Chapter 8

Structs (Value Types)

Classes are used to implement most objects. Sometimes, however, it may be desirable to create an object that behaves like one of the built-in types (such as int, float, or bool)—one that is cheap and fast to allocate and doesn’t have the overhead of references. In that case, you can use a value type, which is done by declaring a struct in C#.

Structs act similarly to classes, but with a few added restrictions. They can’t inherit from any other type (though they implicitly inherit from object), and other classes can’t inherit from them.¹

A Point Struct

In a graphics system, a value class can be used to encapsulate a point. Here’s how to declare it:

```csharp
using System;
struct Point
{
    public Point(int x, int y)
    {
        m_x = x;
        m_y = y;
    }
    public override string ToString()
    {
        return String.Format("({0}, {1})", m_x, m_y);
    }

    public int m_x;
    public int m_y;
}
class Test
{
    public static void Main()
    {
        Point start = new Point(5, 5);
        Console.WriteLine("Start: {0}", start);
    }
}
```

¹Technically, structs are derived from System.ValueType, but that’s only an implementation detail. From a language perspective, they act like they’re derived from System.Object.
The \( m_x \) and \( m_y \) components of the Point can be accessed. In the \( \text{Main()} \) function, a Point is created using the new keyword. For value types, \text{new} creates an object on the stack and then calls the appropriate constructor.

The call to \text{Console.WriteLine()} is a bit mysterious. If Point is allocated on the stack, how does that call work?

### Boxing and Unboxing

In C# and the .NET Runtime world, a little bit of magic happens to make value types look like reference types, and that magic is called boxing. As magic goes, it’s pretty simple. In the call to \text{Console.WriteLine()}, the compiler is looking for a way to convert start to an object, because the type of the second parameter to \text{WriteLine()} is object. For a reference type (in other words, a class), this is easy, because \text{object} is the base class of all classes. The compiler merely passes an object reference that refers to the class instance.

There’s no reference-based instance for a value class, however, so the C# compiler allocates a reference type “box” for the Point, marks the box as containing a Point, and copies the value of the Point into the box. It is now a reference type, and you can treat it as if it were an object.

This reference is then passed to the \text{WriteLine()} function, which calls the \text{ToString()} function on the boxed Point, which gets dispatched to the \text{ToString()} function, and the code writes the following:

Start: \((5, 5)\)

Boxing happens automatically whenever a value type is used in a location that requires (or could use) an object.

The boxed value is retrieved into a value type by unboxing it.

```csharp
int v = 123;
object o = v;        // box the int 123
int v2 = (int) o;    // unbox it back to an integer
```

Assigning the object \( o \) the value 123 boxes the integer, which is then extracted back on the next line. That cast to \text{int} is required, because the object \( o \) could be any type of object, and the cast could fail.

This code can be represented by Figure 8-1. Assigning the \text{int} to the \text{object} variable results in the box being allocated on the heap and the value being copied into the box. The box is then labeled with the type it contains so the runtime knows the type of the boxed object.

![Figure 8-1. Boxing and unboxing a value type](image)

During the unboxing conversion, the type must match exactly; a boxed value type can’t be unboxed to a compatible type.

```csharp
object o = 15;
short s = (short) o;       // fails, o doesn't contain a short
short t = (short)(int) o;  // this works
```