C# [CSharp] Programming.

C# vs .Net

* C# is a programming Language.

* .Net is a framework for building applications on Windows, the dotnet framework is Not Limited to C#. There are many Languages in it.

.Net :-

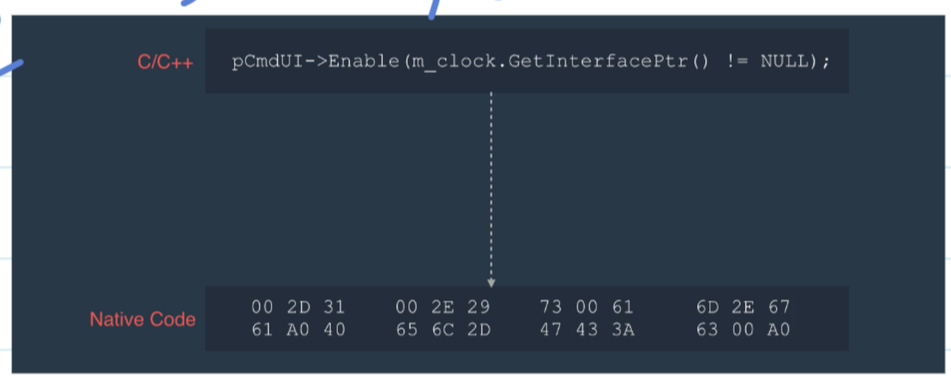
.Net consists of two components.

① CLR [Common Language Runtime]
② Class Library for developing applications.

CLR [Common Language Runtime]

Before going to CLR, we need to know a bit of history of C#.

Before C#, we have two Languages in C family ① C Language ② C++
In the both cases, the compiler will turn the code by converting it into its Native Code of that machine in which it was wrote.



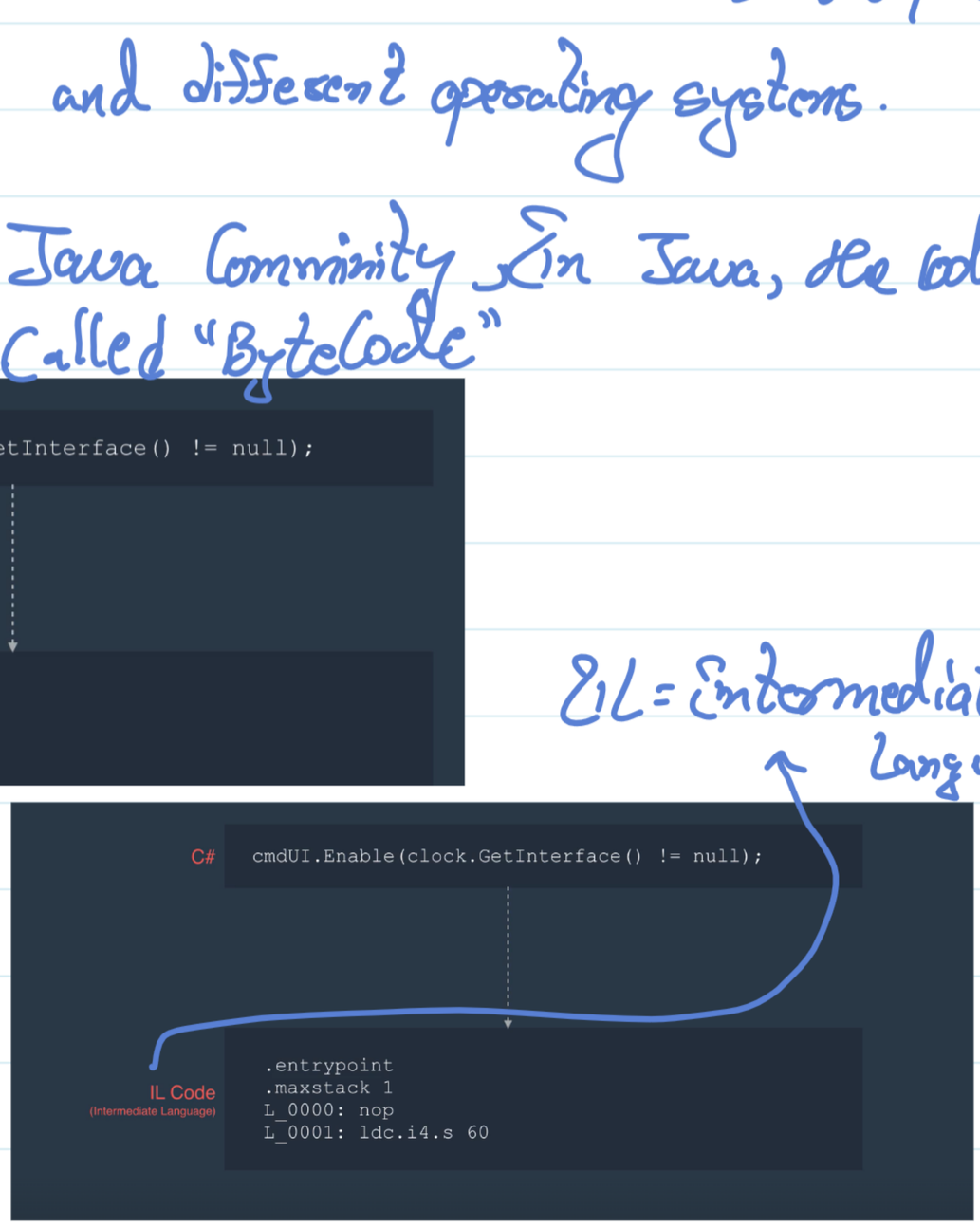
→ It means if we are writing on windows → it will write the native code, only for that particular machine, which has some processor.

Now, we have different hardwares and different operating systems.

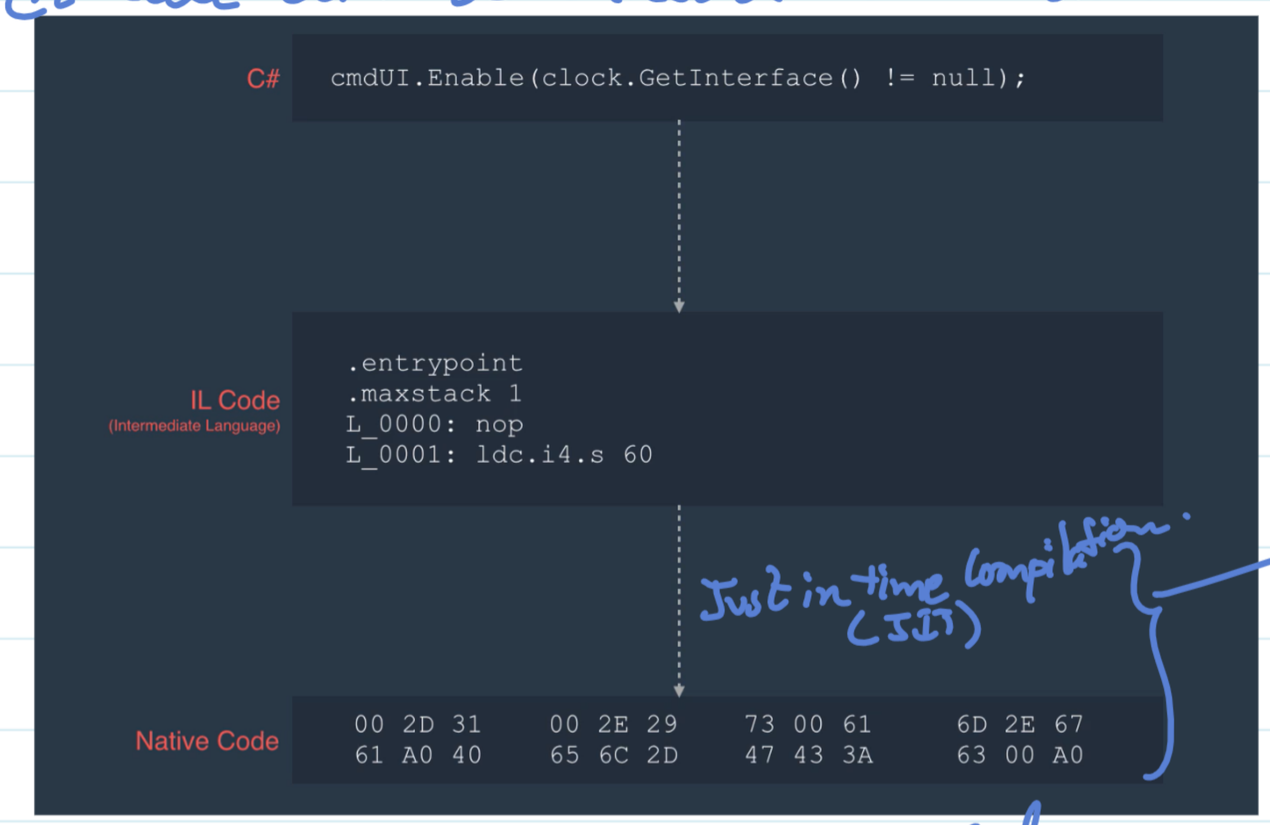
So Microsoft, took an concept of Java Community (In Java, the code will convert to an intermediate Code Called "ByteCode"

IL = Intermediate Language Code

the same Concept, we have in C#



then the IL code will be converted to Native code to run the application.

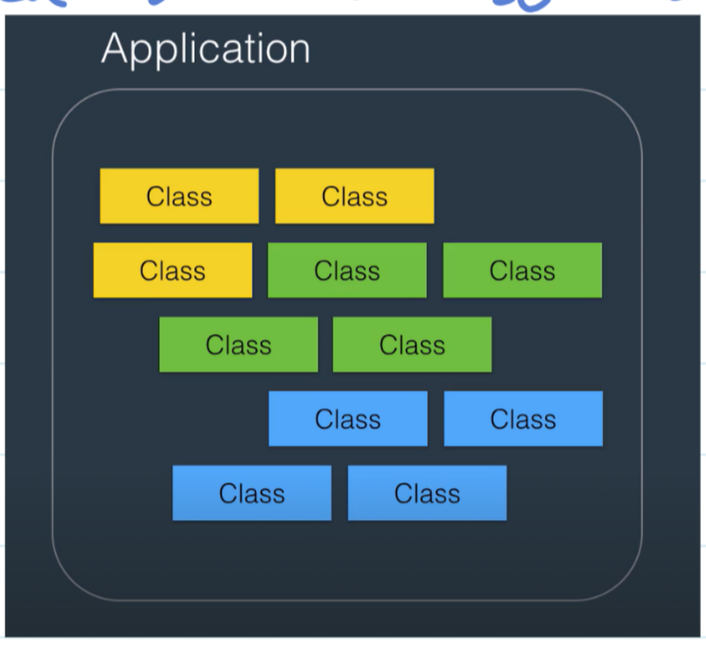


So, CLR is an application, sitting in the memory (Central) → It's job is to convert the IL code to machine code (or) Native code → this process is called "Just in time Compilation [JIT]"

So, in this case we can write a code in windox, & it will work in any machine as long as the other machines has CLR application in those system.

Architecture of .Net Applications.

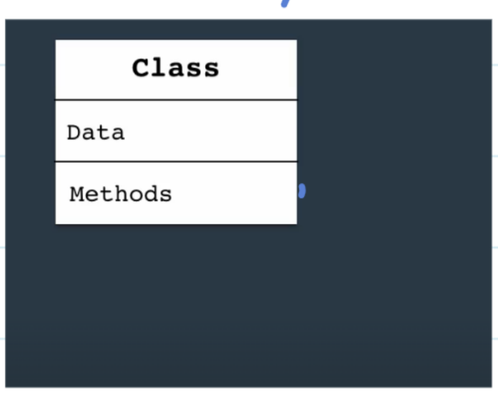
at an very high level, when we built an application. then the application will have building blocks called Class. → these Class Collebarate with each other. at run time

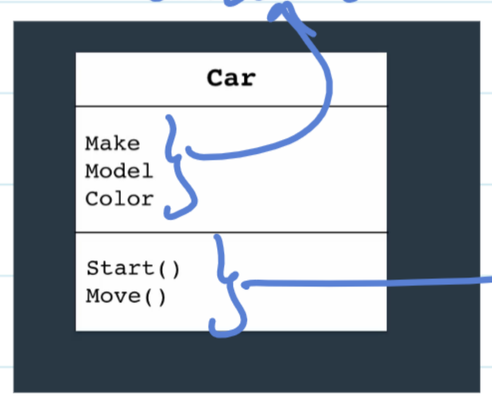


what is a class?

a class is an container that has some data called attributes and some function called methods. the functions (or) methods have the behaviors, they Exibute the Code. they do things for us.

