**Task07\_C#**

**Part01**

**Problem 1**

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

the compiler automatically provides a **default parameterless constructor** if no constructors are explicitly defined in a class. However, once a programmer defines a **custom constructor**, the compiler no longer generates the default one.

This happens because by defining a constructor, the programmer explicitly controls how object initialization should occur. Allowing the compiler to still insert a default constructor could lead to ambiguity or unintended behavior. Therefore, if both a custom and a default constructor are needed, the default constructor must be **explicitly declared** by the programmer.

public class Car

{

public int Id;

public string Brand;

// Custom constructor

public Car(int id, string brand)

{

Id = id;

Brand = brand;

}

class Program

{

static void Main()

{

// This will cause an error, because no default constructor exists:

// Car c1 = new Car(); // can be default instructor

// Must use the custom constructor:

Car c2 = new Car(101, "BMW");

Console.WriteLine($"Id = {c2.Id}, Brand = {c2.Brand}");

}

}

**Problem 2**

**How does method overloading improve code readability and reusability?**

**Method overloading** improves both **readability** and **reusability** in several ways:

**Improved readability**:

* Instead of creating multiple methods with different names for similar operations, overloading allows the same method name to be reused with different parameter lists.
* This makes the code easier to read and understand, since the method name clearly describes the operation, and the parameters clarify how it is applied.

Example:

int Add(int a, int b);

double Add(double a, double b);

All methods perform an “addition-like” operation but on different data types.

**Improved reusability**:

* Overloading lets developers reuse the same method logic in different contexts without duplicating code under new names.
* It reduces redundancy and provides flexibility, since the same operation can be adapted for multiple data types or parameter combinations

**Problem 3**

**What is the purpose of constructor chaining in inheritance?**

The purpose of **constructor chaining in inheritance** is to ensure that:

1. **Each class in the inheritance hierarchy properly initializes its own fields**:
   * When a derived class is created, the **base class constructor must run first** to set up the base class's part of the object.
2. **Code reuse and maintainability**:
   * Instead of duplicating initialization logic in every derived class, constructor chaining lets you reuse the base class constructor to handle base class initialization.
3. **Consistency and reliability**:
   * It guarantees that all necessary setup in base classes is performed **before** the derived class adds its own initialization, ensuring a fully constructed and valid object.

Example:

public class Parent

{

public int X;

public Parent(int x)

{

X = x;

}

}

public class Child : Parent

{

public int Y;

public Child(int x, int y) : base(x) // chaining to Parent constructor

{

Y = y;

}

}

**Problem 4**

**How does new differ from override in method overriding?**

both new and override are used when a derived class defines a method with the same name as one in the base class, but they work differently:

1. **override**
   * Used to **extend or modify** the behavior of a method that is marked as virtual (or abstract) in the base class.
   * Supports **polymorphism** → which means the method that gets executed is decided at **runtime**, depending on the actual object type.
   * Requires the base method to be virtual, abstract, or already override.
2. **new**
   * Used to **hide** the base class method (method hiding), not override it.
   * Breaks polymorphism → the method that gets executed is decided at **compile time**, based on the **reference type**.
   * Does not require the base method to be virtual.

**Problem 5**

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

every class inherits the ToString() method from the base Object class. By default, this method returns the class name, which is usually not meaningful for developers or users.

**Overriding ToString() in custom classes is important because:**

1. **Improves readability and debugging**
   * It provides a human-friendly description of the object’s state instead of just the class name.
   * This is especially useful when printing objects in logs or debugging sessions.
2. **Better representation of data**
   * You can display the most relevant properties of the class in a concise and meaningful format.
3. **Increases usability**
   * When objects are passed to methods like Console.WriteLine() or string concatenation, the overridden ToString() gives clear and useful output without needing extra formatting code.

**Part02**

**Problem 1**

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

1. **Type Category**

* **Class** → **Reference type** (stored on the heap, variable holds a reference).
* **Struct** → **Value type** (stored on the stack, variable holds the actual data).

1. **Memory Allocation**

* **Class**: When you assign one class variable to another, they both point to the same object in memory.
* **Struct**: When you assign one struct variable to another, a copy of the value is made.

1. **Inheritance**

* **Class:** Supports inheritance (can inherit from another class, implement interfaces).
* **Struct:** Cannot inherit from another struct or class (but can implement interfaces).

1. **Default Constructor**

* **Class**: You can define custom constructors (including parameterless ones).
* **Struct**: Always has a default parameterless constructor (cannot explicitly define one yourself).

1. **Nullability**

* **Class**: Being reference types, they can be null.
* **Struct**: Being value types, they cannot be null (unless defined as Nullable<T> or T?).

1. **Use Cases**

* **Class**: Being reference types, they can be null.
* **Struct**: Being value types, they cannot be null (unless defined

1. **Access Modifiers**

* **Class :** allow all access modifiers
* **Struct :** allow (public,private,internal)

1. **Default Constructor**

* **Class** : create default constructor empty parmater

Public Name\_Class(){}

Process is initiated by new when create object

Name\_Class obj = new Name\_Class()

* **Struct** : create default constructor initialize all parameters

In default value Public Name\_Struct(int x , string y){

X = 0;

Y = null ;

}

**Problem 2**

**If inheritance is relation between classes clarify other relations between classes?**

**Yes**

1. **Association** – A general connection between two classes, showing that one class is aware of or interacts with another. For example, a *Teacher* teaches a *Student*.
2. **Aggregation** – A special form of association that represents a "has-a" relationship, where one class contains another but both can exist independently. Example: a *Department* has *Professors*.
3. **Composition** – A stronger form of aggregation that also represents a "has-a" relationship, but here the lifetime of the contained object depends on the container. Example: a *Car* has an *Engine*. If the car is destroyed, the engine no longer exists.
4. **Dependency** – A temporary relationship where one class uses another to perform a task. Example: a *Report Generator* depends on a *Printer* to print a report.