Systemnahes Programmieren
Strings, Classes, Structs and how they pass parameters...

About Strings
A string is an object of type String whose value is text. Internally, the text is stored as a sequential read-only collection of Char objects. There is no null-terminating character at the end of a C# string; therefore a C# string can contain any number of embedded null characters ('\0'). The Length property of a string represents the number of Char objects it contains, not the number of Unicode characters. To access the individual Unicode code points in a string, use the StringInfo object.

string vs. System.String
In C#, the string keyword is an alias for String. Therefore, String and string are equivalent, and you can use whichever naming convention you prefer. The String class provides many methods for safely creating, manipulating, and comparing strings. In addition, the C# language overloads some operators to simplify common string operations. For more information about the keyword, see string (C# Reference). For more information about the type and its methods, see String.

Immutability of String Objects
String objects are immutable: they cannot be changed after they have been created. All of the String methods and C# operators that appear to modify a string actually return the results in a new string object. In the following example, when the contents of s1 and s2 are concatenated to form a single string, the two original strings are unmodified. The += operator creates a new string that contains the combined contents. That new object is assigned to the variable s1, and the original object that was assigned to s1 is released for garbage collection because no other variable holds a reference to it.

Structs vs. Classes
Structs may seem similar to classes, but there are important differences that you should be aware of. First of all, classes are reference types and structs are value types. By using structs, you can create objects that behave like the built-in types and enjoy their benefits as well.

Heap or Stack?
When you call the New operator on a class, it will be allocated on the heap. However, when you instantiate a struct, it gets created on the stack. This will yield performance gains. Also, you will not be dealing with references to an instance of a struct as you would with classes. You will be working directly with the struct instance. Because of this, when passing a struct to a method, it's passed by value instead of as a reference.

What is a Reference Type?
In .NET (and therefore C#) there are two main sorts of type: reference types and value types. They act differently, and a lot of confusion about parameter passing is really down to people not properly understanding the difference between them. Here's a quick explanation:

A reference type is a type which has as its value a reference to the appropriate data rather than the data itself. For instance, consider the following code:

```
StringClass stringClassA = new StringClass();
```

`StringClass` is used as a random example of a reference type. Here, we declare a variable `stringClassA`, create a new `StringClass` object, and assign to `stringClassA` a reference to the object. The value of `stringClassA` is not the object itself – it's the reference. Following statement finally assigns a String value to an attribute of this reference type:

```
stringClassA.val = "A0";
```

What is a Value Type?
While reference types have a layer of indirection between the variable and the real data, value types don't. Variables of a value type directly contain the data. Assignment of a value type involves the actual data being copied. Take a simple struct, for example:

```
public struct StringStruct
{
    public string val;
}
```

Wherever there is a variable of type `StringStruct`, the value of that variable contains all the data - in this case, the single `string` value. An assignment copies the value, as demonstrated here:

```
StringStruct stringStructA = new StringStruct();
stringStructA.val = "A0";
```

Simple types (such as float, int, char), enum types and struct types are all value types.

Note that many types (such as string) appear in some ways to be value types, but in fact are reference types. These are known as immutable types. This means that once an instance has been constructed, it can't be changed. This allows a reference type to act similarly to a value type in some ways - in particular, if you hold a reference to an immutable object, you can feel comfortable in returning it from a method or passing it to another method, safe in the knowledge that it won't be changed behind your back. This is why, for instance, the String.Replace doesn't change the string it is called on, but returns a new instance with the new string data in - if the original string were changed, any other variables holding a reference to the string would see the change, which is very rarely what is desired.

Value Parameters
By default, parameters are value parameters. This means that a new storage location is created for the variable in the function member declaration, and it starts off with the value that you specify in the function member invocation. If you change that value, that doesn't alter any variables involved in the invocation.

```
String strA = "A0";
StringProd1(strA);

public static void StringProd1(String a)
{
    a = "A1";
}
```

But why is not just the reference of String a copied? This is in fact a bit confusing. To understand this behavior, read section “Immutability of String Objects”. Here, `strA` keeps its value “A0”.

