A string is basically a sequence of characters. Each character is a Unicode character in the range U+0000 to U+FFFF (more on that later). The string type (I'll use the C# shorthand rather than putting System.String each time) has the following characteristics:

It is a reference type

It's a common misconception that string is a value type. That's because its immutability (see next point) makes it act sort of like a value type. It actually acts like a normal reference type. See my articles on parameter passing and memory for more details of the differences between value types and reference types.

It's immutable

You can never actually change the contents of a string, at least with safe code which doesn't use reflection. Because of this, you often end up changing the value of a string variable. For instance, the code s = s.Replace ("foo", "bar"); doesn't change the contents of the string that s originally referred to - it just sets the value of s to a new string, which is a copy of the old string but with "foo" replaced by "bar".

It can contain nulls

C programmers are used to strings being sequences of characters ending in '\0', the nul or null character. (I'll use "null" because that's what the Unicode code chart calls it in the detail; don't get it confused with the null keyword in C# - char is a value type, so can't be a null reference!) In .NET, strings can contain null characters with no problems at all as far as the string methods themselves are concerned. However, other classes (for instance many of the Windows Forms ones) may well think that the string finishes at the first null character - if your string ever appears to be truncated oddly, that could be the problem.

It overloads the == operator

When the == operator is used to compare two strings, the Equals method is called, which checks for the equality of the contents of the strings rather than the references themselves. For instance, "hello".Substring(0, 4)=="hell" is true, even though the references on the two sides of the operator are different (they refer to two different string objects, which both contain the same character sequence). Note that operator overloading only works here if both sides of the operator are string expressions at compile time - operators aren't applied polymorphically. If either side of the operator is of type object as far as the compiler is concerned, the normal == operator will be applied, and simple reference equality will be tested.