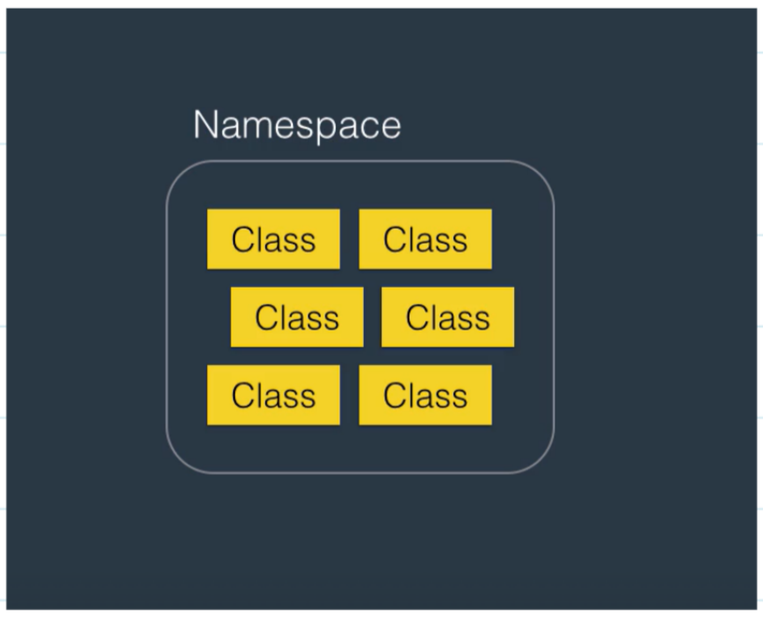
Data represents the state of the application → Example Car → the Car has some attributes.





As the number of classes grows in an application, we need to organize these class. So, we use "NameSpaces"

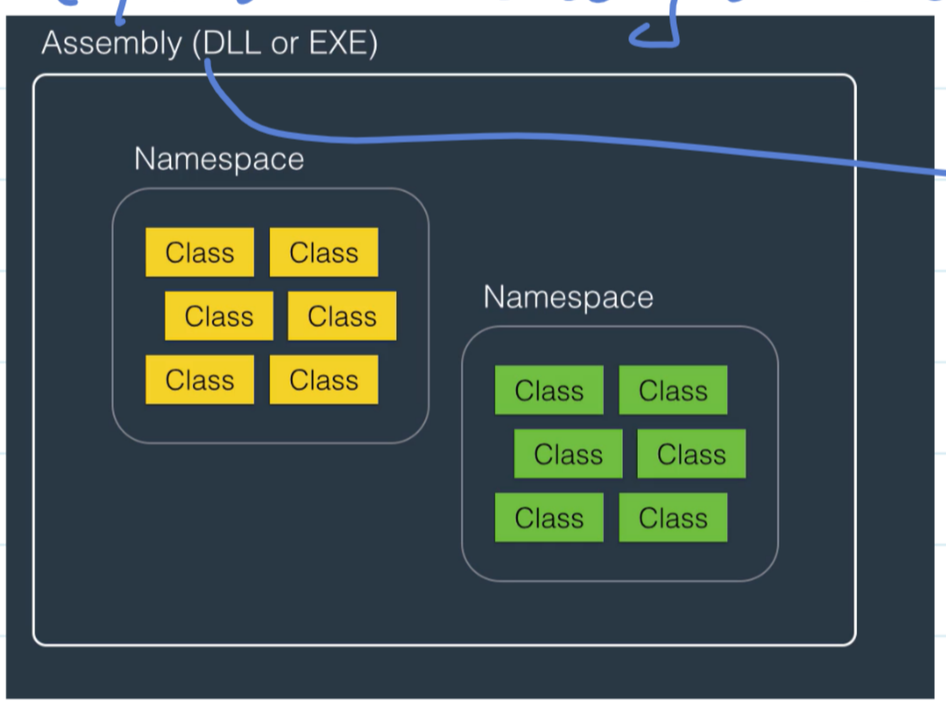
NameSpaces - It is an containes for related classes. for Example in .Net framework, we have NameSpaces with related classes.



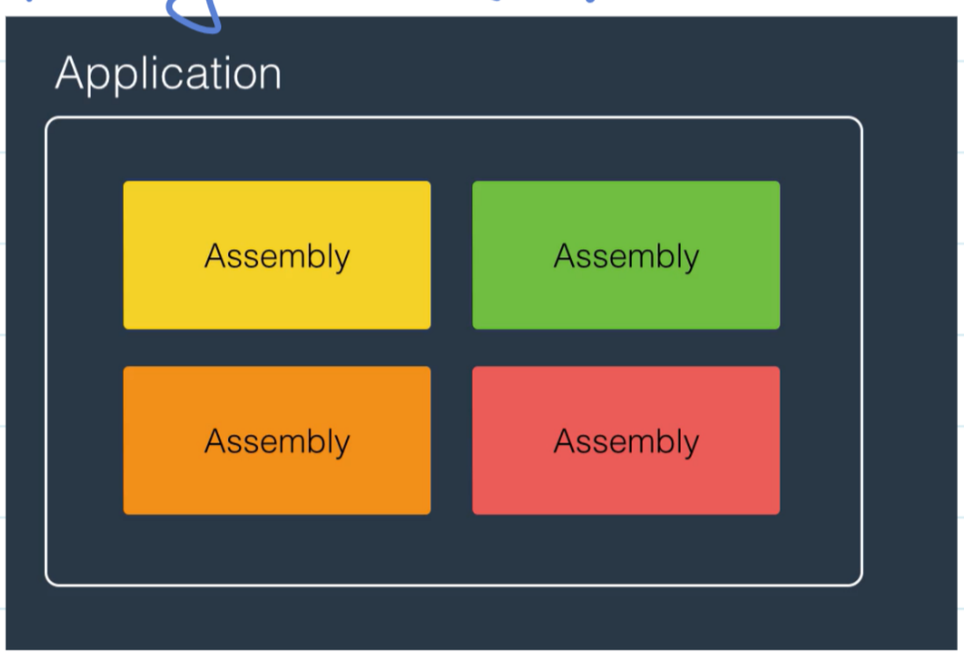
we have namespaces for data, images, Graphs, Security ....

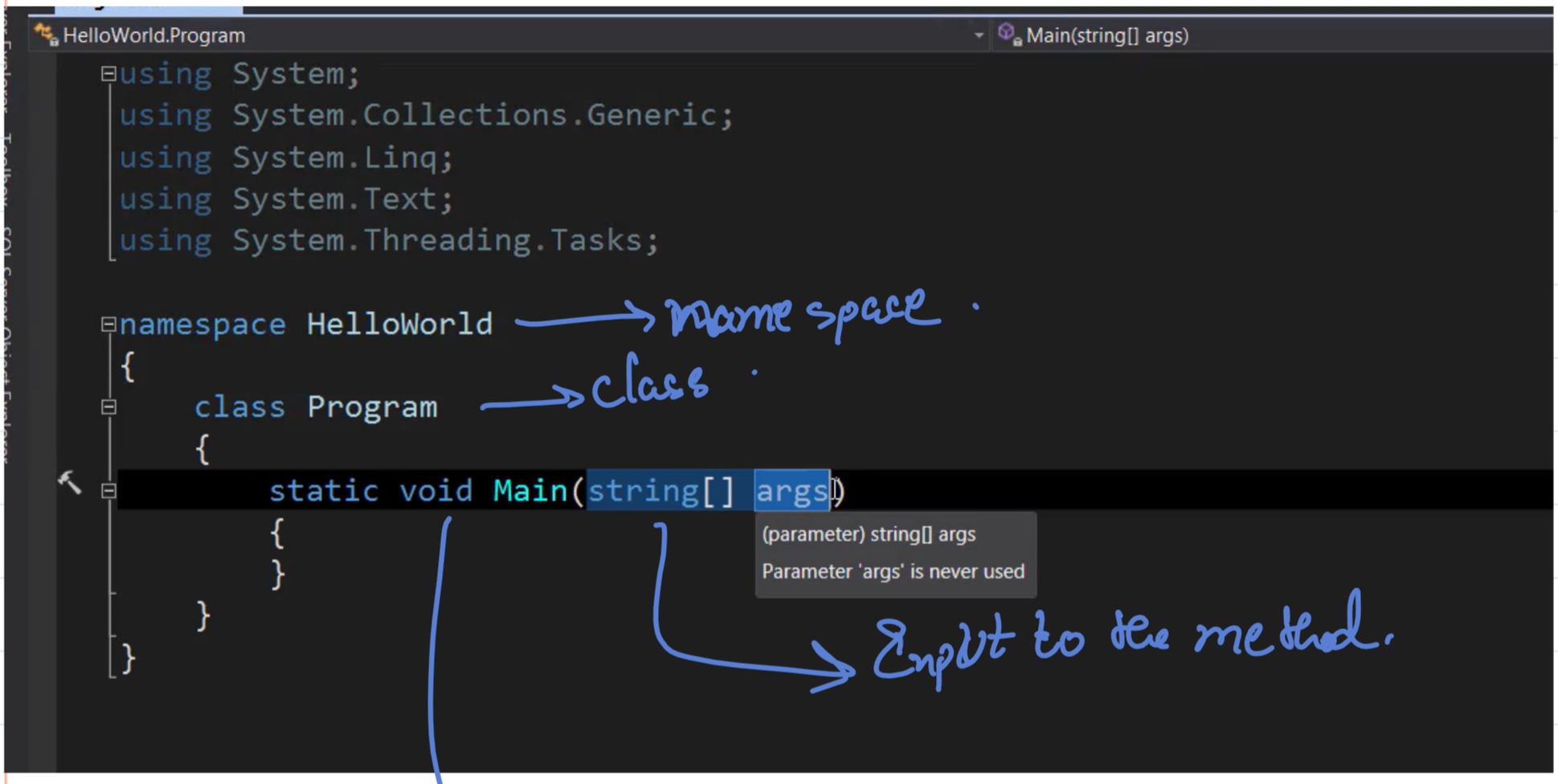
So, we have Many Name spaces in the application, so we need to partition the Namespaces - that is why we use assembly → it is an containes of different Namespaces.

DLL = Dynamically Linked library.



So, when we compile an application, the compiler will build one or more Assembly depending on How you partition our code.



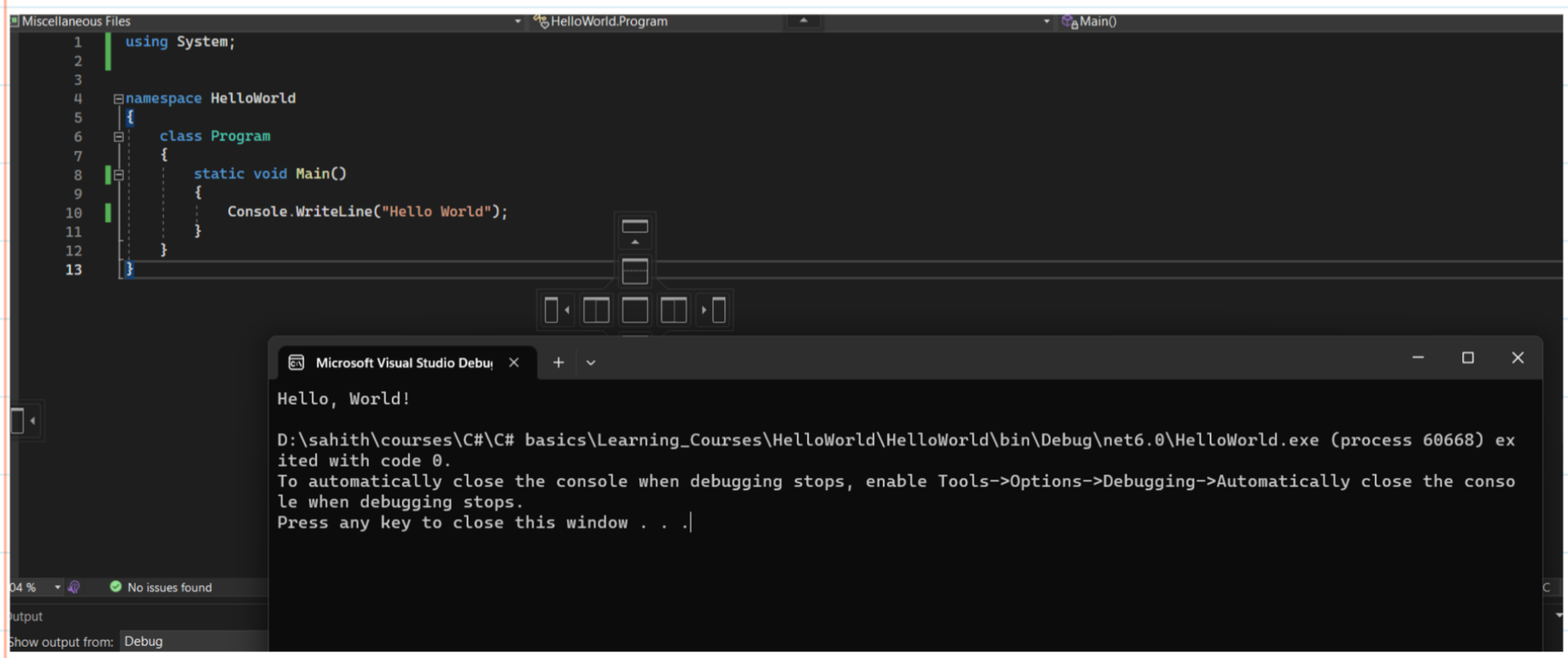


we have a class called Console. → which is used to read data (or) write data.
It has many methods, we can access with dot Notation.

