The String class is special in C# for several reasons: For one, it can be instantiated in two different ways unlike all other classes.

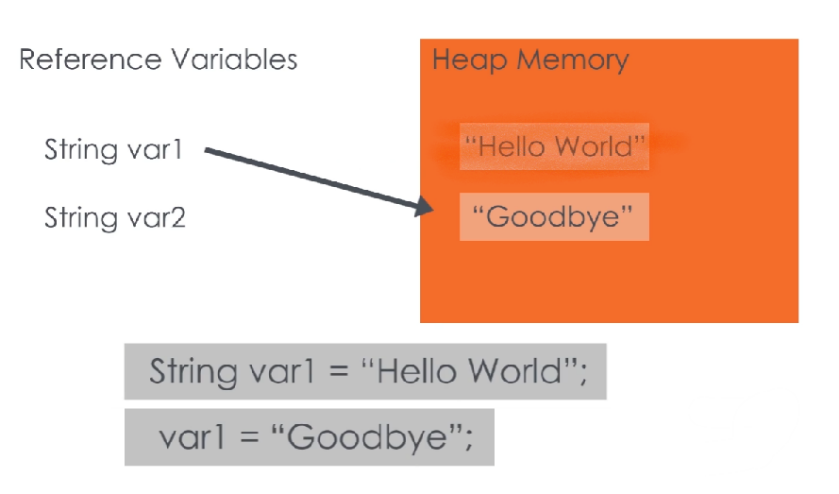
But most importantly, string objects are immutable, which means that they cannot be changed once they are created.

When your code tries to change the value in a string, the CLR actually creates a new string object in memory with the resulting value, and redirects your string reference variable to the new string object.

It does this so that it can recycle duplicate strings, saving memory in large applications.

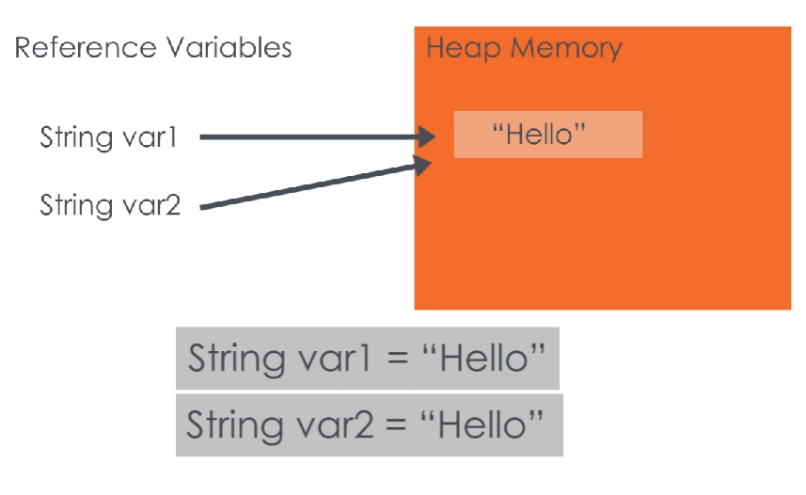
Here’s an illustration of how this works. On the left, we have our reference variables.

Remember that a reference variable in C# is not an object in its own right, it’s a link to an instance of an object in memory.

On the right, we have a crude representation of the CLR’s heap memory, where objects are stored.

When we create a String, either in the “primitive” style or as a new object, a String object is created in memory that holds the value we assign it.

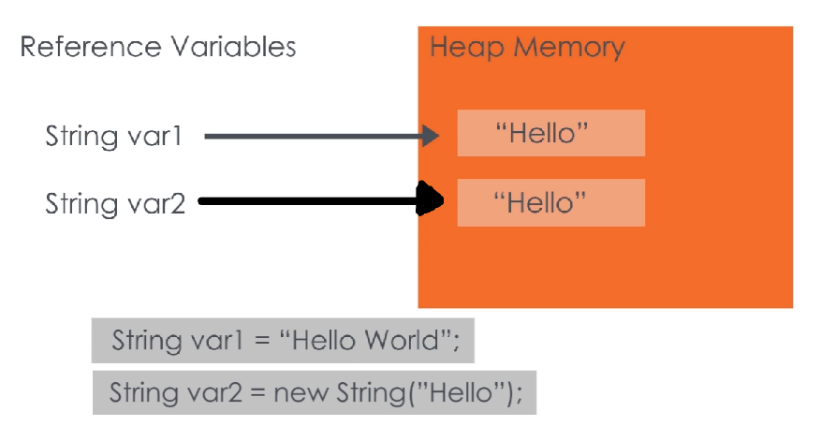
Strings are immutable, so if we try to change the value of the String in our reference variable, the CLR actually creates a new String object, and redirects our reference. The orphaned String object will be cleaned up by the garbage collection process later, to free up the memory it’s using.



