parameters without duplicating logic.

---

3) What is the purpose of constructor chaining in inheritance?

Constructor chaining ensures that the base class initializes its own members before the derived class adds its initialization. This prevents code duplication, guarantees proper setup of inherited members, and maintains a consistent object state across the inheritance hierarchy.

---

4) How does new differ from override in method overriding?

- new: **Hides** the base class method. If you call the method through a base class reference, the base class version is executed.
- override: **Redefines** the base class method. Even if you call it through a base class reference, the derived version is executed.

---

5) Why is ToString() often overridden in custom classes?

By default, `ToString()` only returns the class name, which is not informative. Overriding `ToString()` provides a meaningful string representation of an object's state (e.g., its properties), making debugging and logging easier and the output more readable.

---

6) What is the difference between class and struct in C#?

- **Class**: A **reference type**, stored on the heap, supports inheritance, and is accessed via references.
- **Struct**: A **value type**, usually stored on the stack, does not support inheritance (except for interfaces), and is lightweight, making it better for small data structures.

Part03 Bonus

◈ Static Binding (Early Binding)

- **Definition**: The method call is resolved **at compile time**.
- **How it works**: The compiler knows exactly which method will be executed.
- **Examples**:
  - **Method overloading** (same method name with different parameter lists).
  - Methods **without** **`virtual`/`override`** (normal methods, including `new` keyword hiding).
- **Advantages**:
  - Faster execution (no runtime lookup).
  - Less overhead.
- **Disadvantages**:
  - Less flexible (cannot decide behavior based on object type at runtime).

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

Parent p = new Child();
p.Show();   // Output: Parent  (decided at compile time)
```

🔷 Dynamic Binding (Late Binding)

- **Definition**: The method call is resolved **at runtime**.
- **How it works**: The actual object type decides which method to execute.
- **Examples**:
    - **Method overriding** using `virtual` and `override`.
    - **Polymorphism** scenarios.
- **Advantages**:
    - More flexible (allows runtime polymorphism).
    - Makes code extensible.
- **Disadvantages**:
    - Slightly slower due to runtime method lookup.

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

Parent p = new Child();
p.Show();   // Output: Child  (decided at runtime)
```