

**Username:** Pralay Patoria **Book:** Under the Hood of .NET Memory Management. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC 107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

---

## Structs

The struct type is a fundamental, user-defined value type. In C#, this type is defined with the `struct` keyword. It should be used sparingly, and only when the type represents a single value consisting of 16 bytes or more. It should also be immutable, and like all value types, infrequently boxed.

You may wonder about the 16 byte limit, especially since this restriction is not enforced. The reasoning comes from the overhead of creating an object. On a 32-bit machine, 12 bytes are used just for overhead – an 8-byte header and a 4-byte reference. Therefore, when an object does not have at least 16 bytes of data, you may have a very inefficient design. Consider converting such objects to structs which do not have such heavy overhead.

Even though structs can be larger than 16 bytes, we want to be careful with how large they get to be. Structs are created on the stack rather than the heap. Performance is enhanced when moving smaller data structures around on the stack. If the struct represents an atomic value, this is usually not an issue. However, those wishing to gain value semantics on a large object quickly run into memory issues.

Structs inherit from `System.Struct` and cannot be further inherited. This may limit your design options since structs cannot have a custom-defined base class and cannot have any derived classes. A default constructor is provided by the CLR and cannot be custom defined, and field values cannot be initialized. This requires the struct to be valid with an initial state for all of its members.

Even though a struct cannot participate in inheritance, it can implement an interface. You need to be careful doing this, though; type casting a struct to an interface implicitly boxes the struct, which incurs the overhead of boxing and unboxing, and also has an interesting side effect.

If you type cast a struct to an interface and then perform operations on the interface, the original struct is not affected. This can lead to very subtle logic errors and cause much confusion. The problem is that the interface now refers to a "boxed" copy of the original struct, and not to the same bits that the original struct occupied. Consider the example in [Listing 4.29](#), where we define a simple interface and a struct as well as a class that implements the interface.

```
public interface IShape
{
    Point Top { get; set; }
    void Move(Point newLocation);
}

public struct SimpleShape : IShape
{
    public Point Top
    {
        get; set;
    }

    public void Move(Point newLocation)
    {
        Top = newLocation;
    }
}

public class ClassShape : IShape
{
    public Point Top
    {
```

```

    get; set;
}

public void Move(Point newLocation)
{
    Top = newLocation;
}
}

```

**Listing 4.29:** A struct and a class implementing a common interface.

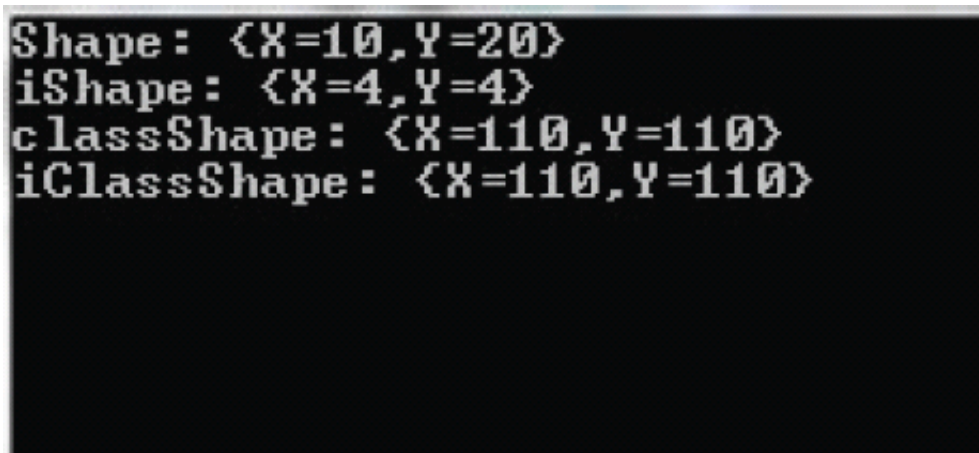
Notice that the interface and the class have the exact same implementation. In the example in [Listing 4.30](#), notice how different the output is.

```

var shape = new SimpleShape();
shape.Top = new Point(10, 20);
var classShape = new ClassShape();
classShape.Top = new Point(40, 50);
IShape iShape = shape as IShape;
iShape.Move(new Point(4, 4));
IShape iClassShape = classShape as IShape;
iClassShape.Move(new Point( 110, 110));
Console.WriteLine("Shape: " + shape.Top);
Console.WriteLine("iShape: " + iShape.Top);
Console.WriteLine("classShape: " + classShape.Top);
Console.WriteLine("iClassShape: " + iClassShape.Top);

```

**Listing 4.30:** Structs are value types and boxing creates a new copy in the reference.



```

Shape: {X=10,Y=20}
iShape: {X=4,Y=4}
classShape: {X=110,Y=110}
iClassShape: {X=110,Y=110}

```

**Figure 4.4:** You would expect `Shape` and `iShape` to have the same values, but they won't.

Because of boxing, `Shape` and `iShape` refer to different memory locations. They now have different values because the processing is done on different entities. On the other hand, with the class, `classShape` and `iClassShape` refer to the same entity, and updates to one affect both. This is the expected behavior for both cases. The unexpected behavior with the struct can lead to subtle logic errors.

