### Part02

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

The key differences between **class** and **struct** in C# lie in their behavior, memory allocation, and usage. Here's an explanation:

**1. Type**

* **Class**: A reference type, meaning it is stored on the heap, and variables store a reference to the memory location.
* **Struct**: A value type, meaning it is stored on the stack, and variables store the actual data.

**2. Inheritance**

* **Class**: Supports inheritance (can inherit from another class or be a base class).
* **Struct**: Does not support inheritance from other structs or classes (except implicit inheritance from System.ValueType).

**3. Default Constructor**

* **Class**: Can have a parameterless constructor (explicitly or implicitly created by the compiler).
* **Struct**: Cannot have an explicitly defined parameterless constructor; a default constructor is implicitly provided by the compiler to initialize fields to their default values.

**4. Memory Allocation**

* **Class**: Stored on the heap, managed by the garbage collector, suitable for large and complex objects.
* **Struct**: Stored on the stack, automatically deallocated, suitable for small, lightweight data structures.

**5. Performance**

* **Class**: Reference-type behavior may lead to performance overhead due to heap allocation and garbage collection.
* **Struct**: Value-type behavior can provide better performance in certain scenarios due to stack allocation and no need for garbage collection.

**6. Nullability**

* **Class**: Can be assigned null since it is a reference type.
* **Struct**: Cannot be assigned null unless declared as a nullable type (e.g., int?).

**7. Usage**

* **Class**: Used for objects with complex behaviors and relationships, and for scenarios requiring polymorphism.
* **Struct**: Used for lightweight, immutable data types like coordinates, points, or color values.

In essence, **classes** are designed for complex, dynamic objects, while **structs** are better suited for simple, efficient, and immutable data representations.

### If inheritance is relation between classes clarify other relations between classes

In addition to **inheritance**, which defines an **"is-a" relationship** between classes, there are several other types of relationships between classes in object-oriented programming. Here’s an explanation of these relationships:

**1. Association (Uses-A Relationship)**

* **Definition**: Represents a general connection between two classes where one class **uses** or interacts with another.
* **Example**: A Person and an Address class where a Person uses an Address object but does not own it.
* **Usage**: Objects work together temporarily, but their lifetimes are independent.

class Person

{

public void DisplayAddress(Address address)

{

Console.WriteLine(address.City);

}

}

**2. Aggregation (Has-A Relationship)**

* **Definition**: A specialized form of association where one class **has** or contains another class as a part. The contained object can exist independently of the container.
* **Example**: A Car class has an Engine object, but the Engine can exist without the Car.
* **Usage**: Models a whole-part relationship with shared ownership.

class Car

{

public Engine Engine { get; set; }

}

**3. Composition (Owns-A Relationship)**

* **Definition**: A stricter form of aggregation where one class **owns** or contains another class, and the contained object cannot exist independently of the container.
* **Example**: A House class contains Room objects, and if the House is destroyed, the Rooms no longer exist.
* **Usage**: Models a whole-part relationship with exclusive ownership.

class House

{

private List<Room> rooms = new List<Room>();

public void AddRoom(Room room)

{

rooms.Add(room);

}

}

**4. Dependency (Depends-On Relationship)**

* **Definition**: Represents a weaker relationship where one class depends on another to fulfill a task but does not store it as a member.
* **Example**: A Printer class depends on a Document class to print.
* **Usage**: Used for loose coupling and flexibility in code.

class Printer

{

public void Print(Document doc)

{

Console.WriteLine(doc.Content);

}

}