C#

From Scratch

HASSAN HABIB

C#: From Scratch

Hassan Habib

Free to read online at hassanhabib.com/books/csharp-from-scratch

Thank you, Lord for all you taught me. Forgive me if I forget anything

Contents

0	Intro
1	Starting A new Project
2	Variables
3	If Statements
4	Switch Cases
5	For-loops
5.5	The Infinite ‘for-loop’
6	While And Do-While
7	Foreach
8	Methods
9	Classes & Objects (Object Oriented Programming)
10	Course Recap

CHAPTER 0

Intro

Welcome to C# from Scratch. Here you will find the most comprehensive guide to developing your C# skills. I have been perfecting this craft for the last 17 years, and I intend to share every piece of my knowledge with you.

The great news is that you don't need to know a thing about algorithms, mathematics, or programming. You only need a working computer that has Visual Studio installed on it (just make sure to have the right version for your operating system, be it Mac, Linux, or Windows).

Now, I understand there are a lot of other resources out there that are teaching you the same thing. And you can certainly go check some of those out.

What you will find though, is that those resources don't give you the knowledge to THINK like a programmer. Or to THINK like a software engineer. You can learn the ins and outs of the language absolutely, but without the proper mindset, that information will be nothing more than a bunch of facts for you to memorize.

And that's not what you're looking for.

If you can't take a real-life problem, visualize it, and simulate it into a problem you can solve using programming language, then you're not getting the most out of C#.

I look forward to teaching you how to think, act, and solve problems like a REAL programmer.

CHAPTER 1

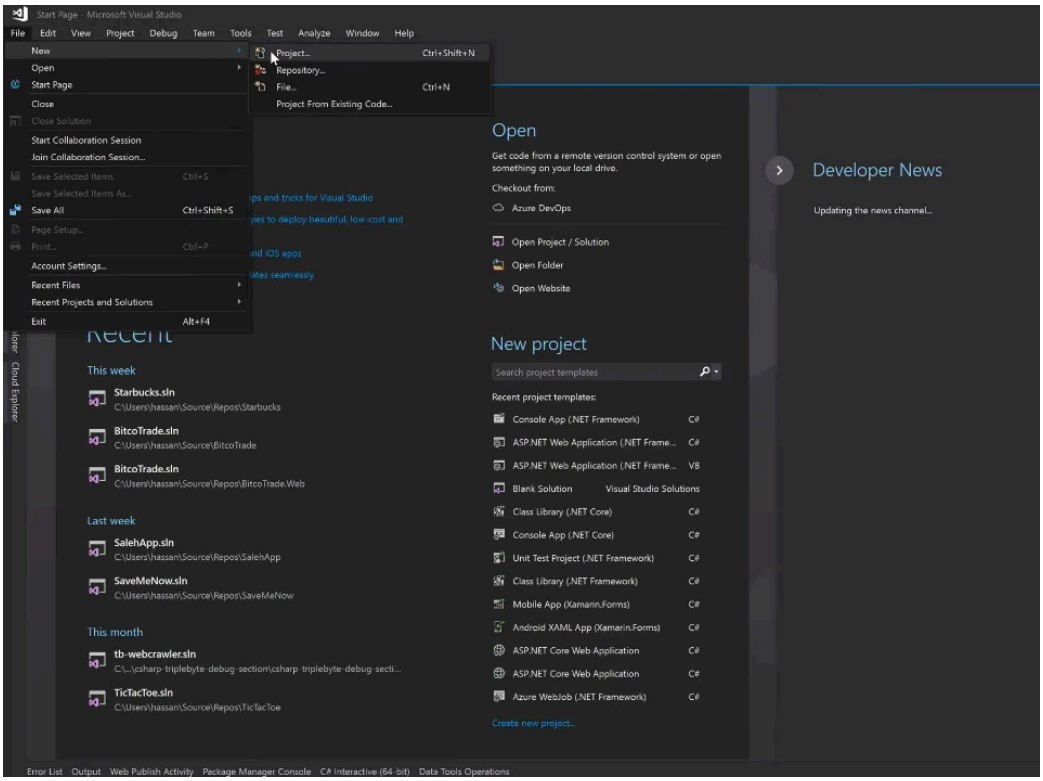
Starting A new Project

Now it's time to learn how to write in C#. Like I said before you need to download Visual Studio(known as an integrated development environment).

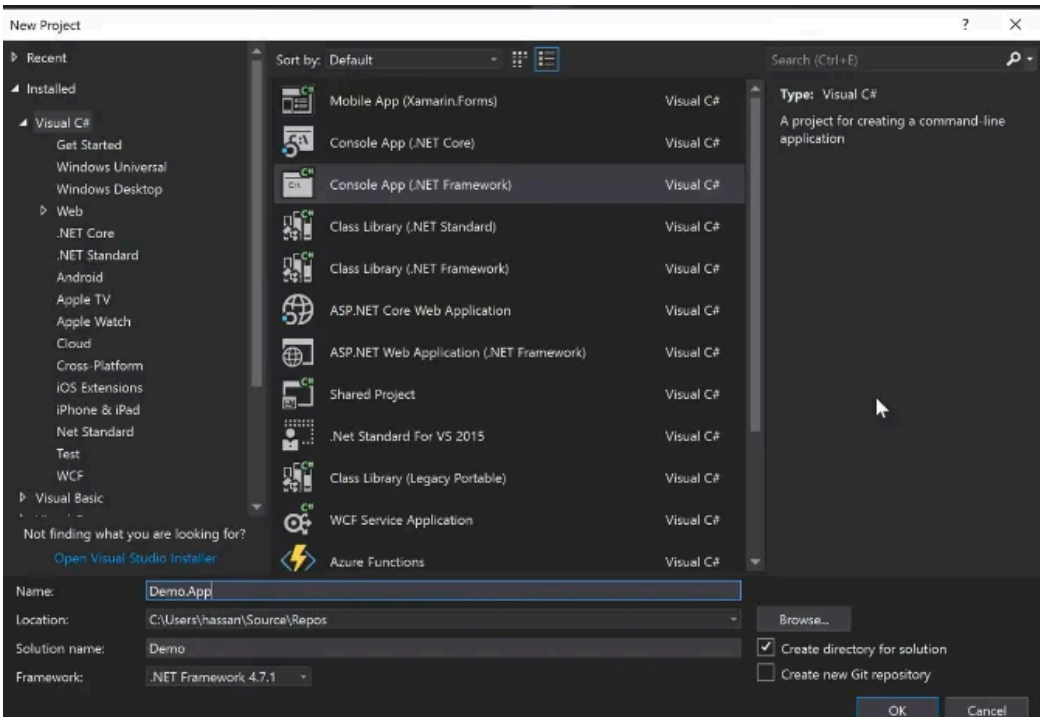
This will help you write C# code and you can turn that into whatever you want, be it a desktop, mobile, or web application.

Let's go ahead and start Visual Studio. For the sake of this example I will be using the 2017 version. Now the 2019 preview is out, and you can download it and give it a shot if you like. It has tons of great features. But for the sake of this book I'd recommend having the 2017 version.

Once you start Visual Studio 2017, at the top left of the program go ahead and hit File → New → Project. Since we are dealing with the most basic programming right now, let's start with a console application.



Go to Visual C# → Console App. At the bottom of the screen you will find the solution name. We're going to call our solution "Demo". Now you want to be as consistent as possible to stay organized so if your solution is called demo, then your app should be demo.app.



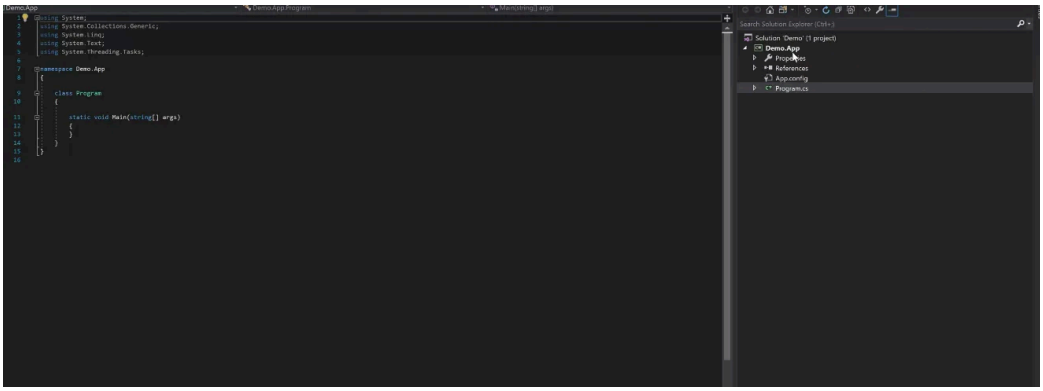
The organization of the solution is very important. You want your solution name and all the projects that come under that to make sense.

The way this works is the that the solution gets a name, then everything after that will take the solution name, then the project name.

For example, if you're building a system for a coffee shop, the solution name will be "coffeeshop". But then you will have extensions of that solution name - coffeeshop.web, coffeeshop.mobile, coffeeshop.android, coffeeshop.ios are some examples.

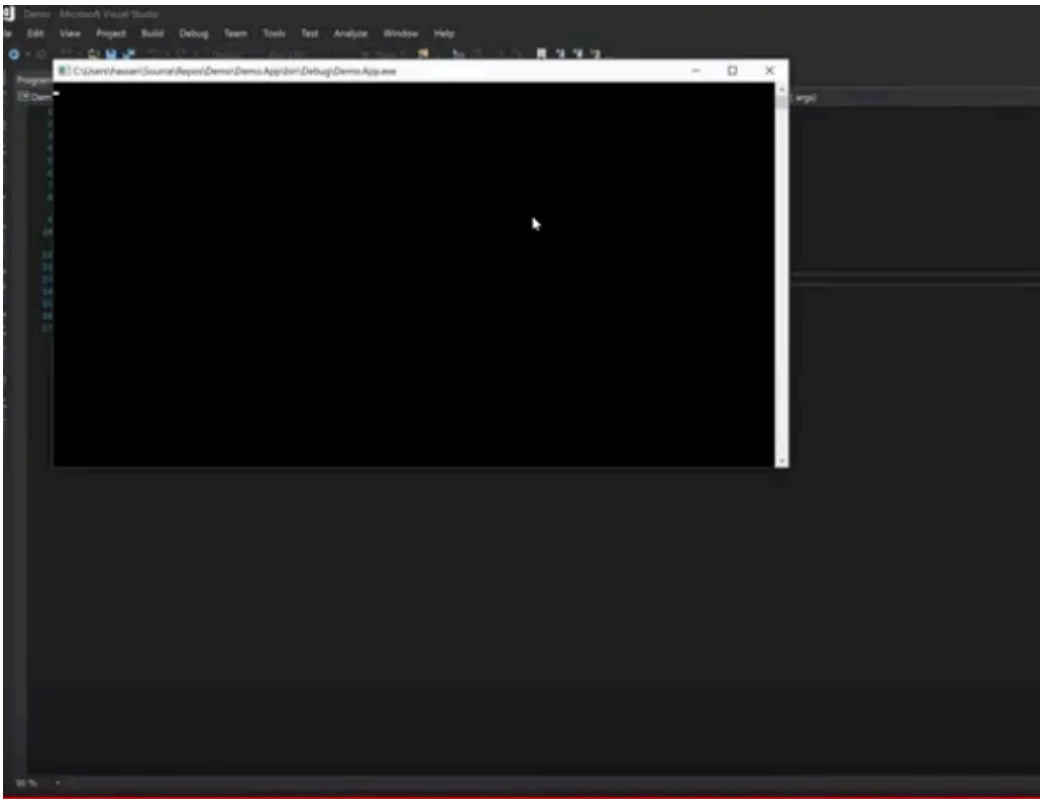
Make sure your project is ordered and organized this way so it's easier for you to know what projects are doing what. On top of that, it's also easier for other people to follow and understand what your project is doing and where to look for things when they're looking for a specific item or feature.

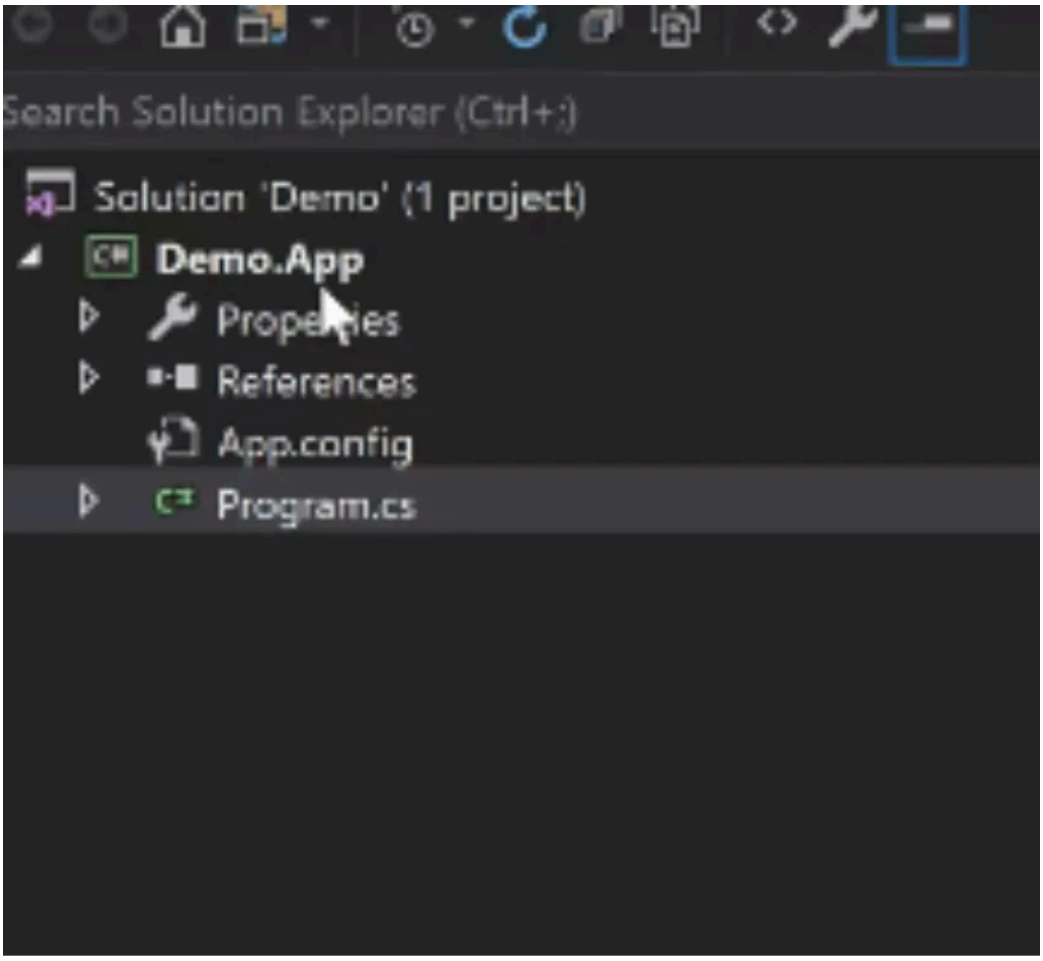
Now you'll be taken to this screen.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Program.cs will always be your entry point for console applications. It is the file that executes or runs. Console apps are the apps that just show a black screen - just pops up and shows you some stuff in it - console application(like the image below).


Demo - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window Help
C:\Users\hassan\Source\Repos\Demo\Demo.App\bin\Debug\Demo.App.exe



You will learn about different types of apps such as web apps and what-not.

So that was program.cs option.

Let's go through the other options here. You've got properties - these are basic static values that you may need in the future to connect to a database or to a storage. We will visit properties later.

The references option holds the libraries that you want to use. And the App.config is your application configuration. Don't worry about these right now. We will address them later.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Let's now look at the contents of program.cs.

At the top of the code(highlighted in blue), you will see the libraries that you may or may not need to write some code.

