.NET/C# Syntax.

Modern .NET/C# Course

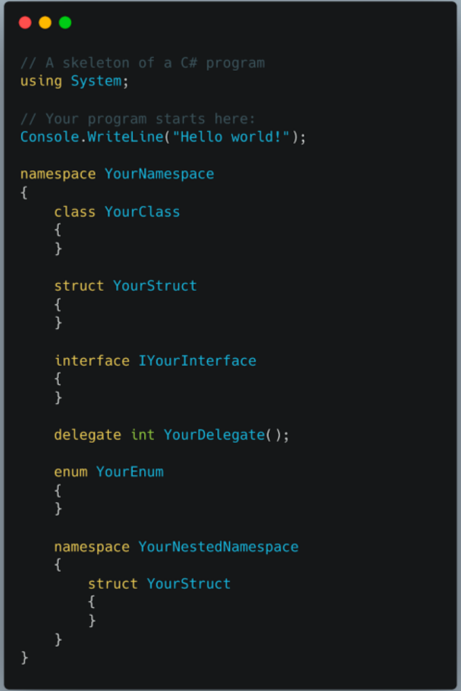
.NET Academy

Content

- Program structure.
- Top-level statements– program without Main method.
- Variables and Datatype.
- Types Aliases.
- Strings.
    - Formatting.
    - Multiline string.
    - UTF-8 string literals.

.NET Academy

Program structure.



```
// A skeleton of a C# program
using System;

// Your program starts here:
Console.WriteLine("Hello world!");

namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }
}
```

C# programs consist of one or more files. Each file contains zero or more namespaces. A namespace contains types such as classes, structs, interfaces, enumerations, and delegates, or other namespaces. The following example is the skeleton of a C# program that contains all of these elements.

.NET Academy

Top-level operators - programme without Main method

Top-level statements enable you to avoid the extra ceremony required by placing your program's entry point in a static method in a class. We can use top-level statements for scripting scenarios, or to explore. Once you've got the basics working, you can start refactoring the code and create methods, classes, or other assemblies for reusable components you've built. Top-level statements do enable quick experimentation and beginner tutorials. They also provide a smooth path from experimentation to full programs.

Top-level statements are executed in the order they appear in the file. Top-level statements can only be used in one source file in your application. The compiler generates an error if you use them in more than one file. Top-level statements make it easier to create simple programs for use to explore new algorithms. You can experiment with algorithms by trying different snippets of code. Once you've learned what works, you can refactor the code to be more maintainable.

Variables and Datatype

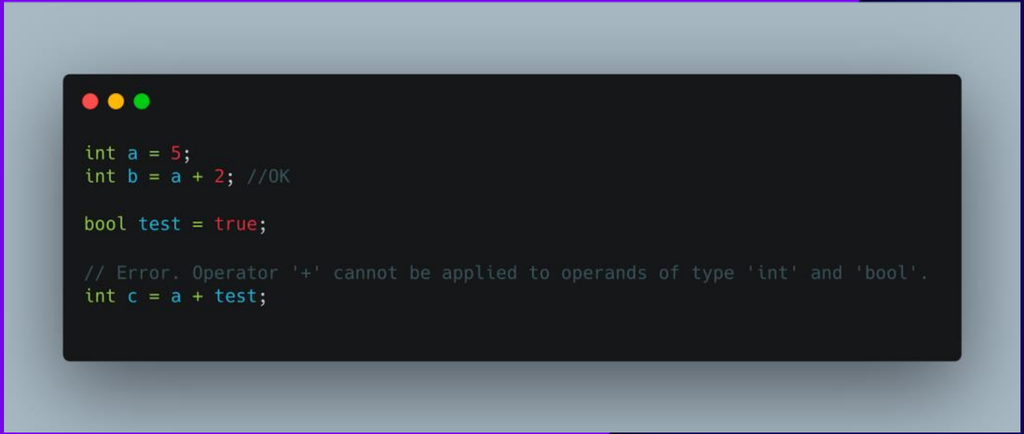
C# is a strongly typed language. Every variable and constant has a type, as does every expression that evaluates to a value. Every method declaration specifies a name, the type and kind (value, reference, or output) for each input parameter and for the return value. The .NET class library defines built-in numeric types and complex types that represent a wide variety of constructs.

The information stored in a type can include the following items:

- The storage space that a variable of the type requires.
- The maximum and minimum values that it can represent.
- The members (methods, fields, events, and so on) that it contains.
- The base type it inherits from. The interface(s) it implements.
- The kinds of operations that are permitted.