In the following example we pass “real” reference types (`StringClass`) to the method. `StringClass` is a self-made wrapper class that contains a single string attribute. The behavior is the same as with reference parameters – even if the parameters are “value parameters”. (In this case, the “value” is just the reference).

```
StringClass stringClassA = new StringClass();
stringClassA.val = "A0";
StringClassProd1(stringClassA)

public static void StringClassProd1(StringClass a)
{
    a.val = "A1";
}
```
Here, because a (copy of the) reference to `stringClassA` is passed rather than its value, changes to the value of parameter `a` are immediately reflected in `stringClassA`. In the above example, `a` ends up being “A1”.

But be careful: If you overwrite `a` with a new instance, it won’t reflect any changes to `stringClassA`. The reason for this behavior is very simple: We just got a copy of the reference to object `stringClassA` – not a reference to the reference! The new `StringClass` reference has another storage location in the Heap than the previous object.

```
public static void StringClassProd1(StringClass a)
{
    a = new StringClass();
    a.val = "A1";
}
```

Reference Parameters
Reference parameters don't pass the values of the variables used in the function member invocation - they use the variables themselves. Rather than creating a new storage location for the variable in the function member declaration, the same storage location is used, so the value of the variable in the function member and the value of the reference parameter will always be the same. Reference parameters need the `ref` modifier as part of both the declaration and the invocation - that means it's always clear when you're passing something by reference. Let's look at our previous examples, just changing the parameter to be a reference parameter:

```
String strA = "A0";
StringProd2(ref strA);

public static void StringProd2(ref String a)
{
    a = "A2";
}
```

Here, because a reference to `strA` is passed rather than its value, changes to the value of parameter `a` are immediately reflected in `stringClassA`. In the above example, `a` ends up being "A2"
Compare this with the result of the same code without the `ref` modifiers.

Output Parameters
Like reference parameters, output parameters don't create a new storage location, but use the storage location of the variable specified on the invocation. Output parameters need the out modifier as part of both the declaration and the invocation - that means it's always clear when you're passing something as an output parameter.
Output parameters are very similar to reference parameters. The only differences are:

- The variable specified on the invocation doesn't need to have been assigned a value before it is passed to the function member. If the function member completes normally, the variable is considered to be assigned afterwards (so you can then "read" it).
- The parameter is considered initially unassigned (in other words, you must assign it a value before you can "read" it in the function member).
- The parameter *must* be assigned a value before the function member completes normally.

Have a look at the following example. We first create an empty string object, `strA`. It's not necessary to instantiate it as long as it is an immutable object. `StringProd3` will change `strA` to "A3".

```
String strA;
StringProd3(out strA);

public static void StringProd3(out String a)
{
    a = "A3";
}
```

But again: String is a very special data type, an immutable class! Let's take another reference type, e.g. our `StringClass`. This class needs to be instantiated within StringClassProd3 before it is used or assigned. If you don't care about that, you will get the compiler error "use of unassigned parameter".

```
StringClass stringClassA = new StringClass();
stringClassA.val = "A0";
StringClassProd3(out stringClassA);

public static void StringClassProd3(out StringClass a)
{
    a = new StringClass();
    a.val = "A3";
}
```

What is the difference between
<u>passing a value object by reference</u> and <u>passing a reference object by value</u>?
You may have noticed that passing a struct by reference, had the same effect as passing a class by value. This doesn't mean that they're the same thing, however. Consider the following code:

```
String... y = new String...();
y.val = "A";
Method(??? y);

public static void Method(??? String... x)
{
    x = new String...();
}
```

- In the case we use `StringStruct` (i.e. a value type) and the parameter is a <u>reference parameter</u> (i.e. replace ??? with `ref` above), `y` ends up being a new `StringStruct` value - i.e. y.val is 0.
- In the case we use `StringClass` (i.e. a reference type) and the parameter is a <u>value parameter</u> (i.e. remove ??? above), the value of y isn't changed - it's a reference to the same object it was before the function member call.