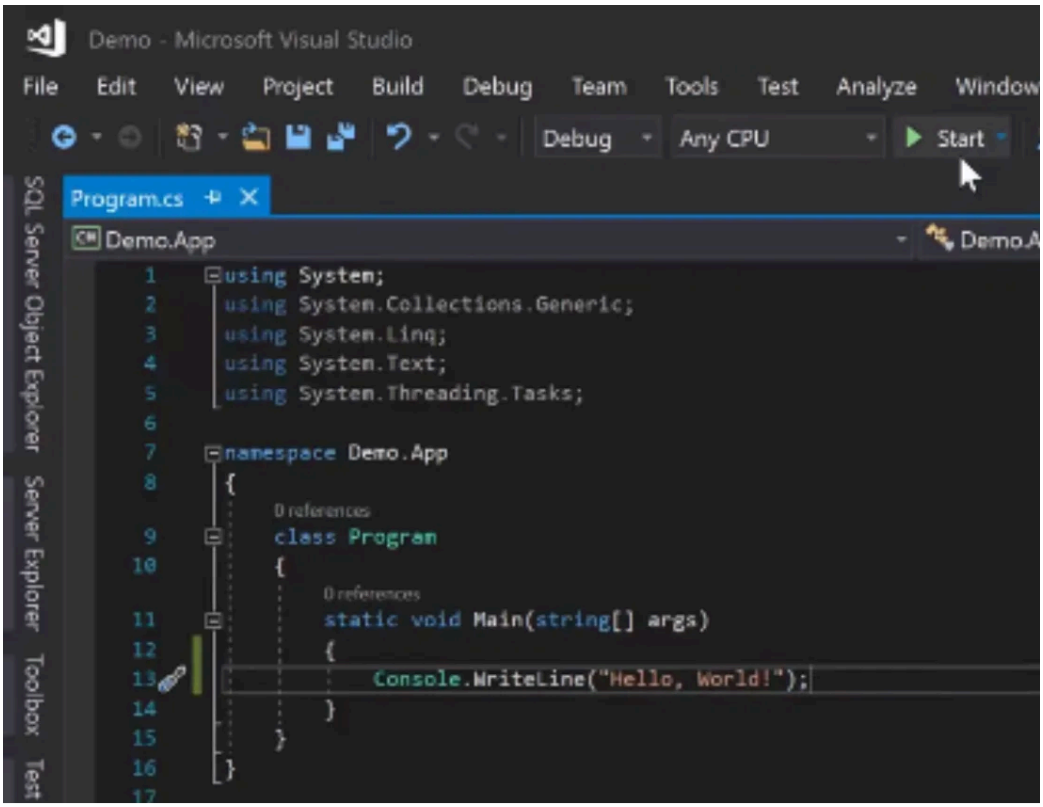
At the bottom of that code there is a namespace - that is a scope - if I hover over it you will see a lighter color - all of that lives in the namespace scope. We will cover this later.

```
using System.Threading.Tasks;

namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Further down there is also a class. C# as an object-oriented programming, is a bunch of classes that have few properties and few actions which we call methods.

```
namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

This main class is called `class Program` - that's your program that you're running and within that main method is where all our code is going to be for the next couple of sessions.

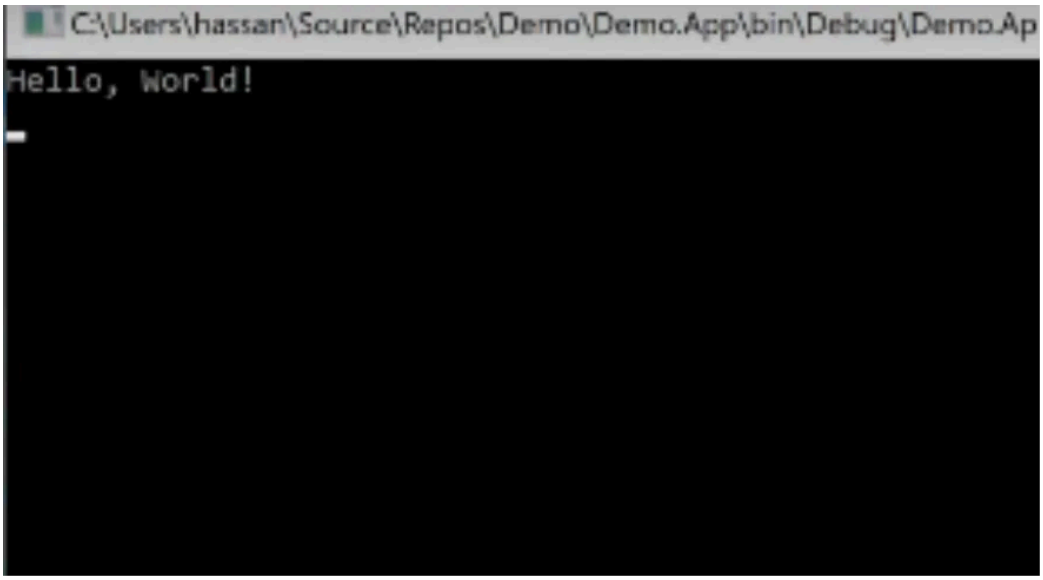



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

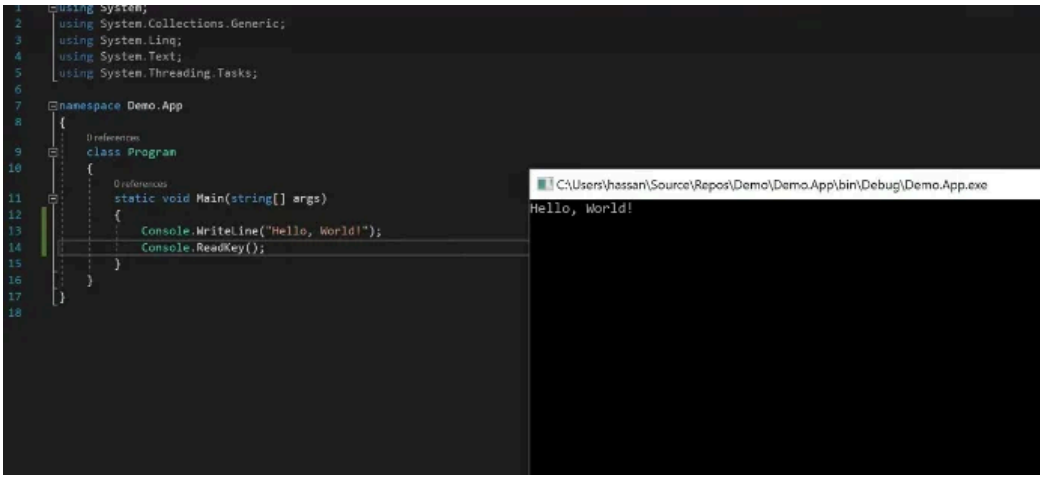
Your main method is where you begin. If I go in and type something like

```
Console.WriteLine("Hello, World!");
```

and then I click start at the top, the black monitor will pop up real quick and then it's gone. It shows `"Hello, World!"`, and then it disappears.



But if I add `Console.ReadKey();` - now the window is going to stay open until I press any key.




```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
            Console.ReadKey();
        }
    }
}
```

The Console window will always show you the message you have typed.

Now that I've given you some information, what I'd like you to do is look around, and try to build a project like this. Become familiar with how it works, and try to print different messages using `Console.WriteLine();` and `Console.ReadKey();`

CHAPTER 2

Variables

Now we are going to talk about variables and programming. Variables are basically spaces in memory that we use to store certain values. Think of programming as a way to simulate something in the real world. Don't think about the code you're looking at. Think about what problem you are trying to solve, and then use that code to solve that problem in the real world.

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            int itemOnePrice = 10;
            int itemTwoPrice = 20;

            Console.WriteLine("Hello, World!");
            Console.ReadKey();
        }
    }
}
```

For instance, if I want to run a program that calculates the prices of a bunch of items that I have, I can go and say:

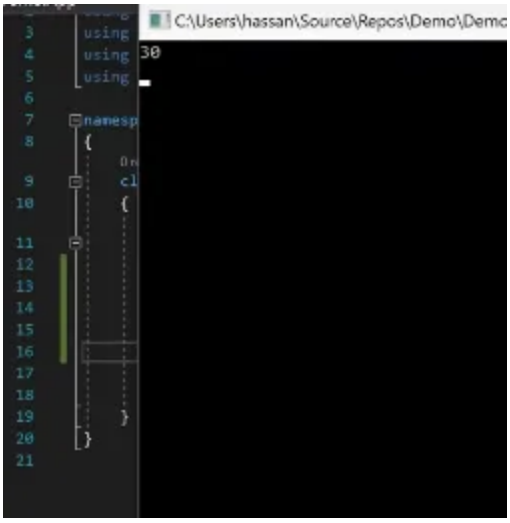
```
int itemOnePrice = 10;
int itemTwoPrice = 20;
```

And so on. If I want to calculate that value, I could store all these values in another variable - let's call it total price. And then I can add `itemOnePrice + itemTwoPrice` and then print that out as:

```
Console.WriteLine(totalPrice);
```

```
namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            int itemOnePrice = 10;
            int itemTwoPrice = 20;
            int totalPrice = itemOnePrice + itemTwoPrice;
            Console.WriteLine(totalPrice);
            Console.ReadKey();
        }
    }
}
```

Now when I run my program it will print out that total value of both of these prices. 10 + 20 = 30



Programming is basically the art of describing computer behaviors to solve a problem. When you're trying to solve problems you're going to run into the situation where you need to save some values in places so you can manipulate that data.

These values could come in different types. There are primitive types such as integers, strings, bytes, and unassigned integers. And then there are complex types that are made out of these types.

If I want to store a name for instance - I could store a name in a string value. (name = "Hassan")

```
namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            int itemOnePrice = 10;
            int itemTwoPrice = 20;

            string name = "Hassan";

            int totalPrice = itemOnePrice + itemTwoPrice;
            Console.WriteLine(totalPrice);
            Console.ReadKey();
        }
    }
}
```

Notice something different in here. When I store a string I use double quotes but when I stored numeric values I didn't have to use anything.

```
int itemOnePrice = 10;
int itemTwoPrice = 20;

string name = "Hassan";

int totalPrice = itemOnePrice + itemTwoPri
Console.WriteLine(totalPrice
Console.ReadKey();
```

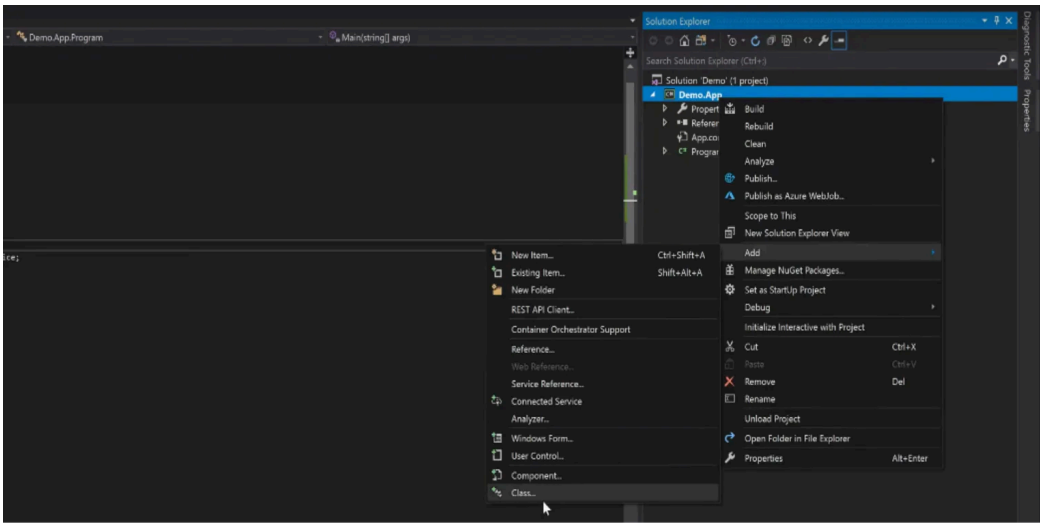
That's because strings, structurally, are fundamentally different than integers.

Programming is all about knowing how to describe the things that matter the most in the problem that you're trying to solve, and then visualizing that to your end user.

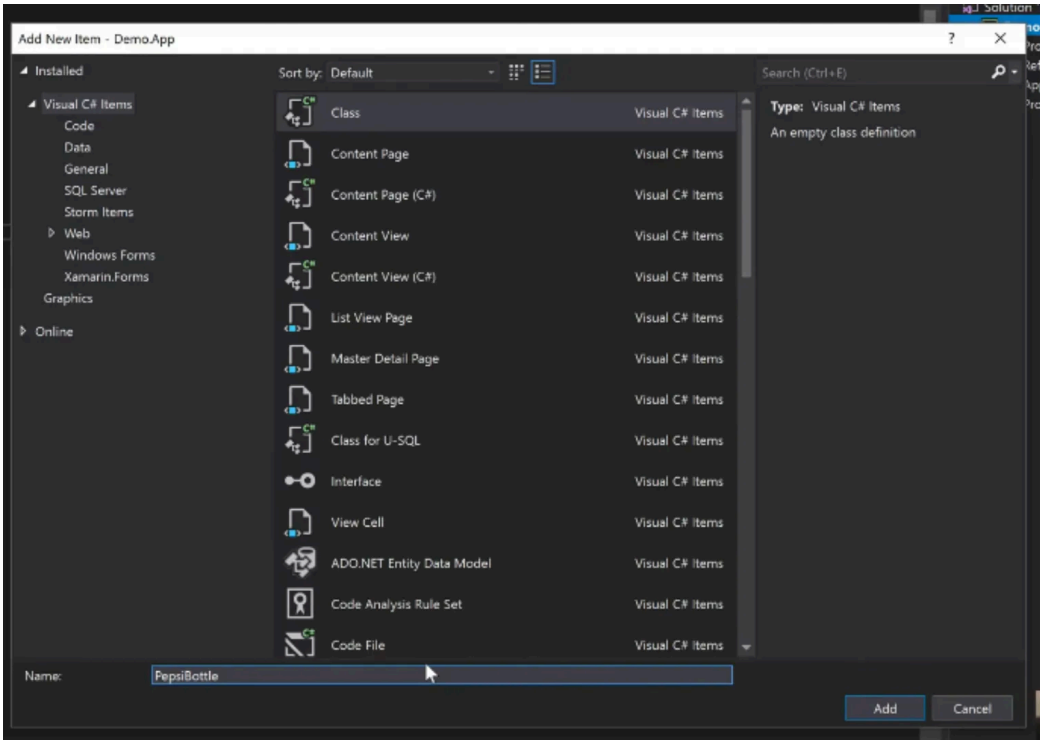
Don't get caught too much in mind teasing and trying to solve unrealistic problems. Try to find a problem in real life, and resemble it.

Example - Let's say I have a bottle of Pepsi, and I want to determine whether I can buy this bottle or not based on some ingredients in this bottle. So when I look at it, I see the values of protein, sodium, and total fat. All of these properties are integer values that I can contain in what I call complex type.

So I want to resemble this. Let's go in and create a complex type.



Let's call it `PepsiBottle`. And now this `PepsiBottle` can have a bunch of things in it.



I could go and say how much sodium is in this, how much sugar is in this, how much fat, etc. You can add these values to determine whether you resemble that or not.

The get; set; in the code below describes two functions, the first is the get which means the property on that class can be retrieved and read.

The second is the set, which means the property on this class can be assigned a value as we are going to show in the next few pages.

```
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class PepsiBottle
    {
        public int Sodium { get; set; }
        public int Sugar { get; set; }
        public int Fat { get; set; }

    }
}
```

If I want to use those values in my method, I could go and set values in this complex type variable.

```
pepsiBottle.Fat = 0;
pepsiBottle.Sodium = 10;
pepsiBottle.Sugar = 20;
```

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            int itemOnePrice = 10;
            int itemTwoPrice = 20;

            string name = "Hassan";

            PepsiBottle pepsiBottle = new PepsiBottle();
            pepsiBottle.Fat = 0;
            pepsiBottle.Sodium = 10;
            pepsiBottle.Sugar = 20;

            int totalPrice = itemOnePrice + itemTwoPrice;
```

```
            Console.WriteLine(totalPrice);
            Console.ReadKey();
        }
    }
}
```

So now you get to resemble your pepsi.

That's all I want you to learn in this chapter - How to look at things in the real world and how to resemble them.

Learn to use these primitive variables as part of these complex variables.

For your homework, go find out all of the different primitive data types that you can use for instance: booleans, characters, unassigned integers and many others, then try to resemble a real problem in your life.

