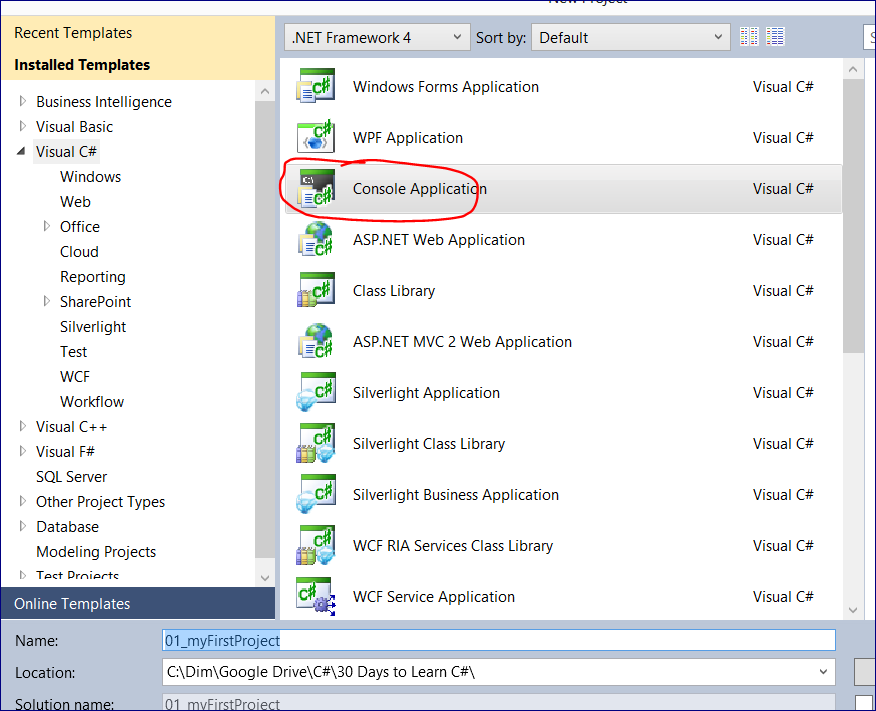
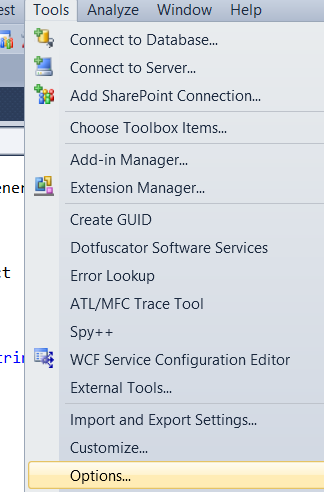
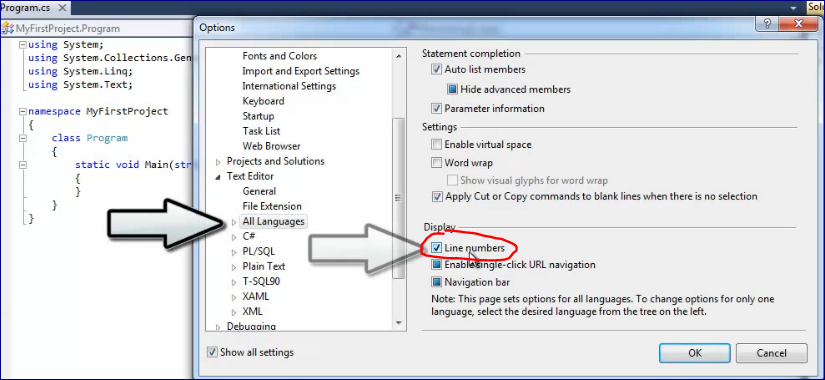
**30 Days to Learn C# -- Notes**

## 30 Days to Learn C# - Lesson 1 (Setting up Your Environment)



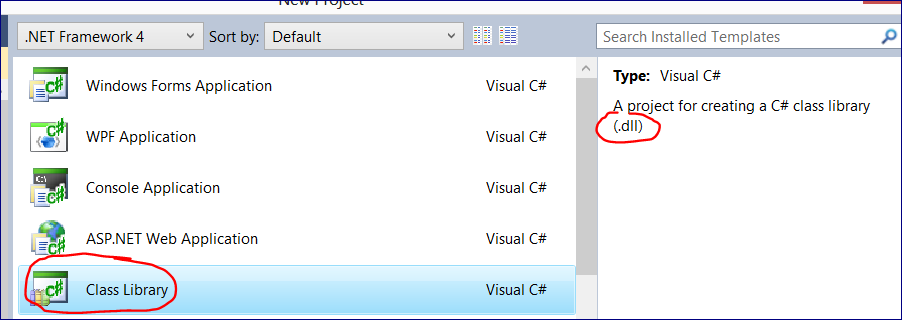
Show the code lines: Tools 🡪 Options 🡪





**One more option is to create a Class Library. If you use the same code over and over again.**

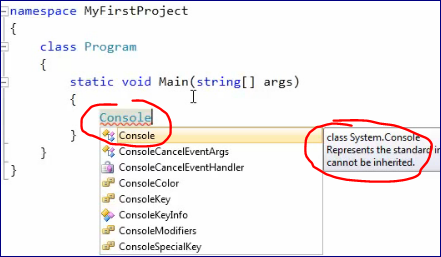
**That will be compiled into .dll file. You can use the same code in Visual Basic as well!!!**



## 30 Days to Learn C# - Lesson 2 (Hello, World! in C#)

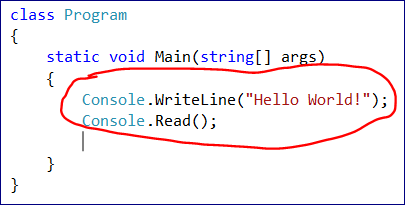
**C# and the .Net framework is completely object oriented.**

It has a **class (static) called “Console”**, so we can use this class to print what we want to.



**Object has:**

* Properties (describes the object)
* And methods (actions -- do something with the object)

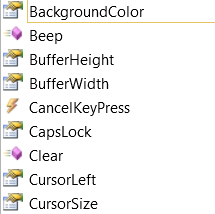


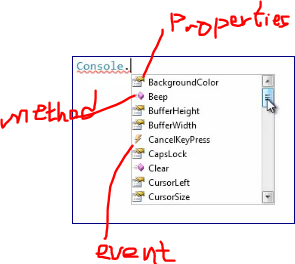
**“Console” is a class, so we can see all properties and methods that are available for that class through intellisense.**

Console.WriteLine("Hello World!");

Console.Read();

We send “Hello World!” statement to the WriteLine method.





## 30 Days to Learn C# - Lesson 3 (Variables and Primitive Data Types)

**Variables and Primitive Data Types**

There are **Static and Dynamic** programming languages.

