C# Fundamentals

By: Mosh Hamedani

Non-primitive Types

Classes

Classes are building blocks of our applications. A class combines related variables (also called fields, attributes or properties) and functions (methods) together.

Note: fields and properties are technically different in C# but conceptually they mean the same thing. They represent attributes about a class. I'll explain the difference between fields and properties in detail in my C# Intermediate course.

An *object* is an instance of a class. At runtime, many objects collaborate with each other to provide some functionality. As a metaphor, think of a supermarket. At a supermarket, there are multiple people working together to provide services to customers. Each person has a role and is focused only on one area of functionality. Software is exactly the same. A role in a supermarket is like a class in a C# application. A person filling that role during work hours, is like an object in an application at runtime.

Note that even though there is a slight different between the word Class and Object, these words are often used interchangeably.

To create a class:

```
public class Person
{
}
```

Here, public is what we call an *access modifier*. It determines whether a class is visible to other classes or not. Access modifiers are beyond the scope of this course and I've covered them in the second part of this course: C# Intermediate.

Here is a class with a field and a method:

```
public class Person
{
	public string Name;

	public void Introduce()
	{
		Console.WriteLine(“My name is “ + Name);
	}
}
```

Here void means this method does not return a value.

To create an object, we use the new operator:

```
Person person = new Person();
```

A cleaner way of writing the same code is:

```
var person = new Person();
```

We use the new operator to allocate memory to an object. In C# you don’t have to worry about de-allocating the memory. CLR has a component called Garbage Collector, which automatically removes unused objects from memory.

Once we have an object, we can access its fields and methods with the dot notation:

```
Person person = new Person();
person.Name = “Mosh”;
person.Introduce();
```

Static modifier

When applied to a class member (field or method), makes that member accessible only via the class, not any objects. So in the earlier example, if the Introduce method was static, we could access it via the Person class:

```
Person.Introduce();
```

We use static members in situations where we want only one instance of that member to exist in memory. As an example, the Main method in every program is declared as static, because we need only one entry point to the application.

In the real-world, it's best to stay away from static as much as you can because that makes writing automated tests for applications hard. Automated testing is the topic for another course, but for now, just remember how to use members that are already declared as static, and prefer not to declare your own class members as static.

Structs

A struct (structure) is a type similar to a class. It combines related fields and methods together.

```
public struct RgbColor
{
     public int Red;
     public int Green;
     public int Blue;
}
```

Use structs only when creating small lightweight objects. That is for a subtle performance optimization. In the real-world, 99% of the time, you create new types using classes, not structures.

In .NET, all primitive types are declared as a structure. They are small and lightweight. The biggest primitive type doesn't take more than 16 bytes.

Arrays

An array is a data structure that is used to store a collection of variables of the same type.

For example, instead of declaring three int variables (that are related), we can create an int array like this:

```
int[] numbers = new int[3];
```

An array in C# is actually an instance of the Array class. So, that's why here we have to use the new operator to allocate memory to this object.

Here, the number 3 specifies the size of the array. Once an array is created, its size cannot be changed. If you need a list with dynamic size, you need to use the List class (explained later in the course).

To access elements in an array, we use the square bracket notation:

```
numbers[0] = 1;
```

Note that in C# arrays are zero-indexed. So the first element has index 0.

Strings

A string is a sequence of characters. In C# a string is surrounded by double quotes, whereas a character is surrounded by a single quote.

```
string name = "Mosh";
```

```
char ch = 'A';
```

There are a few different ways to create a string:

Using a string literal:

```
string firstName = "Mosh";
```

Using concatenation: useful if you wanna combine two or more strings.

```
string name = firstName + " " + lastName;
```

Using string.Format: cleaner than concatenating multiple strings since you can see the output.

```
string name = string.Format("{0} {1}", firstName, lastName);
```

Using string.Join: useful when you have an array and would like to join all elements of that array with a character:

```
var numbers = new int[3] { 1, 2, 3 }

string list = string.Join(",", numbers);
```

C# strings are immutable, which means once you create them, you cannot change their value or any of their characters. The String class has a few methods for modifying strings, but all these methods return a new string and do not modify the original string.

String vs string

Remember, all types in C# map to a type in .NET Framework. So, the "string" type in C# (all lowercase), maps to the String class in .NET, which means we can declare a string in either of the following ways:

```
string name;

String name;
```

The only difference is that if you use the String type, you need to import the System namespace on top of the file, because that's where the String class is defined.

```
using System;
```

Escape Characters

There are few special characters in C# called escape characters:

Escape Character	Description
\n	New line
\t	Tab
\\	The \ character itself
\’	The ‘ (single quote) character
\"	The “ (double quote) character

So if you want to have a new line in your string, you use \n.

Since the backslash character is used to prefix escape characters, if you want to use the backslash character itself in your string (eg path to a folder), you need to prefix it with another backslash:

```
string path = “c:\\folder\\file.txt”;
```

Verbatim Strings

Sometimes if there are many escape characters in a string, that string becomes hard to read and understand.

```
var message = “Hi John\nLook at the following path:c:\\folder1\
\folder2”;
```

Note the \n and double backslashes (\\) here. We can re-write this string using a *verbatim string.* We simply prefix our string with an @ sign, and get rid of escape characters:

```
var message = @“Hi John

Look at the following path:

c:\folder1\folder2”;
```

Reference Types and Value Types

In C#, we have two main types from which we can create new types: classes and structures (structs).

Classes are *Reference Types* while structures are *Value Types.*

Value Types

When you copy a value type to another variable, a copy of the value stored in the source variable is taken and stored in the target variable. Hence, these two variables will be independent.

```
var i = 10;

var j = i;

j++;
```

Here, incrementing j does not impact i.

In practical terms, it means: if you pass an argument to a method and that argument is a value type, its value will be copied. So any modifications made to that argument in the method will be lost upon returning from that method.

Remember: Primitive types are structures so they are value types. Any custom structure you define will also be a value type.

Reference Types

With a reference type, however, the reference (or memory address) of the object is copied to the target variable. This means: if you copy a reference type to another variable, any changes you make to the object referenced by either of these variables, will be visible through the other variable.

```
var array1 = new int[3] { 1, 2, 3 };

var array2 = array1;

array2[0] = 0;
```

Here, both array1 and array2 reference (or point) the same array object in memory. So, after the third line, the first element of both array1 and array2 will be 0.

Remember: arrays and strings are classes, so they are reference types. Any custom classes you define will also be a value type.

Enums

An enum is a data type that represents a set of name/value pairs. Use enums when you need to define multiple related constants.

```
public enum ShippingMethod

{
     Regular = 1,
     Express = 2

}
```

Now we can declare a variable of type ShippingMethod enum and use the dot notation to initialize it:

```
var method = ShippingMethod.Express;
```

Enums are internally integers. So you can easily cast them to and from an int:

```
var methodId = 1;

var method = (ShippingMethod)methodId;
```

```
var method = ShippingMethod.Express;

var methodId = (int)method;
```

To convert an enum to a string use the ToString method. Every object in C# has this method and can be converted to a string:

```
var method = ShippingMethod.Express;
```

```
var methodName = method.ToString();
```

To convert a string to an enum (called parsing), use Enum.Parse:

```
var method = (ShippingMethod)Enum.Parse(typeof(ShippingMethod),
methodName);
```