Try to resemble a computer mouse for instance - what can you look at in a computer mouse that you can resemble in a class?

CHAPTER 3

If Statements

Now it's time to talk more about variables, and we will also introduce a concept called the "if" statements in the programming language. You're going to use 'if' statements throughout your entire life as long as you are writing code. They are very important.

Let's backtrack a bit to the last chapter and go over variables again.

It's very important to understand that when you're using variables, do not create variables if you're going to use their values only once.

For instance, I have a string variable called 'message = hello world'.

```
string message = "Hello, World";
```

```
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            string message = "Hello, World!";

            Console.WriteLine(message);
            Console.ReadKey();
        }
    }
}
```

If I'm using that message only once, it doesn't make sense to define a variable.

It would be better to simply take that value and throw it into the message, getting rid of the variable. That makes it easier and cleaner.

```
using System.Text;
using System.Threading.Tasks;
```

```
namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
            Console.ReadKey();
        }
    }
}
```

We delete the string message and just put the message right into our code:

```
Console.WriteLine("Hello, World!")
```

But if you're going to use the value of that message multiple times, then that's different. You can go and say,

```
if (message == ("Hello,World"), Console.WriteLine(message);
else
Console.WriteLine("Something else!");
```

```
string message = "Hello, World!";

if (message == "Hello,World")
{
    Console.WriteLine(message);
}
else
{
    Console.WriteLine("Something else!");
}

Console.ReadKey();
}
}
```

If your

`(message == ("Hello,World")`, then the console will print

`"Hello,World"`.

`Else`, it will write

```
"Something else!"
```

What I am saying is that I am checking the value of the message in

`if (message == "Hello,World")`

```
Program

atic void Main(string[] args)

string message = "Hello, World!";

if (message == "Hello,World")
{
    Console.WriteLine(message);
}
else
{
    Console.WriteLine("Something else!");
}

Console.ReadKey();
```

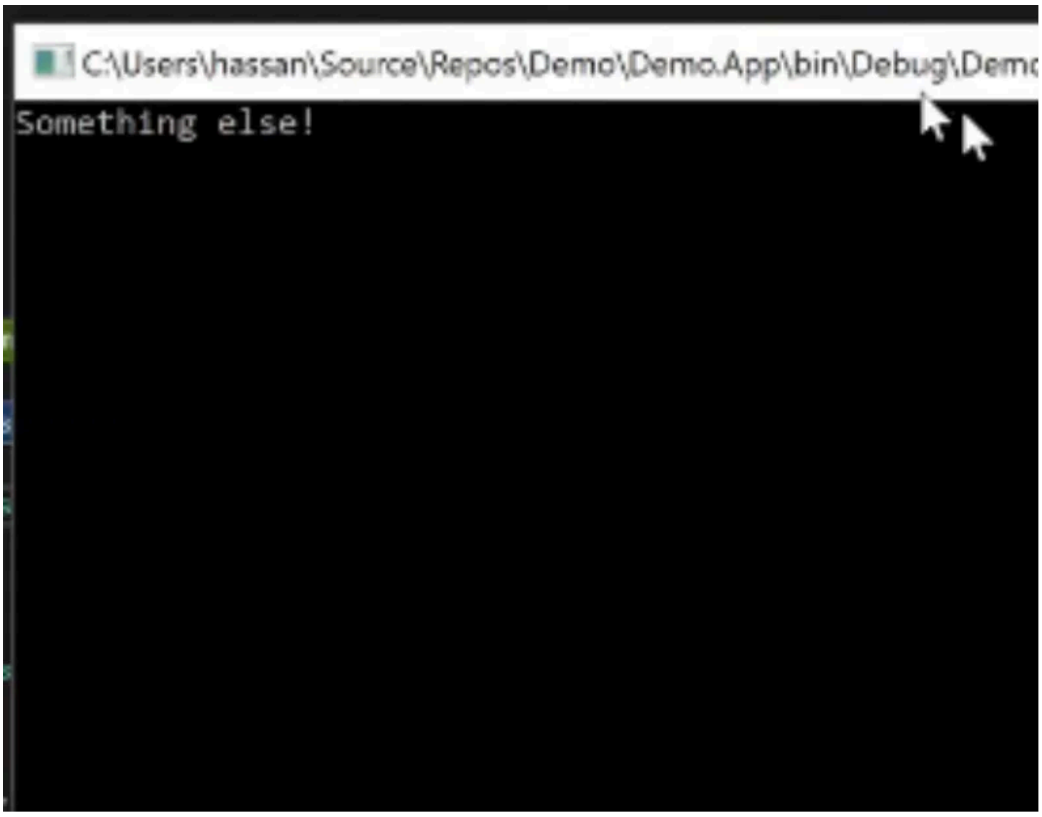
and I am writing it out in

```
Console.WriteLine(message);
```

If the message is `"Hello,World"`, then it will display `"Hello,World"`

If not, then it will write `"Something else!"`

In this case, I set the value to




```
"Hello, World!"
```

Since the input message is `"Hello,World"`, it displays `"Something else!"`

What's happening here? Why is it `"Something else!"`? Because there is an extra space and an exclamation mark in the message `"Hello, World!"`. The message needs to say exactly `"Hello,World"`.

Programming is very precise down to the space, make sure to watch that your spaces and punctuation are all exactly what you intended them to be.

So I used an 'if' statement here. What exactly is an 'if' statement?

It's a way for you to control the flow of your program.

A simple example:

If person enters number 1, then output number 1.

If person enters number anything else, then output "Wrong answer!"

With 'if' statements, you're teaching your program how to react towards different types of input.

```
Another example for you - let's say I'm getting in a number input from the user. We
will ask our user to give us a number through Console.ReadLine();
```

(ie: `Console.ReadLine();` outputs the user input)

ReadLine is basically a function that when called it will wait for the user to type in their input and then we can use that input in our program by assigning a variable to it and reuse it wherever we want.

```
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Write a number: ");
            string input = Console.ReadLine();

            if (message == "Hello, World!")
```

In the next few paragraphs I'm going to show you how we are going to put that in action.

Now let's suppose you want to know if the number that is inputted is greater than or less than zero.

You want to convert a string value into a value that your program can access as a number. This is why we have discussed integers.

```
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Write a number: ");
            string input = Console.ReadLine();
```

```
            int num = int.Parse()

            if (messag
            {
                Consol
            }
            else
            {
                Console.WriteLine("Something else!");
            }

            Console.ReadKey();
        }
    }
}
```

```
static void Main(string[] args)
{
    Console.WriteLine("Write a number: ");
    string input = Console.ReadLine();

    int num = int.Parse(input);

    if (message == "Hello, World!")
    {
        Console.WriteLine(message);
    }
    else
    {
        Console.WriteLine("Something else!");
    }

    Console.ReadKey();
}
}
}
```

The number entered in `int.Parse()` will be something that you parse, meaning you want to convert that string into an integer so you can run comparison arithmetic operations off of it.

We talked earlier on how strings and integers are very fundamentally different data types, we can for instance run arithmetic operations on integers but not strings, we can create strings of characters but not integers, each and every data type has its use based on what it can and cannot do.

Now, since the user input comes in a string format, we will need to convert that string value into an integer so we can run a comparison arithmetic operation on it.

So if I pass an `input` into `int.Parse()`, that input becomes a number - an integer.

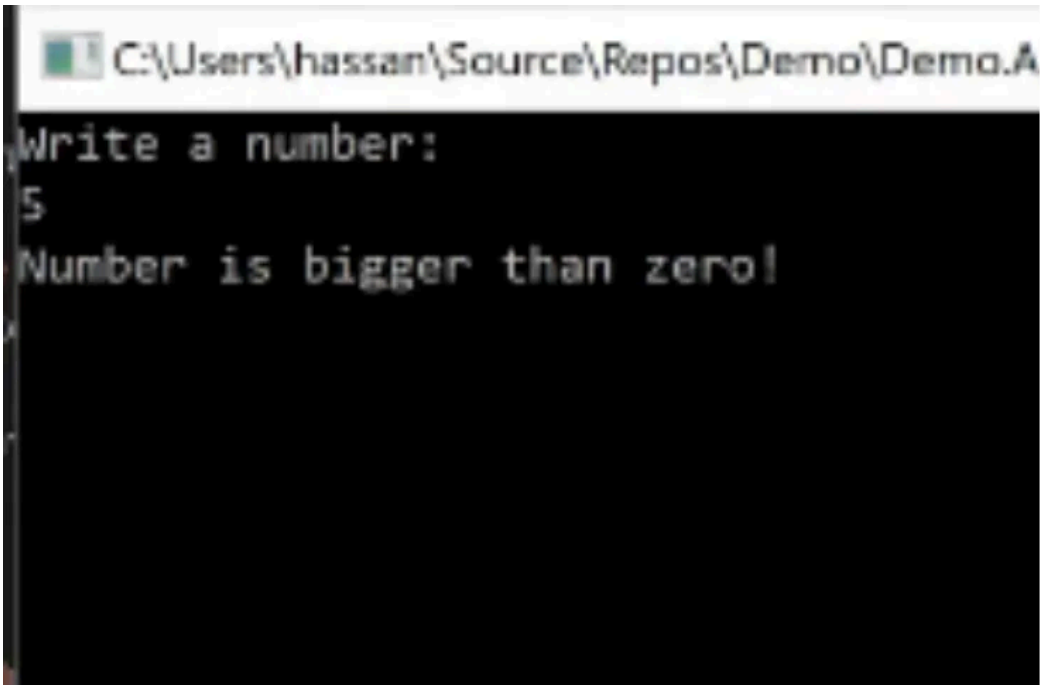
And if this number is greater than zero, then,

```
Console.WriteLine("Number is bigger than zero!");
else
Console.WriteLine("Number is not bigger than zero!");
```

```
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Write a number: ");
            string input = Console.ReadLine();

            int num = int.Parse(input);

            if (num > 0)
            {
                Console.WriteLine("Number is bigger than zero!");
            }
            else
            {
                Console.WriteLine("Number is not bigger than zero!");
            }

            Console.ReadKey();
        }
```

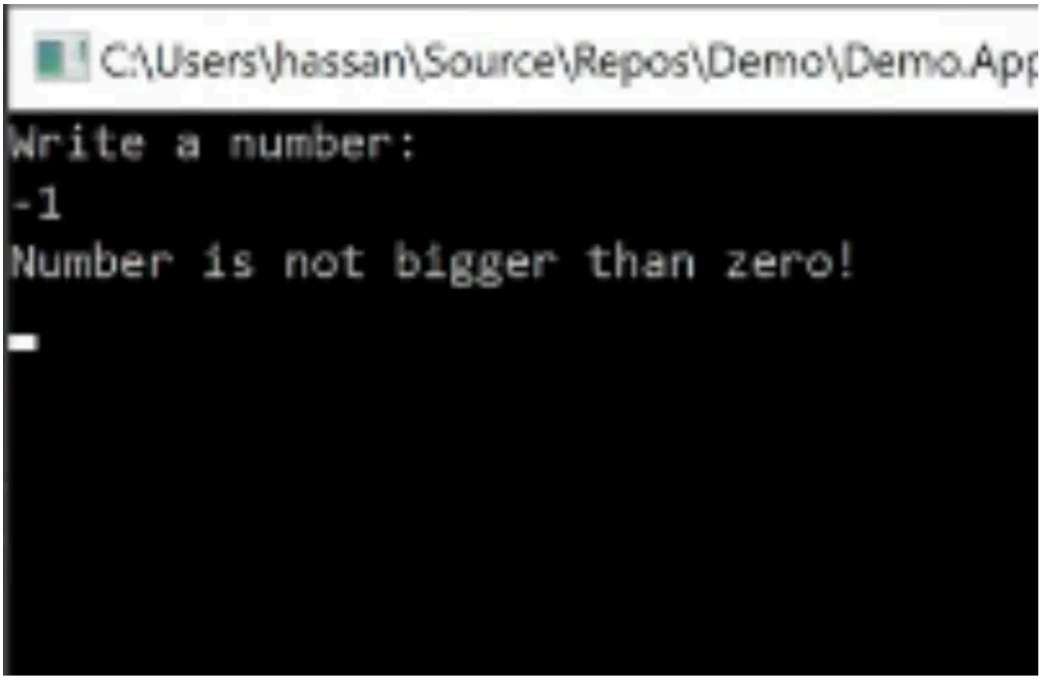
Let's give it a try and run our program(Hit that Start button at the top!). If i put 5, it reads,

```
Number is bigger than zero!
```



If i put -1,

```
Number is not bigger than zero!
```



This is how you interact with your program.

You have set commands using ‘if’ statements and your program is responding appropriately to your inputs.

On your own, try putting in the cases where the number is equal to zero and where the number is smaller than zero. Print a message that says “The number you input is equal to zero” and a message that says “The number you input is smaller than zero”.

How do you put these conditions in here? Think about the problem.

CHAPTER 4

Switch Cases

Let's continue our 'if' statement discussion from last chapter, and move into switch cases. For now, understand you can always do more than just 'if,else'. You can do 'if', 'else if', and 'else', again for as long as you want.

So this is where we are at right now:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Write a number: ");
            string input = Console.ReadLine();

            int num = int.Parse(input);

            if (num > 0)
            {
                Console.WriteLine("Number is bigger than zero!");
            }
            else
            {
                Console.WriteLine("Number is not bigger than zero!");
            }

            Console.ReadKey();
        }
    }
}
```

Continuing our if, else statement...if we want to check for numbers that are less than zero, then instead of

```
"Number is not bigger than zero!"
```

we can do an `else if(num < 0)` statement that will print out:

```
"Number is lesser than zero!"
```

```
            if (num > 0)
            {
                Console.WriteLine("Number is bigger than zero!");
            }
            else if(num < 0)
            {
                Console.WriteLine("Number is lesser than zero!");
            }
            else {
                Console.WriteLine("Number is equal to zero!")
            }

            Console.ReadKey();
        }
    }
}
```

Then your final else statement can be

`("Number is equal to zero!")`".

In this case you get to see all the different if,else statements.

```
If (num > 0,
else if(num < 0,
else
```

What this code is saying:

If the number is greater than zero, print `"Number is bigger than zero!"`, else if the number is less than zero print `"Number is lesser than zero!"`, else print `"Number is equal to zero!"`

If, else if, else. You get to go through all of these and based upon your what you need from your program that you're using to model a real life problem.

```
        static void Main(string[] args)
        {
            Console.WriteLine("Write a number: ");
            string input = Console.ReadLine();
```

```
            int num = int.Parse(input);

            if (num  == 1)
            {
                Console.WriteLine("Number is bigger than zero!");
            }
            else if(num == 2)
            {
                Console.WriteLine("Number is lesser than zero!");
            }
            else {
                Console.WriteLine("Number is equal to zero!");
            }
```

Now, here's another case for you.

Let's check and see if the number is 1 or 2 or 3, and that it is equal to 1 or 2 or 3.

One way to do that is using 'if,else' statements. If it's 1, print 1, else if it's 2, print 2, etc.

The picture above would get pretty messy using nothing but 'if, else' statements for each individual number.

There's a better way to do this. If you're looking for the same value that's exactly equivalent to another value is to use something in C# called "switch statements".

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Write a number: ");
            string input = Console.ReadLine();

            int num = int.Parse(input);

            Console.ReadKey();
        }
```

```
    }
}
```

Switch statements are a lot easier to use when you want to do a condition like this.

Go ahead and delete your 'if,else' statements.

Add in `switch (num)`.

```
        static void Main(string[] args)
        {
            Console.WriteLine("Write a number: ");
            string input = Console.ReadLine();

            int num = int.Parse(input);

            switch (num)
            {

            }

            Console.ReadKey();
```

If we're checking just for the value, you can have your program check a switch statement. It will go "switch, based on this certain number...

...represented as `switch (num)`...

```
        int num = int.Parse(input);

        switch (num)
        {
            case 1:
                Console.WriteLine("The number is one")
        }

        Console.ReadKey();
    }
```

...find all the cases that this number may fall in".

and we can say "in case, if this number is equal to 1...

```
case 1:
```

```
    switch (num)
    {
        case 1:
```

```
            Console.WriteLine("The number is one");
        case 2:
            Console.WriteLine("The number is two");
    }

    Console.ReadKey();
}
```

...then print out:

```
("The number is one")
```

We then have our second case:

```
case 2:
Console.WriteLine("The number is two");
```

```
switch (num)
{
    case 1:
        Console.WriteLine("The number is one");
        break;
    case 2:
        Console.WriteLine("The number is two");
        break;
```

When you have cases it is important to “break” them so that you can exit that loop.