In **Dynamic** PL you can create a variable and assign to it a number and then a string. So, the variable type will be adjusted accordingly.

In **Static** languages you declare the variable type first.

After that you cannot assign to that variable anything that is of another data type.

Console.BackgroundColor = ConsoleColor.DarkBlue;

Console.Clear();

Console.WriteLine("Hello World!");

Console.Title = "Wohoo";

//**Primitive data types (int, float, double, decimal, char, bool)**

//declare and initialize a variable

int foo = 10;

int bar; // declare a variable

bar = 17; // initialize with a value

// declare several variable of THE SAME type using a comma ','

int x = 12

, y = 14

, hello = foo

, world = bar;

// use "f" to explicitly convert double to float -- Inaccurate type

float x1 = 3.14f;

double x2 = 3.14;

// use "m" to explicitly convert double to decimal -- Accurate type

// we can use "decimal" data type for currency as it's very accurate.

decimal x3 = 3.14m;

char x4 = 's'; // Must contain just ONE character

bool x5 = false; // true or false

int x6 = 3;

Console.WriteLine(foo.ToString() + ' ' + bar.ToString());

Console.ReadLine();

## 30 Days to Learn C# - Lesson 4 (Numbers, Arithmetic, Comments, and Value Types)

Commenting



Uncommenting



## 30 Days to Learn C# - Lesson 5 (Strings)

**Strings**

Use **double qoutes(“”)** for strings (“abcd”) and single quotes (‘’) for characters (‘a’)

**// to explicitly say it's an empty string instead of ""**

string myString5 = **string.Empty**;

//---------------------------------------------------------------------------------

// 30 Days to Learn C# - Lesson 5

**// Strings methods**

**// \ -- Escape character**

string myString1 = "I asked, \"Isn't this a string?\"";

Console.WriteLine(myString1);

**// \t -- Tabulation**

string myString2 = "Using tabulation \t in the text";

Console.WriteLine(myString2);

**// \n -- New line**

string myString3 = "Using new line \n as a separate line";

Console.WriteLine(myString3);

**// @ -- Verbatum string --> Display everyting that's in this string**

string myString4 = @"Using @ sign to make it a verbatum string";

Console.WriteLine(myString4);

**// to explicitly say it's an EMPTY STRING instead of ""**

string myString5 = string.Empty;

**// Concatenate strings with "+" operator**

string myString6 = "This is a " + "string" + "!" + 7 + true;

Console.WriteLine(myString6);

**// we split one line of code into the multiple lines**

string myString7 = "This is a "

+ "string" // add a string

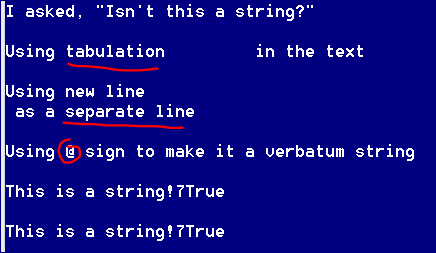
+ "!" // a character will be converted to a string

+ 7 // a number will be converted to a string

+ true; // a boolen value will be converted to a string

Console.WriteLine(myString7);

Console.ReadLine();



## 30 Days to Learn C# - Lesson 6 (Simple String Manipulation)

**String methods**

Strings are NOT value types. A string is a REFERENCE type.

**You cannot modify (mutate) a string but you can reference the modified value.**

string foo = "Hello, world!";

int len = foo.**Length**;

int inx1 = foo.**IndexOf**("e");

int inx2 = foo.**LastIndexOf**("l");

Console.WriteLine(foo);

Console.WriteLine(len);

Console.WriteLine("index of letter 'e' = " + inx1.ToString());

Console.WriteLine("index of the LAST letter 'l' = " + inx2.ToString());

// starts from index 3 with the len =5

string part1 = foo.**Substring**(3, 5);

Console.WriteLine("Substring part1 = " + part1);

// remove 7 characters staring from 0

// doesn't change "foo"

string part2 = foo.**Remove**(0, 7);

Console.WriteLine("Substring part2 = " + part2);

// we can do the same with "Replace" method

string bar = foo.**Replace**("Hello, ", string.Empty);

Console.WriteLine(bar);

// replace "l" with "123"

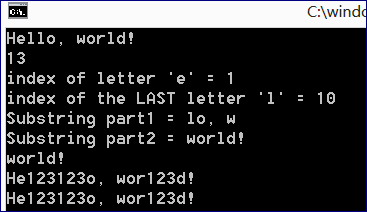
string part3 = foo.Replace("l", "123");

Console.WriteLine(part3);

// If we want to change th actual value of foo

foo = foo.Replace("l", "123");

Console.WriteLine(foo);



## 30 Days to Learn C# - Lesson 7 (Creating Objects)

**Create an object of the class (StringBuilder)**

In general we can concatenate strings with a “+” operator.

However, if we have many pieces that we want to combine in one string the “+” methodology becomes less efficient. We can use a **StringBuilder** to append strings as it's more efficient.

We want to create an object and assign it to a variable in order to reference it any time we want to.

int x = 5;

It looks similar to how we defined a type for a variable.

StringBuilder sb = new StringBuilder()

**To create objects in C#:**

* We define a Data Type (StringBuilder)
* and a variable name (sb)
* Then we want to say that we want to create a NEW StringBuilder object. In order to do that we use new command
* And then we call a "constructor" StringBuilder() (that is like a special method) of the StringBuilder class (we do not have an object already to call a constructor. That's why it's somewhat different from a regular method)
* Also the constructor name has the same name as the DataType.

StringBuilder sb = new StringBuilder();

DateTime dt = new DateTime(); // just another example

StringBuilder x = new StringBuilder()

,y = x; // Contains a reference to the object

// two variable (x and y) are pointing to the same object

// any changes in y we will see in x and vice versa

x.Append("Hello, ");

y.Append("world!");

Console.WriteLine(x.ToString());

// assign NULL -- can be used with reference types (not value types)

// create a nullable datatype using "?". Useful for databases.

x = null;

int? foo = null;

// another example: if we put "int myInt = 5; myInt = null;" it wouldn't work

// creating a NULLABLE INT

int? myInt = 5;

myInt = null;

// it will be converted to an empty string. Prints a blank line

Console.WriteLine(x); Console.WriteLine(y.ToString());

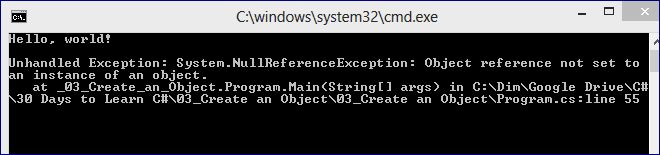
Console.WriteLine("foo = " + foo);

------------------------------------

x = null;

Console.WriteLine(**x.ToString()**);

Console.WriteLine(y.ToString());



We get this error because we try to use the ToString method with the object that is NULL.