Program:-
to Print "Hello world"

```
using System;

namespace Helloworld
{
    class Program.
    {
        static void Main( )
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

output.
Hello world.



Questions:-

1 / 3

What is a namespace?

- A type that contains code for the program.
- A container for classes. ✓
- A deployable unit of application.

2 / 3

What is JIT Compilation?

- The compilation of IL code to native machine code at run-time. ✓
- The compilation of C# code to native code.
- The compilation of C# code to IL code.

3 / 3

What is an Assembly?

- A piece of code that is executed by CLR.
- Container of one or more classes.
- A single unit of deployment of .NET applications. ✓

Summary

So in this section, you learned the basics of C#.

C# vs .NET

C# is a programming language, while .NET is a framework. It consists of a run-time environment (CLR) and a class library that we use for building applications.

CLR

When you compile an application, C# compiler compiles your code to IL (Intermediate Language) code. IL code is platform agnostics, which makes it possible to a take a C# program on a different computer with different hardware architecture and operating system and run it. For this to happen, we need CLR. When you run a C# application, CLR compiles the IL code into the native machine code for the computer on which it is running. This process is called Just-in-time Compilation (JIT).

Architecture of .NET Applications

In terms of architecture, an application written with C# consists of building blocks called classes. A class is a container for data (attributes) and methods (functions). Attributes represent the state of the application. Methods include code. They have logic. That's where we implement our algorithms and write code.

A namespace is a container for related classes. So as your application grows in size, you may want to group the related classes into various namespaces for better maintainability.

As the number of classes and namespaces even grow further, you may want to physically separate related namespaces into separate assemblies. An assembly is a file (DLL or EXE) that contains one or more namespaces and classes. An EXE file represents a program that can be executed. A DLL is a file that includes code that can be re-used across different programs.

In the next section, you'll learn about basics of the C# language, including variables, constants, type conversion and operators.

Primitive Types and Expressions :-

Variables and Constants :-

Variable :- A name given to a storage Location in Memory.
Constant :- a immutable value.
↳ which means it cannot be changed.
Ex :- Pi = 3.14.

Declaring Variables / constants :-
int number ; → semicolon.
↳ type of the variable. → name of the variable.
→ C++ is an case sensitive.

int Number = 1;
Const float Pi = 3.14f; → value.
↳ const ↳ Datatype → name(or) identifier

rules for identifiers :-
1. Cannot start with a number Ex :- 1Route ✗, OneRoute ✓
2. Cannot include a whitespace ; Ex :- first Name ✗, FirstName ✓.
3. Cannot be an reserved Keyword. Ex : int ✗, @int ✓
4. Always use meaning Names Ex :- fn ✗, firstName ✓

In terms of Naming Conventions, these are three popular Naming Conventions.
1. Camel Case :- firstName
2. Pascal Case :- FirstName.
3. Hungarian Notation :- strFirstName.
↳ datatype → we don't use in C++

So, we can use, Camel Case (or) Pascal Case.
Ex :- int number; → for Variables.
Const int MaxZoom = 5; for constants.

Primitive Types

	C# Type	.NET Type	Bytes	Range
Integral Numbers	**byte**	Byte	1	0 to 255
	short	Int16	2	-32,768 to 32,767
	int	Int32	4	-2.1B to 2.1B
	long	Int64	8	…
Real Numbers	**float**	Single	4	-3.4×10^{38} to 3.4×10^{38}
	double	Double	8	…
	decimal	Decimal	16	-7.9×10^{28} to 7.9×10^{28}
Character	**char**	Char	2	Unicode Characters
Boolean	**bool**	Boolean	1	True / False

Real Numbers

	C# Type	.NET Type	Bytes	Range
Real Numbers	**float**	Single	4	-3.4×10^{38} to 3.4×10^{38}
	double	Double	8	…
	decimal	Decimal	16	-7.9×10^{28} to 7.9×10^{28}

```
float number = 1.2f;

decimal number = 1.2m;
```

to make as float.

Char is initialized in (' ') [single quotes]
String " " [" "] [double quotes]

overflowing :-

byte numbers = 255; → the Largest va.
number = number + 1; // 0. → we have Exceeded the size of byte.
so, we get output "0"

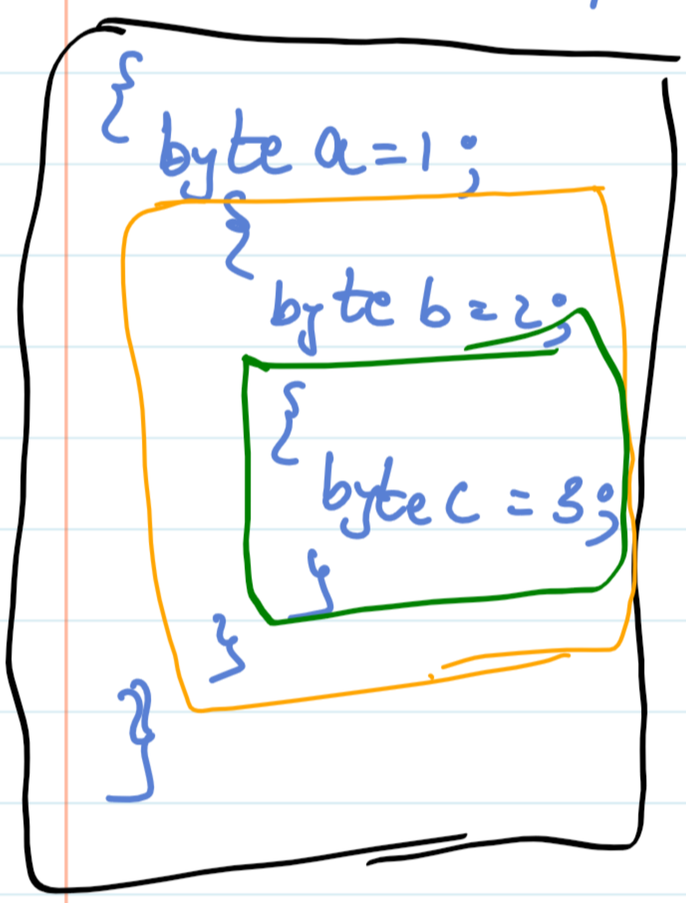
If you want to stop the overflow, we need to use "checked" keyword.

```
checked
{
  byte numbers = 255;
  number = number + 1;
}
```

} with this overflow will not happen at runtime. instead an Exception will be throoced. & Program will be crashed, if we don't handle it the Exception properly.

Scope :-

* where a Variable / constant has meaning and accessable.

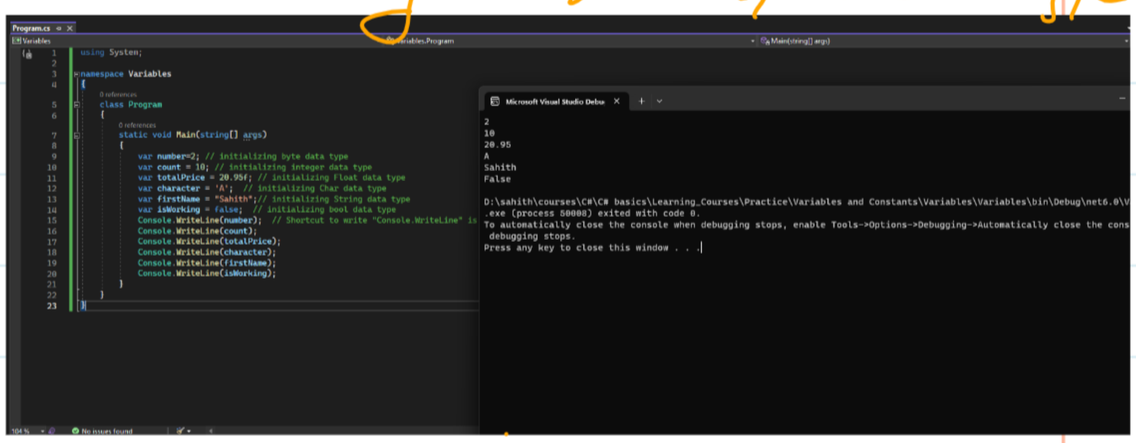


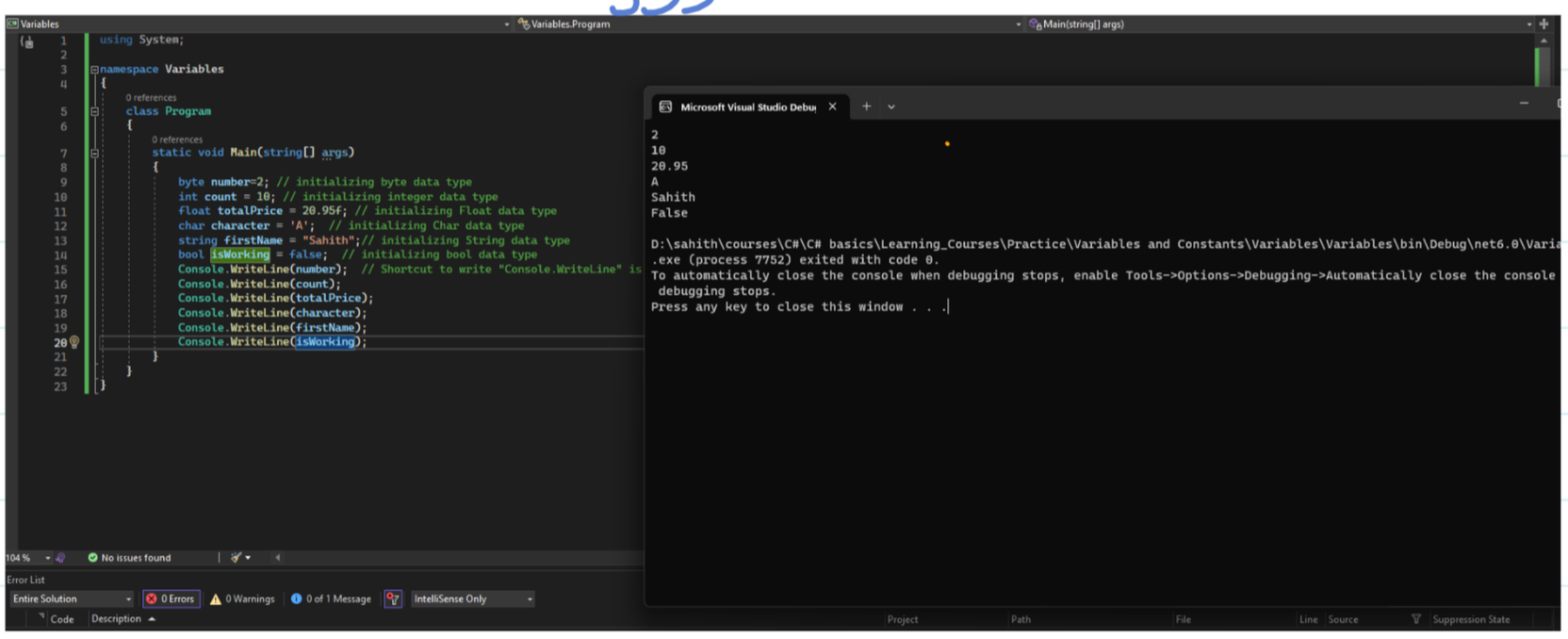
} Here we have 3 blocks of code.

Write an C# program to initialize and print all the data types :-

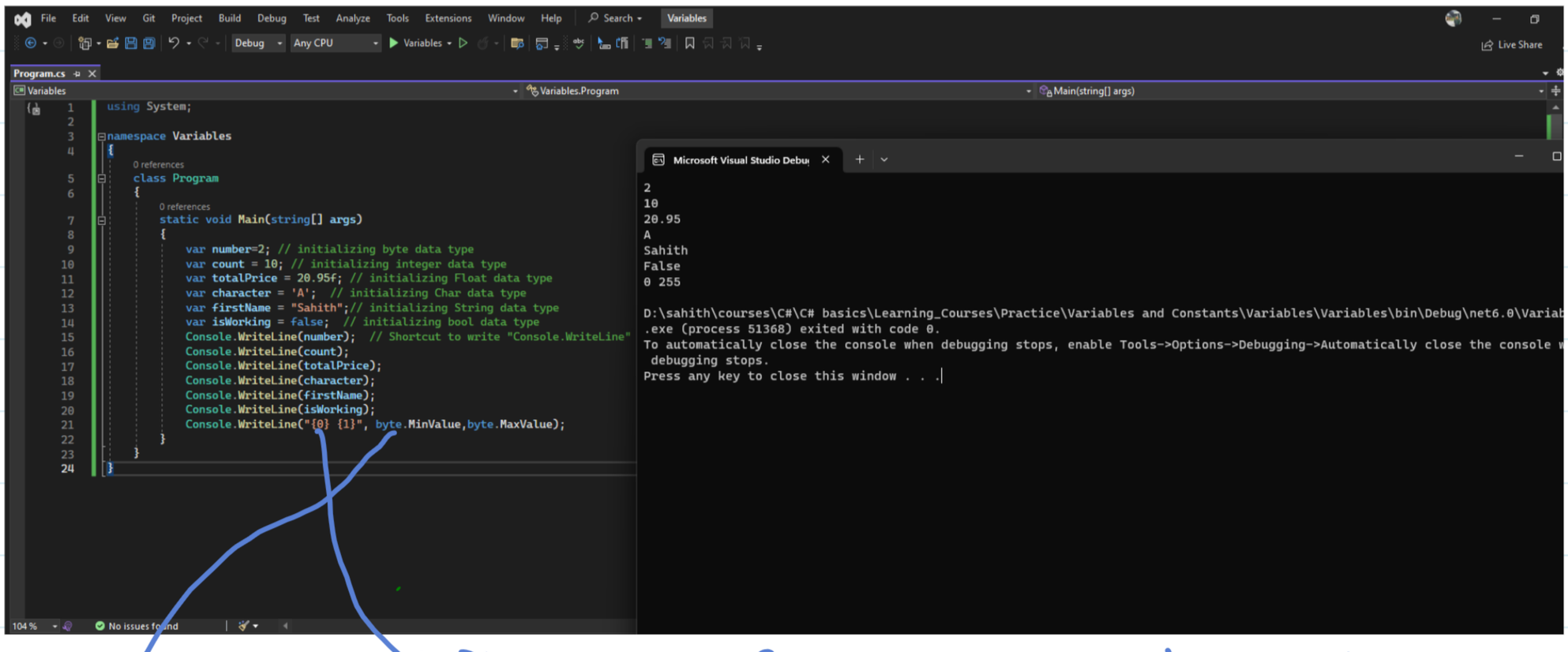
```
using System;                                   → name of the project
namespace Variable                              → class name
{
    Class Program                               → method name
    {
        static void Main(string[] arg)
        {
            byte number=2;
            int count=10;
            float totalPrice=20.95f;
            Char character='A';
            string firstName="Sahith"
            bool isWorking = false;
            Console.writeLine(number);
            Console.writeline(count);
            Console.writeline(totalPrice);
            Console.writeline(character);
            Console.writeline(firstName);
            Console.writeline(isWorking);
        }
    }
}
```

In C#, we have keyword "var", we no need to write byte, int, float ---
we can just write var, the compiler automatically takes it respective datatype.





Shortcut :- Press "Control + x" to delet the particular line.



we can do it to float, int...

It is a format for string. {0} = 1st argument i.e minvalue.
{1} = 2nd argument i.e maxvalue.

Type Conversion:-

we have ① Implicit type Conversion.
② Explicit type Conversion (Casting)
③ Conversion between non-compatible types.

Example for implicit type Conversion:-

