# Strings in C#

Strings in C# are a built-in type, as well as a standard .NET type. The `string` keyword in C# is the same as the `System.String` type, and you can declare variables using either version without any impact on your program's behavior. When a string is instantiated in C#, a sequence of bytes in memory is allocated to it. This allocation cannot be modified after-the-fact; it can only be erased once the string is no longer in use by the program. To change a string, or to create a new string out of one or more others, a new memory allocation is required. Because string values cannot be changed once they are created, strings are said to be *immutable*. This aspect of strings can have performance implications in your programs, so it's helpful to understand.

## Creating Strings

Strings are reference types, which means (among other things) they are null when they are declared, until they are assigned a value. You can declare and assign a value in a single statement, or as separate statements. Both of the following examples result in strings with the same value:

```
string string1; // current value is null
string1 = "Hello";

string string2 = "Hello";
```

```
using System;

class Program
{
    static void Main()
    {
    string string1; // current value is null
    string1 = "Hello";

    string string2 = "Hello";
    Console.WriteLine(string1);
    Console.WriteLine(string2);
    }
}
```

**Note** {.note}
In C#, you can add a *line comment* to any line by using to forward slashes: `//` . Any thing on that line following these characters will be ignored.

If you need an *empty* string, you can create one in two different ways:

```
string empty1 = "";
string empty2 = String.Empty;
```

```
using System;

class Program
{
    static void Main()
    {
        string empty1 = "";
        string empty2 = String.Empty;

        Console.WriteLine(empty1.Length);
        Console.WriteLine(empty2.Length);
    }
}
```

`String.Empty` is a built-in constant value for a string with zero length.

## Empty or null

It's important to understand the difference between an *empty* string and a *null* string (or other reference type). In C#, a variable's value is null if it is not set, or if it has been explicitly set to null. A null value typically means the value is not known, or has not yet been set, and is frequently a source of errors in programs, because methods cannot be invoked on null instances of types. An empty string is simply a string with a length of zero. It's a valid instance of a string, but one that currently has no characters in it. Take care when working with nulls in your programs, especially if they may be passed to other methods, and be sure to confirm that arguments your methods work with are not null before working with their members.

> **Tip** {.tip .newLanguage }
> Null values are frequent sources of bugs even for experienced programmers. Be wary of assuming that variables are not null, especially when working with parameters or other variables you're not responsible for directly instantiating.

A simple example should help demonstrate the difference between empty and null strings.

```
string emptyString = String.Empty;
string nullString = null;

Console.WriteLine(emptyString); // prints nothing
Console.WriteLine(nullString); // prints nothing

// this line will print 0
Console.WriteLine($"1st string is {emptyString.Length} characters long.");

// this line will throw an exception (uncomment it to confirm)
// Console.WriteLine($"2nd string is {nullString.Length} characters long.");
```

```
using System;

class Program
```

```
{
    static void Main()
    {
        string emptyString = String.Empty;
        string nullString = null;

        Console.WriteLine(emptyString); // prints nothing
        Console.WriteLine(nullString); // prints nothing

        // this line will print 0
        Console.WriteLine($"1st string is {emptyString.Length} characters long.");

        // this line will throw an exception (uncomment it to confirm)
        // Console.WriteLine($"2nd string is {nullString.Length} characters long.");
    }
}
```

If you try to work with the members of an object that isn't set (one that is null), your program will throw a `NullReferenceException` with a message "Object reference not set to an instance of an object". These are very common, and often very frustrating, errors in C# applications. You can mitigate them by ensuring you're always setting a default value to your instances rather than allowing them to remain null.

You can check to see if a string value is null by comparing it to null (for instance, `(nullstring == null)` would evaluate to `true` ). There are also some built-in helper functions for detecting whether a string is either null or empty.
Both `(String.IsNullOrEmpty(emptyString))` and `(String.IsNullOrEmpty(nullString))` would return `true` as well. This demonstrates another common pattern in C#, which is the use of static *extension methods* on a type for functionality that is useful to apply to null instances of the type. You will learn more about extension methods in Defining and Calling Methods lesson.

## String Operations

There are many built-in ways to manipulate strings in C#. One of the simplest is concatentation. You can concatenate two strings by using the `+` operator:

```
string one = "abc";
string two = "123";
string combined = one + two; // "abc123"
```

```
using System;

class Program
{
    static void Main()
    {
        string one = "abc";
        string two = "123";
        string combined = one + two; // "abc123"
        Console.WriteLine(combined);
    }
}
```

Note that if you need to build up a string by performing many dozens or more concatenations, the allocating and deallocating of so many instances can become a performance issue. In that case, there is a special type called `StringBuilder` that can be used more efficiently for this purpose.