Variables and Datatype

The compiler uses type information to make sure all operations that are performed in your code are type safe. For example, if you declare a variable of type int, the compiler allows you to use the variable in addition and subtraction operations. If you try to perform those same operations on a variable of type bool, the compiler generates an error, as shown in the following example:



```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

.NET Academy

Variables and Datatype

The compiler embeds the type information into the executable file as metadata. The common language runtime (CLR) uses that metadata at run time to further guarantee type safety when it allocates and reclaims memory.

When you declare a variable or constant in a program, you must either specify its type or use the var keyword to let the compiler infer the type.

After you declare a variable, you can't redeclare it with a new type, and you can't assign a value not compatible with its declared type.

C# provides a standard set of built-in types. These represent integers, floating point values, Boolean expressions, text characters, decimal values, and other types of data. There are also built-in string and object types. These types are available for you to use in any C# program.

We can use the struct, class, interface, enum, and record constructs to create our own custom types.

C# Keyword	.NET Type	Range	Size
bool	System.Boolean	true-false	1 bit
byte	System.Byte	0 to 255	8-bit
sbyte	System.SByte	-128 to 127	8-bit
char	System.Char	-	16-bit
decimal	System.Decimal	28-29 digits	128-bit
double	System.Double	~15-17 digits	64-bit
float	System.Single	~6-9 digits	32-bit
int	System.Int32	-2,147,483,648 to 2,147,483,647	32-bit
uint	System.UInt32	0 to 4,294,967,295	32-bit
nint	System.IntPtr	Depends on platform	32 or 64 bit
nuint	System.UIntPtr	Depends on platform	32 or 64 bit
long	System.Int64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	64-bit
ulong	System.UInt64	0 to 18,446,744,073,709,551,615	64-bit
short	System.Int16	-32,768 to 32,767	16-bit
ushort	System.UInt16	0 to 65,535	16-bit
object	System.Object	-	32-bit address
string	System.String	2^31 characters	16-bit per char.
dynamic	System.Object	-	Depends on type.
class	-	-	32-bit address
delegate	System.Delegate	-	32-bit address
interface	-	-	32-bit address
struct	-	-	Depends on members
enum	System.Enum	-	8-bit

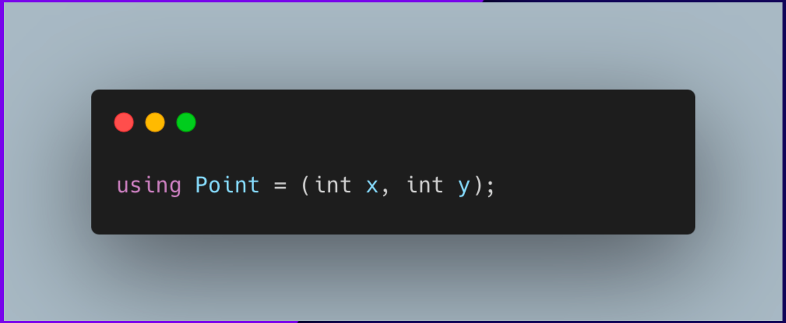
Types Aliases

Relax the using_alias_directive to allow it to point at any sort of type, not just named types. This would support types not allowed today, like: tuple types, pointer types, array types, etc.

.NET Academy

Types Aliases

For example, this would now be allowed:



```
using Point = (int x, int y);
```

For ages, C# has had the ability to introduce aliases for namespaces and named types (classes, delegated, interfaces, records and structs). This worked acceptably well as it provided a means to introduce non-conflicting names in cases where a normal named pulled in from using_directives might be ambiguous, and it allowed a way to provide a simpler name when dealing with complex generic types. However, the rise of additional complex type symbols in the language has caused more use to arise where aliases would be valuable but are currently not allowed. For example, both tuples and function-pointers often can have large and complex regular textual forms that can be painful to continually write out, and a burden to try to read. Aliases would help in these cases by giving a short, developer-provided, name that can then be used in place of those full structural forms.

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's 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.

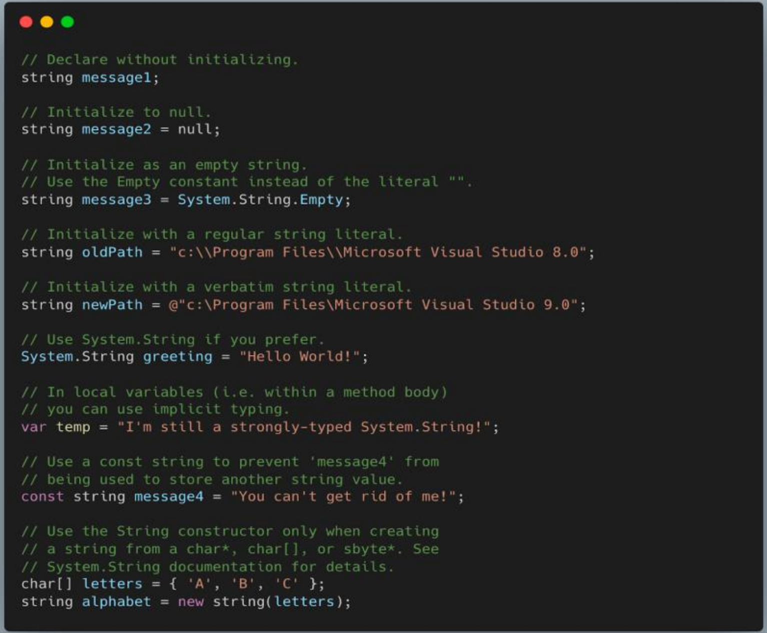
.NET Academy

Strings

In C#, the string keyword is an alias for String; therefore, String and string are equivalent. It's recommended to use the provided alias string as it works even without using System;. 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. For more information about the type and its methods, see String.

Strings

You can declare and initialize strings in various ways, as shown in the following example:

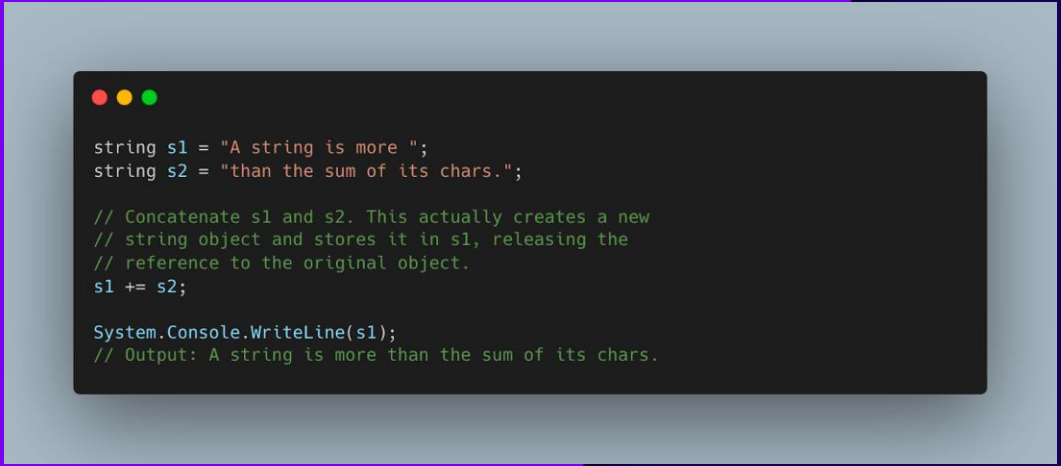


```csharp
// Declare without initializing.
string message1;

// Initialize to null.
string message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

// Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";

// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

.NET Academy

Strings

String objects are immutable: they can't be changed after they've 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.

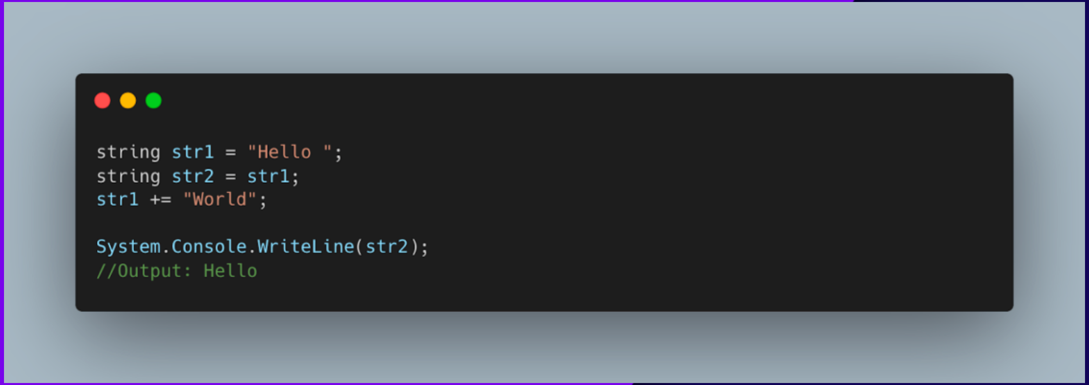


```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

.NET Academy

Strings

Because a string "modification" is actually a new string creation, you must use caution when you create references to strings. If you create a reference to a string, and then "modify" the original string, the reference will continue to point to the original object instead of the new object that was created when the string was modified. The following code illustrates this behavior:



```
string str1 = "Hello ";
string str2 = str1;
str1 += "World";

System.Console.WriteLine(str2);
//Output: Hello
```

.NET Academy

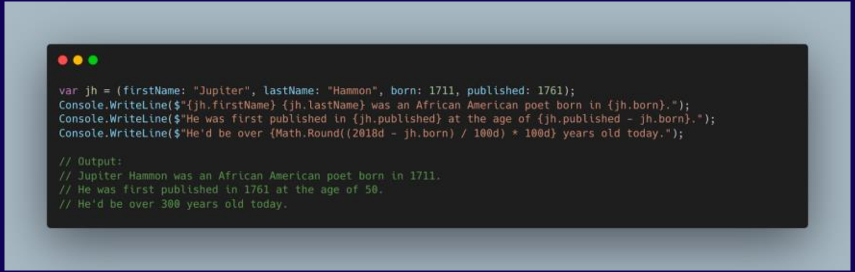
Formatting

A format string is a string whose contents are determined dynamically at run time. Format strings are created by embedding interpolated expressions or placeholders inside of braces within a string. Everything inside the braces ({...}) will be resolved to a value and output as a formatted string at run time. There are two methods to create format strings: string interpolation and composite formatting.

Interpolated strings are identified by the $ special character and include interpolated expressions in braces. If you're new to string interpolation, see the String interpolation - C# interactive tutorial for a quick overview.

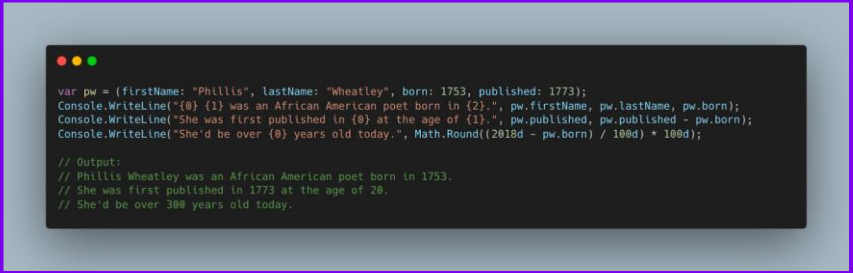
Formatting

Use string interpolation to improve the readability and maintainability of your code. String interpolation achieves the same results as the String.Format method, but improves ease of use and inline clarity.



```csharp
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published: 1761);
Console.WriteLine($"{jh.firstName} {jh.lastName} was an African American poet born in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of {jh.published - jh.born}.");
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) * 100d} years old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.
```

The String.Format utilizes placeholders in braces to create a format string. This example results in similar output to the string interpolation method used above.



```csharp
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published: 1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.", pw.firstName, pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.", pw.published, pw.published - pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d - pw.born) / 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

.NET Academy

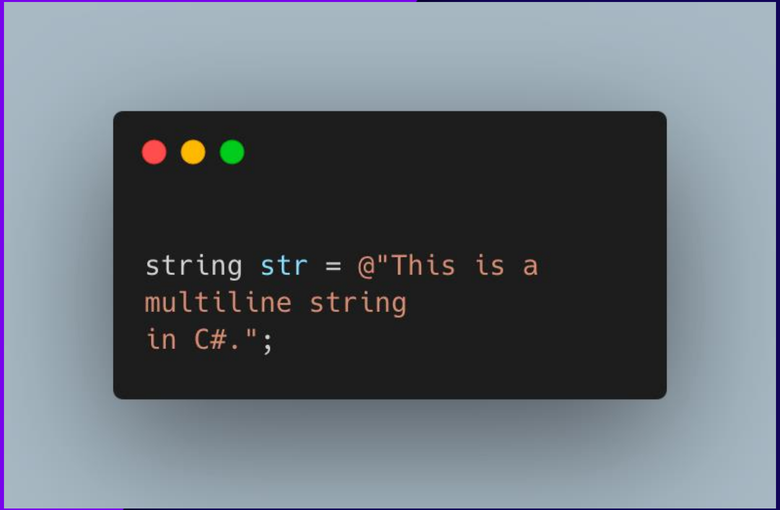
Multilinestring.

