Struct vs Class in C#

Introduction to Structs and Classes in C#

1. Structs:

- **Definition**: A `struct` (structure) is a **value type** in C#. It stores data directly in memory, making it lightweight and efficient for small data structures.
- **Usage**: Typically used for small, immutable data structures like points, coordinates, or simple data containers.

2. Classes:

- **Definition**: A `class` is a **reference type** in C#. It stores data in the heap memory, with variables referencing the memory address.
- **Usage**: Commonly used for complex data structures that may be large, mutable, or require inheritance.

Key Differences Between Struct and Class

Feature	Struct	Class
Type	Value type	Reference type
Memory Allocation	Stored on stack	Stored on heap
Inheritance	Does not support inheritance	Supports inheritance
Default Constructor	Cannot define parameterless constructor	Can define parameterless constructor
Performance	Faster for small data structures	More efficient for larger objects
Nullability	Cannot be `null` (unless nullable)	Can be `null`
Boxing/Unboxing	May require boxing for interface use	No boxing required

Structs in C#

1. Definition:

- A `struct` is a lightweight, value-based data type that's best suited for small data structures.

2. When to Use Structs:

- For small, **immutable** data structures.
- When **performance** is a priority, as they don't incur the overhead of heap allocation.

3. Example: Creating a Simple `struct` for 2D Points

```
public struct Point
{
        public int X { get; }
        public int Y { get; }
        public Point(int x, int y)
        {
        X = x;
        Y = y;
        }
}
// Usage
Point p1 = new Point(3, 4);
Console.WriteLine($"Point: ({p1.X}, {p1.Y})");
```

Output:

Point: (3, 4)

Classes in C#

1. Definition:

- A `class` is a reference type in C# and allows for more complex behavior like inheritance and polymorphism.

2. When to Use Classes:

- For complex data structures or objects that may be **large or mutable**.
- When you need **inheritance** and **polymorphism**.

3. Example: Creating a Simple `class` for a Student

```
public class Student
{
        public string Name { get; set; }
        public int Age { get; set; }
        public Student(string name, int age)
        {
        Name = name;
        Age = age;
        }
        public void DisplayInfo()
        {
        Console.WriteLine($"Student: {Name}, Age: {Age}");
        }
}
// Usage
Student s1 = new Student("Alice", 20);
s1.DisplayInfo();
```

Output:

Student: Alice, Age: 20

Struct vs Class in Action: A Memory Comparison

1. Memory Allocation:

- **Structs**: Stored on the **stack**, making them faster for small, short-lived objects.
- **Classes**: Stored on the **heap**, providing more flexibility but with a performance cost for larger, complex objects.

2. Example of Memory Behavior:

- **Struct Example**:

```
Point p1 = new Point(3, 4);

Point p2 = p1; // Value copy

p2.X = 10;

Console.WriteLine(p1.X); // Output: 3

Console.WriteLine(p2.X); // Output: 10
```

Class Example:

Student s1 = new Student("Alice", 20);

Student s2 = s1; // Reference copy

s2.Name = "Bob";

Console.WriteLine(s1.Name); // Output: Bob

Console.WriteLine(s2.Name); // Output: Bob

Explanation:

- **Struct**: `p1` and `p2` are independent after copying, so changes to `p2` don't affect `p1`.
- **Class**: `s1` and `s2` reference the same object, so changing `s2` affects `s1`.

Choosing Between Struct and Class

Use Struct:

- For small, **lightweight objects** that don’t need inheritance.
- When you need a **value type** with fewer allocations on the heap.

Use Class:

- For **complex** objects or **large datasets** that may need inheritance.
- When object **mutability** and reference sharing are important.

Best Practices:

- **Struct**: Favor immutability to avoid unintended behavior.
- **Class**: Favor for objects that represent complex entities in applications.