Inserting...

```
break;
```

...will tell the program there are no more cases for you to check.

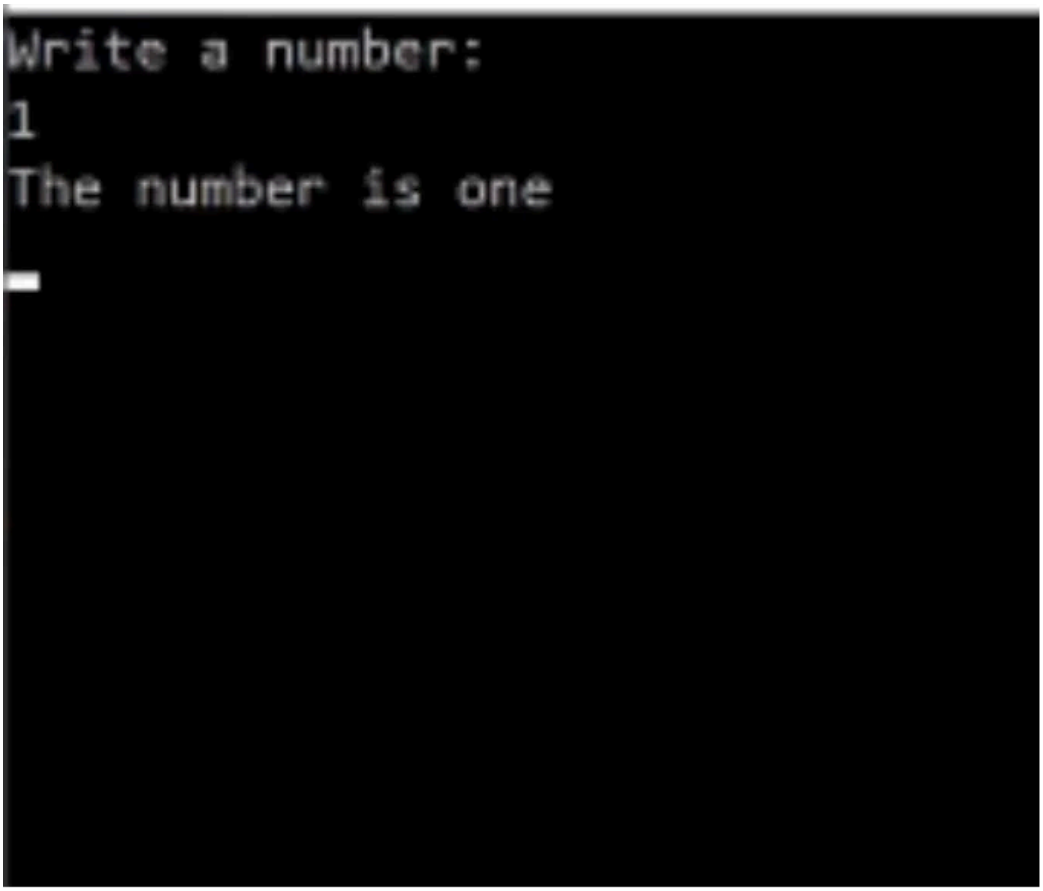
The third case we can have here is the case where neither of those two cases is a possibility.

We can put in a default case that would run if neither of the first two cases are satisfied. Let’s put in a default case where the number is neither one or two, and be sure to put in the break.

```
default:
Console.WriteLine("The number is neither one or two");
break;
```

```
            switch (num)
            {
                case 1:
                    Console.WriteLine("The number is one");
                    break;
                case 2:
                    Console.WriteLine("The number is two");
                    break;
                default:
                    Console.WriteLine("The number is neither one or two");
                    break;

            }

            Console.ReadKey();
        }
```

What's happening here is instead of putting in a huge series of 'if' statements - if, if else, if else, if else.... you're just checking for the number value. This lets you use the much simpler switch statements.



It is another tool that is very similar to 'if' statements. It is a tool that allows you to do stuff based on the value of the parameter you're passing in.

Now if we run this and input the number 1, it says the number is one.

```
Write a number:
5
The number is neither one or two
```

If I enter a 5, it says "The number is neither one or two".

```
Write a number:
2
The number is two
```

If I enter a 2, it says "The number is two".

And that's the end of switch statements for now. Please check out this link if you want to dig deeper:

```
https://bit.ly/2TzLtzW
```

CHAPTER 5

For-loops

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

Now we are going to talk about iteration in C#. We are going to want to have our program iterate for a number of times, through a list of things, and then print these things out.

The point here is to make things simpler for our program. Instead of us repeating the calculation of something over and over again, we can simplify that by iterating it.

For instance if I asked you to write a program based on what you've read so far, and I asked you to write a program to print the numbers from 1-7, you'd go and type:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(1);
        Console.WriteLine(2);
        Console.WriteLine(3);
        Console.WriteLine(4);
        Console.WriteLine(5);
        Console.WriteLine(6);
        Console.WriteLine(7);

        Console.Read();
```

```
    }
}
```

```
Console.WriteLine(1);
Console.WriteLine(2);
Console.WriteLine(3);
Console.WriteLine(4);
```

and so on.

Your program will go and iterate your code, and then it would print all these numbers out on the black monitor.

The beauty of programming though, is that it is always the idea of doing less and accomplishing more.

You want to have tools that will do the heavy lifting for you, so you can focus completely on doing that work that requires a lot of brain power.

For iteration, I'm going to show you three ways you can iterate in programming language(three popular ways anyway). There's more than three, but the ones that we care about and the ones you will typically see are what we call 'for-loops.'

```
    {
        static void Main(string[] args)
        {

            for()

            Console.Read();
        }
    }
}
```

'For-loops' are a way for you to iterate through a list of things or for a certain number of times.

```
static void Main(string[] args)
{

    for(int i = 0; i <= 10; i++)
    {

    }

    Console.Read();
```

In a 'for-loop' you enter some conditions for the loop to follow.

So let's enter some conditions for a 'for-loop'.

Let's say I want to start with 0, this is my start index(`int i =0`). And let's say that my index should stop at 10(`i <=10`), and finally that my index will accumulate 1 step at a time(`i++`). (`i++` means one step up)

```
for(int i = 0; i <=10; i++)
```

These are my 3 conditions. My first variable tells me I'm starting from zero. The second one tells me I'm going to continue when the value of this variable is less than or equal to 10. The third tells me to iterate one step at a time.

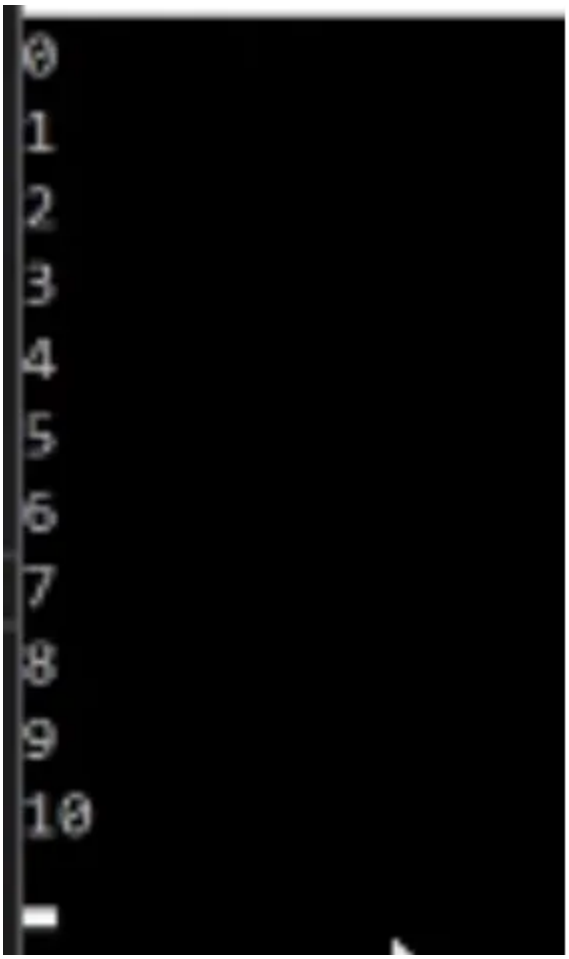
So it will stop at 10.

As long as it is less than ten, the condition is fulfilled. It will continue to iterate, and then iterate just one step everytime.

```
class Program
{
    static void Main(string[] args)
    {

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

        Console.Read();
```

Now let's type in `Console.WriteLine(i);` underneath those 'for-loop' conditions and print 'i', the variable.

Hit the start button and it will print all those numbers for you.



You can go up to any number you want, and it's still going to work for you.

What you want to explore is to try running a number of mathematical operations with this.

Let me show you how to print the entire times table with this.

If you go and type,

`Console.WriteLine($"{i} * 2 = {i * 2}")`,

```
0 * 2 = 0
1 * 2 = 2
2 * 2 = 4
3 * 2 = 6
4 * 2 = 8
5 * 2 = 10
6 * 2 = 12
7 * 2 = 14
8 * 2 = 16
9 * 2 = 18
10 * 2 = 20
```

you will get the results of all of the numbers in your 'for-loop', multiplied by two.

You may ask, what is this dollar sign in our string?

The dollar sign is a mechanism to tell our program that we want to allow variables to be a part of the string we are building.

```
for(int i = 0; i <= 10; i++)
{
    Console.WriteLine($"{i} * 2 = {i * 2}" );
}

Console.Read();
```

Now go and get the hang of this. Try to print all the times table from zero all the way down to ten, multiplied by all numbers from 0 to 10.

Give that a shot. In chapter 6, we discuss other forms of iteration.

CHAPTER 5.5

The Infinite 'for-loop'

I didn't want to put this in the 'for-loop' chapter because it would distract you from learning the basic 'for-loop'.

Let's see what an infinite 'for-loop' looks like. Infinite 'for-loops' can be made with the annotation:

```
for(; ; )
```

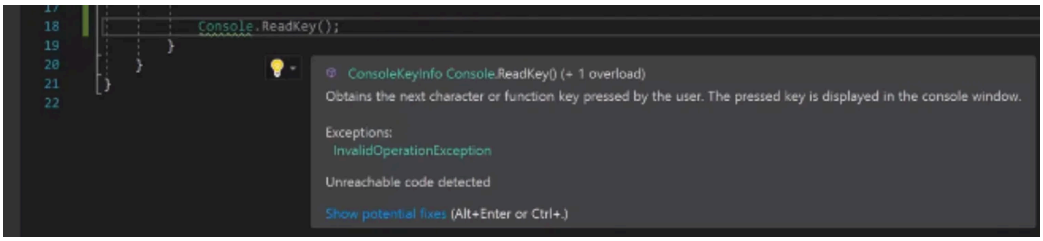
It is an open-ended code style that runs your application forever.

It would never reach your

```
namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            for(; ; )
            {
                Console.WriteLine("Forever");
            }

            Console.ReadKey();
        }
    }
}
```

```
Console.ReadKey();
```

Visual studio will even tell you that the point is unreachable code because this program will keep running forever.




```
        Console.ReadKey();
    }
}
```

If we run the program. It's going to keep going until your computer runs out of memory.

And that's all there is to it!

CHAPTER 6

While And Do-While

Okay, let's talk about a couple more ways to iterate in C#.

There are two more forms of iteration that we shall talk about, and those are `while` and `do while`. Let's discuss the difference between them.

Let's say you have a condition in here, some condition, and this condition is true.

```
class Program
{
    static void Main(string[] args)
    {
        bool condition = true;
    }
}
}
```

In programming, there's a special primitive type that handles whether something is either true or false, boolean data types cannot have any other value but true and false.

For instance you can assign a boolean value such as:

```
bool condition = true;
```

But we can also do something like:

```
bool condition = 1 > 0;
```

The statement here is that if 1 is bigger than 0 then the boolean value of the condition variable is true, otherwise it's false.

You can put while(this condition is true), continue to print (condition is still true).

```
While (condition)
Console.WriteLine("Condition is still true");
```

```
class Program
{
    static void Main(string[] args)
    {
        bool condition = true;
        while (condition)
        {
            Console.WriteLine("Condition is still true");
        }
    }
```

This is also an infinite loop because the condition is hardcoded to true. There's no other possible condition except for true. Therefore, it will keep iterating forever.

Let's use a better example.

```
namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            while (i <= 10)
            {
                Console.WriteLine("Condition is still true");
            }

            Console.ReadKey();
        }
    }
}
```

Let's put an iteration in our code here. If we put 'i' in, starting from 0, and it will stop when 'i' becomes greater than 10.

```
int i = 0;
While (i <= 10)
```

Now in this current state, this will keep running forever because 'i's' value doesn't change. 'i' will always be zero and therefore never become greater than 10.

What we can do to increase the value of that variable with every pass is to use something that we used in the for-loop:

```
{
    int i = 0;
```

```
    while (i <= 10)
    {
        Console.WriteLine("Condition is still true");
        i++;
    }

    Console.ReadKey();
}
```

```
i++;
```

Let's note here what `i++;` actually means. It simple means one step. It can be represented in 3 ways:

```
while (i <= 10)
{
    Console.WriteLine("Condition is still true");
    i = i + 1;
}

Console.ReadKey();
}
```

```
i++;
i = i + 1;
i += 1;
```

```
while (i <= 10)
{
    Console.WriteLine("Condition is still true");
    i += 1;
}

Console.ReadKey();
}
```

Let's now print the value of 'i' in here. That way we can double check the value we are using.

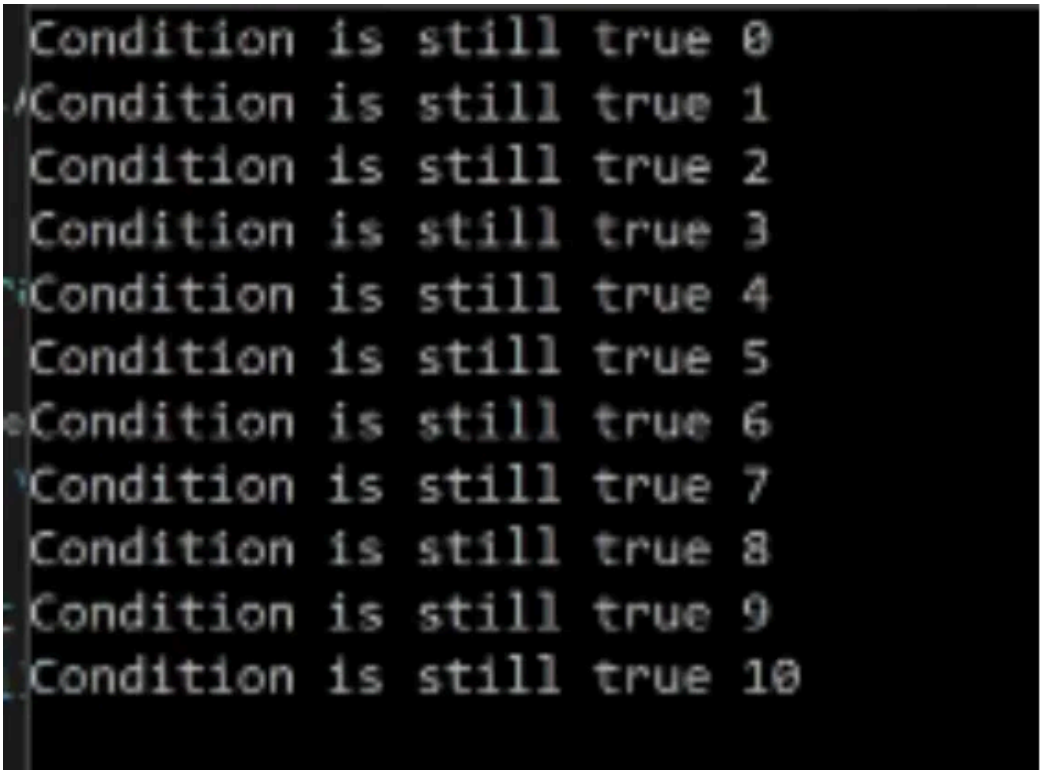
```
Console.WriteLine($"Condition is still true {i}");
i++;
```

```
int i = 0;
while (i <= 10)
{
    Console.WriteLine($"Condition is still true {i}");
    i++;
```

```
}

Console.ReadKey();
```

You can see here that we used the string interpolation dollar sign $ again, to pass in the value of the variable 'i' into our printed out string.

Now when we run this, it will continue to print until it reaches 10 and then it will stop.



'While-loop' is the idea that you continue to iterate until a certain condition isn't true anymore.

If we compare this with the 'for-loop' - in a 'for-loop' you're iterating through a list or a number of things until you reach the end of it.

Let yourself think about the use cases for each of these.

For instance, you can use 'while-loop' to ping a server as long as the server is alive. Once the server does not return that it is alive anymore, then stop pinging that server and shut it down.