If I have a String variable here, and I initialize it to “Hello”, what happens when I create a second String with the value “Hello”?

Well, because Strings are immutable, the CLR decides that it should be safe for these two variables to share the same instance, and so they will both be directed to the same object in memory.

This helps save memory where you might have hundreds of Strings with the same value in large applications.



There is a way to override this though – if we use the new keyword to force a new String object to be instantiated, we can force the CLR to have duplicate Strings in memory.

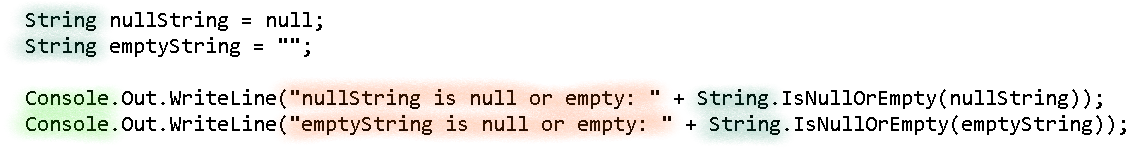
There are a number of useful methods in the string class, and you should become very familiar with them.

Many common interview questions test knowledge of how Strings in C# work, and the best ways to manipulate them.

You can find links to the documentation in the reference section.

Let’s look at some of the most commonly used String methods.

In the first region of code, we have 2 strings.



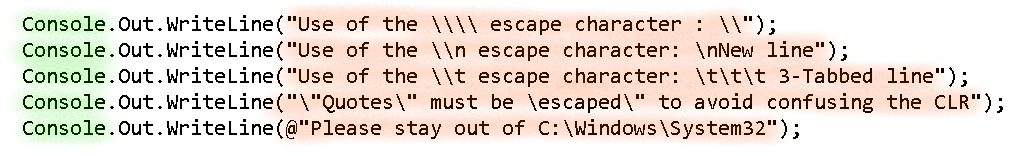
The string declaration called nullString is assigned to the null keyword, implying this reference is pointing to nothing.

The next string declaration is an example of a reference that points to an empty string value.

It’s best-practice is to check that a string reference has a value before using it, especially when a string is from an unknown source.

So static method **String.IsNullOREmpty method** will safely verify string references, in this example both instances evaluate to true and the WriteLine methods are executed.

Next, we show some escape characters. Escape characters are used to print things that are not alphanumeric characters, like a line break or tab indent.



Use of the \\ escape character: \

Use of the \n escape character:

New line

Use of the \t escape character: 3-Tabbed line

"Quotes must be "escaped" to avoid confusing the CLR

Please stay out of C:\Windows\System32

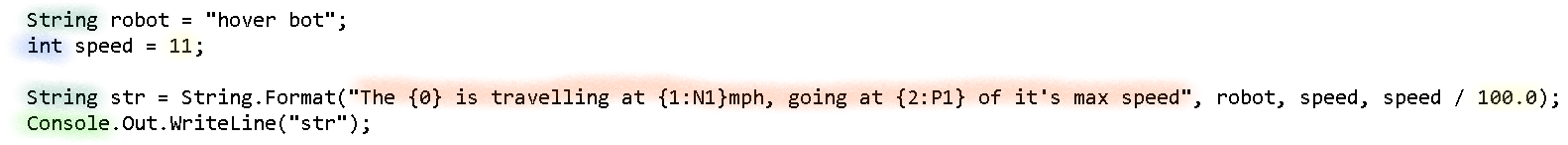
These are indicated by a backslash and another character.

with the backslash and letter ‘t’ is a like pushing the Tab key, while the backslash and letter ‘n’ is a new line like pressing the return key.

To print a single backslash or double quote character requires a backslash before each.