To get rid of this problem do not use ToString() method:

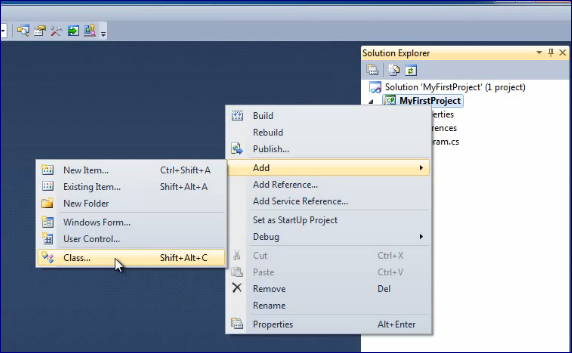
Console.WriteLine(**x**);

------------------------------------

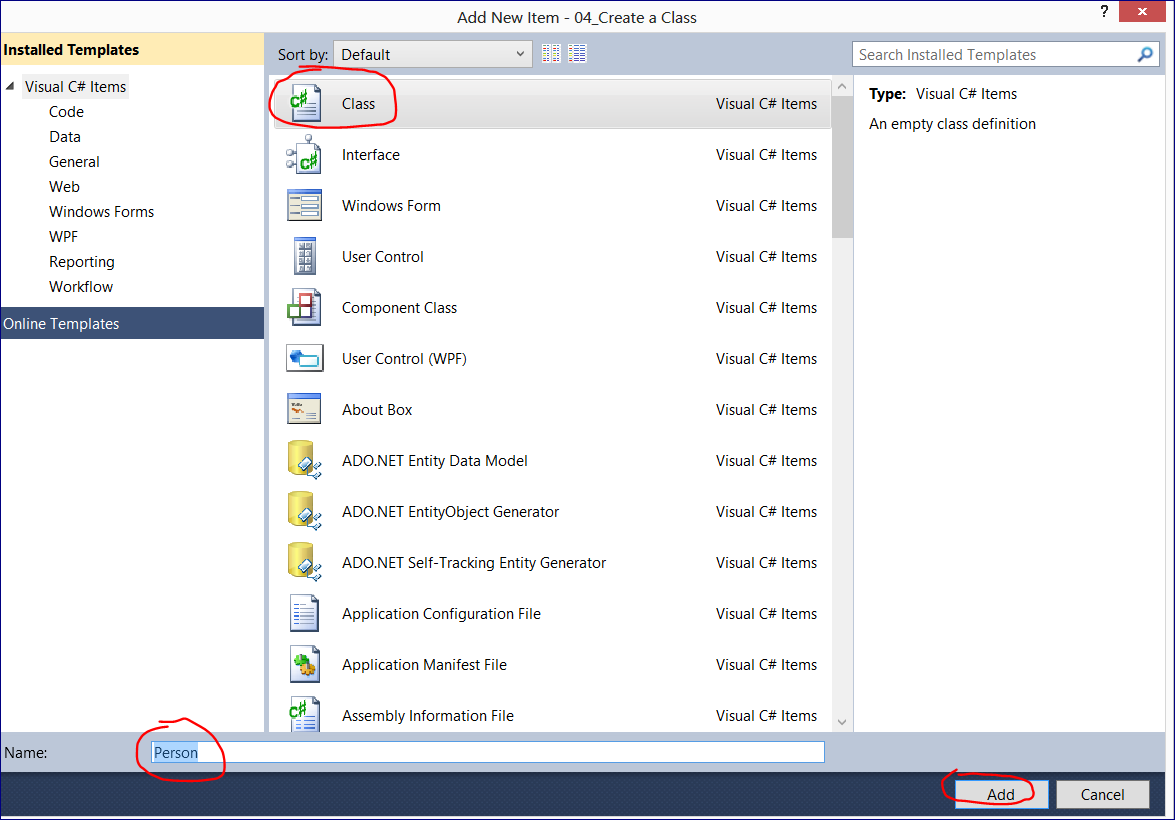
## 30 Days to Learn C# - Lesson 8 (Namespaces, using Statements, and Writing Your First Class)

**Create a Class**

Right-click on the Project name



Select a CLASS, name it, and press “Add”



**Create a Class:**

1. Define a class with a "class" keyword.
2. Put the name of the class then. By convention the name of the class should start with a Capital letter
3. Use Pascal case to name a class, e.g. "class **PersonOfInterest**"
4. Add curly braces inside of the class.
5. Everything that goes inside consider as a part of the Person class (methods, properties, etc.)

**class** Person

{

Put properties and methods here

}

**Define a namespce:**

* Whenever you define a class you want to put it into the **namespace** (a naming container).
* It protects your class from naming collisions with another class
* When we have a "namespace" there is no confusion which "class Person" we want to use (as we may have many Person classes). It works as First name and Last Name

Typically you use the name of your project.

Add curly braces to include the class Person

So, the class Person and the class Program are both in the **MyFirstProject** namespace

**namespace** MyFirstProject

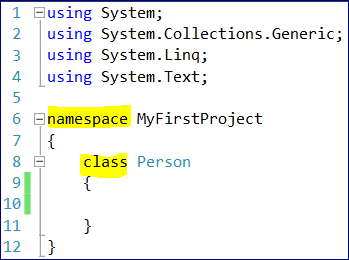
{

class Person

{

}

}



**Levels of namespaces**

To better organize the code we can add “LEVELS” to the namespace of the class.

e.g.:

namespace **AnotherNameSpace.MyFirstDataTypes**

{

class Person

{

}

}

**In order to use it in our main program we have two options:**

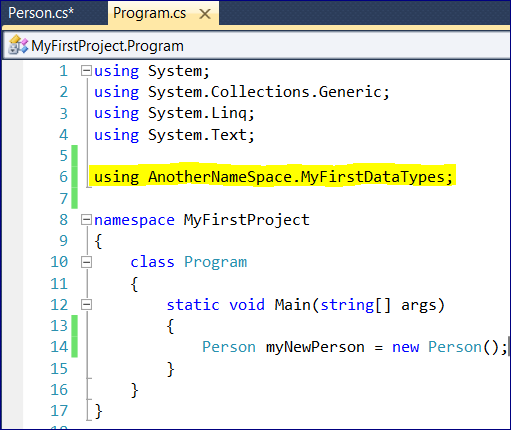
**1. Specify the full path**

AnotherNameSpace.MyFirstDataTypes.Person myNewPerson = new AnotherNameSpace.MyFirstDataTypes.Person();