```
namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            while (i <= 10)
            {
                Console.WriteLine($"Condition is still true {i}");
                i++;
            }

            do
            {
                Console.WriteLine($"Condition is still true {i}");
                i++;
            } while (i <= 10);

            Console.ReadKey();
        }
    }
```

The question now is, what about 'do-while'?

'Do-while' is different from 'while' in that it has to do at least one iteration before it stops.

As per the image above, let's copy and paste the `Console.WriteLine` content from our while loop down into our 'do-while' loop.

Your 'while' loop will not execute until the condition is met. Your 'do-while' loop will execute the code at least one time, before it checks if the condition is met.

'Do-while' would be used in a case where you need to execute the code at least once, and then check back again.

```
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            int i = 0;
            while (i <= 10)
            {
                Console.WriteLine($"Condition is still true {i}");
                i++;
            }

            do
            {
                Console.WriteLine($"Condition is still true {i}");
                i++;
            } while (i <= 10);

            Console.ReadKey();
        }
    }
}
```

```
C:\Users\hassan\Source\Repos\Demo\Demo.A
Condition is still true 0
Condition is still true 1
Condition is still true 2
Condition is still true 3
Condition is still true 4
Condition is still true 5
Condition is still true 6
Condition is still true 7
Condition is still true 8
Condition is still true 9
Condition is still true 10
Condition is still true 11
```

```
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            while (i <= 10)
            {
                Console.WriteLine($"Condition is still true {i}");
                i++;
            }

            do
            {
                Console.WriteLine($"Condition is still true {i}");
                i++;
            } while (i <= 10);

            Console.ReadKey();
        }
    }
}
```

Above is a great example of the difference between the ‘while’ and ‘do-while’ loops. Notice our console counts to 11, not 10.

The ‘while’ loop printed 0 through 10 and stopped, and that left ‘i’ at 11. So by the time our program reaches the ‘do-while’ loop, its condition `i <= 10` is already false.

And yet it still prints 11. Remember, the code will fire at least once before it checks to see if the condition is met. The ‘do-while’ loop runs its body first, printing 11 and moving ‘i’ to 12, and only then checks the condition, finds that 12 is greater than 10, and stops. Put a ‘while’ loop in its place and it would have printed nothing at all, because it checks the condition before it ever runs.

Both loops definitely have their uses, and it completely depends on what you want to do.

CHAPTER 7

Foreach

We have covered three forms of iteration up until this point - 'for', 'while', and 'do-while'.

It's time now to talk about one more form of iteration - 'foreach'.

'While' and 'do-while' can be grouped as one iteration kind, then 'for' and 'foreach' as the other type.

'Foreach' functions just as good as 'for' loops, but from a performance perspective they are a lot slower. From a readability perspective however, they are a lot easier to read than 'for' loops. There are uses for each case.

```
namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] numbers = new int[3] { 1, 2, 3};

            Console.ReadKey();
        }
    }
}
```

Let's say you have a list of 3 numbers - 1, 2, and 3.

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

We are creating an array of numbers, by defining our array in a variable of type int[] this type means that our variable is going to be an array of integers.

But on the right side of the equal sign, we are putting in more specific details about our array, such as the length of the array int[3] we are declaring our array is going to consist of 3 elements, and then we pass in these elements as we define the array {1, 2, 3}.

There are so many ways to add in values to the array such as:

```
numbers[0] = 1;
numbers[1] = 2;
numbers[2] = 3;
```

Notice that arrays are zero based, meaning that when we declare an array to have 3 positions, it means we are starting the assignments of these positions by position 0 then 1 and so on.

You want to iterate through this list of 3 numbers.

```
class Program
{
    static void Main(string[] args)
    {
        int[] numbers = new int[3] { 1, 2, 3};

        foreach(int number in numbers)
        {
            Console.WriteLine(number);
        }

        Console.ReadKey();
    }
}
```

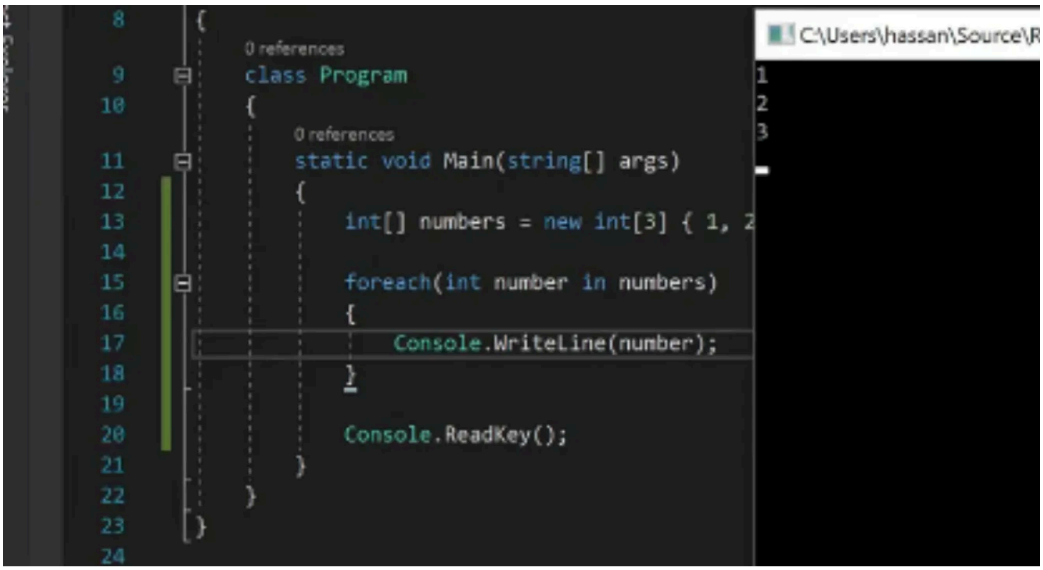
What we're going to type in will be telling the program:

"For each number you come across in your array of numbers, print out each number"

```
foreach(int number in numbers)
```

We defined an array through `int [] numbers = new int[3] {1, 2, 3};`. So it will print 1, 2, and 3.

'Foreach' is nice and easy to read.




```
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] numbers = new int[3] { 1, 2

            foreach(int number in numbers)
            {
                Console.WriteLine(number);
            }

            Console.ReadKey();
        }
    }
}
```

‘Foreach’ works like this: “Foreach number you come across in this list, execute this specific action.”

In this case, ‘foreach’ number you come across, print it out. It will come across 1 and print it out. Then 2, then 3.

But again, ‘foreach’ will affect your performance.

With ‘for’ loops you get a much better performance. If you don’t care about performance, due ‘foreach’ every single time, because they are a lot easier to read and understand.

```
namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] numbers = new int[3] { 1, 2, 3 };

            foreach (int number in numbers)
            {
                Console.WriteLine(number);
            }

            for (int i = 0; i < numbers.Length; i++)
            {
                Console.WriteLine(numbers[i]);
            }

            Console.ReadKey();
        }
    }
```

Compare the 'foreach' code to the 'for' code, and you can see the difference in readability.

'foreach' is much easier to look at.

With 'for' loops, you have to set a start point, you have to set a parameter for you 'i' to stay within, and you have to put in your accumulation value of your 'i' for every iteration.

Represented as...

```
for (int i = 0; i < numbers.Length; i++)
```

Now if I run my code it will print each list.

```
    {
        static void Main(string[] args)
        {
            int[] numbers = new int[3] { 1, 2, 3};

            foreach(int number in numbers)
            {
                Console.WriteLine(number);
            }

            for (int i =0; i<= numbers.Length; i++)
            {
                Console.WriteLine(numbers[i]);
            }

            Console.ReadKey();
        }
    }
}
```

...Or it would have if there wasn't an error in the 'for' loop.

The 'i' parameter is set as

```
i <= numbers.Length
```

Remember how we said earlier that arrays are zero-based. Based on the condition above, i <= numbers.length, we are basically saying that the loop will iterate for as long as the length of this array. It will iterate as long as 'i' is less than or equal the length of the array.

Now in an array of 3 positions, you will notice that 'i' cannot go beyond position 2, since are starting at position 0 then 1 then 2, putting 'i' in a position 3 will cause an error, because that position doesn't exist.

So telling our loop to continue running until reaching the length of the array (3) means that i will be equivalent to the value of 3 which puts our program out of bound of the available positions in that array.

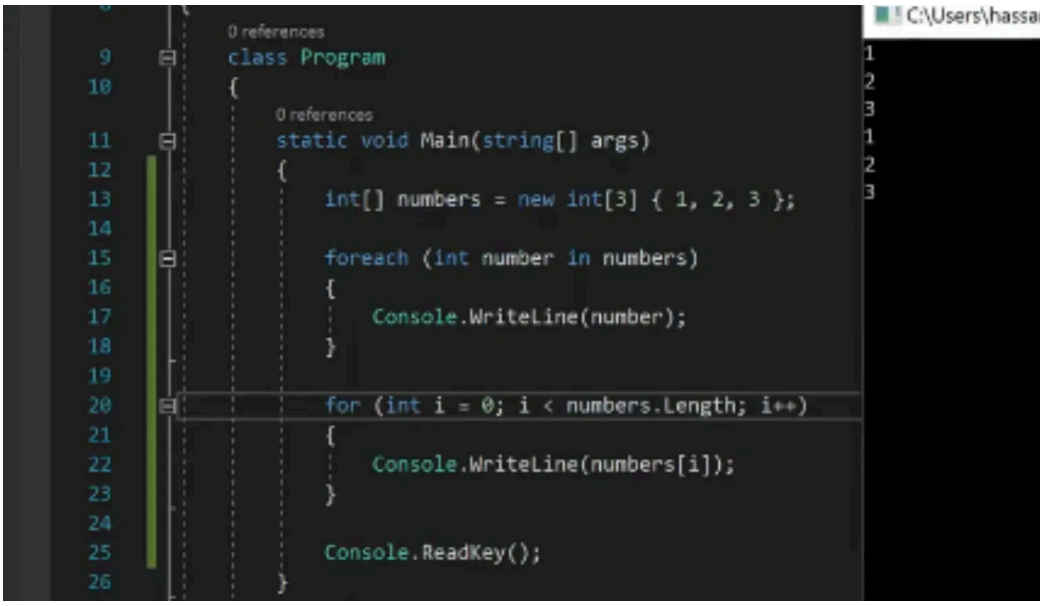
'Foreach' will take away this possibility that you have to backtrack and fix the code in your 'for' loop. With 'foreach', it is just going to tell us what is there('foreach' this, do that). That's it. It is much better from a readability standpoint.

With 'foreach', you don't have to set your values. You don't have to set how many items are on the list, and you don't have to set how many times to iterate.

It doesn't have all those details - 'foreach' is just one straight, simple couple lines of code that tells you exactly what you need to do.

That's it for 'foreach'.

Try to play around with lists, try to play around with different kinds of arrays and see how 'for' versus 'foreach' can perform.




```
class Program
{
    static void Main(string[] args)
    {
        int[] numbers = new int[3] { 1, 2, 3 };

        foreach (int number in numbers)
        {
            Console.WriteLine(number);
        }

        for (int i = 0; i < numbers.Length; i++)
        {
            Console.WriteLine(numbers[i]);
        }

        Console.ReadKey();
    }
```

CHAPTER 8

Methods

It's time to talk about something new in C#.

Among the most important things you need to learn in C# is how to organize your code and how to make your code make sense and easy to read.

An essential skill you will require when you're building your software is trying to encapsulate your code into methods.

Let's get into how we can build a method.

Let's say we're building a method in here that will add 2 numbers. What I expect my method to return when it adds two numbers, is that it will also return a number.

```
namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {

        }

        int Calculate()
    }
}
```

My return value type is going to be an integer in this case, so we start with `int` .

Then I need to give my method a name. When you're writing methods in C#, make sure you are using an upper-case letter when you're naming the method. Let's name our method Calculate and be sure to use that capital C.

```
int Calculate()
```

The next thing in your method to consider is to pass in parameters. If you need your method to do something that requires an input from the outside world, you need to allow or program these as parameters.

```
{
    static void Main(string[] args)
    {

    }

    int Calculate(int firstNumber, int secondNumber)
    {

    }
}
}
```

If I am adding two numbers, I am expecting my user, whoever is going to use my method, to pass me two numbers.

These are called input parameters. Let's call our first parameter 'firstNumber'.

The second one would be 'secondNumber'. With this, we have a method, it's called 'Calculate', and it has two parameters.

```
int Calculate(int firstNumber, int secondNumber)
```

These two parameters are what we're going to use within that method to execute a calculation. Let's say we want to add these two numbers. In order for you to calculate that, let's define a variable and let's call it 'total'.

```
int Calculate(int firstNumber, int secondNumber)
{
    int total = firstNumber + secondNumber;
    return total;
}
}
}
```

Let's say 'total' is 'firstNumber' plus the 'secondNumber', and then we want to give back this total to whoever is using that method.

We will say 'return total'.

```
int total = firstNumber + secondNumber;
return total;
```

This way I've taken two numbers, I'm adding them up inside my method, and pushing that value out.

Someone might say I can simplify this operation by simply saying, "here is what am simply returning". It's one less line of code.

```
}

int Calculate(int firstNumber, int secondNumber)
{
    return firstNumber + secondNumber;
}
}
```

```
return firstNumber + secondNumber;
```

Still works.

```
class Program
{
    static void Main(string[] args)
    {
        int total = Calculate()
    }

    int Calculate(int firstNumber, int secondNumber)
    {
        return firstNumber + secondNumber;
    }
}
```

Let's see how we can use this method we have created.

Let's say I have an integer total and I want to take the value out of that method and put it into a variable.

```
int total = Calculate()
```

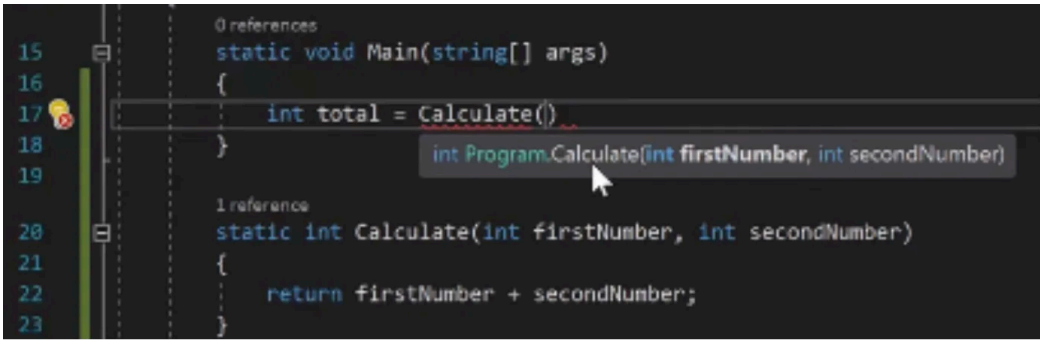
All I have to do is call that method, which is calculate. We are pulling the value out of our method (whatever the total is of firstNumber + secondNumber). Let's put this method in as a static method. We will talk about what static means a little bit later. For now, let's call it static. Type static in to the left of your `int`.

```
static int Calculate(int firstNumber, int secondNumber)
```

```
static void Main(string[] args)
{
    int total = Calculate()
}

static int Calculate(int firstNumber, int secondNumber)
{
    return firstNumber + secondNumber;
}
}
}
```

So this is static, and we want to pass in 2 variables.

In order to know what the parameters are, you can press ctrl+shift+space, and that will show you what the input parameters are.