Strings include many built-in formatting functions, which can be invoked on variables as well as string literals. For instance:

```
string original = "Test string";
string capital = original.ToUpper(); // TEST STRING
string lower = original.ToLower(); // test string
string lower2 = "Another Test".ToLower(); // another test
```

```
using System;

class Program
{
    static void Main()
    {
        string original = "Test string";
        string capital = original.ToUpper(); // TEST STRING
        string lower = original.ToLower(); // test string
        string lower2 = "Another Test".ToLower(); // another test

        Console.WriteLine(original);
        Console.WriteLine(capital);
        Console.WriteLine(lower);
        Console.WriteLine(lower2);
    }
}
```

When accepting user input, it can be useful to trim it, so that it fits into a fixed amount of space, or so that it doesn't contain extraneous characters. The `Trim` methods remove whitespace characters from the start and/or end of a string:

```
string input = " Steve "; // has a space at the start and end.
string clean1 = input.TrimStart(); // "Steve "
string clean2 = input.TrimEnd(); // " Steve"
string clean3 = input.Trim(); // "Steve"
string shortversion = input.Trim().Substring(0,3); // "Ste"
```

```
using System;

class Program
```

```
{
    static void Main()
    {
        string input = " Steve "; // has a space at the start and end.
        string clean1 = input.TrimStart(); // "Steve "
        string clean2 = input.TrimEnd(); // " Steve"
        string clean3 = input.Trim(); // "Steve"
        string shortversion = input.Trim().Substring(0,3); // "Ste"

        Console.WriteLine($"*{input}*");
        Console.WriteLine($"*{clean1}*");
        Console.WriteLine($"*{clean2}*");
        Console.WriteLine($"*{clean3}*");
        Console.WriteLine($"*{shortversion}*");
    }
}
```

Note in the last example that multiple methods are chained together, such that the `input` value is first trimmed, and then a substring consisting of the first three characters is extracted. `SubString` is a very useful String method that starts from a given position (where 0 is the start of the string) and returns a number of characters specified in the second argument (in this case 3).

> **Tip** {.tip .vb }
> Unlike VB, C# doesn't include methods like `Left` and `Right` that return a certain number of characters starting from the left or right side of a string. However, these methods are available as part of the .NET Framework.

## Replacing Parts of Strings

A frequent scenario when working with strings is the construction of a string that is composed of some fixed parts, and a part that is variable. You saw in the last lessonhow you could construct such a string using `$"Hello {name}!` syntax. This is known as *string interpolation*, and is a new feature in C# 6. The same string can also be constructed using concatenation ( `"Hello " + name + "!"` ). Strings also support rich formatting, which at its simplest allows for this kind of replacement. To use string formatting, you specify a format string, which includes special placeholder values, and pass the format string and the replacement values to the `Format` method. Finally, you can use the `Replace` method to replace part of a string with another string. Consider the following examples:

```
string name = "Steve";
string greet1 = $"Hello {name}!"; // Hello Steve!
string greet2 = "Hello " + name + "!"; // Hello Steve!
string greet3 = String.Format("Hello {0}!", name); // Hello Steve!
string greetTemplate = "Hello **NAME**!";
string greet4 = greetTemplate.Replace("**NAME**", name); // Hello Steve!
```

```
using System;

class Program
```

```
  {
      static void Main()
      {
          string name = "Steve";
          string greet1 = $"Hello {name}!"; // Hello Steve!
          string greet2 = "Hello " + name + "!"; // Hello Steve!
          string greet3 = String.Format("Hello {0}!", name); // Hello Steve!
          string greetTemplate = "Hello **NAME**!";
          string greet4 = greetTemplate.Replace("**NAME**", name); // Hello Steve!

          Console.WriteLine(name);
          Console.WriteLine(greet1);
          Console.WriteLine(greet2);
          Console.WriteLine(greet3);
          Console.WriteLine(greetTemplate);
          Console.WriteLine(greet4);
      }
  }
```

Of all of the above, the string interpolation approach ( `greet1` ) is recommended. It's clean, readable, and generally has better performance than the other approaches. However, using format strings or templates is a good option if you need to store the template separately from the scope of the variables that will be used during runtime value replacement.