**Session 1: Types, Keywords, and Operators with C#**

In this first session, we're going to start learning about object-oriented programming with C#. We'll touch on the very basics around variable types, language keywords, and how we can use operators to make variables work together for us.

**Get C# to run locally**

You can get all of the tools to build with C# locally on your Mac or PC by installing [Visual Studio or Visual Studio for Mac](https://visualstudio.com?WT.mc_id=visualstudio-twitch-jefritz). You can also install just the build tools for Windows, Mac, or Linux by following the instructions at [dot.net](https://dot.net?WT.mc_id=visualstudio-twitch-jefritz) For most of our streams to start, we will be writing code using [Jupyter Notebooks and .NET Interactive](https://github.com/dotnet/interactive).

**What is C# ?**

C# is a statically-typed, compiled, object-oriented language used with the .NET frameworks and runtimes. The [official Microsoft C# language reference](https://docs.microsoft.com/dotnet/csharp/language-reference/?WT.mc_id=visualstudio-twitch-jefritz) can be found on docs.microsoft.com. Similar to Java in syntax and structure, C# programs are compiled and executed against a .NET runtime running on a computer. The output of compiling C# code can be called a '.NET program'.



A .NET Runtime is a collection of commands native to the computer operating system that instruct the computer how to interpret and run a .NET program.



There are several different .NET runtimes available that give C# flexibility to run in many different locations.



* .NET Framework - runs on Windows and support desktop user-interface, console, and server development



* .NET Core - runs on Windows, Mac, and Linux with support for desktop user-interface, console, and server development



* Xamarin - runs on iOS and Android devices with support for native application development on those devices



* Unity - runs on Windows, Mac, Linux, iOS, and Android devices with support for game development using the Unity3D tools



* Mono - runs on Windows, Mac, Linux, and Web Assembly



A .NET Framework is a collection of programming instructions and tools that help you write a program of a specific type. Examples of .NET Frameworks include Windows Forms, ASP.NET, Xamarin iOS, and Blazor



C# requires a .NET runtime and frameworks for the appropriate program type to run. The definition of the framework and runtime for a C# program are stored in a .csproj file. We'll learn more about this file and structure in a future lesson. For now, know that the .NET tools will help construct and manage this file for you when you specify what type of program you want to create.



All C# files carry a .cs file extension by default.



**Syntax 101**

Here are the basics of C# code syntax that you should know as we get started.

**C# uses a semi-colon to denote the end of a statement**

This is a hard and fast rule, and you'll get used to it quickly. Every statement in C# needs to end with a semi-colon ; This allows us to also have very flexible spacing in how we structure our code.



**C# is NOT space sensitive**



You can place as many spaces, tabs, or blank lines around your code as you would like.

**C# IS case sensitive**

C# is case-sensitive. All variables, objects, and their interactions must be referenced with the proper casing.



**Comment Syntax**

You can write comments by using the two forward-slash characters to indicate everything after them is a comment.

// This is a comment



You can create comments that span multiple lines by using slash asterisk fencing like the following:



/\*



This is a multi-line comment



​

and this is still commented out

\*/



**Everything in C# is an object**

As C# is an object oriented language, everything we want to work with is an object. Objects can be declared of various **TYPES** and then interacted with. The simplest types in C# are called [**Built-In Types**](https://docs.microsoft.com/dotnet/csharp/language-reference/builtin-types/built-in-types?WT.mc_id=visualstudio-twitch-jefritz)



We can define variables, in-memory storage for a type by preceeding the name of the variable we would like to create with the type of the variable we are creating.





int i;

double j;



char c;



​

display(c);

That's not very exciting, but we created a 32-bit integer named i. We can initialize a variable with an = assignment at the time of declaration.





int i = 10;



​

display(i);

10

**The var keyword**

Sometimes, its a little cumbersome to declare a variable, assign a value, and have to specify the type before it. C# has built-in type inference and you can use the var keyword to force the compiler to detect the actual type being created and set the variable to the type of the value being assigned.





var i = 10;



var someReallyLongVariableName = 9;

var foo = "Something";

​

display(someReallyLongVariableName);

​

var c = 'C';



display(c);

9

C

You can **ONLY** use the var keyword when creating and assigning the variable in one statement.

**Target-Typed new expressions**



Starting with [C# 9 (.NET 5 and later)](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/target-typed-new), types can not be declared and constructed using the new() keyword without any specification for the constructor. If the type can be inferred from it's usage, the new() keyword can be used.



You can still use constructor arguments with new() to initialize the appropriate type.





//List<string> testList = new();

List<string> testList = new() { "Hello", "World" };



​

display(testList);

| ***index*** | **value** |
| --- | --- |
| 0 | Hello |
| 1 | World |

**Real Literals**



We can declare double, float, and decimal types with simple numeric notation, but we need to force the literal numbers we assign to be the correct type to match the variable type expected.