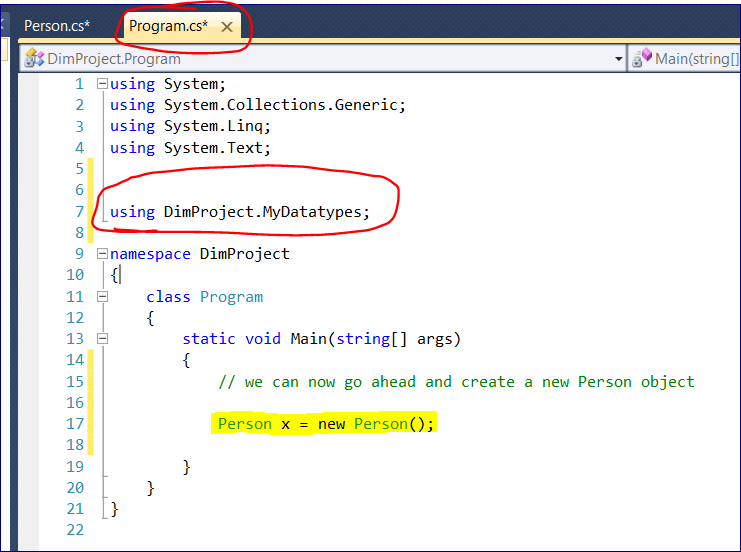
**2. Import the name space**

using AnotherNameSpace.MyFirstDataTypes;

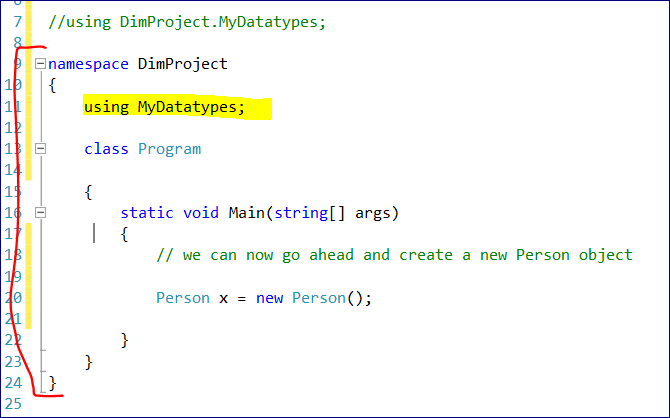
Person myNewPerson = new Person();



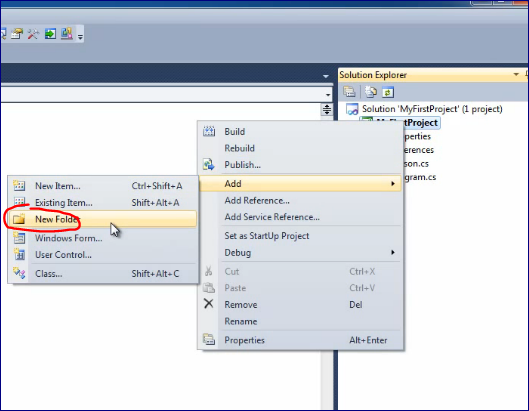
But in this case we need to import this module where we use it.



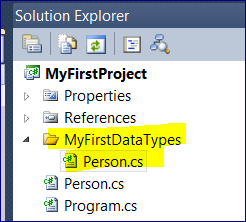
Or we can put this Import inside of the Program class.



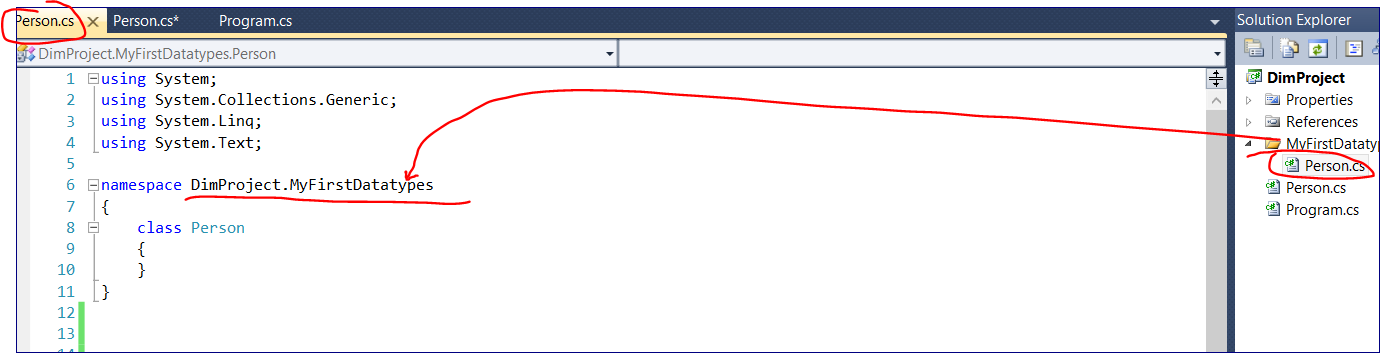
**Let’s add a new folder**



Add a new class Person to the folder



If we create a folder and create a Person class inside the naming will be implemented automatically.



## 30 Days to Learn C# - Lesson 9 (Constructors, Fields, and Methods)

The class “Person” was created

We want our person to have First and Last name whenever the Person Object is created.

If we have an empty class C# will allow us to create an empty object calling a default constructor but it doesn’t help us much as the object will not have properties and methods.

class Person

{

}

**1. We need to write a constructor.**

The constructor name has the same name as the class.

This is the empty constructor.

class Person

{

Person()

{

}

}

We can provide parameters that the constructor will require while creating an object.

class Person

{

Person(string firstname, string lastname)

{

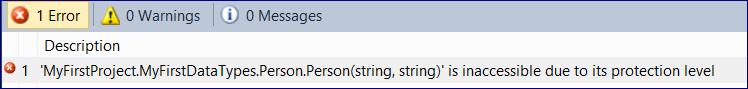
}

}

There is an accessibility issue.

We need to determine what can and cannot be accessed.

Right now our person constructor is not accessible outside of the Person class because we have not defined the accessibility.



Anything that is defined inside of the namespace (a class or any other type of data) is automatically defined as internal

We can specify it explicitly though

internal class Person

To make the class accessible from the other classes we have to make it public.

Any code within this project can now access this class.

So, we can use the Person class anywhere, anytime.

And we need to do the same thing for the constructor.

Because



namespace MyFirstProject.MyFirstDataTypes

{

**public** class Person

{

**public** Person(string firstname, string lastname) // this is a **CONSTRUCTOR**

{

}

}

}

Now we are getting another error:





As we already defined that the constructor requires two arguments.

It works OK if we pass two strings:

Person myNewPerson = new Person("John","Doe");

**2. Constructor Fields**

The data was passed from the parameters to the constructor.