```
byte b=1;
int i=b;
```

byte consists of 1byte of memory } the binary representations.
but integer takes 4 bytes.

00000001

00000000 00000000 00000000 00000001

2nd Example for implicit type Conversion:-

```
int i=1;
float f=i;
```

In this case Not data will get Loss.

3rd Example

```
int i=1;
byte b=i;
```

→ 1 can be converted to byte, But if we have 300 a.... Hence need Explicitly converted.

// won't compile. } Integers is 4 bytes, when we convert 4 bytes to 1byte, there are many chances of data Loss (i.e 3byte)

If we want to forcefully convert the integers to byte. we can use Explicit type.

Ex:- `int i=300;`
`byte b=(byte)i;`

{ we know that data will be Lost, But also if we want to convert, we can use this Explicit type }

↳ This process is called "Casting"

Ex:- `float f=1.0f;`
`int i=(int)f;`

Some, datatypes are not compatable. Like.

Ex:- `String s="1";`
`int i=(int)s;` // won't compile.

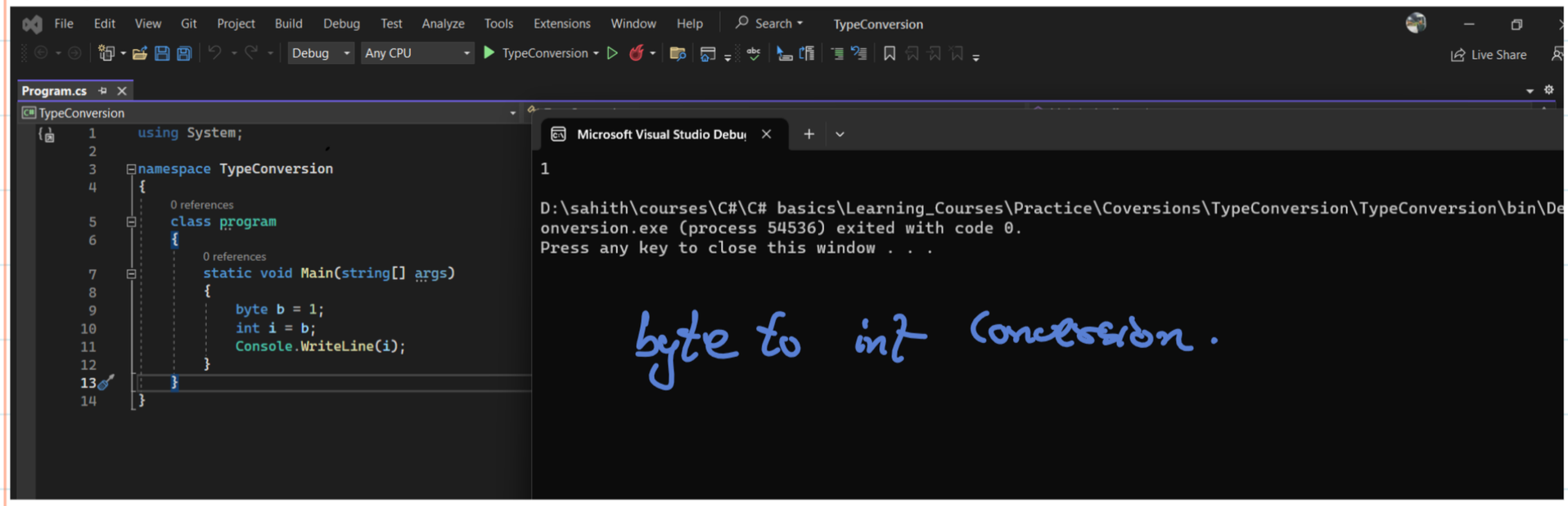
In this, we need an different mechanism, to convert string to int.

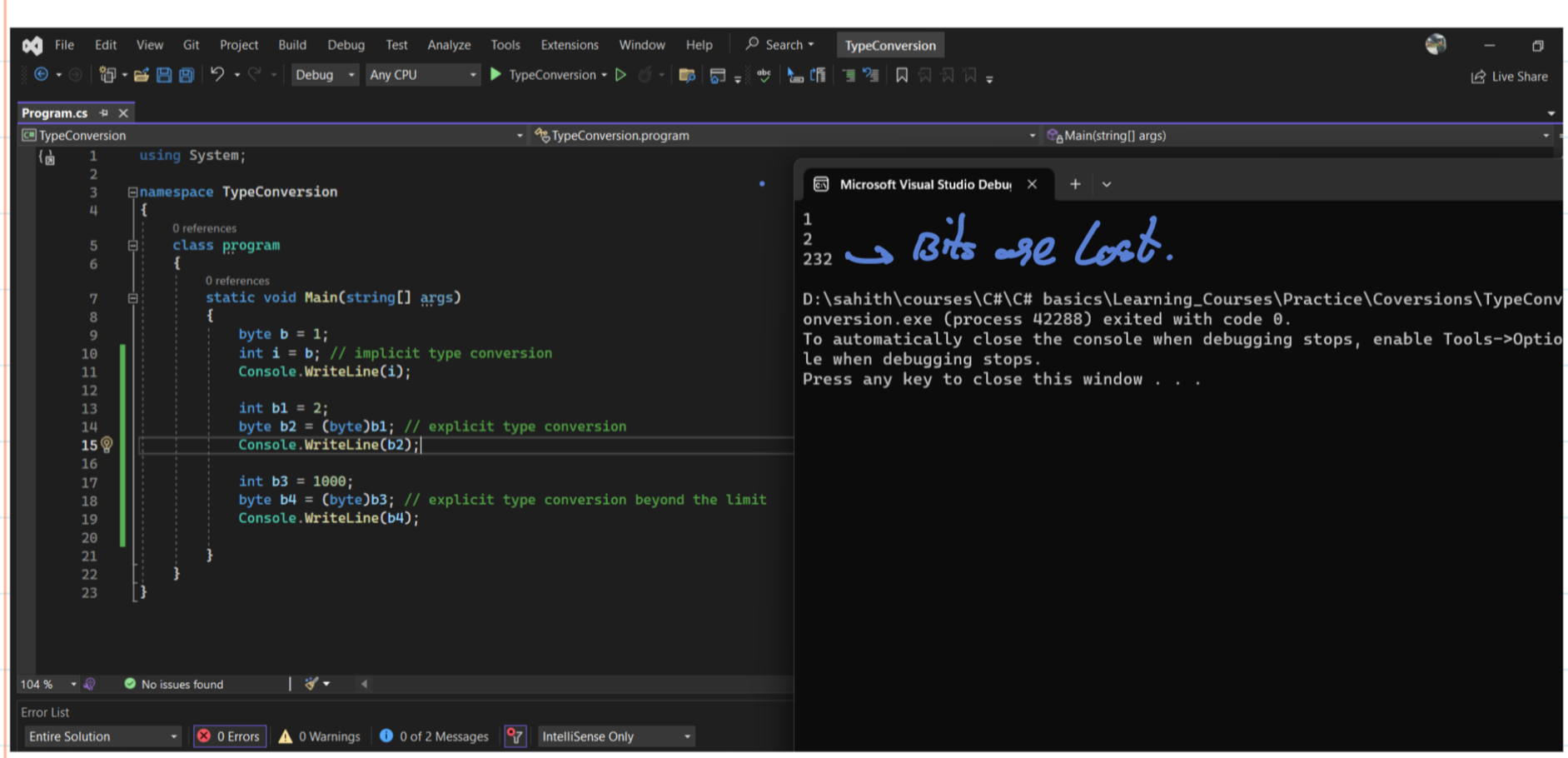
Ex:- `String s="1";`
`int i=Convert.ToInt32(s);` // we can use Convert class, which is in .Net framework
`int j=int.Parse(s);` // (or) we can use Parse method.

the methods, we have in Convert class:-

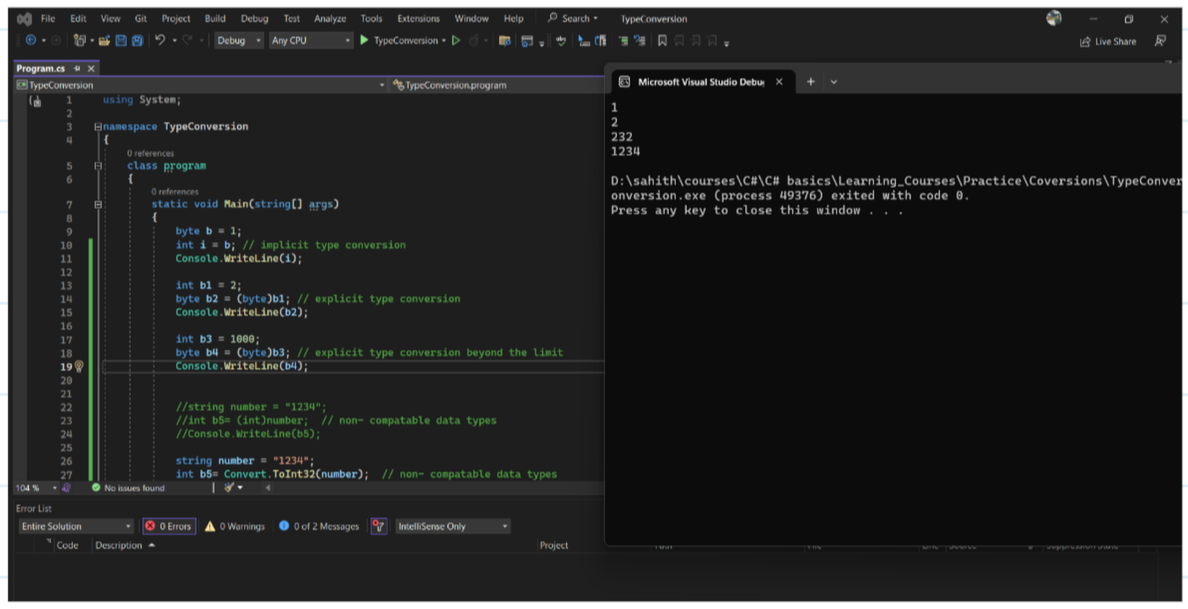
* ToByte()
* ToInt16()
* ToInt32()
* ToInt64()

Program implicite type conversion:-





using System;




```
namespace TypeConversion
{
   class program
   {
      static void Main(string[] args)
      {
         byte b = 1;
         int i = b; // implicit type conversion
         Console.WriteLine(i);

         int b1 = 2;
         byte b2 = (byte)b1; // explicit type conversion
         Console.WriteLine(b2);

         int b3 = 1000;
         byte b4 = (byte)b3; // explicit type conversion beyond the limit
         Console.WriteLine(b4);