This illustrates the importance of immutability for structs. An immutable object is one whose values cannot be modified after creation. For an example of this, look at the example of the `DateTime` struct in [Listing 4.31](#).

```

DateTime today = DateTime.Today;

```

```
DateTime tomorrow = today.AddDays(1);
```

**Listing 4.31:** Structs need to be immutable.

Although `DateTime` has a `Day` property, it is read-only. To add a day to a `DateTime`, the `AddDays()` method must be called, which returns a *new* `DateTime`, while the original `DateTime` retains its value.

This has several advantages, not least of which is that an immutable struct will not be subject to the boxing issue we just saw. A `DateTime` value is thread-safe since different threads will always receive the same value; if a `DateTime` is stored in a `HashTable` or other hash-based collection, there is no danger of its state (and therefore its value) being changed while occupying another value's location. If you have these issues, it would be wise to make your structs immutable.

Listing 4.32 is an example of a `GeographicPoint` class that is not immutable.

```
public partial struct GeographicPoint
{
    private float latitude;
    private float longitude;
    private int altitude;

    public float Latitude
    {
        get { return latitude; }
        set { latitude = value; }
    }

    public float Longitude
    {
        get { return longitude; }
        set { longitude = value; }
    }

    public int Altitude
    {
        get { return altitude; }
        set { altitude = value; }
    }

    public GeographicPoint(float latitude, float longitude, int altitude)
    {
        this.latitude = latitude;
        this.longitude = longitude;
        this.altitude = altitude;
    }
}
```

**Listing 4.32:** `GeographicPoint` is not immutable and, as a struct, will cause problems.

This struct can be made immutable by making its fields read-only and removing the property setters. In other projects, it will be necessary to modify any method mutating the state of the instance so that it no longer does so. That will require refactoring ([Listing 4.33](#)) to use local variables or returning new instances of the struct.

```

public partial struct GeographicPoint
{
    private readonly float latitude;
    private readonly float longitude;
    private readonly int altitude;

    public float Latitude
    {
        get { return latitude; }
    }

    public float Longitude
    {
        get { return longitude; }
    }

    public int Altitude
    {
        get { return altitude; }
    }

    public GeographicPoint(float latitude, float longitude, int altitude)
    {
        this.latitude = latitude;
        this.longitude = longitude;
        this.altitude = altitude;
    }

    public GeographicPoint ChangeLatitude (float newLatitude)
    {
        return new GeographicPoint(latitude, Longitude, Altitude);
    }

    public GeographicPoint ChangeLongitude (float newLongitude)
    {
        return new GeographicPoint(Latitude, newLongitude, Altitude);
    }
}

```

Listing 4.33: Making the `GeographicPoint` struct immutable.

Although this struct is now immutable, it will still have an issue with `Hashtable`s. When struct objects are used as `Hashtable` keys, the lookup operation for the `Hashtable` is slower because of the implementation of the `GetHashCode` method used to perform the lookup. When a struct contains only simple value types (`int`, `short`, etc.) the default implementation of `GetHashCode` inherited from `System.Object` causes most of the keys to be stored in the same location. When multiple keys hash to the same location, the hash table must do collision resolution to find the correct entry. In extreme cases, you may have to search out every entry in the `Hashtable` to confirm that your item is not there. To resolve this problem, we need to override `Object.GetHashCode()` to ensure proper storage. We should also override `Object.Equals()` in the process. The default `Object.Equals()` works, and we could leave it alone. However, it is rather slow because it uses reflection to check each property. By implementing our own `Equals()` method we can optimize it and eliminate reflection and boxing. While we're implementing the overridden instance `Equals()`, we should complete the struct by implementing the other equality operators, `==` and `!=` (see [Listing 4.34](#)).

```

public partial struct GeographicPoint
{

```

```
public static bool operator ==(GeographicPoint first, GeographicPoint second)
{
    return first.Equals(second);
}

public static bool operator !=(GeographicPoint first, GeographicPoint second)
{
    return !first.Equals(second);
}

public override int GetHashCode()
{
    return (int)this.latitude ^ (int)this.longitude ^ this.altitude;
}

public override bool Equals(object obj)
{
    if (obj is GeographicPoint)
    {
        return Equals((GeographicPoint)obj);
    }

    return false;
}

public bool Equals(GeographicPoint other)
{
    return this.latitude == other.latitude
        && this.longitude == other.longitude
        && this.altitude == other.altitude;
}
}
```

**Listing 4.34:** A fully implemented `GeographicPoint` optimized for hashing and safe from boxing errors.

With the struct fully implemented, we now have more useful semantics. We can now ensure `GeographicPoint` is thread-safe, `Hashtable` friendly, and works with equality operators.