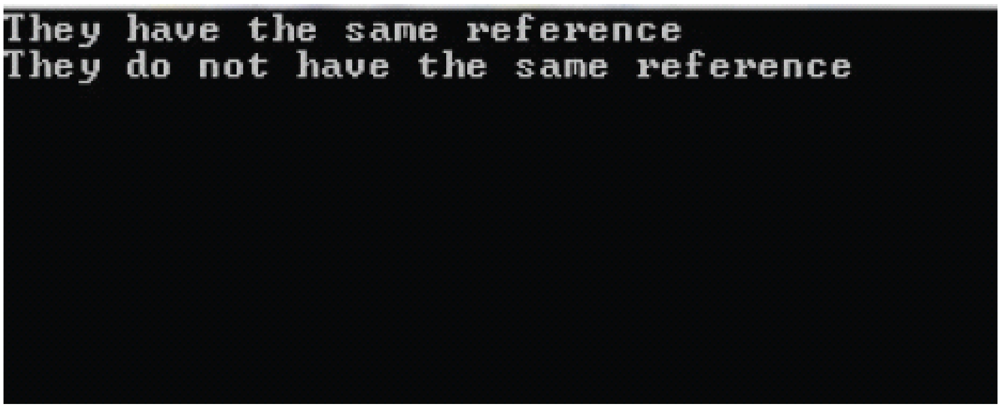
Strings

Strings are reference types that behave in many ways like a value type. Assignment and comparison works as you would expect with a value type, because they have value equality implemented, and sometimes they may even have the same reference but, in general, they will behave more like value types. This is a likely source of bugs among novice programmers, since the behavior may not be immediately obvious.

```
public void Test()
{
 string s = “Original”;
 s.ToLower();
 if (object.ReferenceEquals(s, “Original”))
   Console.WriteLine(“They have the same reference”);
 else
   Console.WriteLine(“They do not have the same reference”);
 s = s.ToLower();
 if (object.ReferenceEquals( s, “Original”))
   Console.WriteLine(“They have the same reference”);
 else
   Console.WriteLine(“They do not have the same reference”);
}
```

Listing 4.19: Showing the effects of immutable strings on reference equality.



```
They have the same reference
They do not have the same reference
```

Figure 4.3: The output from the code in Listing 4.19.

The output is unexpected. The confusion here lies in understanding mutability, rather than reference versus value types. Since strings are immutable, methods from the `string` class return a new string rather than modify the original string's memory. This can both improve code quality by reducing side effects, and reduce performance if you are not careful.

The first pair of strings has the same reference, because of interning, which we will explain later. The second pair of strings does not have the same reference, because the `ToLower` method created a new object.

The value for a string is stored in the heap. This is important because the stack can only hold 1 MB, and it would not be very difficult to imagine a single string using all of that space.

This string immutability can potentially create other memory usage issues. Every method call will result in a new string, and the old string will need to be garbage collected. In most cases, there will be no references to the old string, but the memory will not be reclaimed until the GC runs. Even though this memory will be reclaimed, the result is that the GC will need to run more frequently.

Intern pool

When comparing the identity of a string literal, you may be surprised to discover that they can have the same reference, despite having two unique declarations. We saw this earlier, in Listing 4.4.

```
string first = “text”;
string second = “text”;
bool areSame = object.ReferenceEquals(first, second);
Assert.IsTrue(areSame);
```

Listing 4.20 Two strings can have the same reference even when they are declared separately.

Strings are reference types, so it is not apparent that this would be true. In fact, if these strings were never written as string literals in the program, `Object.ReferenceEquals` would be false. Since the strings were written as string literals, they are stored in the **intern pool** by the CLR.

The intern pool is an optimization to speed up string literal comparisons. Using Reflector, we see that `String.Equals` has the implementation shown in Listing 4.21. The `EqualsHelper` is a private method in the String class using unsafe pointer manipulation to optimize the string comparison.

```
public static bool Equals(string a, string b)
{
 return ((a == b) ||
   (((a != null) && (b != null)) && EqualsHelper(a, b)));
}
```

Listing 4.21: Implementation of the `Equals` method in the String class.