We need to save it somewhere in the person object.

We need so called “FIELDS”

Fields can be used throughout the class code.

private string \_firstname;

we can assign a default value

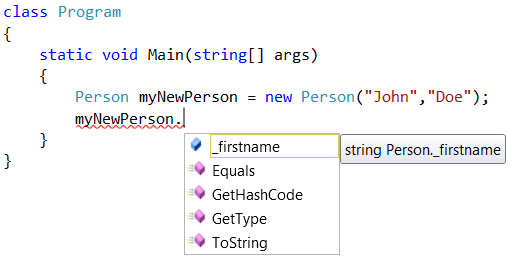
private string \_firstname = string.Empty;

Because they are private they are not accessible outside of the Person class.

If we assign “firstname” to be public

public string \_firstname;

we can access this property through the object from the main program.



**But we don’t want this to happen. We want to restrict access to the properties and use just methods of the class to change them.**

namespace MyFirstProject.MyFirstDataTypes

{

public class Person

{

**private** string \_firstname;

**private** string \_lastname;

public Person(string firstName, string lastName)

{

}

}

}

**3. Assign values to the fields inside of the constructor**

We need to use **this** keyword 🡪 it refers to the object

namespace MyFirstProject.MyFirstDataTypes

{

public class Person

{

private string \_firstname;

private string \_lastname;

public Person(string firstName, string lastName)

{

**this**.\_firstname = firstName;

**this**.\_lastname = lastName;

}

}

}

We now have a constructor getting the firstName and lastName and assigning them to the PRIVATE FIELDS \_firstname and \_lastname.

These fields are unique for each person object that we create.

**3. Create a method**

We want our object to say “Hello!”

Two rules for the method creation:

1. The name has to be different from the class name (as it’s not a constructor)
2. It has a return type

public string SayHello(string name)

{

return "Hello, " + name;

}

**This method**

* is public
* returns a string (I we do not want method to return anything we put void)
* requires a string parameter (name)

namespace MyFirstProject.MyFirstDataTypes

{

public class Person

{

private string \_firstname;

private string \_lastname;

public Person(string firstName, string lastName)

{

this.\_firstname = firstName;

this.\_lastname = lastName;

}

public string SayHello(string name)

{

return "Hello, " + name;

}

}

}

**Now we can create a Person object and call the SayHello method from the main program.**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using MyFirstProject.MyFirstDataTypes;

namespace MyFirstProject

{

**class Program**

{

static void Main(string[] args)

{

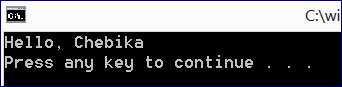
**Person** myNewPerson = new **Person**("John","Doe");

Console.WriteLine(myNewPerson.**SayHello**("Chebika"));

}

}

}



## 30 Days to Learn C# - Lesson 10 (GET and SET methods)

We need some way to make the information about properties available.

In order to access a property we can make it “PUBLIC”.

So, instead of: private string \_firstname;

we write: public string \_firstname;

But 

Because it gives a full control over the properties.

But we need a way to access these PRIVATE values outside of the class.

**// Writing "GET" and "SET" methods (accessors)**

// Use Pascal Case to name methods and classes

// Use Camel Case to name variables

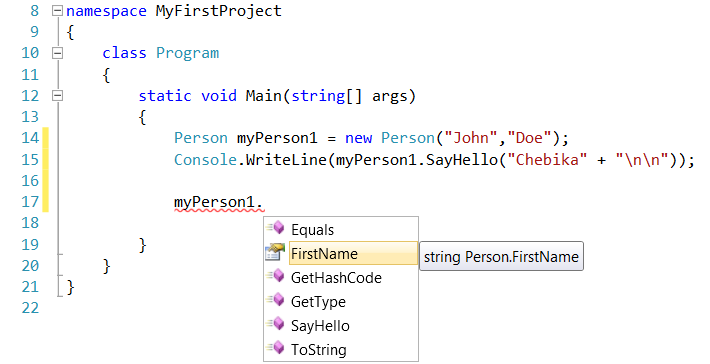
public string FirstName

{

get { return \_firstname; }

}

Now in the main program we can use “FirstName” method available for the Person Class.



We also have a SET method.

We can implement ANY logic in the Set method.

For example, we can change the LAST name while assigning a value to the FIRST NAME.

public string FirstName

{

get { return \_firstname; }

set { \_lastname = value; }

}

public string LastName

{

get { return \_lastname; }

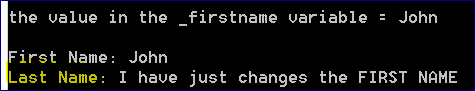
}

--======================

myPerson1.FirstName = "I have just changes the FIRST NAME";

Console.WriteLine("First Name: " + myPerson1.FirstName);

Console.WriteLine("Last Name: " + myPerson1.LastName);



--======================

These are the GET and SET methods for “\_firstname” and “\_lastname”

public class Person

{

private string \_firstname;

private string \_lastname;

public Person(string firstName, string lastName)

{

this.\_firstname = firstName;

this.\_lastname = lastName;

}

// Writing "GET" and "SET" methods

public string FtNm

{

**get** { return \_firstname; }

**set** { \_firstname = value; }

}

public string LtNm

{

**get** { return \_lastname; }

}

**We can use a simplified syntax for the GET and SET methods:**

public class Person

{

public string FtNameProperty { get; private set; }

public string LtNameProperty { get; private set; }

public Person(string firstName, string lastName)

{

FtNameProperty = firstName;

LtNameProperty = lastName;

}

--============================================================================

**SIMPLIFIED SYNTAX**

namespace MyFirstProject.MyFirstDataTypes

{

public class Person

{

// using simplified syntax for GET and SET methods

public string FtNameProperty { get; private set; }

public string LtNameProperty { get; private set; }

public Person(string firstName, string lastName)

{

FtNameProperty = firstName;

LtNameProperty = lastName;

}

**// Overload 1**

public string SayHello(string name)

{

return "Hello " + name;

}

**// Overload 2**

public string SayHello(Person x)

{

return "Hello " + x.FtNameProperty + " " + x.LtNameProperty + "!!!\n";

}

}

}

--============================================================================

namespace MyFirstProject

{

class Program

{

static void Main(string[] args)

{

Person john = new Person("John","Brown");

Person jane = new Person("Jane", "Green");

**// SayHello Overload 1**

Console.WriteLine(john.SayHello("Chebika" + "\n"));

**// SayHello Overload 2**

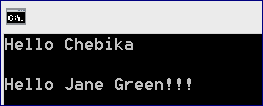
string greetings = john.SayHello(jane);

Console.WriteLine(greetings);

}

}

}

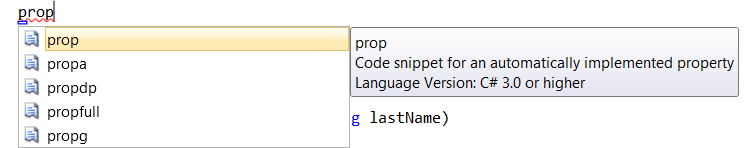


--======================

Snippets:

--======================

Print “prop” and press Tab button twice



Prop <Tab> <Tab>



You’ll get a template to fill in. Once you finish press <Enter>.