         //string number = "1234";
         //int b5= (int)number;  // non- compatable data types
         //Console.WriteLine(b5);

         string number = "1234";
         int b5= Convert.ToInt32(number);  // non- compatable data types
         Console.WriteLine(b5);


         try
         {
            string number1 = "1234";
            int b6 = Convert.ToByte(number);  // the code will crash so, we can use try
                                              // and catch method
            Console.WriteLine(b6);
         }
         catch(Exception)
         {
            Console.WriteLine("the number could not be converted to byte");
         }
      }
   }
}
```

C# Operators :-

1. Arithmetic Operators.
2. Comparision Operators.
3. Assignment Operators.
4. Logical Operators.
5. Bitwise Operators.

1) Arithmetic operators.

	Operator	Example
Add	**+**	a + b
Subtract	**-**	a - b
Multiply	*****	a * b
Divide	**/**	a / b
Remainder	**%**	a % b

	Operator	Example	Same as
Increment	**++**	a++	a = a + 1
Decrement	**--**	a--	a = a - 1

the increment and decrement can be done in two ways 1) Postfix
2) Prefix.

Example for Postfix:- `int a=1;` `int b=a++;` o/p = a=2, b=1

Example for Prefix:- `int a=1;` `int b=++a;` o/p :- a=2, b=2

2) Comparision Operators :-

	Operator	Example
Equal	**==**	a == b
Not Equal	**!=**	a != b
Greater than	**>**	a > b
Greater than or equal to	**>=**	a >= b
Less than	**<**	a < b
Less than or equal to	**<=**	a <= b

(3) Assignment Operators :-

	Operator	Example	Same as
Assignment	**=**	a = 1	
Addition assignment	**+=**	a += 3	a = a + 3
Subtraction assignment	**-=**	a -= 3	a = a - 3
Multiplication assignment	***=**	a *= 3	a = a x 3
Division assignment	**/=**	a /= 3	a = a / 3

(4) Logical Operators :-

	Operator	Example
And	**&&**	a && b
Or	**II**	a || b
Not	**!**	!a

(5) Bitwise operators :-

	Operator	Example
And	**&**	a & b
Or	**I**	a | b

What are Logical Operations?

Logical operations are part of Boolean algebra, which is often taught to students of computer science and electronic engineering at University. So, if you're not familiar with Boolean algebra, here is a brief introduction.

In Boolean algebra, the value of variables can only be **true** or **false**, also denoted 1 and 0 respectively. Unlike elementary algebra, where the main operations are addition, subtraction, etc, the main operations of Boolean algebra are **conjunction** (AND), **disjunction** (OR) and **negation** (NOT).

Logical AND

Let's assume we have two variables: **x** and **y**. In C#, the logical AND operator is indicated by &&. We can define a Boolean expression as follows:

```
z = x && y
```

In this expression, **z** is true if *both* **x** and **y** are true; otherwise, it'll be false.

What is a real-world example of this in programming? Imagine you're developing a loan application. The provider only offers loans to applicants who are over 18 and are citizen of the given country. In this example, we have two variables:

x = applicant being over 18
y = application being a citizen

z = is eligible to apply for loan = x && y

If *both* **x** and **y** are true, the applicant is eligible to apply for a loan.

Later, when we get to conditional statements, you can check to see if the above expression evaluates to true or false, and then, can change the flow of your application.

So, here is the rule of thumb with logical AND: if *both* **x** and **y** are true, **x && y** will be true; otherwise, it'll be false.

Logical OR

In C#, logical OR is indicated by two vertical lines (||). Considering the following expression:

```
z = x || y
```

z will be true, if *either* **x** or **y** is true.

What is a real-world example of this? Imagine you're building software for a recruiter. For a given job application, applicants can apply if they have a degree in computing, or more than 5 years of experience in the field. You can model this using a Boolean expression as follows:

x = applicant has a degree in computing
y = applicant has more than 5 years of experience

z = application is eligible = x || y

If either **x** or y is true, z will be true.

So, unlike the logical AND, where both variables must be true, with logical OR, if at least one of them is true, the result will be true.

Logical NOT

The NOT operator in C# is indicated by an exclamation mark (!) and it reverses the value of a given variable or expression:

```
y = !x
```

So, here, if **x** is true, **y** will be false, and if **x** is false, **y** will be true.

Clean Coding

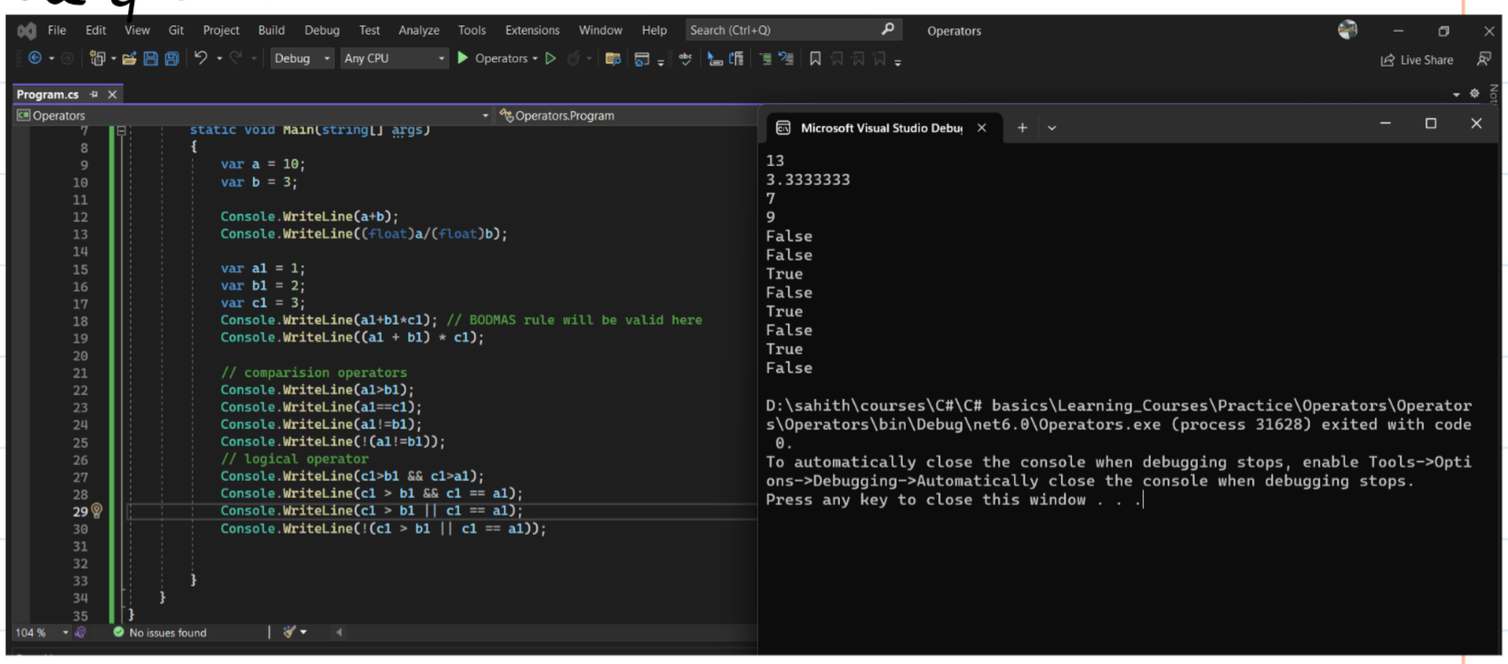
In all examples here, I used variables **x** and **y**, mainly to relate programming to Boolean algebra. But when it comes to coding, you should avoid using variable names such as x, y, z as they don't give a clue to other developers reading your code (or even yourself). Instead, use meaningful names. For instance, in the first example, you can replace **x**, **y** and **z** as follows:

```
x: isOver18
y: isCitizen
z: isEligible
```

Often, it's a good practice to prefix Boolean names with IS or HAS (if possible).

Program to perform all the operators :-

```
using System;

namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            var a = 10;
            var b = 3;

            Console.WriteLine(a+b);
            Console.WriteLine((float)a/(float)b);

            var a1 = 1;
            var b1 = 2;
            var c1 = 3;
            Console.WriteLine(a1+b1*c1); // BODMAS rule will be valid here
            Console.WriteLine((a1 + b1) * c1);

            // comparision operators
            Console.WriteLine(a1>b1);
            Console.WriteLine(a1==c1);
            Console.WriteLine(a1!=b1);
            Console.WriteLine(!(a1!=b1));
            // logical operator
            Console.WriteLine(c1>b1 && c1>a1);
            Console.WriteLine(c1 > b1 && c1 == a1);
            Console.WriteLine(c1 > b1 || c1 == a1);
            Console.WriteLine(!(c1 > b1 || c1 == a1));


        }
    }
}
```



Comments :-

Single-line Comment	`// Here is a single-line comment` `int a = 1;`

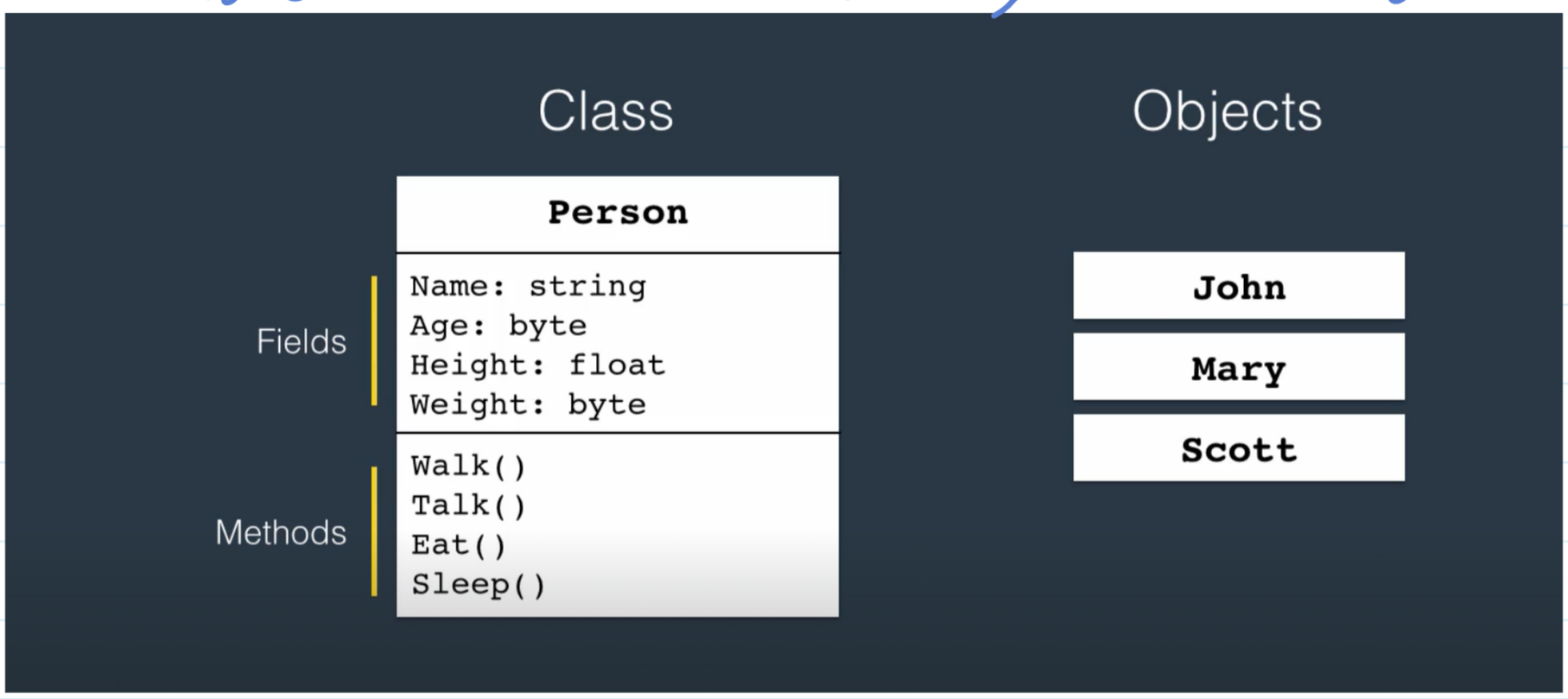
Multi-line Comments	`/*`
	`  Here is a multi-line`
	`  comment`
	`*/`
	`int a = 1;`

Non-Primitives :-

Classes :- Classes are the building Blocks of the Application.

* Classes combine related variables (fields) and functions (methods)

Ex :-



Here, I've created a class called Person, with 4 fields [Name, Age, Height, Weight] and also, it has 4 functions (or) methods [Walk(), Talk(), Eat(), Sleep()].
This Class is a type (or) Blueprint from which we create objects. So Object is an instance of a class. In this Example we can create 3 instance or Objects [John, Mary, Scott]. So when we run the project these objects collaborate Each to get output.

Syntax to create Class in C#

```
public Class Person
{
    Public string Name;

    public void Introduce()
    {
        Console.WriteLine("Hi, my name is" + Name);
    }

}
```

- public → access modifier. It shows, Intro can access the code.
- Class keyword → identifies.
- defining an field (or) variable.
- * Classes can also contain method.

Declaring Classes.

