**What is a string?**

A string is basically a sequence of characters. Each character is a [Unicode](http://www.unicode.org/) character in the range U+0000 to U+FFFF . The string type 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.

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 soriginally 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.

**Interning**

.NET has the concept of an "intern pool". It's basically just a set of strings, but it makes sure that every time you reference the same string literal, you get a reference to the same string. This is probably language-dependent, but it's certainly true in C# and VB.NET, and I'd be very surprised to see a language it didn't hold for, as IL makes it very easy to do (probably easier than failing to intern literals). As well as literals being automatically interned, you can intern strings manually with the Intern method, and check whether or not there is already an interned string with the same character sequence in the pool using the IsInterned method. This somewhat unintuitively returns a string rather than a boolean - if an equal string is in the pool, a reference to that string is returned. Otherwise, null is returned. Likewise, the Intern method returns a reference to an interned string - either the string you passed in if was already in the pool, or a newly created interned string, or an equal string which was already in the pool.

**Literals**

Literals are how you hard-code strings into C# programs. There are two types of string literals in C# - regular string literals and verbatim string literals. Regular string literals are similar to those in many other languages such as Java and C - they start and end with ", and various characters (in particular," itself, \, and carriage return (CR) and line feed (LF)) need to be "escaped" to be represented in the string. Verbatim string literals allow pretty much anything within them, and end at the first " which isn't doubled. Even carriage returns and line feeds can appear in the literal! To obtain a " within the string itself, you need to write "". Verbatim string literals are distinguished by having an @ before the opening quote. Here are some examples of the two types of literal, and what they amount to:

|  |  |  |
| --- | --- | --- |
| **Regular literal** | **Verbatim literal** | **Resulting string** |
| "Hello" | @"Hello" | Hello |
| "Backslash: \\" | @"Backslash: \" | Backslash: \ |
| "Quote: \"" | @"Quote: """ | Quote: " |
| "CRLF:\r\nPost CRLF" | @"CRLF: Post CRLF" | CRLF: Post CRLF |

Note that the difference is only for the compiler's sake. Once the string is in the compiled code, there's no such thing as a verbatim string literal vs a regular string literal.

The complete set of escape sequences is as follows:

* \' - single quote, needed for character literals
* \" - double quote, needed for string literals
* \\ - backslash
* \0 - Unicode character 0
* \a - Alert
* \b - Backspace
* \f - Form feed
* \n - New line
* \r - Carriage return
* \t - Horizontal tab