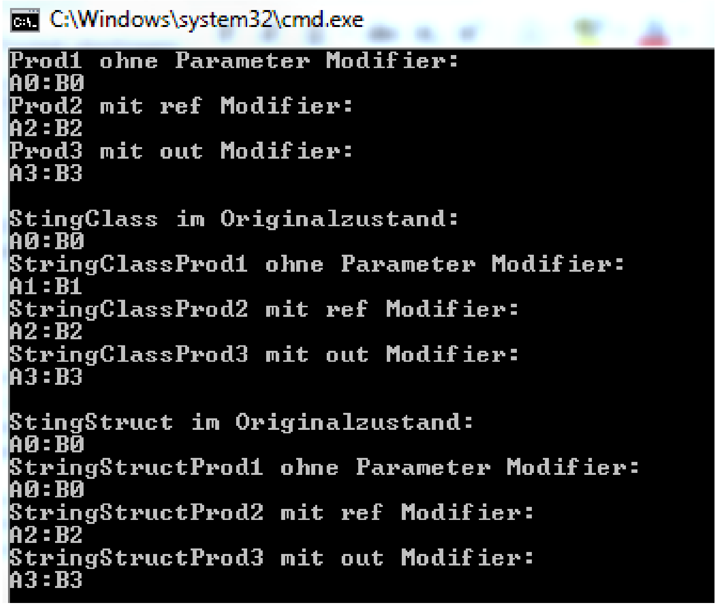
This difference is absolutely crucial to understanding parameter passing in C#, and is why I believe it is highly confusing to say that objects are passed by reference by default instead of the correct statement that object references are passed by value by default.

Sample Code
Following code tries to demonstrate the different parameter passing behaviors described in the sections above.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Gen
{
    class Prod
    {
        public static void StringProd1(String a, String b)
        {
            a = "A1";
            b = "B1";
        }

        public static void StringProd2(ref String a, ref String b)
        {
            a = "A2";
            b = "B2";
        }

        public static void StringProd3(out String a, out String b)
        {
            a = "A3";
            b = "B3";
        }

        public static void StringClassProd1(StringClass a, StringClass b)
        {
            // Folgende Zeile hebt die kopierte Referenz auf.
            // Änderungen an a werden demzufolge nicht zurückgeschrieben.
            //a = new StringClass();

            a.val = "A1";
            b.val = "B1";
        }

        public static void StringClassProd2(ref StringClass a, ref StringClass b)
        {
            // Folgende Zeile instanziert die neue Klasse direkt in die alte Referenz.
            // Änderungen an a werden demzufolge zurückgeschrieben.
            // a = new StringClass();
            a.val = "A2";
            b.val = "B2";
        }

        public static void StringClassProd3(out StringClass a, out StringClass b)
        {
            // Vorsicht: Compiler Fehler "Use of unassigned parameter"
            // falls a oder b genutzt werden wollen, bevor sie instanziert wurden!

            StringClass stringClassA = new StringClass();
            stringClassA.val = "A3";
            StringClass stringClassB = new StringClass();
            stringClassB.val = "B3";

            a = stringClassA;
            b = stringClassB;
        }

        public static void StringStructProd1(StringStruct a, StringStruct b)
        {
            a.val = "A1";
            b.val = "B1";
        }
```

```
public static void StringStructProd2(ref StringStruct a, ref StringStruct b)
{
    //a = new StringStruct();
    a.val = "A2";
    b.val = "B2";
}

public static void StringStructProd3(out StringStruct a, out StringStruct b)
{
    a.val = "A3";
    b.val = "B3";
}

public static void Main()
{
    //STRING TESTS
    String strA = "A0";
    String strB = "B0";
    Console.WriteLine("Strings im Originalzustand:");
    Console.WriteLine(strA + ":" + strB);