```
public class Calculator
{
    public int Add(int a, int b)
    {
        return a+b;
    }
}
```

→ which takes parameters.

Creating Objects:-

int number; → type → identifier

Person person = new Person();

→ memory allocation.

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

we can write the code as:-

```
var person = new Person();
person.Name = "Nash";
person.Introduce();
```

Static Modifies:-

Ex:-

```
public class Calculator
{
    public int Add(int a, int b)
    {
        return a+b;
    }
}
```

→ static

we can access like this.

```
int result = Calculator.Add(1,2);
```

Program to create an class.

```
using System;
namespace CSharpFundamentals
{
    public class Person
    {
        public string FirstName;
        public string LastName;
        public void Introduce()
        {
            Console.WriteLine("My name is " + FirstName + " " + LastName);
        }
    }
    class Programm
    {
        static void Main(string[] args)
        {
            var john = new Person();
            john.FirstName = "John";
            john.LastName = "Smith";
            john.Introduce();
        }
    }
}
```

O/P

My name is John Smith.

Arrays:

* Array is a data structure to store a collection of variables of the same type.

Ex:

```
int number1;
int number2;
int number3;
```

→ this can be replaced by

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

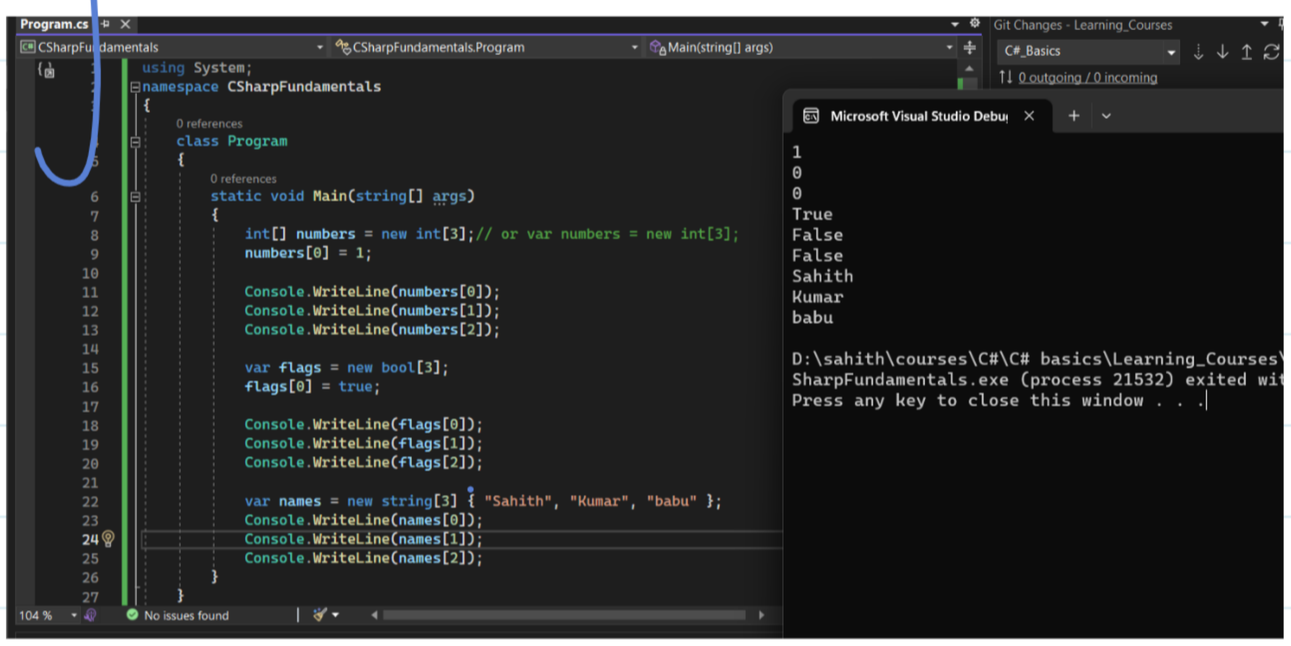
↳ size of the array.

Accessing Array Elements:-

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

(or)

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




```
using System;
namespace CSharpFundamentals
{
   class Program
   {
      static void Main(string[] args)
      {
         int[] numbers = new int[3];// or var numbers = new int[3];
         numbers[0] = 1;

         Console.WriteLine(numbers[0]);
         Console.WriteLine(numbers[1]);
         Console.WriteLine(numbers[2]);

         var flags = new bool[3];
         flags[0] = true;

         Console.WriteLine(flags[0]);
         Console.WriteLine(flags[1]);
         Console.WriteLine(flags[2]);

         var names = new string[3] { "Sahith", "Kumar", "babu" };
         Console.WriteLine(names[0]);
         Console.WriteLine(names[1]);
         Console.WriteLine(names[2]);
      }
   }
}
```

Strings:- It is a sequence of characters. Exs- "Hello World".

Creating Strings:-

Ex:- string firstName = "Sahith";
we can also concatinate the strings with "+" operator.
Ex:- string name = firstName + " " + LastName;
we can use string format
Ex:- String name = string.Format ("{0} {1}", firstName, LastName);

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

{0} {1} → placeholders.
string.Format → an method.
{0} {1} → used like an template

* So times we may have array of objects.

Ex:- var numbers = new int[3] { 1,2,3 };
string List = string.Join (",", numbers);

* If we have an string, we can read there individual characters.

Ex:- string name = "Nosh"
char firstChar = name[0];

→ Strings are Immutable:- which means, Once we create them, we cannot change them.

Paths in strings

Escape Characters

Char	Description
\n	New Line
\t	Tab
\\	Backslash
\'	Single Quotation Mark
\"	Double Quotation Mark

Verbatim Strings

```
string path = "c:\\projects\\project1\\folder1";
(or)
string path = @"c:\projects\project1\folder1";
```

```
using System;

namespace CSharpFundamentals
{
    class Program
    {
        static void Main(string[] args)
        {
            string firstName = "Sahith Kumar";
            var lastName = "Singari";

            var fullName = firstName + " " + lastName; // concatination

            // concatination using string.format
            var myFullName = string.Format("My name is {0} {1}", firstName, lastName);

            var names = new string[3] { "John", "Jack", "Mary" };
            var formattedNames = string.Join(", ", names);
            Console.WriteLine(formattedNames);

            var text = "Hi John \n Look into the following paths \n c:\\folder1\\folder2\n c:\\folder3\\folder4";
            Console.WriteLine(text);

            // we can use an verbatim string for the above case
            var text1 = @"Hi John
                Look into the following paths
                c:\folder1\folder2
                c:\folder3\folder4";
            Console.WriteLine(text1);
        }
    }
}
```

Enum :- It is a set of names/value pairs [constants]
If we have multiple constants, we can use Enum.

```
const int RegularAirMail = 1;
const int RegisteredAirMail = 2;
const int Express = 3;


public enum ShippingMethod
{
    RegularAirMail = 1,
    RegisteredAirMail = 2,
    Express = 3;
}
```

we can use it with dot notation.

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

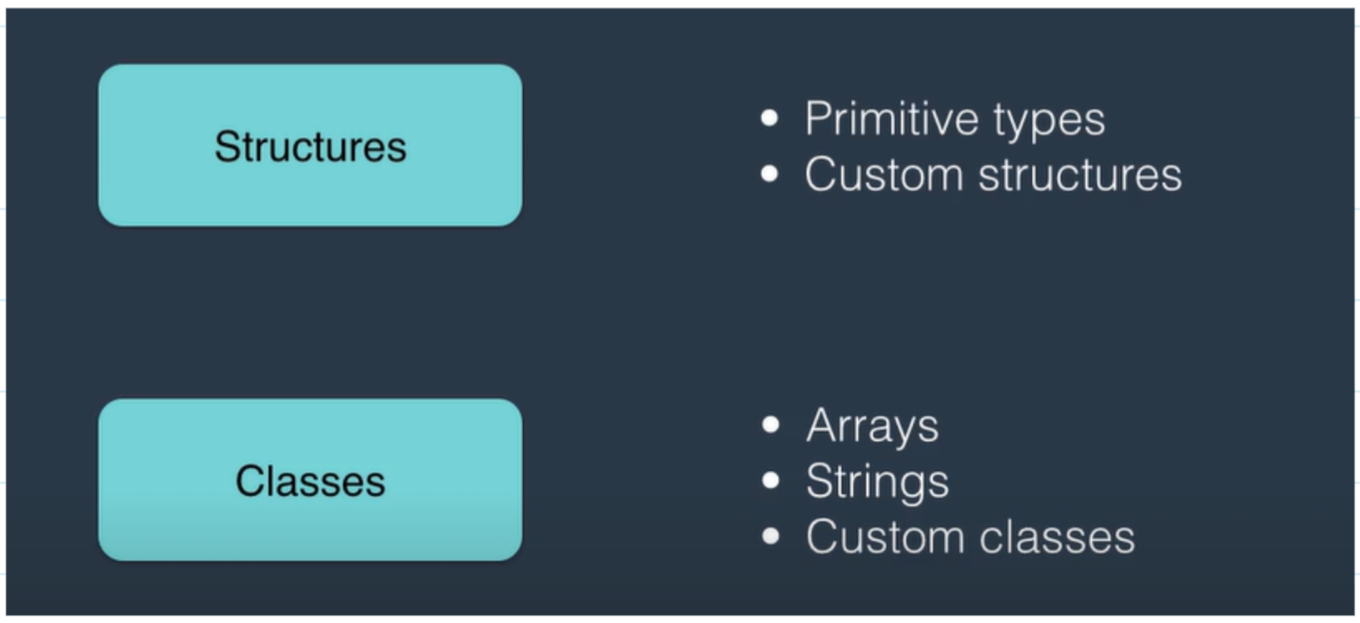
```
using System;

namespace CSharpFundamentals
{
  public enum ShippingMethod
  {
    RegularAirMail =1,
    RegisteredAirMail=2,
    Express=3
  }

  class Program
  {
    static void Main(string[] args)
    {
      var method = ShippingMethod.Express;
      Console.WriteLine((int)method);

      var methodId = 3;
      Console.WriteLine((ShippingMethod)methodId);

      // converting to an string
      Console.WriteLine(method.ToString());

      // converting a string to enum

      var methodName = "Express";
      var ShippingMethod1=Enum.Parse(typeof(ShippingMethod), methodName);
    }
  }
}
```

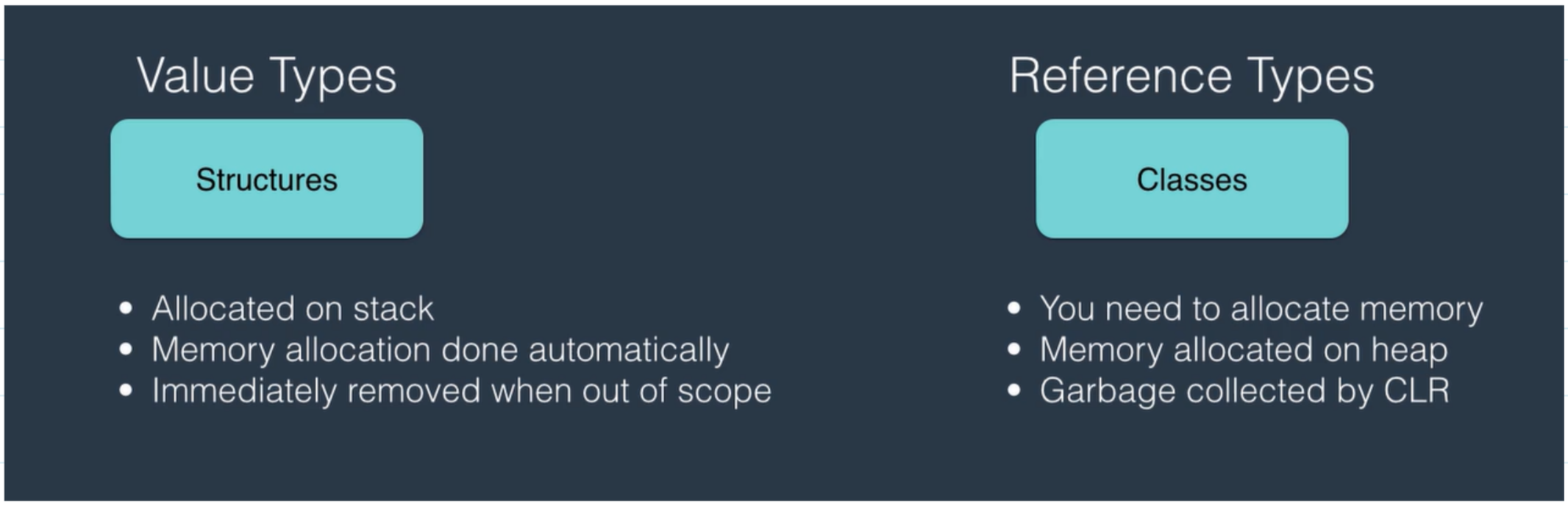
Reference Types and Value Types:-

Types [Primitive] & [Non Primitive]

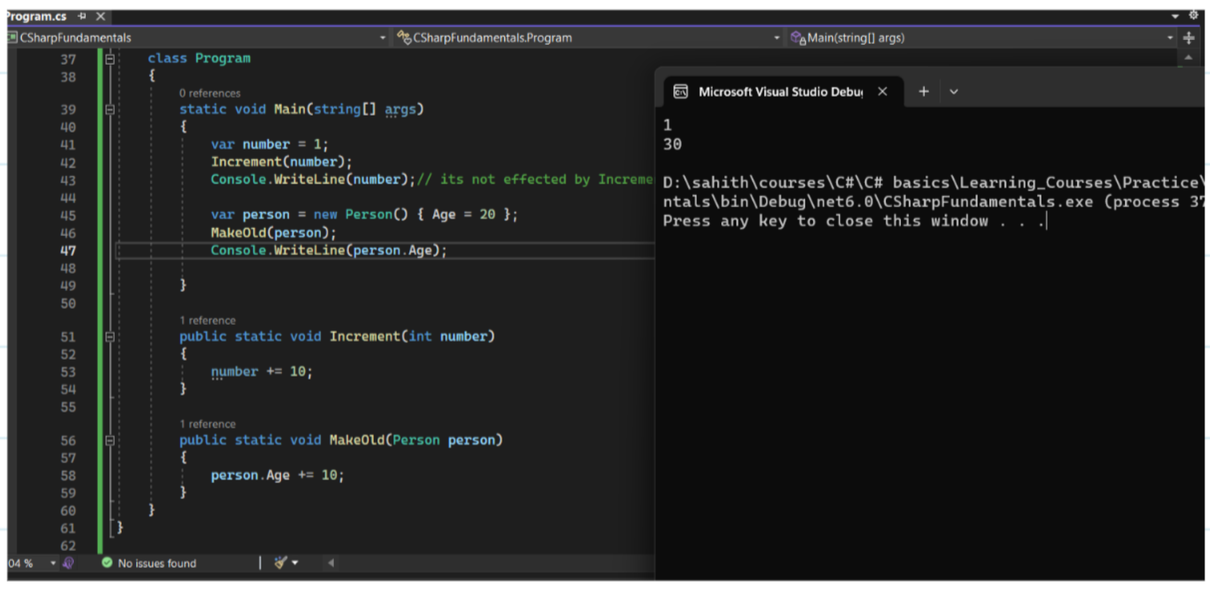
Primitive	Non Primitive
int	Classes
char	structures
float	arrays [System.Array]
bool.	strings [System.String]

In dot Net all the Primitive and Non-Primitive are classified into structures and classes.