```
static void Main(string[] args)
{
    int total = Calculate()
}

static int Calculate(int firstNumber, int secondNumber)
{
    return firstNumber + secondNumber;
}
```

Make sure your input parameters make sense and are named properly. This is so the user who is going to use your method knows what to do with it.

```
static void Main(string[] args)
{
```

```
    int total = Calculate(1, 2);
}
```

Let's say I'm passing in 1, and 2. Let's add a couple lines of code:

`Console.WriteLine(total);`

```
class Program
{
    static void Main(string[] args)
    {
        int total = Calculate(1, 2);
        Console.WriteLine(total);
        Console.ReadKey();
    }

    static int Calculate(int firstNumber, int secondNumber)
    {
        return firstNumber + secondNumber;
    }
}
```

```
Console.ReadKey();
```

Now if you run your method, what will happen is it is going to calculate 1+2=3. With

`Calculate(1, 2);` you've set values to your

`int firstNumber,` and `int secondNumber`. And it will print it on the console and the console will stay until you press any key. It prints a 3 as expected.

Also a best practice for you when using methods, is to use the alias of these parameters. If you're passing in raw data like 1 and 2, I highly recommend that you use the alias.

An alias here is a name of the input parameter that you're passing to your function, so if you're passing a number such as 1, define that by using the colon to indicate what value you are passing.

In our case here Calculate(firstNumber:1 ...)

It indicates that we are passing the value 1 to the firstNumber input parameter of the Calculate method.

You should always do this unless your number is defined in a variable, such as:

```
int firstNumber = 1;
```

Calculate(firstNumber, ...)

Since your variable already has a name, you don't have to use an alias for that.

The alias makes it clear what value you're passing in. This way your method is a lot more readable, and makes a lot of sense.

```
static void Main(string[] args)
{
    int total = AddTwoNumber(firstNumber: 1, secondNumber: 2);
    Console.WriteLine(total);
    Console.ReadKey();
}

static int AddTwoNumber(int firstNumber, int secondNumber)
{
    return firstNumber + secondNumber;
}
```

If you can recall from before, I mentioned that you want your method name to encapsulate what the method is doing.

'Calculate' is not very descriptive. Let's call it:

`AddTwoNumber`.

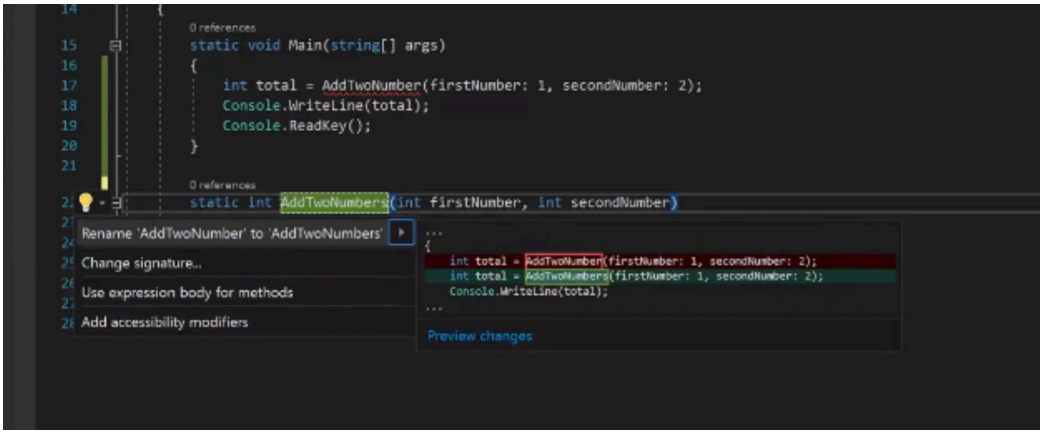
When you change your method name, you can press ctrl+period, and it will let you change that name across all of your code.

```
static void Main(string[] args)
{
    int total = AddTwoNumber(firstNumber: 1, secondNumber: 2);
    Console.WriteLine(total);
    Console.ReadKey();
}

static int AddTwoNumber(int firstNumber, int secondNumber)
{
    return firstNumber + secondNumber;
}
```

Here's another example. If we change `AddTwoNumber` to `AddTwoNumbers`...

```
}

static int AddTwoNumbers(int firstNumber, int secondNumber)
{
    return firstNumber + secondNumber;
}
```

...say we want to change the method name across all the code. You'll see that visual studio gives you that option.




```
static void Main(string[] args)
{
    int total = AddTwoNumber(firstNumber: 1, secondNumber: 2);
    Console.WriteLine(total);
    Console.ReadKey();
}

static int AddTwoNumbers(int firstNumber, int secondNumber)
```

If you go to ctrl+period, it will rename all the occurences. If you have a massive code base and this method is being used all over, this will stop you from changing it all the time. Visual studio can automatically rename your code this way.

Methods allow you to add, remove, due whatever you want to do with it. Just encapsulate that code somewhere far,far away from what your code is trying to.

Your assignment from this chapter is to do all kinds of different methods. Can we call or define a method within an existing method?

CHAPTER 9

Classes & Objects (Object Oriented Programming)

Okay, let's take a look at everything you have learned so far.

If statements, switch cases, for-loops and the other types of iteration, and functions(or methods).

Now it's time to put some of these ideas together. How can we put all these concepts into action?

Back in chapter 1 we talked about visualization. We discussed how you want to visualize something in real life, then bring that as a model to your program to solve a real-life problem.

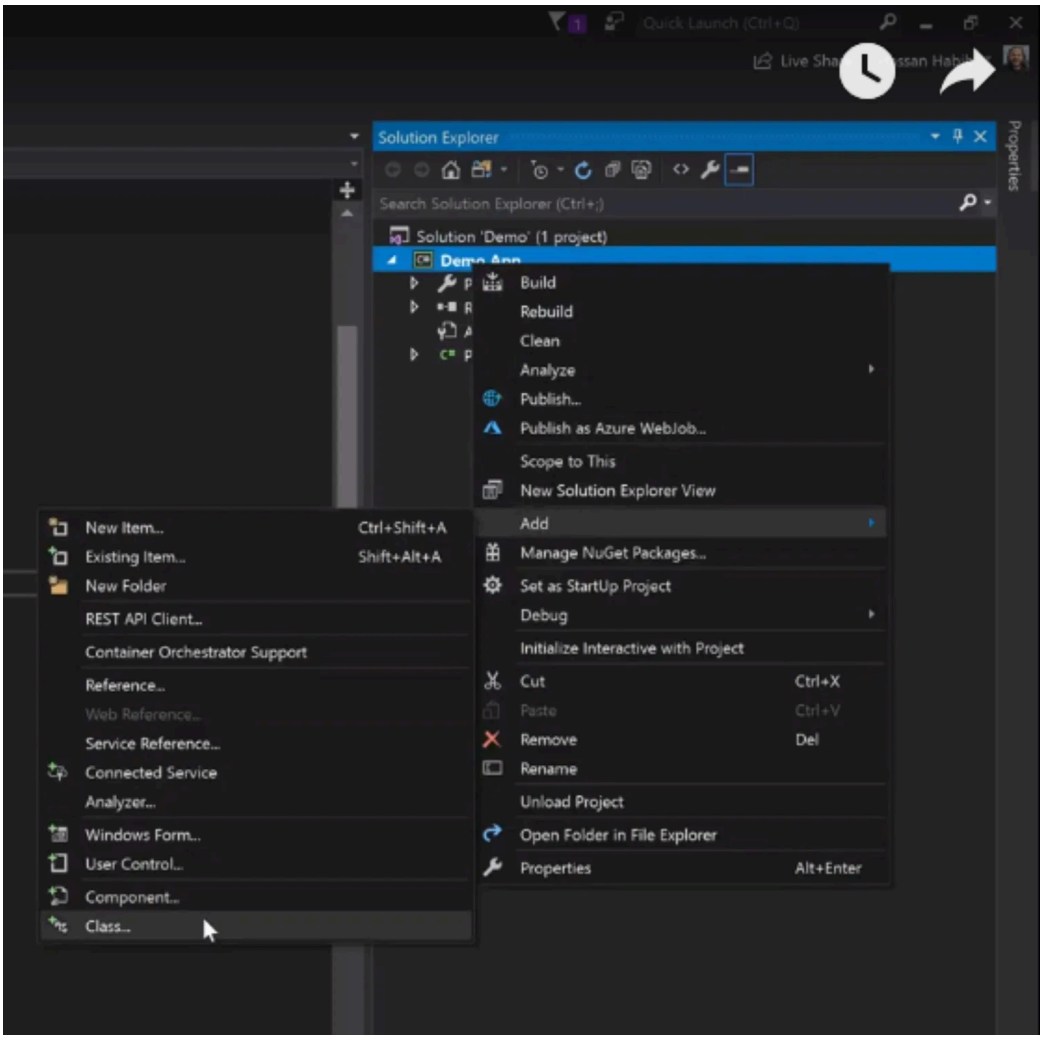
Visualization is the number one thing you want to care about as an engineer. Do not worry too much about the technology or the code that you're using.

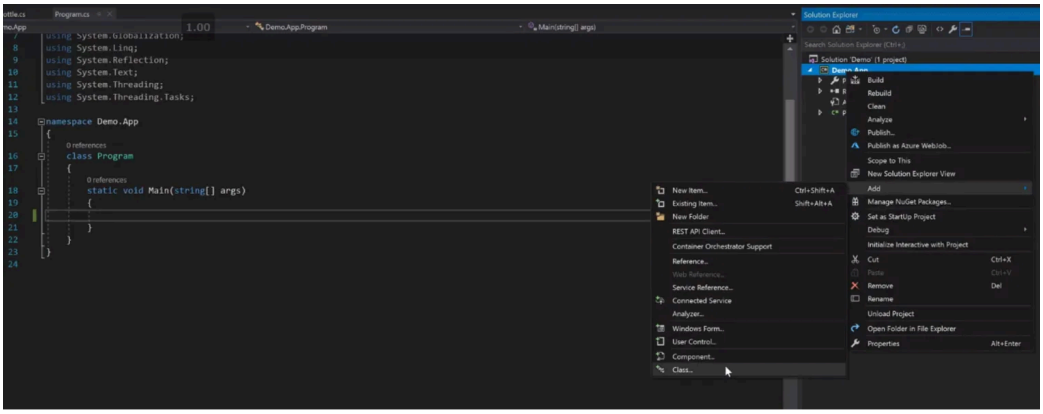
You are learning C# because it's an amazing language, but its purpose is to help you solve real-life problems. Focus on "How can I capture something in the world, visualize that, and translate that into code?"

It happens in the same way as when you take a picture of something and put it on Instagram or Facebook. We are taking a picture of something in the real world, and we are putting it as code and software so that we can solve problems more effectively.

Now, it's time to put everything you have learned into action. Let's visualize a human being. You want to represent people and their actions. An example we can use would be people at a coffee shop.

If you're going to visualize a human being, you'll need to learn about something called classes. Go ahead and build a new class from the right-hand pane.



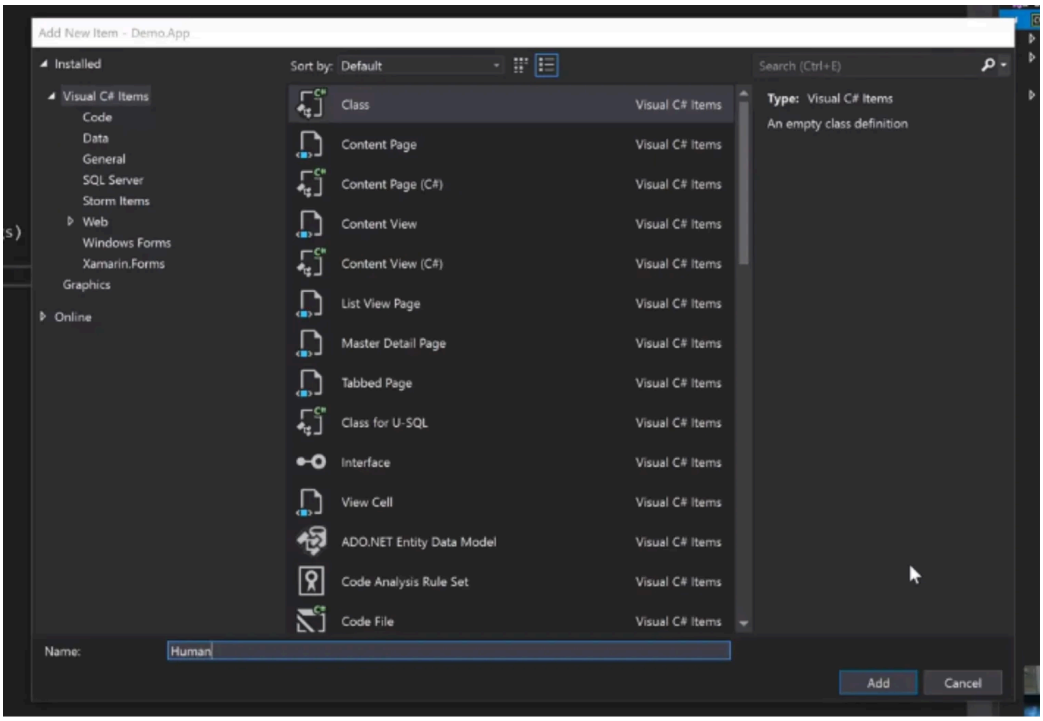



```
using System.Globalization;
using System.Linq;
```

```
using System.Reflection;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

Classes are something that generally encapsulate the type of the thing that you're trying to represent in the real world.

In our case we're just building a human model(human class). Classes and models are the same concept. Type 'Human' for your name.



We are building a human - what does a human have? They have properties - such as a name or a date of birth.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Human
    {
        public int MyProperty { get; set; }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Human
    {
    }
}
```

```
public String Name {get; set;}
public DateTime DateOfBirth{get; set;}
```

You will see here again us using get; set; is a way for us to indicate that we can retrieve or get the value of the class members and set them.

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Human
    {
        public String Name { get; set; }
        public DateTime DateOfBirth { get; set; }

    }
}
```

But humans also have actions, such as eating, drinking, swimming ...etc.

We define these actions as methods, these methods can manipulate existing class properties such as scratching one their own head, or affect other properties of other classes, such as drinking coffee is it's own class, and human object is consuming that coffee by Drink(Coffee) method or actions.

If we're talking about a coffee shop, all we care about is an action that is related to that. Let's say ordering a coffee.

`OrderCofee()` is a method, and this method will do something. It builds something and it does something. In this case, ordering a coffee.

```
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Human
    {
        public String Name { get; set; }
        public DateTime DateOfBirth { get; set; }

        public void OrderCofee()
        {

        }
    }
}
```

```
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
    class Human
    {
        public String Name { get; set; }
        public DateTime DateOfBirth { get; set; }

        public void OrderCofee(string coffeeType)
        {

        }
    }
}
```

Ordering coffee means that you're going to pass in, for instance, what kind of coffee you want. Let's call it `coffeeType`.

```
public void OrderCofee(string coffeeType)
```

Let's switch gears for a second and discuss enums.

```
        public void OrderCofee(string coffeeType)
        {

        }

        enum CoffeeType
        {
            Mocha,
            Espresso
        }
    }
}
```

Enums allow you to put a restriction on the text that your user can and cannot pass. In our case, `Mocha` and `Espresso.`

So the coffee type I put in here can be mocha or it can be espresso. So instead of a string we will pass a `CoffeeType`.

```
public void OrderCofee(CoffeeType coffeeType)
```

So what does this code mean?

We have defined strictly a bunch of types for coffee in an enum type. And we used that type to define what OrderCoffee method can and cannot order.

In our case here, since you can only pass a value of CoffeeType, this human can only order the types defined in the CoffeeType enum, such as Mocha and Espresso.

```
        public String Name { get; set; }
        public DateTime DateOfBirth { get; set; }

        public void OrderCofee(CoffeeType coffeeType)
        {

        }

        enum CoffeeType
        {
            Mocha,
```

```
            Espresso
        }
    }
```

We have used enums to put a restriction on the text the user can or cannot pass. They can only pass Mocha or Espresso. Now these are the only 2 types of coffee he can order.