## 30 Days to Learn C# - Lesson 11 (Method Overloading)

Let’s say we want our person to greet another person.

We can create a string variable “greetings” that will concatenate the First and Last name.

Person jane = new Person("Jane", "Doe");

string greetings = (john.SayHello(jane.FtNm + " " + jane.LtNm));

Console.WriteLine(greetings + "\n");

However, there is more efficient way, especially if we have many different persons.

**Method overloading:**

Basically, we are changing the current method.

We write a different version of the existing method.

Currently, our “SayHello” accepts the string.

We can overload this method to make it accept a Person object.

1. Write another method that will have the SAME name

2. Provide a different set of parameters (parameter signature) to this method

public string **SayHello**(**string** name)

{

return "Hello, " + name;

}

public string **SayHello**(**Person** person)

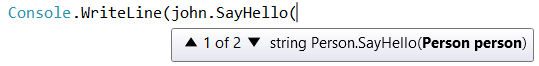
{

return "Hello, " + person.\_firstname + " " + person.\_lastname;

}

Now we will have two options for the method “SayHello”

We can either pass a string or a Person object to this method.





To make the overload more efficient we can have one **MASTER** overload and all others will be calling it with a different set of parameters (parameter TYPE)

We cannot have to “SayHello” methods with

Now we can change “Hello” to “Hello there” only in one place.

public string **SayHello**(string name)

{

return **"Hello there, "** + name;

}

public string **SayHello**(Person person)

{

return **SayHello**(person.\_firstname + " " + person.\_lastname);

}

We **cannot** have method with the same datatype even though the variable is different

public string SayHello(string foo)

**But** we can have string and person datatypes together (as it’s a **different signature**)

public string SayHello(**string foo, Person person**)

The **reverse order** is also a different signature

public string SayHello(**Person person, string foo**)

**We can also overload CONSTRUCTORS as those are special methods.**

The same rule applies: the constructor has to have a different signature than the other constructors.

Let’s allow the creation of a Person object with the First Name only.

public **Person**(string firstName)

{

this.\_firstname = firstName;

}

public **Person**(string firstName, string lastName)

{

this.\_firstname = firstName;

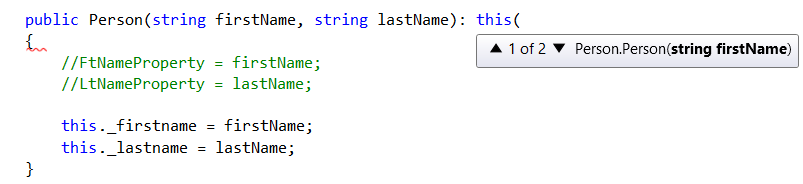
this.\_lastname = lastName;

}





We can also use a MASTER CONSTRUCTOR but the syntax is different that we have with methods.



private string \_firstname;

private string \_lastname;

**// Main constructor**

public Person(string firstName)

{

this.\_firstname = firstName;

}

**//Overload #1**

public Person(string firstName, string lastName): **this(firstName)**

{

//this.\_firstname = firstName;

this.\_lastname = lastName;

}

So, when we call the Overload #1 we pass two parameters: fisrtName and lastname

It will call the MASTER constructor with the firstName parameter

Then it will proceed with the secondName

namespace MyFirstProject

{

class Program

{

static void Main(string[] args)

{

Person john = new Person("John","Doe");

Person jane = new Person("Jane", "Doe");

// approach with concatenating parameters

string greetings = (john.SayHello(jane.FtNm + " " + jane.LtNm));

Console.WriteLine(greetings + "\n");

// approach with overloading

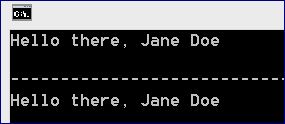
Console.WriteLine("-----------------------------------------------------");

Console.WriteLine(john.SayHello(jane)+"\n");

}

}

}



## 30 Days to Learn C# - Lesson 12 (Default Optional Parameters)

We can assign an optional (default) value to the parameter.

So, we do not need to pass one while calling a method.

private string \_firstname;

private string \_lastname;

public Person(string firstName, string lastName **= ""**)

{

this.\_firstname = firstName;

this.\_lastname = lastName;

}

In this case we have just one constructor and we can call it with one OR two parameters:

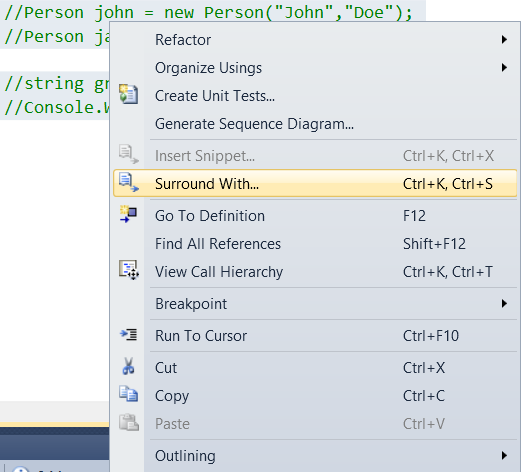
Person john = new Person("John","Doe");

Person jane = new Person("Jane");

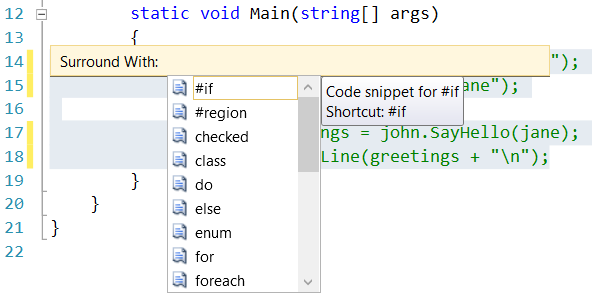
## [30 Days to Learn C# - Lesson 13 (Conditions](http://www.youtube.com/watch?v=j-gMupaKWaY&list=PLvRMjkCT20cFYVgtpbLkkJ9pGmm1yO1Id&index=13): IF, WITH)

A “condition” evaluates to either “true” or “false”.