```
using System;

//example 1
/*namespace CSharpFundamentals
{
   class Program
   {
      static void Main(string[] args)
      {
         var a = 10;
         var b = a;
         b++;
         Console.WriteLine(string.Format("a: {0},b:{1} ", a,b));
         //if we copy a value type to different variable, the copy
         //of that variable is copied and stored in an target variable
         //that is why we call them as "value types"

         var array1 = new int[3] { 1, 2, 3 };
         var array2 = array1;
         array2[0] = 0;
         Console.WriteLine(array2[0]);

      }
   }
}

*/
```




```
// example 2

namespace CSharpFundamentals
{
   public class Person
   {
      public int Age;
   }
   class Program
   {
      static void Main(string[] args)
      {
         var number = 1;
         Increment(number);
         Console.WriteLine(number);// its not effected by Increment method

         var person = new Person() { Age = 20 };
         MakeOld(person);
         Console.WriteLine(person.Age);

      }

      public static void Increment(int number)
      {
         number += 10;
      }

      public static void MakeOld(Person person)
      {
         person.Age += 10;
      }
   }
}
```

Summary Non-Primitive Types:-

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 `using System;` it's where the String class is defined.

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);
```

Control Flow :-

Conditional Statements :-

1. If / else statements
2. switch / case.

we also have Conditional operator: a ? b : c [shortcut for If/else]

If/else.

```
if (condition)
    someStatement

else if (anotherCondition)
    anotherStatement.
else
    yetAnotherStatement.
```

[If we have more statements we can use {}

```
if (condition)
{
    someStatements
}
else.
{
    otherStatements
}
```

Nested If statements -

```
if (condition)
{ if (another condition)
    ----
  else
    ----
}
```

Switch / case.

```
switch (role)
{
  case Role.Admin:
    ----
      break;
  case Role.Moderator:
      ----
      break;
  default:
      ---
      break;
}
```

```
/*
namespace Conditionals
{
   class Program
   {
      static void Main(string[] args)
      {
         int hour = 10;

         if (hour >0 && hour < 12)
         {
            Console.WriteLine("It's morning");
         }
         else if (hour >=12 && hour<18)
         {
            Console.WriteLine("It's afternoon");
         }
         else
         {
            Console.WriteLine("It's evening");
         }
      }
   }
}
*/


//namespace Conditionals
//{
//   class Program
//   {
//      static void Main(string[] args)
//      {
//         bool isGoldCstomer = true;

//         //float price;
//         //if(isGoldCstomer)
//         //{
//         //   price = 19.95f;
//         //}
//         //else                              // control +k+c --> to comment multiple lines
//         //{
//         //   price = 29.95f;

//         //}
//         // the above if else can be written as follows
//         float price = (isGoldCstomer) ? 19.95f : 29.95f;

//         Console.WriteLine(price);
//      }
//   }
//}


// switch case

namespace Conditionals
{

   class Program
   {
      static void Main(string[] args)
      {
         var season = Season.Autumn;

         switch(season)
         {
            case Season.Autumn:
               Console.WriteLine("Its autumn and a beautiful season.");
               break;

            case Season.Summer:
               Console.WriteLine("It's perfect time to go to beach");
               break;

            default: Console.WriteLine("I don't understand that season!");
               break;


      }

   }
```

1- Write a program and ask the user to enter a number. The number should be between 1 to 10. If the user enters a valid number, display "Valid" on the console. Otherwise, display "Invalid". (This logic is used a lot in applications where values entered into input boxes need to be validated.)

```
namespace NumberValidity
{
    class Program
    {
        static void Main(string[] args)
        {
            var Number = 9;

            if (Number >= 1 && Number <= 10)
            {
                Console.WriteLine("Valid");
            }
            else
                Console.WriteLine("Invalid");
        }

    }
}
```

2. Write a program which takes two numbers from the console and displays the maximum of the two.

```
namespace Maximum
{
    class Program
    {
        static void Main(string[] args)
        {
            var number1 = 10;
            var number2 = 20;

            if (number1 > number2)
            {
                Console.WriteLine("Number1 is maximum");
            }
            else if (number1 < number2)
            {
                Console.WriteLine("Number2 is maximum");
            }
        }
    }
}
```

Iteration statements :-

1. for loop.
2. Foreach loop
3. while loops
4. Do-while loops.

① For Loop :-

```
for (var i=0; i<10; i++)
{
  ---
}
```

② Foreach loop

```
foreach (var number in numbers)
{
  ---
}
```

③ While Loop

```
while (i<10)
{
  ---
  i++;
}
```

④ Do-while loops

```
do
{
  ---
  i++;
} while (i<10);
```

Break and Continue

- `Break`: jumps out of the loop
- `continue`: jumps to the next iteration

for loop

```
namespace Iterations
{
    class Program
    {
        static void Main(string[] args)
        {
            for (var i=1;i<=10; i++)
            {
                if (i%2==0)
                {
                    Console.WriteLine(i);
                }
            }

            for (var i=10; i>1; i--)
            {
                if(i%2==0)
                {
                    Console.WriteLine(i);
                }
            }
        }
    }
}
```

<foreach Loop> :-

```
using System;

namespace Iterations
{
    class Program
    {
        static void Main(string[] args)
        {
            var name = "John Smith";

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

            foreach (var character in name)
            {
                Console.WriteLine(character);
            }
        }
    }
}
```

while loop: -

```
using System;

namespace Iterations
{
    class Program
    {
        static void Main(string[] args)
        {
            //for (var i = 1; i <= 10; i++)
            //{
            //    if (i% 2 == 0)
            //        Console.WriteLine(i);
            //}

            var i = 0;
            while (i <= 10)
            {
                if (i % 2 == 0)
                    Console.WriteLine(i);

                i++;
            }
        }
    }
}
```

```
using System;

namespace Iterations
{
    class Program
    {
        static void Main(string[] args)
        {
            while (true)
            {
                Console.Write("Type your name: ");
                var input = Console.ReadLine();

                if (String.IsNullOrWhiteSpace(input))
                    break;

                Console.WriteLine("@Echo: " + input);
            }
        }
    }
}
```

Break & Continue.

```
using System;

namespace Iterations
{
    class Program
    {
        static void Main(string[] args)
        {
            while (true)
            {
                Console.Write("Type your name: ");
                var input = Console.ReadLine();

                if (!String.IsNullOrWhiteSpace(input))
                {
                    Console.WriteLine("@Echo: " + input);
                    continue;
                }

                break;
            }
        }
    }
}
```

Random Class.

```
using System;

namespace CSharpFundamentals
{
    class Program
    {
        static void Main(string[] args)
        {
            var random = new Random();
            for (var i = 0; i < 10; i++)
                Console.WriteLine(random.Next());
        }
    }
}
```

op.

```
531361047
1474176480
1427000922
918495352
1501143261
2092252804
943784030
815330425
1690889627
1635677101
Press any key to continue . .
```

```
using System;

namespace CSharpFundamentals
{
    class Program
    {
        static void Main(string[] args)
        {
            var random = new Random();

            var buffer = new char[10];
            for (var i = 0; i < 10; i++)
                buffer[i] = (char)('a' + random.Next(0, 26));

            var password = new string(buffer);

            Console.WriteLine(password);
        }
    }
}
```

Arrays :-

① Single dimensional array
② multi- dimensional array ⟶ rectangular array
⟶ Jagged array.

Multi Dimension Arrays

Rectangular

3x5

0	0	1	2	3	4
	0	1	2	3	4
	0	1	2	3	4

Jagged

0	0	1	2	3	
	0	1	2	3	4
	0	1	2		

Syntax:-

Syntax (Rectangular 2D)

```
var matrix = new int[3, 5];
var matrix = new int[3, 5]
{
    { 1, 2, 3, 4, 5 },
    { 6, 7, 8, 9, 10 },
    { 11, 12, 13, 14, 15 }
};
```

* to access the elements

Ex:- var element = matrix[0,0];

Syntax for [Rectangular 3D]

var cdouss = new int[3,5,4]

Syntax for Jagged:-

Jagged

0	1	2	3	
0	1	2	3	4
0	1	2		

```
var array = new int[3][];
```

array[0] = new int[4];
array[1] = new int[5];
array[2] = new int[3];

to access :-

array[0][0] = 1;

The difference between Rectangular array and Jagged array. is :-

Jagged	`var array = new int[3][];`
Rectangular	`var array = new int[3, 5];`

```
namespace CSharpFundamentals
{
   class Program
   {
      static void Main(string[] args)
      {
         var numbers = new[] { 3, 7, 9, 2, 14, 6 };
         //length property
         Console.WriteLine("Length: "+ numbers.Length);

         // Index() used to find the position the the element in
the array

         var index = Array.IndexOf(numbers, 9);
         Console.WriteLine("Index of 9: " +  index);

         // Clear()
         Array.Clear(numbers, 0, 2);
         Console.WriteLine("Effect of Clear()");
         foreach (var n in numbers)
         {
            Console.WriteLine(n);
         }

         // Copy()
         int[] another = new int[3];
         Array.Copy(numbers, another, 3);
         Console.WriteLine("Effect of Copy()");
         foreach(var n in another)
         { Console.WriteLine(n); }

         // Sort()
         Array.Sort(numbers);
         Console.WriteLine("Effect of Sort()");
         foreach (var n in numbers)
               Console.WriteLine(n);

         // Reverse()
         Array.Reverse(numbers);
         Console.WriteLine("Effect of Reverse()");
         foreach(var n in numbers)
         {
            Console.WriteLine(n);
         }
      }
   }
}
```

```
Length: 6
Index of 9: 2
Effect of Clear()
0
0
9
2
14
6
Effect of Copy()
0
0
9
Effect of Sort()
0
0
2
6
9
14
Effect of Reverse()
14
9
6
2
0
0
```

Arrays vs Lists

- Array: fixed size
- List: dynamic size

Creating a List
=> var numbers = new List<int>();

```
var numbers = new List<int>();
var numbers = new List<int>() { 1, 2, 3, 4 };
```