In C#, a multiline string is a string that spans multiple lines. It's different from a single line string, which only consists of characters on one line. A multiline string in C# is created using string literals, which are a series of characters enclosed in double quotes. To form multiline strings, we need to use a special type of string literals called verbatim string literals.

.NET Academy

Multilinestring.

To create a multiline string literal in C#, we use verbatim strings. Verbatim strings are prefixed with the @ symbol and allow us to include line breaks, special characters, and even whitespace without using escape sequences.

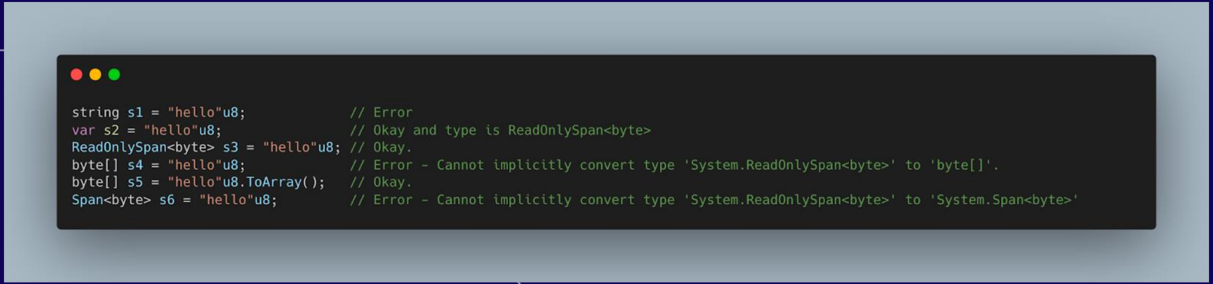


```
string str = @"This is a
multiline string
in C#.";
```

.NET Academy

UTF-8 string literals.

This proposal adds the ability to write UTF8 string literals in C# and have them automatically encoded into their UTF-8 byte representation.
The language will provide the u8 suffix on string literals to force the type to be UTF8. The suffix is case-insensitive, U8 suffix will be supported and will have the same meaning as u8 suffix.
When the u8 suffix is used, the value of the literal is a ReadOnlySpan<byte> containing a UTF-8 byte representation of the string. A null terminator is placed beyond the last byte in memory (and outside the length of the ReadOnlySpan<byte>) in order to handle some interop scenarios where the call expects null terminated strings.



```
string s1 = "hello"u8;             // Error
var s2 = "hello"u8;                // Okay and type is ReadOnlySpan<byte>
ReadOnlySpan<byte> s3 = "hello"u8; // Okay.
byte[] s4 = "hello"u8;             // Error - Cannot implicitly convert type 'System.ReadOnlySpan<byte>' to 'byte[]'.
byte[] s5 = "hello"u8.ToArray();   // Okay.
Span<byte> s6 = "hello"u8;         // Error - Cannot implicitly convert type 'System.ReadOnlySpan<byte>' to 'System.Span<byte>'
```

.NET Academy

Questions for the study of the topic

1. What is a top-level operator in C#? What is its role in a programme?
2. What language constructs allow you to create a programme in C# without the Main method?
3. What is the main difference between top-level operators and traditional Main method?
4. What language elements are used to declare variables in C#?
5. What are Type Aliases in C#? What is their purpose and how are they declared?
6. How to perform string formatting in C#? What methods or operators are used for this?
7. What are multiline strings and how to create them in C#?

.NET Academy

Homework

1. What are the two main ways of creating a programme in C# that you have learnt in the lesson?
2. How do top-level operators differ from the traditional Main method?
3. How to declare a variable in C#? Give an example.
4. What data types can you use to declare variables? Give 5 examples.
5. What are Type Aliases and how can they be useful in your code?
6. What operations can you perform on string type variables in C#?
7. How can you create a multi-line string in C#? Give an example.
8. What methods for working with strings did you learn in class? Describe what they can be used for.
9. What are the advantages and disadvantages of using top-level operators when creating programmes without the Main method?

Thank you for your attention.

.NET Academy