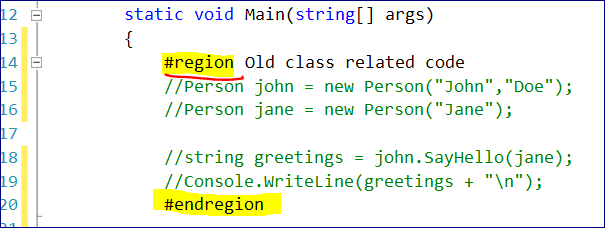
Use the feature of Visual Visio



Or Press Ctrl + K + S



Select **#region**



This is just to hide a block of code

// != ; == ; <= ; < ; > ; >=

bool result = 6 > 5; // TRUE

bool str1 = string.IsNullOrEmpty("Test"); // FALSE

bool str2 = string.IsNullOrEmpty(""); // TRUE

bool str3 = string.IsNullOrEmpty(string.Empty); // TRUE

bool str4 = string.IsNullOrEmpty(null); // TRUE

// we can test against "null" values

string nullString = null;

bool str5 = nullString == null;

// Compare two objects with the same names

Person bob = new Person("Bob", "Bob");

Person bob2 = new Person("Bob", "Bob");

Console.WriteLine("Compare two objects with the same fisrt and last names: " + (bob == bob2).ToString());

Console.WriteLine(bob.FtNm == bob2.FtNm);

Console.WriteLine("----------------------------------");

// Logical AND: &&

// Both conditions have to be TRUE to return TRUE

Console.WriteLine (

(bob.FtNm == bob2.FtNm) && (bob.LtNm == bob2.LtNm)

);

// Logical OR: ||

// At LEAST ONE of the conditions have to be TRUE to return TRUE

// Logical NOT: !

// NOT True = False, NOT False = True

Console.WriteLine(result);

Console.WriteLine(str1);

string myStr = "test";

**if** (myStr != "test")

{

Console.WriteLine("This is true");

}

**else**

{

Console.WriteLine("This is false");

}

// Another Compariosn operator -- **SWITCH**

// It tests just "equality". Not >, < , etc.

// Whenever it finds a match the code in switch statement stops executing

string myStr2 = "hTest";

switch (myStr2)

{

case "test":

Console.WriteLine(@"The value is ""test""");

break;

case "tEst":

Console.WriteLine(@"The value is ""tEst""");

break;

case "Test":

Console.WriteLine(@"The value is ""Test""");

break;

default:

Console.WriteLine("No value was found \n");

break;

}

## 30 Days to Learn C# - Lesson 14 (Validating Parameters)



namespace MyFirstProject.MyFirstDataTypes

{

**public class Person**

{

public string FirstName { get; private set; }

public string LastName { get; private set; }

public Person(string firstName, string lastName = "")

{

// validating the firstName in the Constructor

if (string.IsNullOrEmpty(firstName) || string.IsNullOrWhiteSpace(firstName))

{

// if the string is null, whitespace, or empty we want to throw a new exception

// if this excecutes ALL code execution WILL STOP

throw new ArgumentException("firstName cannot be null, empty, or whitespace");

}

// validating the lastName in the Constructor

if ((lastName == null || lastName.Trim() == string.Empty) && lastName != string.Empty)

{

throw new ArgumentException("lastName cannot be null or whitespace");

}

FirstName = firstName;

LastName = lastName;

}

// Overload 1

public string SayHello(string name)

{

**// check for null**

**// empty**

**// whitespace**

if (string.IsNullOrEmpty(name) || string.IsNullOrWhiteSpace(name))

{

// if the string is null, whitespace, or empty we want to throw a new exception

// if this excecutes ALL code execution WILL STOP

throw new ArgumentException("name cannot be null, empty, or whitespace");

}

return "Hello there, " + name;

}

// Overload 2

public string SayHello(Person person)

{

if (person == null)

{

**// throwing an exception**

**// exceptions are bsically "errors"**

throw new ArgumentNullException("person cannot be null");

}

return SayHello(person.FirstName + " " + person.LastName);

}

}

}

--===============

namespace MyFirstProject

{

**class Program**

{

static void Main(string[] args)

{

Person john = new Person("John","Doe");

Person jane = new Person("Jane");

//string greetings = john.SayHello(jane);

//Console.WriteLine(greetings + "\n");

// It will handle the following exceptions:

//Person person = new Person(null, null);

//Person person = new Person(string.Empty, null);

//Person person = new Person(" ", null);

//Person person = new Person("Jeremy", null);

//Person person = new Person("Jeremy", " ");

Person person = new Person("Jeremy", "daldsjals");

Console.WriteLine(person.SayHello("Jeremy"));

}

}

}

## [30 Days to Learn C# - Lesson 15 (Loops](http://www.youtube.com/watch?v=1SxsgbWqNec&list=PLvRMjkCT20cFYVgtpbLkkJ9pGmm1yO1Id&index=15))

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// FOR Loop -- you know how many incrementation you know

// for (initialization; condition; incrementation)

string str = "Hello, World!";

for (int counter = 0; counter < str.Length; counter++)

{

// "ii" is a good name for the counter as it's easier to find in the text

//int i = counter++; // assigns a value and THEN increment

//int z = ++counter; // increments the value and assign

Console.WriteLine(counter);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// WHILE Loop -- executes WHILE smth is TRUE

int index = str.IndexOf("l");

while (index > -1)

{

str = str.Remove(index, 1);

index = str.IndexOf("l");

}

Console.WriteLine(str);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// DO WHILE Loop

do

{

} while (index > -1);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// FOR EACH Loop -- used with arrays or strings

// for each character in the string "str" perform some action

str = "Hello, World!";

foreach (char c in str)

{

Console.WriteLine(c);

}

## 30 Days to Learn C# - Lesson 16 (Arrays)

An array could of a specific data type.

The length of the array is predefined.