Useful Methods in Lists :-

- Add()
- AddRange()
- Remove()
- RemoveAt()
- IndexOf()
- Contains()
- Count

```
using System.Collections.Generic;

namespace CSharpFundaments
{
   class Program
   {
      static void Main(string[] args)
      {
         var numbers = new List<int>() { 1,2,3,4};
         numbers.Add(1);// in Arrays we don't have add
                        // method but in Lists we can do it

         numbers.AddRange(new int[3] { 5, 6, 7 });

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

         Console.WriteLine();
         Console.WriteLine("Index of 1: "+ numbers.IndexOf(1));
         Console.WriteLine("Last Index of 1: " + numbers.LastIndexOf(1));

         Console.WriteLine("Count:"+ numbers.Count);

         numbers.Remove(1);

         for (var i=0; i < numbers.Count; i++)
            {
            if (numbers[i] == 1)
            {
               numbers.Remove(numbers[i]);
            }
         }

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

         numbers.Clear();
         Console.WriteLine("Count:" + numbers.Count);
      }
   }
}
```

```
Microsoft Visual Studio Debu
1
2
3
4
1
5
6
7

Index of 1: 0
Last Index of 1: 4
Count:8
2
3
4
5
6
7
Count:0

D:\sahith\courses\C#\C# basics\Learning_Courses\Practice\Arrays and lists-
in\Debug\net6.0\CSharpFundamentals.exe (process 36912) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Opt
le when debugging stops.
Press any key to close this window . . .
```

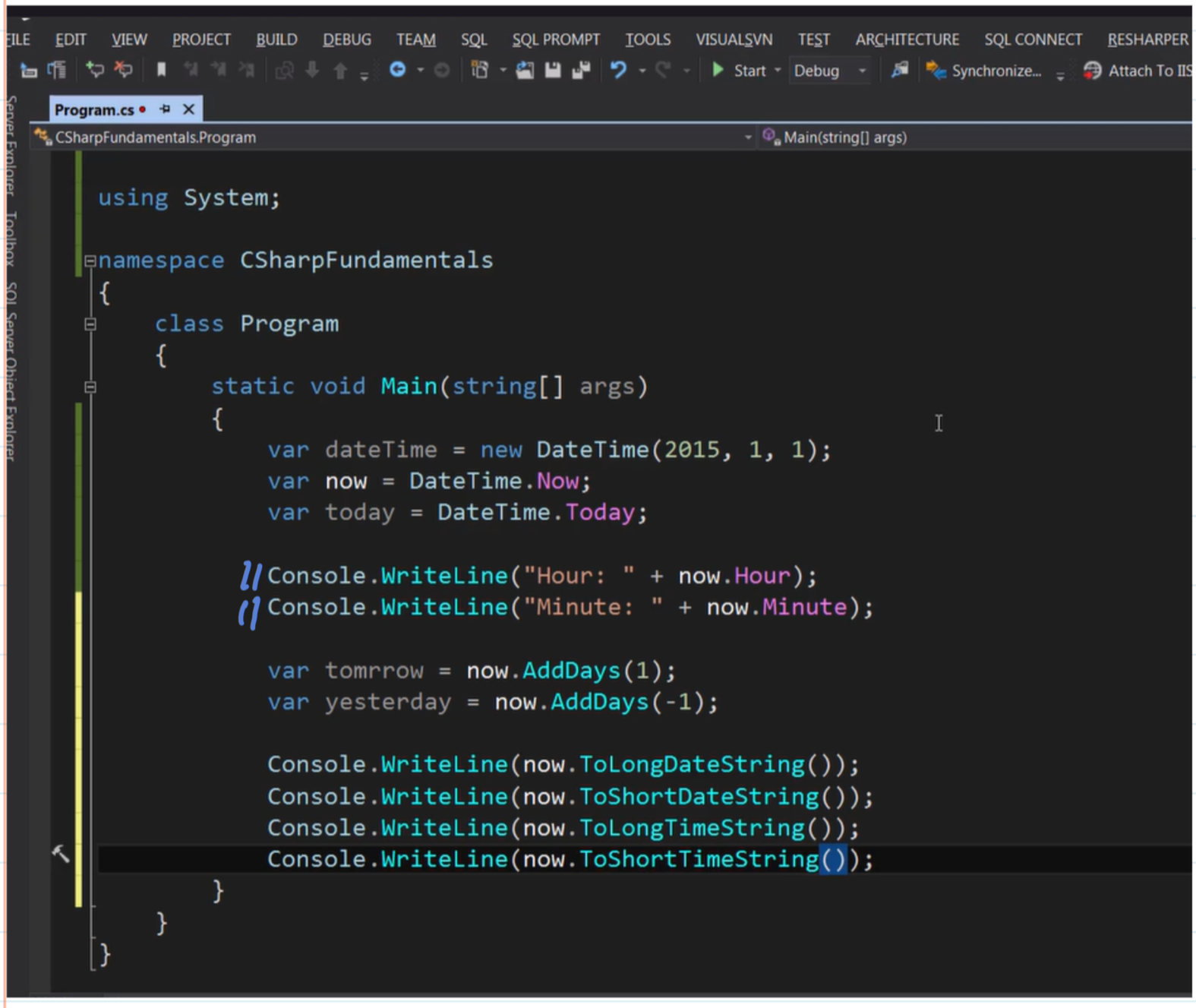
DateTime :-

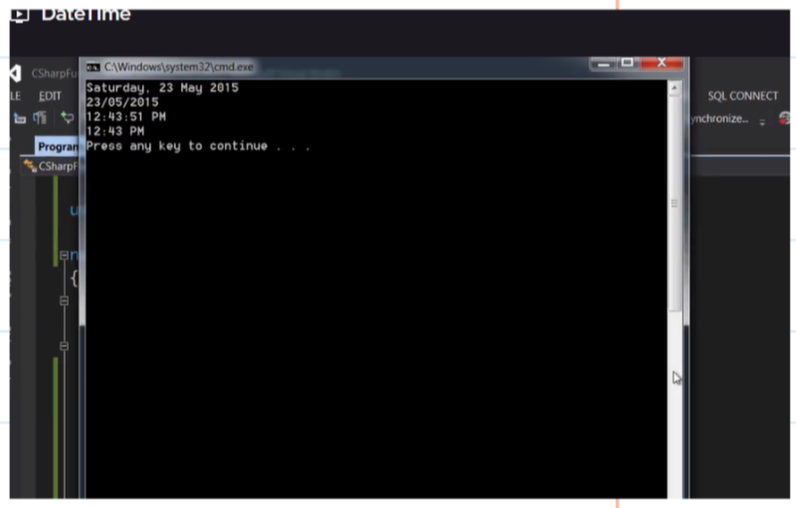
DateTime

CSharpFundamentals (Debug|Any CPU) - Microsoft Visual Studio

```
using System;

namespace CSharpFundamentals
{
    class Program
    {
        static void Main(string[] args)
        {
            var dateTime = new DateTime(2015, 1, 1);
            var now = DateTime.Now;
            var today = DateTime.Today;

            Console.WriteLine("Hour: " + now.Hour);
            Console.WriteLine("Minute: " + now.Minute);
        }
    }
}
```

```
C:\Windows\system32\cmd.exe
Hour: 12
Minute: 40
Press any key to continue . . .
```





Strings → There are immutable, It is an Class.

Formatting

```
ToLower()        // "hello world"
ToUpper()        // "HELLO WORLD"
Trim()
```

Searching

```
IndexOf('a')
LastIndexOf("Hello")
```

Substrings

```
Substring(startIndex)
Substring(startIndex, length)
```

Replacing

```
Replace('a', '!')
Replace("mosh", "moshfegh")
```

Null checking

```
String.IsNullOrEmpty(str)
String.IsNullOrWhiteSpace(str)
```

Splitting

```
str.Split(' ')
```

Converting Strings to Numbers

```
string s = "1234";
int i = int.Parse(s);
int j = Convert.ToInt32(s);
```

Converting Numbers to Strings

```
int i = 1234;
string s = i.ToString();          // "1234"
string t = i.ToString("C");       // "$1,234.00"
string t = i.ToString("C0");      // "$1,234"
```

Format Strings

Format Specifier	Description	Example
c or **C**	Currency	123456 (C) -> $123,456
d or **D**	Decimal	1234 (D6) -> 001234
e or **E**	Exponential	1052.0329112756 (E) -> 1.052033E+003
f or **F**	Fixed Point	1234.567 (F1) -> 1234.5
x or **X**	Hexadecimal	255 (X) -> FF

String Builder :-

StringBuilder

- Defined in System.Text
- A mutable string
- Easy and fast to create and manipulate strings

Not for Searching

- IndexOf()
- LastIndexOf()
- Contains()
- StartsWith()
- …

String Manipulation Methods

- Append()
- Insert()
- Remove()
- Replace()
- Clear()

Working with files and directories :-

* In dotNet we have an namespace system.IO.
  some of the classes in it * File, FileInfo.
  * Directory, DirectoryInfo.
  * Path

File, FileInfo.

* Provide methods for creating, copying, deleting, moving, and opening of files.

FileInfo :- provides instance methods.
File :- provides static methods.

File, FileInfo

- Create()
- Copy()
- Delete()
- Exists()
- GetAttributes()
- Move()
- ReadAllText()

Directory, DirectoryInfo -

Directory = provides static methods.
DirectoryInfo :- provides instance methods.

Directory, DirectoryInfo

- CreateDirectory()
- Delete()
- Exists()
- GetCurrentDirectory()
- GetFiles()
- Move()
- GetLogicalDrives()

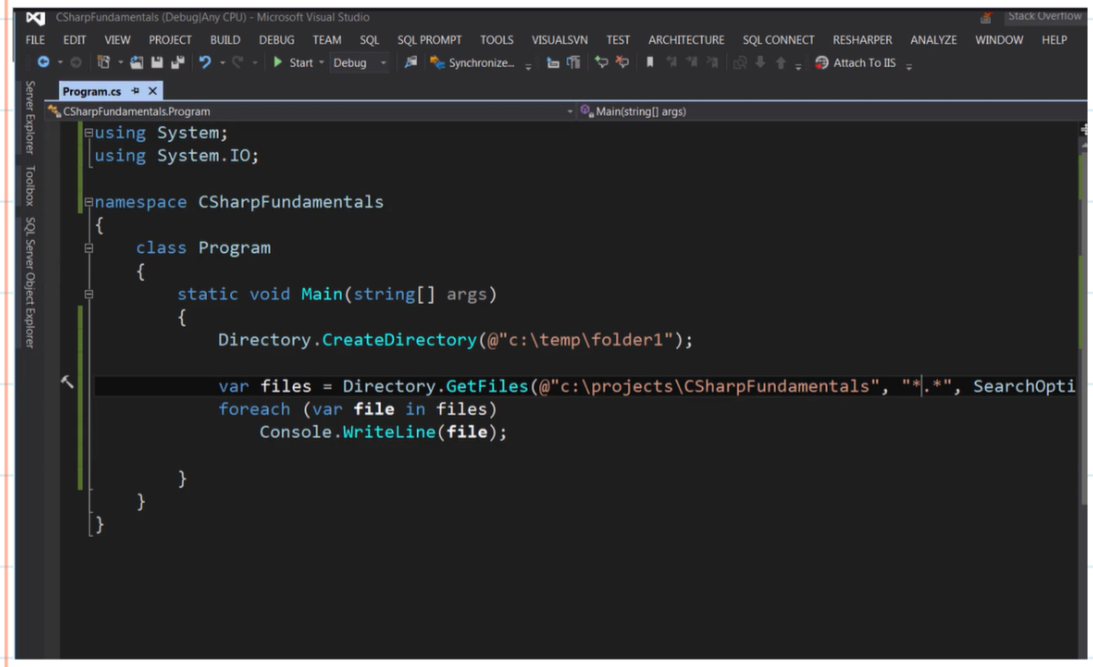
Paths :-

- GetDirectoryName()
- GetFileName()
- GetExtension()
- GetTempPath()

Program for File & FileInfo Classes.

```
namespace CSharpFundamentals
{
   class Program
   {
      static void Main(string[] args)
      {
         var path = @"c:\somefile.jpg";
         File.Copy(@"c:\temp\myfile.jpg", @"d:\temp\myfile.jpg", true);

         //Delete method
         File.Delete(path);

         if(File.Exists(path) )
         {
            //
         }
         var content = File.ReadAllBytes(path);

         var fileInfo = new FileInfo(path);

         fileInfo.CopyTo("...");

         fileInfo.Delete();

         if(fileInfo.Exists )
         {
            //....
         }


      }
   }
}
```

Program directory & directory Info :-



program Path :-

```
using System;
using System.IO;

namespace CSharpFundamentals
{
    class Program
    {
        static void Main(string[] args)
        {
            var path = @"C:\Projects\CSharpFundamentals\HelloWorld\HelloWorld.sln";

            var dotIndex = path.IndexOf('.');
            var extension = path.Substring(dotIndex);

            Console.WriteLine("Extension: " + Path.GetExtension(path));
            Console.WriteLine("File Name: " + Path.GetFileName(path));
            Console.WriteLine("File Name without Extension: " + Path.GetFileNameWithoutExtensio
            Console.WriteLine("Directory Name: " + Path.GetDirectoryName(path));



        }
    }
}
```

Debug :-- F9 → to set break point.
F10 → Step over
F11 → Step Into.
Shift + F11 → to come out of the Loop.
Debug → windows → watch → to watch in the Console.

F5 ⇒ to Run in debug mode
Shift + F5 ⇒ stop the debug mode.