In C#, the main differences between **class** and **struct** are as follows:

**1. Memory Allocation:**

* **Class**: A class is a reference type. Objects of a class are stored on the **heap**, and a reference to the object is stored on the **stack**.
* **Struct**: A struct is a value type. It is stored directly on the **stack** (or inline in other structures or arrays), and copying a struct creates a full copy of the data.

**2. Default Behavior:**

* **Class**: A class can have a **default constructor**, and its fields are initialized to **default values** (e.g., null for reference types, 0 for numeric types).
* **Struct**: A struct **cannot** have a default constructor (unless it’s a parameterless constructor in C# 10+), and its fields are automatically initialized to **default values**.

**3. Inheritance:**

* **Class**: Classes support **inheritance**. A class can inherit from another class and implement interfaces.
* **Struct**: Structs **cannot** inherit from another struct or class, but they can implement interfaces. In other words, structs do not support **class inheritance**.

**4. Nullability:**

* **Class**: A class object can be **null** because it is a reference type.
* **Struct**: A struct cannot be **null** unless it is defined as a **nullable struct** (e.g., int?).

**5. Performance:**

* **Class**: Classes involve more overhead because of heap allocation and garbage collection.
* **Struct**: Structs can be more efficient when dealing with small, lightweight data because they avoid heap allocation and garbage collection, especially when used in collections or as local variables.

**Other Relations Between Classes:**

In object-oriented programming, inheritance is one way classes can relate, but there are other key relations:

1. **Association**:
   * Represents a relationship where one class uses or has a reference to another class.
   * Example: A **Car** class may have an association with a **Wheel** class, meaning the car "uses" wheels.
2. **Aggregation**:
   * A form of association where one class **owns** or is made up of other objects, but the lifetime of the contained objects is not strictly dependent on the lifetime of the container.
   * Example: A **Library** class may aggregate **Book** objects. If the library is destroyed, the books may still exist elsewhere.
3. **Composition**:
   * A stronger form of association than aggregation. In composition, if the containing object is destroyed, the contained objects are also destroyed.
   * Example: A **House** class might have a **Room** class. If the house is destroyed, the rooms are also destroyed.
4. **Dependency**:
   * A class depends on another if it uses its functionality, typically through methods or properties.
   * Example: A **Client** class might depend on a **Server** class to retrieve data, meaning the client cannot function without the server.
5. **Interface Implementation**:
   * A class can implement an **interface**, which is a contract that defines methods and properties but doesn't provide any implementation.
   * Example: A **Dog** class might implement an **IAnimal** interface, guaranteeing that it provides implementations for the methods defined in IAnimal.

In summary:

* **Classes** support inheritance, but **structs** do not.
* **Relations between classes** include association, aggregation, composition, dependency, and interface implementation.

The bonus

**Self-Study Report 2: Static vs Dynamic Binding in C#**

**Static Binding**:

* **Definition**: Static binding (also called early binding) occurs at **compile-time**. The method or property that will be called is determined based on the type of the object at compile-time, not runtime.
* **Example**: In method overloading, if the method signature is clear, the compiler determines which method to call before the program runs.
* class Example
* {
* public void Display(int x) { Console.WriteLine("Integer: " + x); }
* public void Display(string x) { Console.WriteLine("String: " + x); }
* }

In this case, when you call Display(10) or Display("Hello"), the compiler knows which method to use at compile time because of the method signature (the parameter type).

* **Key Points**:
  + Happens at compile-time.
  + The method or property is selected by the compiler.
  + Usually applies to method overloading or when accessing properties and methods of a class.

**Dynamic Binding**:

* **Definition**: Dynamic binding (also called late binding) occurs at **runtime**. The method or property to be called is determined at runtime, typically in the case of polymorphism.
* **Example**: In method overriding, the decision about which method to call is made based on the **actual object type** at runtime.
* class Animal
* {
* public virtual void Speak() { Console.WriteLine("Animal speaks"); }
* }
* class Dog : Animal
* {
* public override void Speak() { Console.WriteLine("Dog barks"); }
* }
* class Program
* {
* static void Main()
* {
* Animal animal = new Dog();
* animal.Speak(); // This will call Dog's Speak method at runtime
* }
* }

In this example, even though the reference type is Animal, at runtime, the Speak() method of the Dog class is called because the actual object is of type Dog.

* **Key Points**:
  + Happens at runtime.
  + The method to be called is determined based on the actual object type.
  + Used in polymorphism and method overriding.

**Comparison**:

* **Static Binding** is faster because it occurs at compile-time, and the compiler can optimize method calls.
* **Dynamic Binding** is more flexible because it allows for behavior to be determined at runtime, making it essential for polymorphic behavior.

**Summary**:

* **Static Binding**: Compile-time, determined by the method signature (e.g., method overloading).
* **Dynamic Binding**: Runtime, determined by the actual object type (e.g., method overriding with polymorphism).