You might ask, what is void? Why did we change it to Coffee?

```
        public String Name { get; set; }
        public DateTime DateOfBirth { get; set; }

        public Coffee OrderCofee(CoffeeType coffeeType)
        {

        }

        enum CoffeeType
        {
            Mocha,
            Espresso
        }
    }
```

```
public void OrderCofee(CoffeeType coffeeType)
```

Changes to...

```
public Coffee OrderCofee(CoffeeType coffeeType)
```

Void means the method is going to return nothing, it will run whatever action it has and then return nothing, we can only observe the side-effects of that method to whichever variables it's consuming.

We changed the return type to 'Coffee' to get the object of coffee that human is ordering, since we defined the type, now we can go take that type and build a coffee object of which the human would likes to consume.

Now, a little explanation of what is happening with our code in this image.

```
public Coffee OrderCofee(CoffeeType coffeeType)
{

}
```

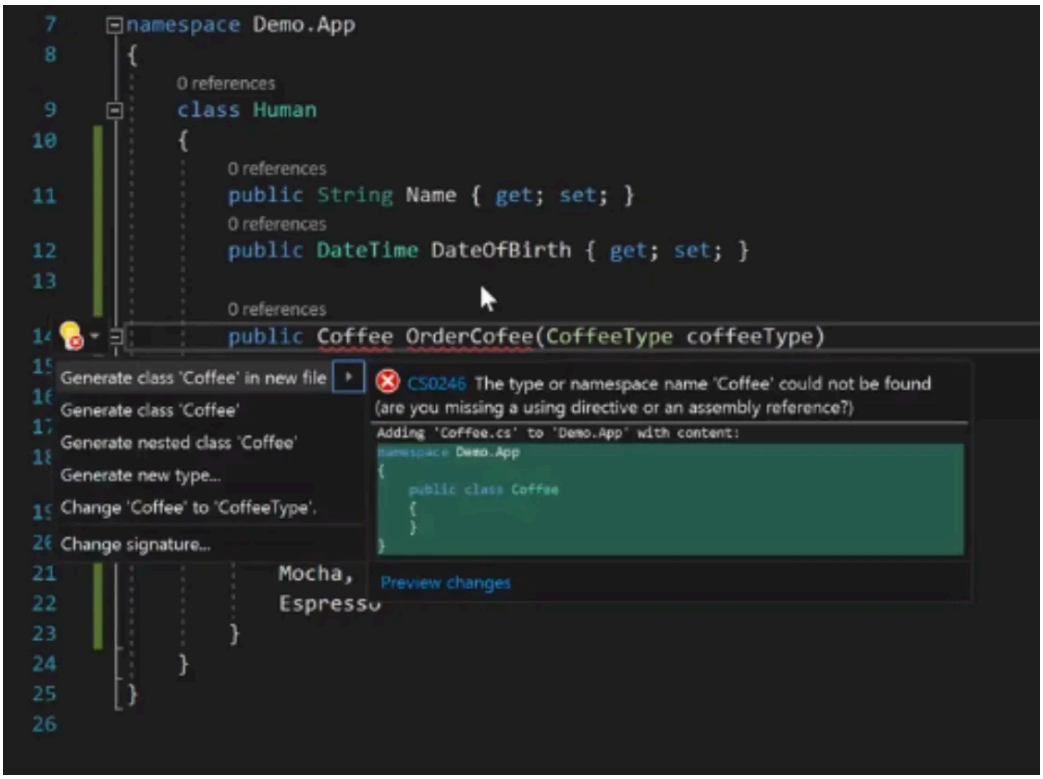
```
enum CoffeeType
{
Mocha,
Espresso
}
```

A guy is going to order a `Coffee`, by requesting which type `Coffee` he wants(`CoffeeType`). And the two types he could order are `Mocha` or `Espresso`.

The lowercase coffee type(`coffeeType`) here is a placeholder for whatever is going to be passed by the consumers of this method.

Let's go build another object. We see how the code is starting to fan out. You're ordering coffee but what you're ordering is returning something.

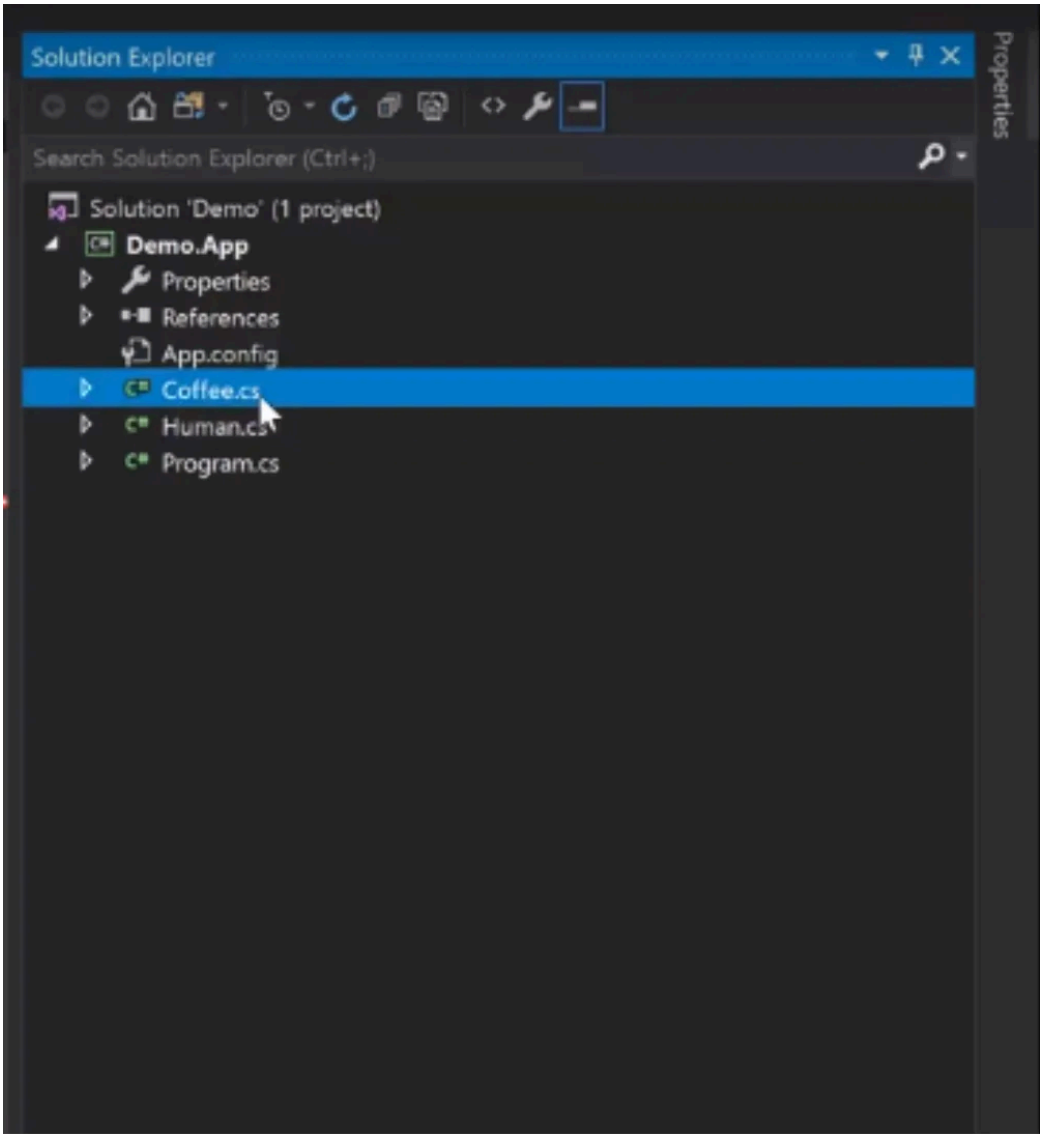
Since we haven't really defined a class of Type Coffee, the IDE(integrated development environment) will throw an error and suggest defining a new class Coffee.




```
namespace Demo.App
{
```

```
class Human
{
    public String Name { get; set; }
    public DateTime DateOfBirth { get; set; }

    public Coffee OrderCofee(CoffeeType coffeeType)
```

Let's build a new class in here with ctrl+period.



Let's generate a new class. Access it from the right side menu as shown to the right.

```
namespace Demo.App
{
    public class Coffee
    {
    }
}
```

So we have `public class Coffee`.

Let's say our coffee has a name, represented by a string, and a size represented by an integer.

```
namespace Demo.App
{
    public class Coffee
    {
        public string Name { get; set; }
        public int Size { get; set; }
    }
}
```

```
public string Name {get; set; }

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

Now let's return to our human object. Find the Human.cs option in the right-hand pane.

You can see from the color switch from white to green that `Coffee` is now a class.

```
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{
```

```
    class Human
    {
        public String Name { get; set; }
        public DateTime DateOfBirth { get; set; }

        public Coffee OrderCofee(CoffeeType coffeeType)
        {
            switch (coffeeType)
            {
                case CoffeeType.Mocha:
                    return new Coffee();
            }
        }

        enum CoffeeType
        {
            Mocha,
            Espresso
        }
    }
}
```

Okay, we're returning coffee,

`return new Coffee();`. Based on the coffee type, `CoffeeType`, this is how we are building our coffee.

When someone orders a coffee, we are going to give them either a mocha or espresso

If it's mocha, then,

```
case CoffeeType.Mocha:
return new Coffee();
```

If it's espresso, then,

```
        public Coffee OrderCofee(CoffeeType coffeeType)
        {
            switch (coffeeType)
            {
                case CoffeeType.Mocha:
                    return new Coffee();
                case CoffeeType.Espresso:
                    return new Coffee();
                default:
                    return null;
```

```
            }
        }
```

```
case CoffeeType.Espresso:
return new Coffee();
```

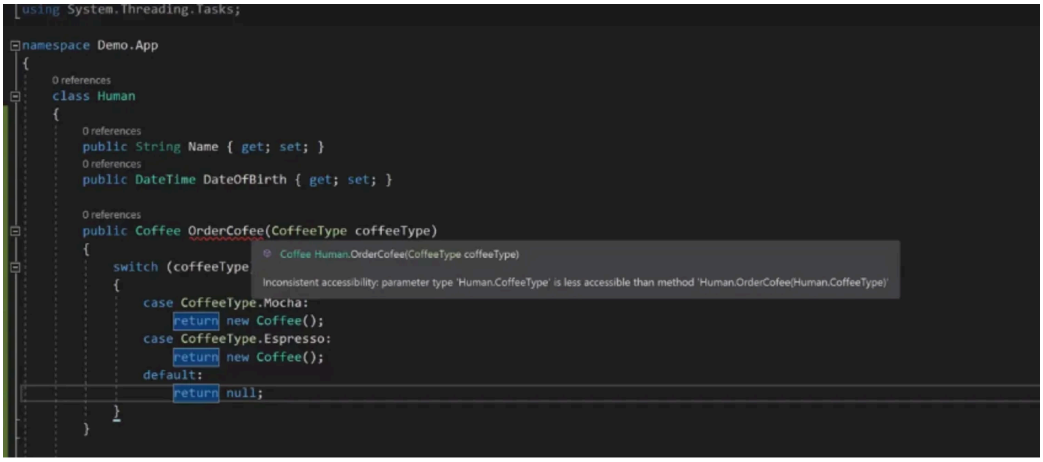
If it's neither, then we use a default case like you learned in switch cases.

```
default:
return null;
```

Null in this case simply means you didn't order anything.

Do you see how I am visualizing this with you? We are visualizing the idea: How can I put something into perspective? And how can I simulate something that happens in the real world?

Now if you have followed along, you will notice there is an error. As shown by the red line underneath `OrderCofee`.




```
using System.Threading.Tasks;

namespace Demo.App
{
    class Human
    {
        public String Name { get; set; }
        public DateTime DateOfBirth { get; set; }

        public Coffee OrderCofee(CoffeeType coffeeType)
        {
            switch (coffeeType)
```

```
            {
                case CoffeeType.Mocha:
                    return new Coffee();
                case CoffeeType.Espresso:
                    return new Coffee();
                default:
                    return null;
            }
        }
```

We have an “inconsistent accessibility” error. Your `class Human` is less public than `public Coffee.`

```
namespace Demo.App
{
    public class Human
    {
        public String Name { get; set;
        public DateTime DateOfBirt
```

If your method is public, `public Coffee`, that means it’s accessible from outside of this class.

As you can see here, a class is a concept of an object that we would like to represent in our program. It can be anything - a concept of a car, airplane, soccer field or anything around you that has properties or features can be represented in a class.

An apple is a class. It has a color, size, weight and price for instance.

In our example here, your class may as well be public, meaning it can be accessed from outside of this class. In this case, your class, `class Human`, isn’t public, making your method public(`public Coffee`) and your class not public(`class Human`), causing this error.

```
namespace Demo.App
{
    public class Human
    {
        public String Name { get; set; }
        public DateTime DateOfBirth { get; set; }

        public Coffee OrderCofee(CoffeeType coffeeType)
        {
            switch (coffeeType)
            {
```

```
                case CoffeeType.Mocha:
                    return new Coffee();
                case CoffeeType.Espresso:
                    return new Coffee();
                default:
                    return null;
            }
        }

        public enum CoffeeType
        {
            Mocha,
            Espresso
        }
    }
}
```

