

Everything to know about C# Syntax in 10 minutes

(The information below was obtained from the Train To Code YouTube channel, all credit goes to them. I summarized what I learnt from them below.)

- It's part of the c programming languages (Aka java,go,r,objective-c)
- Use's semicolons and curly braces
- Developed by Microsoft but is now open-source and works on all platforms
- Can be built to run on Linux, Windows, Mobile, Mac and Webassembly thanks to [Xamarin](#) and [Blazer](#)

```
void SayHello(string name)
{
    Console.WriteLine($"Hello {name}");
}

SayHello("James");
```

- The function has a return type of void, has the name SayHello and has an argument of string name.
- The body of the function is enclosed in curly brackets and each line ends in a semi-colon
- Comment with two forward slashes "//comment here"

```
var myString = "string value";

var myNumber = 123;
```

- Var in C# is not the same as in JS(Javascript), nowadays devs use var for everything because they want the compiler to work out what variable it is for you. It doesn't mean that the variable can be any type like in JS
- [Implicitly typed variables](#) are good to use
- Data Types in C# (string, int, float, double,bool, DateTime etc)

```
public class MyClass
{
    protected void MyMethod()
    {
        Console.WriteLine("Hello");
    }
}
```

```
public class MyClass
private
protected
internal
file
```

- You can also create your own data types through:

Classes(These are defined with the class keyword and can have 1 of 5 access modifiers that define the visibility of the class. *Public*(visible to everyone, everywhere) *Private*(Just within the scope of where the class is declared), *Protected*(The declaring scope of anything inherited from the class), *Internal*(Means just as assembly) and *File*(Means just as File).

Classes can have methods, methods can have their own access modifiers as long as they are more restrictive than that of the class, so you can't have a public method in a private class.

```
public abstract class BaseClass
{
    public abstract void MyMethod();
}

public class MyClass: BaseClass
{
    public override void MyMethod()
    {
        Console.WriteLine("Hello");
    }
}
```

You can inherit classes, through a special keyword called **abstract**(denotes that this class is only for inheriting. You can put it on the class itself and on any method if you want it to be implemented in any overriding class.)

```
public interface IMyInterface
{
    void MyMethod();
}

public class MyClass: IMyInterface
{
    public void MyMethod()
    {
        Console.WriteLine("Hello");
    }
}
```

Classes can also be implemented as any interfaces, it just defines methods and properties that the implementing classes must contain.

```
public class MyClass
{
    public string MyProperty { get; set; }

    public int OtherProperty { get; set; }
}
```

C# has a nice shorthand syntax for declaring properties by using the get and set keywords which you can expand by adding some custom logic to it

```

public class MyClass
{
    public string MyProperty { get; set; }

    private int _otherProperty;
    public int OtherProperty
    {
        get { return _otherProperty; }
        set { _otherProperty = value; }
    }
}

```


By default, you just need to write get and set and the compiler will create the property for you.

```

public class MyClass
{
    public string MyProperty { get; set; }

    public int OtherProperty { get; private set; }
}

```



You can add restrictive modifiers to the get and set.

```

public class MyClass
{
    public string MyProperty => "VALUE";

    public string MyPropertyA
    {
        get
        {
            return "VALUE";
        }
    }
}

```

There is a useful shorthand for read-only properties

```

public class MyClass
{
    public string ValueA { get; }

    public MyClass(string valueA)
    {
        ValueA = valueA;
    }

    public void Deconstruct(out string valueA)
    {
        valueA = this.ValueA;
    }
}

```

Classes also have *constructors* (Eg: MyClass, which takes a bunch of parameters into the class when a new instance is created) and *destructors* (Spits out a bunch of parameters from within the class)

```


var instance = new MyClass("one", "two");
var (one, two) = instance;

public class MyClass
{
    public string ValueA { get; }
    public string ValueB { get; }

    public MyClass(string valueA, string valueB)
    {
        ValueA = valueA;
        ValueB = valueB;
    }

    public void Deconstruct(out string valueA, out string valueB)
    {
        valueA = this.ValueA;
        valueB = this.ValueB;
    }
}

```




Use a constructor when you want to create a new instance of the class (Eg: var instance=new...)

To use a deconstructor, you just wrap the variables you want in deconstruct (Eg: var(one,two)=instance;)

```

var instance = new MyRecord("one", "two");
var (one, two) = instance;
record MyRecord(string one, string two);

```



Record types can be deconstructed the same way

```
record MyRecord(string fieldA, string fieldB);

struct MyStruct
{
    public string FieldA;
    public string FieldB;
}

class MyClass
{
}
```

Structs(used for creating more primitive data types, the advantage of it is that they exist in the execution stack of your program. They are not stored in heap memory like other objects. Good for storing readily accessible data or small data types) and

Records(a quick and easy way of representing the shape of data. They are immutable so once created you can't change it)

Now that we are done going over the object-oriented stuff, let's look at the control flow of the program:

- Control statements in C# include *if and else* statements, curly brackets are required if you have more than one line in the if statement.


```

var value = 55;

string MyFunction(int value)
{
    switch (value)
    {
        case 1: return "one";
        case 2: return "two";
        case > 10: return "large number";
        default: return "unknown";
    }
}

```

- We also have the switch statements.

```

for(var i = 0; i < 10; i++)
{
    Console.WriteLine(i);
}

var j = 0;
do
{
    Console.WriteLine(j);
    j++;
} while (j < 10);

```



- We also have *for loops* and *do while loops*.

```

var values = Enumerable.Range(1, 10).ToArray();

foreach(var value in values)
{
    Console.WriteLine(value);
}

```

- The most common type of loop is the *foreach*, C# uses *while* JS uses *of*

Let's look at collection types:

```
var myArray = new string[5];|
```

- C# has arrays

```
using System.Collections;

var myList = new List<string>();


var myDictionary = new Dictionary<int, int>();

var myQueue = new Queue<string>();

// Plus loads more!
```

- It also has a bunch of collection methods.

```
using System.Collections;


class MyEnumerable : IEnumerable 
{
    private readonly string[] _data = new string[] { "one", "two"};

    public IEnumerator GetEnumerator()
    {
        return _data.GetEnumerator();
    }
}
```

- All of these collection methods and arrays implement an interface called IEnumerable.
- IEnumerable is the interface that the foreach statement looks at and decides if something can be used in a forloop.

```
using System.Collections;

IEnumerable<int> InfiniteArray()
{
    var i = 0;
    while (true)
    {
        yield return i;
        i++;
    }
}
```



```
using System.Collections;

IEnumerable<int> SquareEach(IEnumerable<int> data)
{
    foreach(var item in data)
    {
        var squared = Math.Pow(item, 2);
        yield return item;
    }
}
```

- You can create your own class that implements IEnumerable if you want to and also a function that returns IEnumerable and generates functions using the yield statement.
- The function above uses a yield statement to create an infinite array.
- C# comes with a bunch of functions that you can use on IEnumerable collections, such as *Select* (basically a map in other languages) and *Aggregate* (a reduce function).

```
async Task SomeAsyncFunction()
{
    await Task.Delay(1000);

    await Task.Delay(500);
}
```

C# also supports the use of Async and Await to control asynchronous code execution. We use these by adding *async* to the function name and by calling *await* inside the function whenever we want to await for the execution of another asynchronous piece of code.

Generics are also included in C#, they are used in Triangle brackets on any function name or class definition, you can use the keyword `where` after the argument list to add restrictions to your generic type argument.

```
string GenericFunction<T>() where T : IThing
{
    return typeof(T).GetType().Name;
}
```

Above we see that argument `T` has to be something that implements the interface(`IThing`).