In the case of matching interned strings, the comparison stops at `a==b` , which can potentially be a substantial performance boost (especially when comparing large strings). The implementation of `EqualsHelper` first checks to see if the references are the same. If they are, the comparison can conclude with success.

You can retrieve an instance from the intern pool with the `String.IsInterned` method, which will return null if the string is not in the pool. The `String.Intern` method will check the pool, add the string if it's not there, and return a reference to the string from the pool.

```
string first = String.Intern(String.Concat(“auto”, “dog”));
string second = String.IsInterned(String.Concat(“auto”, “dog”));

Assert.AreSame(first, second);
```

Listing 4.22: Showing the effects of explicitly interning a string.

While string literals are automatically added to the intern pool, instances of strings are not. They must be explicitly added with a call to `String.Intern` .

This sounds like a great way to improve the performance of your application, but be careful. There are important memory considerations. Items in the intern pool are not likely to be garbage collected. These references are accessible throughout the CLR and so can survive your application, and can survive every scope within your application. Another important factor is that the string is created separate from the intern pool before it is added to the pool. This will increase the overall memory footprint.

So, adding many strings to the intern pool could substantially increase your memory footprint and this segment of memory will not be reclaimed until you recycle the entire run time.

If you feel that string interning is adversely affecting the memory management in your application, you can disable it in individual assemblies with the `CompilationRelaxationsAttribute` .

Concatenation

A common task when working with strings is adding them together. The `string` class has an operator overload for `+` that converts the call to a `String.Concat` . The C# compiler optimizes the code by concatenating string literals. So Listing 4.23 is optimized as shown in Listing 4.24.

```
string panda = “bear” + “cat”;
```

Listing 4.23: A common string concatenation.

```
string panda = “bearcat”;
```

Listing 4.24: How the compiler will optimize it.

The compiler also converts concatenation with variables and non-string literals into a call to `String.Concat` . So Listing 4.25 becomes Listing 4.26.

```
string panda = “bear”;
panda += “cat”;
```

Listing 4.25: Forcing the compiler to concatenate.

```
string panda = “bear”;
panda = String.Concat(panda, “cat”);
```

Listing 4.26: The compiler explicitly calling the `String.Concat` method.

As you can see, the compiler is doing its best to help out the developer, but this is not without pitfalls. For example, perhaps the most pervasive challenge with string concatenation involves building a large string composed of many known elements. Approaching this problem with simple concatenation techniques leads to the creation of many small objects that must be garbage collected, potentially incurring unnecessary cost (see Listing 4.27).

```
public static string SerializeGeometry(IEnumerable<GeographicPoint> points)
{
  string result = “<Coordinates>”;

 foreach (var point in points)
 {
   result += “\t<Coordinate>” + point + “</Coordinate>”;
 }

 result += “<Coordinate>”;
 return result;
}
```

Listing 4.27: String concatenation in a loop.

The `SerializeGeometry` method creates an XML string based on an `IEnumerable` of `GeographicPoint`. There are an unknown number of elements in the sequence, as it is iterated, appending as many strings as needed to the resulting variable. Since `string` is immutable, this causes a new string to be allocated at every pass through the loop. This problem can be prevented by using the `StringBuilder` class.

```
public static string SerializeGeometry(IEnumerable<GeographicPoint> points)
{
  StringBuilder result = new StringBuilder();
 result.AppendLine("<Coordinates>");

 foreach (var point in points)
 {
  result.AppendLine("\t<Coordinate>" + point + "</Coordinate>");
 }

 result.AppendLine("<Coordinate>");
 return result.ToString();
}
```

Listing 4.28: Using `StringBuilder` in a loop.

`StringBuilder` will build the string with a single object for the GC to have to keep track of. This will not only reduce the total memory footprint, but also simplify the job that the GC has to do.

Naturally, we also need to be careful when using the `StringBuilder`, as there is a fair amount of logic in it. Thus, in simple cases, it can be overkill and adversely impact performance. As a good rule of thumb, you should probably never use a `StringBuilder` outside of a loop. Also don't do any string concatenation inside of the calls to the Append method, as this will just reintroduce the problems with multiple objects that we are trying to avoid in the first place.