```
        public enum CoffeeType
        {
            Mocha,
            Espresso
```

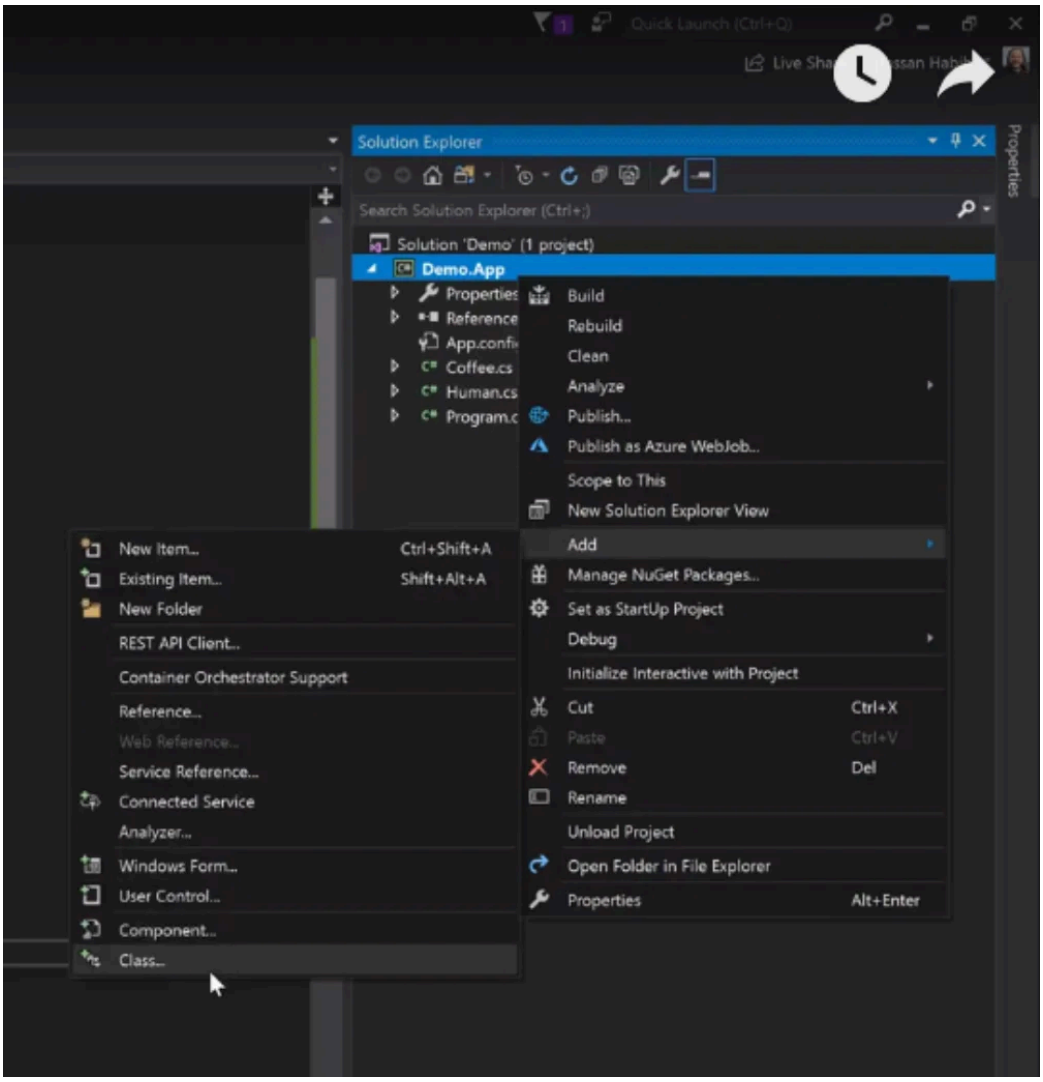
The same thing applies to the enum type.

Go ahead and put `public` before each of your classes and methods.

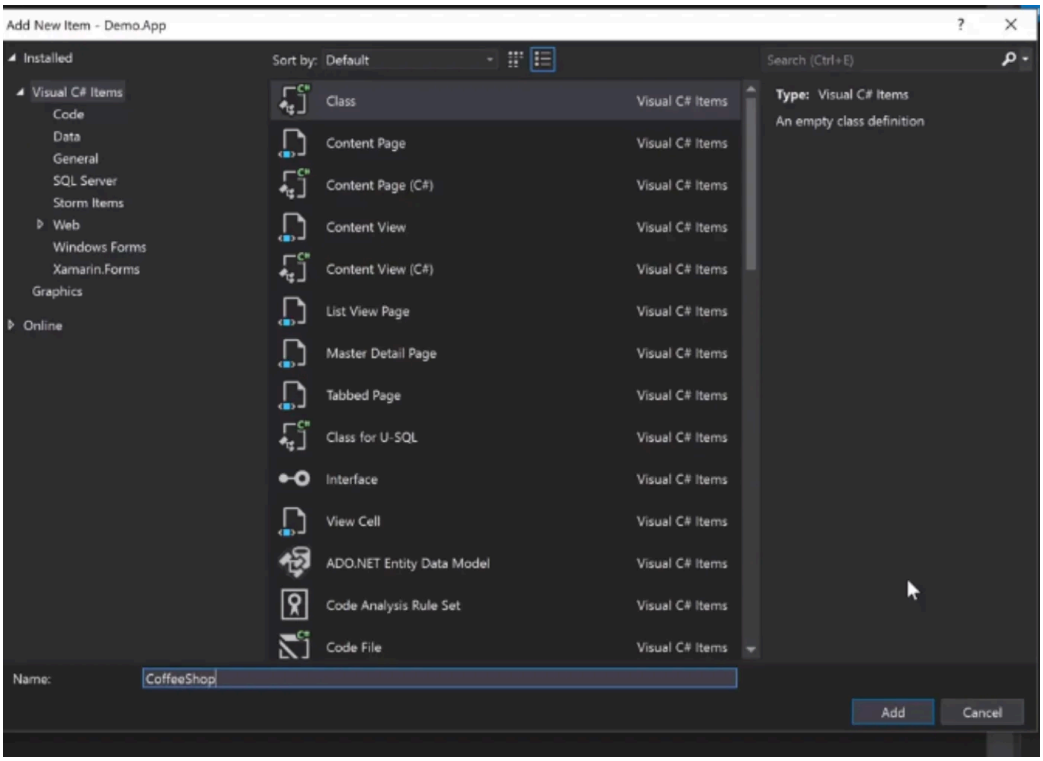
Now everything should be good. It's public across the board.

So we have coffee, we have coffee types, and we have a person that will go and order coffee.

The question now is - order coffee from who? There has to be somebody there that will take that coffee order and process it.




```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Demo.App
{

    class CoffeeShop
    {
    }
}
```



Let's go ahead and build another class called "Coffeeshop".

The coffee shop will have a function that takes in that coffee object that someone will order, and start making that coffee.

Since our coffee order is already created, this method doesn't need to return anything.

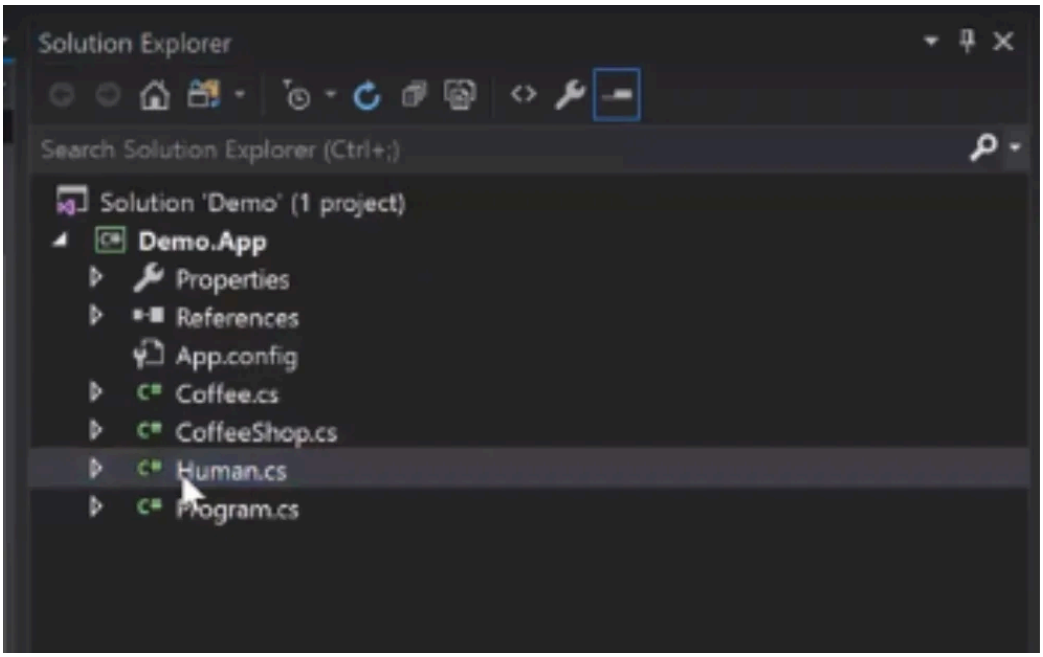
All the changes will happen on the coffee object already created, we will not need to do anything other than applying the making of coffee on that existing object.

```
namespace Demo.App
{
    class CoffeeShop
    {
        public void MakeCoffee(Coffee coffee)
        {
            return;
        }
    }
}
```

So we represent nothing with the type void, meaning there's nothing expected to be returned, just apply some changes on the coffee object.

Now that we have our coffee shop set up, let's return to our human class.

```
namespace Demo.App
{
    public class Human
    {
        public String Name { get; set; }
        public DateTime DateOfBirth { get; set; }

        public Coffee OrderCofee(CoffeeType coffeeType)
        {
            switch (coffeeType)
            {
                case CoffeeType.Mocha:
                    return new Coffee();
                case CoffeeType.Espresso:
                    return new Coffee();
                default:
                    return null;
            }
        }

        public void Drink(Coffee coffee)
        {

        }

        public enum CoffeeType
        {
            Mocha,
            Espresso
        }
    }
}
```



Now our human can also do something else in our coffee shop. He can drink the coffee.

```
public void Drink(Coffee coffee)
{
return;
}
```

And once again, it doesn't matter what it returns, because once we drink coffee we don't have to go any further in our coffee ordering/drinking process, so nothing needs to be returned.

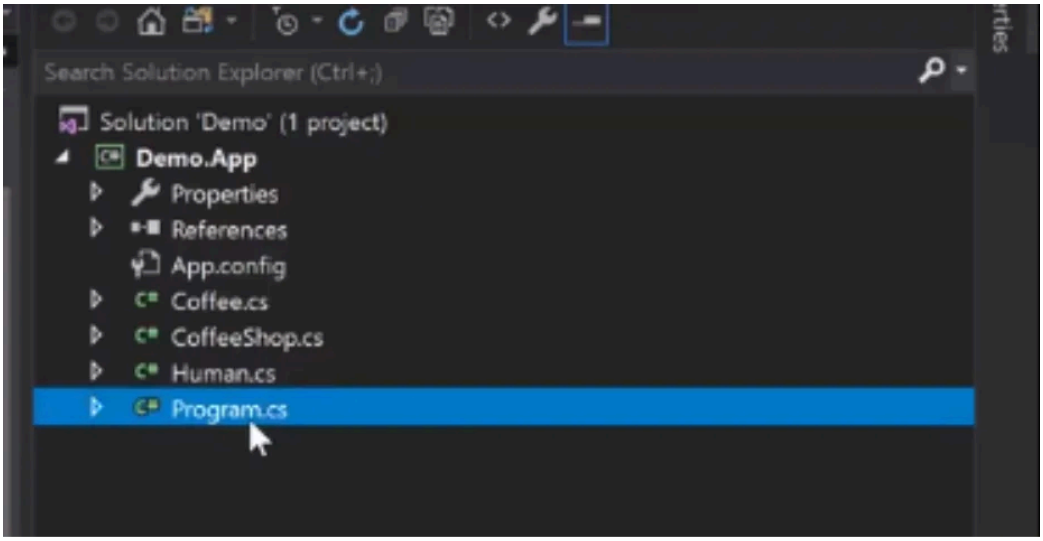
```
            return null;
    }
}

public void Drink(Coffee coffee)
{
    return;
}
```

```
            return null;
    }
}

public void Drink(Coffee coffee)
```

```
{

}
```

```
using System.Globalization;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

How do we visualize that scenario, a human ordering coffee in real life? Let's build a new human. Back to program.cs.



I'm going to put in a new human, say Hassan. And Hassan equals new human. And this human has a name and a date of birth.

```
namespace Demo.App
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            Human hassan = new Human();
            hassan.Name = "Hassan";
            hassan.DateOfBirth = new DateTime();

            Coffee coffee = hassan.OrderCofee(Human.CoffeeType.Espresso);
        }
    }
}
```

```
Human hassan = new Human();
hassan.Name = "Hassan";
hassan.DateOfBirth = new DateTime();
```

Now we want this guy to order a coffee. Let's type a line that says, " The coffee that I want is what Hassan is ordering. And Hassan is ordering an espresso."

And just like all the variables we talked about before, you will notice for this class Coffee, we define the type in the upper case fashion, and define the variable in the lower case fashion.

Variable names can be anything, we can say: Coffee cupOfCoffee; this will still be correct, but class names has to be exactly what we have previously defined.

`Coffee coffee = hassan.OrderCofee(Human.CoffeeType.Espresso);`

```
namespace Demo.App
{
    class Program
    {
        static void Main(string[] args)
        {
            Human hassan = new Human();
            hassan.Name = "Hassan";
            hassan.DateOfBirth = new DateTime();

            Coffee coffee = hassan.OrderCofee(Human.CoffeeType.Espresso);
        }
    }
}
```

The next thing we want here is a place where this coffee is being made, like a coffee shop.

The coffee shop will take that order, and make the coffee for us:

```
CoffeeShop shop = new CoffeeShop();
shop.MakeCoffee(coffee);
```

```
        static void Main(string[] args)
        {
            Human hassan = new Human();
            hassan.Name = "Hassan";
            hassan.DateOfBirth = new DateTime();

            Coffee coffee = hassan.OrderCofee(Human.CoffeeType.Espresso);
            CoffeeShop shop = new CoffeeShop();
            shop.MakeCoffee(coffee);

        }
    }
}
```

Once that coffee is made, now it's time for Hassan to drink that coffee.

Let's explain this line by line:

1. We created a human, in a variable that we called hassan.
2. We gave our human Name property a value, that is "Hassan".
3. We assigned a date of birth to hassan's DateOfBirth property.
4. We created a coffee object, by using hassan OrderCoffee function.
5. Then we created a coffee shop at the same time we created the Hassan object.
6. Then we used the MakeCoffee function in the CoffeeShop variable to make the coffee.

```
hassan.Drink(coffee);
```

```
            Coffee coffee = hassan.OrderCofee(Human.CoffeeType.Espresso);
            CoffeeShop shop = new CoffeeShop();
            shop.MakeCoffee(coffee);

            hassan.Drink(coffee);

        }
    }
```

You see what I did there? I visualized a situation where someone - a human whose name is Hassan, is ordering an espresso. And we have a coffee shop that makes the order. The coffee shop will make the coffee and then Hassan will drink that

coffee.

For an exercise, try to look at something around you, a problem that you have. Let's say your room, your instance. You decide you want to organize your room.

How can you visualize all of the elements in your room so you can have the computer find the best way to organize these elements based on the perimeter of your room?

Let's say your room is x wide by y length. You have a desk and a dresser and the various furniture pieces in your room. You will need dimensions because you will need to input the space of your room.

Then let the computer find out all of the possible ways to organize your room for you. The computer will tell you where to put each item in your room.

CHAPTER 10

Course Recap

Our final chapter here will be a recap of the C# Starting From Scratch course. The image below outlines the main topics we have covered - Variables, Flow Control, Iteration, and classes and objects of which is also known as Object Oriented Programming.

```
using Demo.App;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net;

namespace Hassan.Social.YouTube.Channel
{
    class Program
    {
        static void Main(string[] args)
        {
            #region Variables
            // …
            #endregion

            #region Flow Control
            // …
            #endregion

            #region Iteration
            // …
            #endregion

            #region OOP
            // …
            #endregion

            Console.Read();
        }

        #region OOP Class
        // …
        #endregion
```

```
    }
}
```

These will be the basics and the foundation for all the following C# courses. I am sharing with you the concepts that are very basic. These are the things that just get you off the ground. I assume no previous knowledge whatsoever with programming or C# or the IDE.

Starting off, we talked about variables, how we save numbers and how we save words, strings, booleans, how to save things and arrays, how to reserve that place in memory and then reuse these values into doing other things with our program.

```
static void Main(string[] args)
{
    #region Variables
    // …
    #endregion

    #region Flow Control
    // flow control
    if (booleans)
    {
        // do something
    }
    else
    {
        // do something else
    }

    switch (strings)
    {
        case "blah":
            // do something
            break;
        case "other":
            // do something
            break;
        default:
            break;
    }
```

```
static void Main(string[] args)
{
    #region Variables
    int numbers = 10;
    string strings = "Hello!";
    bool booleans = true;
    int[] arraysOfNumbers = new int[] { 1, 2, 3, 4 };
```

```
    string[] arraysOfStrings = new string[] { "hi", "hola", "hala" };
    #endregion

    #region Flow Control
    // …
    #endregion

    #region Iteration
    // …
    #endregion
```

We also talked about flow control - 'if' statements and switch cases are tools that you can use to have your program perform an action based on a specific input. As well as what to do with the data and how to act upon it.

```
    #region Flow Control
    // …
    #endregion

    #region Iteration
    while (booleans)
    {
        // do something
    }

    do
    {
        // do something
    } while (booleans);

    for (int i = 0; i < arraysOfNumbers.Length; i++)
    {
        // do something
    }

    foreach (string number in arraysOfStrings)
    {
        // do something
    }

    #endregion

    #region OOP
    // …
    #endregion

    Console.Read();
}
```

The third portion of this was the iteration.

If you have a list of things or you want to do something for a certain number of times.

You have 'while', 'do-while', 'for', and 'foreach' options.

These are the very basic concepts that you could use to iterate through a bunch of numbers, or iterate through an array and perform an action.

We also talked about an intro into object oriented programming.

```
            #region Flow Control
            // …
            #endregion

            #region Iteration
            // …
            #endregion

            #region OOP
            Student josh = new Student();
            josh.Name = "Josh McCall";
            josh.Studies("Algorithms");
            #endregion

            Console.Read();
        }

        #region OOP Class
        class Student
        {
            public String Name { get; set; }
            public void Studies(String subject)
            {

            }
        }
        #endregion
    }
}
```

We discussed what the classes are and how you visualize something in real life and turn that into either properties or methods, things that the visualized object could do, or things that can be done to that object.

Also things like properties, things that describe that object from an information perspective ilke a student for instance - what their name, age, date of birth, what class they're in, and you use that data and manipulate it for your own program to

serve a specific need.

Go through these things - these are the fundamentals of starting to learn C#. The next course for this one is going to be “Scratching The Surface With C#”, where I’m going to dive deeper into these subjects and programming.

There will be a lot to learn from a control flow perspective, from an iteration, from variables, I called it flow control.

Play around with the fundamentals you have learned in this book and I’ll see you in the next one - “Scratching The Surface With C#”!