To do this, we add a d, f, or m suffix to a number being assigned.





var myNumber = 4m;



myNumber.GetType()



[System.Decimal](https://docs.microsoft.com/dotnet/api/system.decimal?view=net-5.0)



**Type Casting**





int valueA = 10;



decimal valueB = valueA; // Implicit conversion



​

display(valueB);

display(valueB.GetType());

​

decimal valueC = 20;



//int valueD = valueC; // This errors out because int cannot be implicitly converted to by a decimal



int valueD = (int)valueC; // Explicitly convert valueC to int with the (int) modifier



​

display(valueD);

10



[System.Decimal](https://docs.microsoft.com/dotnet/api/system.decimal?view=net-5.0)



20



**Operators**



Now that we have some basic types and can create variables, it would sure be nice to have them interact with each other. [Operators](https://docs.microsoft.com/dotnet/csharp/language-reference/operators/) can be used to interact with our variables.



Let's start by declaring two variables, apples and oranges and interact with them using different operators. Try changing some of the values and tinkering with the operators in the following code snippets.





var apples = 100m; // Decimal value



var oranges = 30m; // Decimal value



Basic arithmetic operators and assignment are available:





display(apples + oranges);



130





display(apples - oranges);



70



display(apples \* oranges);



3000





display((int)apples / 7m);



47.571428571428571428571428571



You can use the equals character = to assign values, and prefix it with an arithmetic operator to modify and assign the resultant value.





display(apples += 10);



​

display(apples -= 10);



​



display(apples \*= 10);



​

display(apples /= 3m);



​

display(30.0d == 30);



110

100



1000



333.33333333333333333333333333



True

C# makes the inequality operators available as well, and a test for equality using ==





display(apples > oranges);



True





display(apples >= oranges);



True





display(apples < oranges);



False



display(apples <= oranges);



False





display(apples == oranges);



False





display(apples != oranges); // The not-equals operator



True



**Date Types**

Dates are a more complex data type that you can interact with by using the DateTime type. We can assign a new Date and time with a new statement to construct the DateTime type. [Complete documentation of the DateTime type](https://docs.microsoft.com/dotnet/api/system.datetime?view=net-6.0&WT.mc_id=visualstudio-twitch-jefritz) is available at docs.microsoft.com





DateTime today = new DateTime(2020, 8, 1, 9, 15, 30); // Create a date for August 1, 2020 at 9:00am



​

display(today);

display(today.Hour); // We can reference parts of the DateTime as properties on the variable



display(today.Minute);

display(today.Second);

2020-08-01 09:15:30Z

9

15

30

There are several properties on the DateTime object that allow us to interact with the variable type itself:





display(DateTime.Now); // Display the current local time



​

display(DateTime.MaxValue); // Display the maximum date value



display(DateTime.MinValue); // Display the maximum date value



2021-08-09 10:37:30Z

9999-12-31 23:59:59Z



0001-01-01 00:00:00Z



We can also add durations to our date like days and hours. The TimeSpan type is available to define a time duration that we can interact with.





display(today.AddDays(7)); // Add a week to August 1



​

TimeSpan ThreeHours = TimeSpan.FromHours(3); // Define a TimeSpan of 3 hours



display(ThreeHours); // Show 3 hours as a string

​

display(today.Add(ThreeHours)); // Add 3 hours to 'today' and display the result



​



2020-08-08 09:15:30Z

03:00:00

2020-08-01 12:15:30Z



**DateTimeKind**



A DateTime type carries a Kind property to indicate if it is a Local time or a reference to Utc time.





var thisDate = new DateTime(2021, 8, 10, 10, 0, 0);



display(thisDate.Kind);

Unspecified



var utcNow = DateTime.UtcNow;



display(thisDate.Kind);



Unspecified



var utcKindDate = new DateTime(2021, 8, 10, 10, 30, 0, DateTimeKind.Utc);



display(utcKindDate.Kind);



​



display(utcKindDate.ToLocalTime());



display(utcKindDate.ToUniversalTime());



Utc

2021-08-10 06:30:00Z



2021-08-10 10:30:00Z



**Tuples**



A [Tuple](https://docs.microsoft.com/dotnet/csharp/language-reference/builtin-types/value-tuples?WT.mc_id=visualstudio-twitch-jefritz) can be used to group together related data elements in a lightweight structure.



Use parenthesis with the value-types to combine:





(int, int) point = (2, 3);



display(point);



| **Item1** | **Item2** |
| --- | --- |
| 2 | 3 |

You can access the items in the Tuple with the Item# properties of the Tuple instance:





display(point.Item1);



display(point.Item2);



2

3

Tuples can have names assigned to the properties as well:





(decimal Latitude, decimal Longitude) Philadelphia = (39.95233m, -75.16379m);



display(Philadelphia);



​

display(Philadelphia.Latitude);



display(Philadelphia.Longitude);



| **Item1** | **Item2** |
| --- | --- |
| 39.95233 | -75.16379 |

39.95233



-75.16379

