Question Answering Assignment

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

1. Type Category

- **Class**: Reference type, stored on the heap. When you assign a class instance to a new variable or pass it to a method, a reference to the original object is passed (not a copy).
- **Struct**: Value type, typically stored on the stack (though it can be on the heap if part of another object). Assigning a struct to a new variable or passing it to a method creates a copy of the struct.

2. Memory Allocation

- **Class**: Allocated on the heap, managed by the garbage collector.
- **Struct**: Allocated on the stack (or inline within other objects), not directly managed by the garbage collector, leading to potentially better performance for small, short-lived objects.

3. Default Values

- **Class**: A class variable is null if uninitialized (since it's a reference type).
- **Struct**: A struct cannot be null unless explicitly made nullable (e.g., struct?). All fields are initialized to their default values (e.g., 0 for numbers, false for booleans).

4. Inheritance

- **Class**: Supports inheritance and can inherit from other classes (single inheritance) or implement interfaces. All classes implicitly inherit from System.Object.

- **Struct**: Does not support inheritance (cannot inherit from another struct or class) but can implement interfaces. Structs implicitly inherit from System.ValueType (which itself inherits from System.Object).

5. Default Constructor

- **Class**: Can have parameterless constructors defined explicitly. If none is defined, a default parameterless constructor is provided.
- **Struct**: Cannot have an explicit parameterless constructor. The compiler provides a default constructor that initializes all fields to their default values. You can define constructors with parameters, but they must assign all fields.

6. Nullability

- **Class**: Can be assigned null.
- **Struct**: Cannot be null unless explicitly declared as a nullable value type (e.g., int? or Nullable<T>).

7. Copy Behavior

- **Class**: Assignment copies the reference, so both variables point to the same object. Modifying one affects the other.
- **Struct**: Assignment copies the entire struct, creating an independent copy. Modifying one does not affect the other.

8. Use Cases

- **Class**: Best for larger objects, objects with shared state, or when inheritance/polymorphism is needed. Common for complex entities like business objects (e.g., Customer, Order).
- **Struct**: Best for small, lightweight data structures with value semantics, like Point, DateTime, or Rectangle. Ideal when you want immutability or copy-by-value behavior.

9. Performance Considerations

- **Class**: Heap allocation and garbage collection can introduce overhead, but reference semantics are useful for shared state.
- **Struct**: Stack allocation can be faster for small objects, but passing large structs by value can be expensive due to copying.

10. Mutability

- **Class**: Typically mutable, though you can design immutable classes.
- **Struct**: Preferably immutable (as recommended by Microsoft) to avoid confusion with value semantics, but mutable structs are allowed (though changes to a copy don't affect the original).

2. If inheritance is relation between classes clarify other relations between classes

In object-oriented programming (OOP) in C#, **inheritance** is one type of relationship between classes, where one class (derived) inherits properties and behaviors from another class (base). However, there are other important relationships between classes that define how they interact or depend on each other.

1. Association

- **Definition**: A general relationship where one class uses or interacts with another class, typically through references or method calls. It represents a "has-a" or "uses-a" relationship, often implemented via fields, properties, or method parameters.

- **Characteristics**:
  - Loose coupling: Classes are independent, and the relationship is not necessarily permanent.
  - Can be **one-to-one**, **one-to-many**, or **many-to-many**.
  - No ownership is implied (unlike composition or aggregation).

- **Example**:

```
public class Teacher
{
   public string Name { get; set; }
}

public class Course
{
   public string CourseName { get; set; }
   public Teacher Instructor; // Association: Course uses Teacher
}
```

  - A Course has a reference to a Teacher, but they are independent entities.

2. Composition

- **Definition**: A strong "has-a" relationship where one class (the container) owns another class (the contained). The contained object's lifecycle is tied to the container's lifecycle—if the container is destroyed, so is the contained object.
- **Characteristics**:
  - Strong ownership: The contained object cannot exist without the container.
  - Typically implemented by creating the contained object within the container class.
- **Example**:

```
public class Engine
{
   public void Start() { /* Logic */ }
}

public class Car
```

```
{
   private Engine engine = new Engine(); // Composition: Car owns
Engine
   public void Drive() { engine.Start(); }
}
```

    - The Engine is created and destroyed with the Car. If the Car is destroyed, the Engine is too.

3. Aggregation

- **Definition**: A weaker "has-a" relationship where one class contains another, but the contained object can exist independently of the container. It's a form of association with a "whole-part" relationship.
- **Characteristics**:
    - Loose coupling: The contained object's lifecycle is not controlled by the container.
    - Often implemented by passing the contained object via constructor or setter.
- **Example**:

```
public class Department
{
   public string Name { get; set; }
}

public class University
{
   private List<Department> departments = new List<Department>(); //
Aggregation
   public void AddDepartment(Department dept) {
departments.Add(dept); }
}
```

    - A Department can exist independently of a University (e.g., it could belong to another university or none).

4. Dependency

- **Definition**: A relationship where one class depends on another temporarily, typically through method parameters or local variables, without holding a reference as a field.
- **Characteristics**:
  - Weakest form of relationship: The dependency is transient and exists only during method execution.
  - Indicates that one class uses another but doesn't own or contain it.
- **Example**:

```
public class Logger
{
    public void Log(string message) { /* Log message */ }
}

public class OrderProcessor
{
    public void ProcessOrder(Order order, Logger logger) // Dependency
    {
        logger.Log("Processing order...");
    }
}
```

  - OrderProcessor uses Logger only during the ProcessOrder method call, without storing it.

5. Realization (Interface Implementation)

- **Definition**: A relationship where a class implements an interface, defining a contract that the class must follow. It's a "can-do" relationship, specifying behavior without sharing implementation.
- **Characteristics**:

    - Unlike inheritance, realization doesn't involve inheriting implementation, only agreeing to fulfill a contract.
    - Promotes loose coupling and polymorphism.
- **Example**:

```
public interface IPrintable
{
   void Print();
}

public class Document : IPrintable
{
   public void Print() { /* Print logic */ }
}
```

    - Document realizes the IPrintable interface, promising to implement the Print method.