    Console.WriteLine("Prod1 ohne Parameter Modifier:");
    StringProd1(strA, strB);
    Console.WriteLine(strA + ":" + strB);

    Console.WriteLine("Prod2 mit ref Modifier:");
    StringProd2(ref strA, ref strB);
    Console.WriteLine(strA + ":" + strB);

    Console.WriteLine("Prod3 mit out Modifier:");
    StringProd3(out strA, out strB);
    String newStr;
    StringProd3(out newStr, out newStr);
    Console.WriteLine(strA + ":" + strB);

    //STRING CLASS TESTS
    StringClass stringClassA = new StringClass();
    stringClassA.val = "A0";
    StringClass stringClassB = new StringClass();
    stringClassB.val = "B0";

    Console.WriteLine();
    Console.WriteLine("StingClass im Originalzustand:");
    Console.WriteLine(stringClassA.val + ":" + stringClassB.val);

    Console.WriteLine("StringClassProd1 ohne Parameter Modifier:");
    StringClassProd1(stringClassA, stringClassB);
    Console.WriteLine(stringClassA.val + ":" + stringClassB.val);

    Console.WriteLine("StringClassProd2 mit ref Modifier:");
    StringClassProd2(ref stringClassA, ref stringClassB);
    Console.WriteLine(stringClassA.val + ":" + stringClassB.val);

    Console.WriteLine("StringClassProd3 mit out Modifier:");
    StringClassProd3(out stringClassA, out stringClassB);
    Console.WriteLine(stringClassA.val + ":" + stringClassB.val);

    //STRING STRUCT TESTS
    StringStruct stringStructA = new StringStruct();
    stringStructA.val = "A0";
    StringStruct stringStructB = new StringStruct();
    stringStructB.val = "B0";

    Console.WriteLine();
    Console.WriteLine("StingStruct im Originalzustand:");
    Console.WriteLine(stringStructA.val + ":" + stringStructB.val);

    Console.WriteLine("StringStructProd1 ohne Parameter Modifier:");
    StringStructProd1(stringStructA, stringStructB);
    Console.WriteLine(stringStructA.val + ":" + stringStructB.val);

    Console.WriteLine("StringStructProd2 mit ref Modifier:");
    StringStructProd2(ref stringStructA, ref stringStructB);
    Console.WriteLine(stringStructA.val + ":" + stringStructB.val);

    Console.WriteLine("StringStructProd3 mit out Modifier:");
    StringStructProd3(out stringStructA, out stringStructB);
    Console.WriteLine(stringStructA.val + ":" + stringStructB.val);
```

```
            Console.Read();
        }
    }

    public class StringClass
    {
        public string val;
    }

    public struct StringStruct
    {
        public string val;
    }
}

//Following output is generated by this code:
```



```
Prod1 ohne Parameter Modifier:
A0:B0
Prod2 mit ref Modifier:
A2:B2
Prod3 mit out Modifier:
A3:B3

StingClass im Originalzustand:
A0:B0
StringClassProd1 ohne Parameter Modifier:
A1:B1
StringClassProd2 mit ref Modifier:
A2:B2
StringClassProd3 mit out Modifier:
A3:B3

StingStruct im Originalzustand:
A0:B0
StringStructProd1 ohne Parameter Modifier:
A0:B0
StringStructProd2 mit ref Modifier:
A2:B2
StringStructProd3 mit out Modifier:
A3:B3
```

More Information about this topic
http://msdn.microsoft.com/en-us/library/ms228362.aspx
http://msdn.microsoft.com/en-us/library/aa288471(v=vs.71).aspx
http://www.yoda.arachsys.com/csharp/parameters.html
http://www.albahari.com/valuevsreftypes.aspx