Types

The type system in .NET consists of **value** and **reference** types: a value type is a **primitive** or **struct**, and a reference type is a **pointer.** In practical terms, while a value type will simply represent the current value for whatever variable you are dealing with, reference types are a bit more complicated. There are two components to the data in a reference type, specifically, the **reference** and the actual **data.** The reference is stored in the stack, and it contains the memory location (in the heap) where the data is actually stored.

A type itself describes the possible data that it can hold, along with the operations it can perform on that data. Every variable has a type that fully describes it, and it may also have types that provide a partial description, such as an interface.

```
DateTime valueType = DateTime.Now;
IComparable partialDescription = valueType;
List<string> referenceType = new List<string>();
IEnumerable<string> partialDescription2 = referenceType;
```

Listing 4.1: Switching from the type to an interface.

In this example, `valueType` is a `DateTime` and so can do anything that a `DateTime` can do, but `partialDescription` is an `IComparable` and can only do what an `IComparable` can do, even though both variables have the same data and refer to the same location in memory.

Let's take a look at these two main .NET types, and see how their nature and operations can give rise to bugs and memory leaks.

Value types

Value types consist of **simple** types, **enum** types, **struct** types, and **nullable types.** Value types do not support user-defined inheritance, but they do have an inheritance chain as follows:

- `Enums` inherit from `System.Enum` , which inherits from `System.ValueType`
- nullable types are structs, and structs inherit from `System.ValueType` , which inherits from `System.Object` .

Variables of these types maintain their own copies of their data so, when a value is assigned from one variable to another, the data stored in the memory at one location (i.e. the memory allocated to the initial variable) is copied into the memory allocated for the new variable.

```
int x = 8;
int y = x;
Assert.AreEqual(y, 8);
Assert.AreNotSame(x, y);
```

Listing 4.2: Even if value types have the same value they may, in fact, refer to different locations.

The values may be equal, but modifying the value of `y` (e.g. with the `++` operator) will not affect the `x` variable. This is often the source of bugs when developers forget that changes to a value type parameter will not carry over to the calling method.

The exception to the *One variable, one memory location* rule is usage of the `ref` and `out` keywords, which we can see in Listing 4.3.

```
private void Increment(int value)
  {
   value++;
  }

  private void Increment(ref int value)
  {
   value++;
  }

  private void IncrementOut(out int value)
  {
   value = default(int);
   value++;
  }

  [TestMethod]
  public void OutRefTest()
  {
   int x = 7;
   Increment(x);
   Assert.AreEqual(7, x);
   Increment(ref x);
   Assert.AreEqual(8, x);
   IncrementOut(out x);
   Assert.AreEqual(1, x);
  }
```

Listing 4.3: The effects of `out` and `ref` parameters.

The first `Increment(x)` call copies the value into the parameter for the method and, as the value is never returned, the variable `x` remains the same. The second call is to the `ref` version of `Increment.`

The `ref` keyword allows a value to be passed in, modified, and then passed out. This method call works as desired and increments the number. Calling the `ref` function with an initialized variable will result in a compile-time error.

The `out` parameter is another source of confusion for many. It works like the `ref` keyword in that it passes the value out, but the value for the incoming parameter is not copied. The parameter for the method does not need to be initialized before the call, but it must be initialized within the body of the method. This causes the `IncrementOut` method to always return 1. Calling the `out` function with an initialized variable will cause that value to be discarded.

Reference types

Reference types include **class** types, **interface** types, **array** types, and **delegate** types. They differ from value types in that reference types store references to their values, and require memory allocation on the managed heap. Individual variables can reference the same value or object, and perform operations that affect other variables that have the same reference.

```
var person = new Person { FirstName = “Thomas”, LastName = “Jefferson” };
var other = person;
person.LastName = “Paine”;
Assert.AreEqual(“Paine”, other.LastName);
```

Listing 4.4: Two reference values can refer to the same memory location.

In this previous example, the only thing copied during the assignment statement was the reference – the `person` and `other` variables both refer to the same memory location on the heap.

Boxing and unboxing

Value types can be converted to reference types through a process known as **boxing**, and back into value types through **unboxing.** In C#, this is as simple as casting the value type to a corresponding base type: `System.Enum` for `enum` types, and both `System.ValueType` and `System.Object` work for all value types. Boxed `nullable` types will have a null reference if the `HasValue` property is false, and will otherwise perform a bitwise copy of its `Value` property.

```
int x = 8;
// Box
object y = (object)x;
// Unbox
int z = (int)y;
```

Listing 4.5: A simple case of boxing and unboxing.

Boxing is sometimes necessary, but it should be avoided if at all possible, because it will slow down performance and increase memory requirements. For example, when a value type is boxed, a new reference type is created and the value is copied from the value type to the newly created reference type; this takes time and extra memory (a little bit more than twice the memory of the original value type).

Boxing can be avoided by using parameterized classes and methods, which is implemented using generics; in fact this was the *motivation* for adding generics.

```
public void BoxedCall(object value)
{
 // Perform operation on value
}

public void NonboxedCall<T>(T value)
{
 // Perform operation on value
}
```

Listing 4.6: A method signature that may require boxing and one that would not.

Calls to the `BoxedCall` method will perform a boxing operation on value types, and calls to the `NonboxedCall` method will not. With generic inference, the type can be determined at compile time. This will improve the performance of code and will also prevent the creation of an object on the heap to be collected.

```
int x = 7;
BoxedCall(x);
NonboxedCall(x);
```

Listing 4.7: Calling the two methods on a simple value type.

The type parameter for the call to `NonboxedCall` is inferred based on its usage. It could also have been written as in Listing 4.8.

```
int x = 7;
BoxedCall(x);
NonboxedCall<int>(x);
```

Listing 4.8: Calling the two methods on a simple value type without generic type inference.

As long as the compiler can infer the type parameter, leave it out. Specifying it in such cases does not add any clarity and will often create confusion.