Sometimes you will want to ignore escape sequences by using the @ sign before a string. This is useful for things like a website address or file path, such as shown here.

Below that is the **String.Format method**, which is useful for creating strings based on a pattern and displaying values with a format like a dollar sign for money.

In this example, the indexing in curly-braces is replaced with the values in order, starting with the string variable robot.

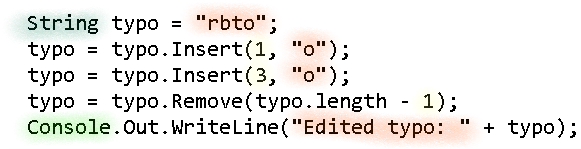
The integer speed will print with a decimal place holder due to the **:N1** characters, and the last value will appear as a percent value due to the **:P1**.

When we run this program,the formatting printed the speed with a decimal place holder and the division as a percentage.

The hover bot is travelling at 11.0mph, going at 11 % of it’s max speed

Now let’s look at some common methods for manipulating strings.

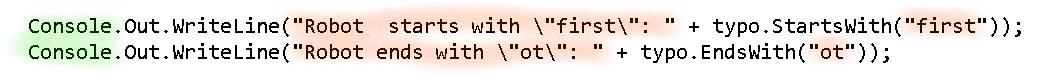
These methods return a new string object as their output, which has the value of the original string, with the requested edits.



The first string declared is misspelled, but we can correct that by using the **Insert method**.

By inserting 2 ‘o’ characters while the **Remove method** deletes the extra ‘o’ character, we correct the spelling of the string.

Moving on to the next region, the **Starts-With method** in this example returns false because the string variable doesn’t start with the word ‘first’, while the **Ends-With method** returns true.



Robot starts with “first”: False

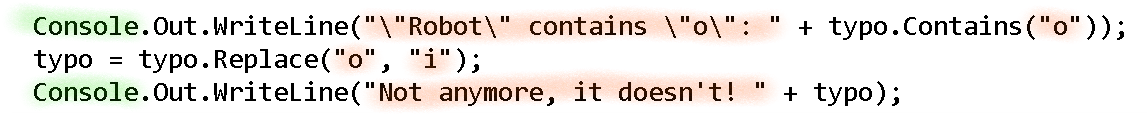
Robot ends with “ot”: True

The **Substring method** returns a snippet starting at an index and with a given length; here, it returns the last 3 characters, “bot”.



Substring: bot

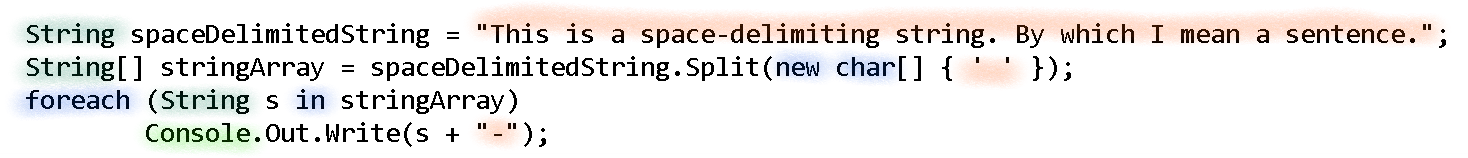
The **Contains method** verifies that a character or substring exists in a string value, here it is true, and then the Replace method swaps all instances of the letter ‘o’ with the given string.



“Robot” contains “o”: True

Not anymore it doesn’t! ribit

The **Split method** returns an array of substrings by separating a string value into pieces by providing a delimiter.



This-is-a-space-delimiting-string.-By-which-I-mean-a-sentence.

In this example, the empty space is the delimiter to separate a sentence into an array of individual words, but you could also use it to separate a file path by the backslashes to extract the directory names or part of a website’s address.

So we’ve seen a few examples of string manipulation using methods from the System.String class, used escape sequences, and learned how to use static methods that can be called without an instantiating an object.

Up to this point we’ve been manipulating immutable strings, and getting new string objects as results.

If you want to use a String-like object that is not immutable, and will incorporate any changes directly into the object,

you want the StringBuilder class, provided by the .Net Framework for creating and manipulating mutable strings.

String are by far the most common object type you will work with – next to the Object type of course – so it pays to understand their functions and features in detail.