// Create an array

// DataType [] = new DataType [length];

char[] foo = new char[5];

foo[0] = 'H';

foo[1] = 'e';

foo[2] = 'l';

foo[3] = 'l';

foo[4] = 'o';

// With FOR Loop

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

Console.WriteLine();

// with FOR EACH Loop

foreach (char x in foo) { Console.Write(x); }

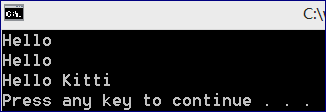
Console.WriteLine();

// Another way to populate the array

// We do not specify the length of the array but populate it at the time of creation

char[] myArr = new char[] { 'H', 'e', 'l', 'l', 'o', ' ', 'K', 'i', 't', 't','i' };

Console.WriteLine(myArr);



## 30 Days to Learn C# - Lesson 17 (Arrays as Parameters)

**“params” key word**

----------------------------------------------------------------------------------------------------

public string SayHello(**params** string[] names)

{

}

----------------------------------------------------------------------------------------------------

string[] names = new string[]

{

"James",

"John",

"Jennifer",

"Jackie"

};

Person jeremy = new Person("Jeremy", "McPeak");

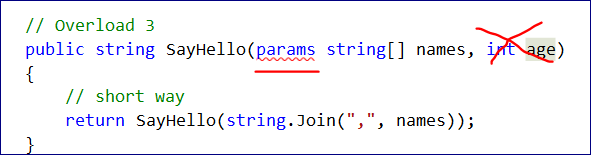
jeremy.SayHello(names);

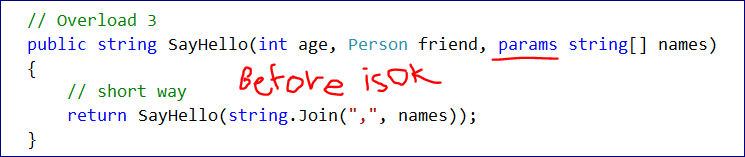
**We want to run SayHello(string) method to several parameters (strings within the array).**

**There are two limitations:**

- The an array contains values of the SAME datatype

- We cannot have parameters after using a “params” key word (we can have them before though)





**//long way**

**// Overload 3**

public string SayHello(params string[] names)

{

StringBuilder builder = new StringBuilder();

string temp = string.Empty;

foreach (string name in names)

{

// name, name, name

builder.Append(name + ",");

}

temp = builder.ToString();

temp = temp.Remove(temp.LastIndexOf(","));

return SayHello(temp);

}

--===========================

**// short way**

**// Overload 3**

public string SayHello(params string[] names)

{

return SayHello(string.Join(",", names));

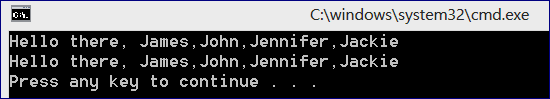
}

--===========================

MAIN Program

Console.WriteLine(jeremy.SayHello(names));

Console.WriteLine(jeremy.SayHello("James", "John", "Jennifer", "Jackie"));

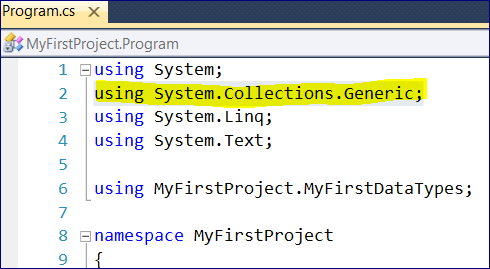


## 30 Days to Learn C# - Lesson 18 (Generic Lists and Dictionaries)

**Lists are similar to arrays but have many extra methods that make them more powerful (“arrays on steroids”).**

The collections in .NET framework are divided into two categories:

* Non-generic collections which are shipped with original version of the .NET framework
* Generic collections that were added in .NET 2.0



Generic collections are far better because of the type safety.

We discussed two of the Generic classes:

* List
* Dictionary

Lists are similar to arrays but we can add as many elements as we need (we do not need to specify the size of the list)

**LISTS**

To create a list we have to specify

* Datatype
* variable name
* =
* new keyword
* constructor()

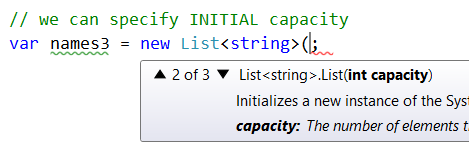
List<string> namesList = new List<string>();

List<int> col1 = new List<int>();

We can use a shortcut for the left part

var names2 = new List<string>();

The compiler understands that we call **List** constructor and will create a List object with the name “names2”



// we can specify INITIAL capacity

var names3 = new List<string>(10);

It will be automatically expanded if we add more than 10 elements to the list.

string[] names = new string[]

{

"James",

"John",

"Jennifer",

"Jackie"

};

List<int> col1 = new List<int>();

List<string> names1 = new List<string>();

// the shortcut "var"

var names2 = new List<string>();

// we can specify INITIAL capacity

var names3 = new List<string>(10);

//-------------------------------------------------------

// features of the list class

//-------------------------------------------------------

List<string> namesList = new List<string>();

// Adding elements to the list

namesList.AddRange(names); // populate using an array

namesList.Add("Dmitry"); // add an element to the list

// how many elements are in the list

Console.WriteLine("list has elemets: " + namesList.Count);

Console.WriteLine("---------------------------------------");

Console.WriteLine(namesList[0]);

// Check if an element is in the list

bool hasName = namesList.Contains("Dmitry");

Console.WriteLine(hasName);

Console.WriteLine("---------------------------------------");

// FindAll method allows to specify a searching criteria inside the list

// we use Lambda expressions for that

// find all elements that have the second letter = 'a'

var newList = namesList.FindAll((s) => s[1] == 'a');

foreach (var name in newList)

{

Console.WriteLine(name);

}

Console.WriteLine("---------------------------------------");

// clear out all elemets

namesList.Clear();

Console.WriteLine("list has elemets: " + namesList.Count);

namesList.TrimExcess();

Console.WriteLine("list has elemets: " + namesList.Count);

**DICTIONARIES**

**Dictionary is a collection of a “Key – Value” pairs.**

//===============================================================

Console.WriteLine("\n----------------------------------------------");

Console.WriteLine("DICTIONARY");

Console.WriteLine("----------------------------------------------");

Dictionary<int, string> dict1 = new Dictionary<int, string>();

var myDict = new Dictionary<int, string>();

myDict.Add(0, "Jeremy");

string value = myDict[0]; // because we've used an integer as "Key" it looks exactly like Index notaion

var dict2 = new Dictionary<string, string>();

dict2.Add("a", "Dmitry");

dict2.Add("boo", "Natalia");

string value2 = dict2["a"]; // as here we use "string" as a "Key"

Console.WriteLine(dict2["a"]);

//dict2.ContainsKey("a");

//dict2.ContainsValue("Dmitry");

//dict2.Remove("a"); // accepts a "Key" not an Index

// Printing values from Dictionary

foreach (var keyValuePair in dict2)

{

Console.WriteLine(keyValuePair.Key + " = " + keyValuePair.Value);

}

//-----------------------------------------------

Console.WriteLine("\n----------------------------------------------");

Console.WriteLine("Multivalue Dictionary");

Console.WriteLine("----------------------------------------------");

var multiValueDict = new Dictionary<string, List<string>>();

multiValueDict.Add("a", new List<string>());

multiValueDict["a"].Add("Dmitry");

multiValueDict["a"].Add("Natalia");

## 30 Days to Learn C# - Lesson 19 (Inheritance)

Everything inherits from the System.Object

It means that everything can be converted or CAST to an object.

System.Object x; // we can declare a variable in this way

Object myVariable1; // or that way

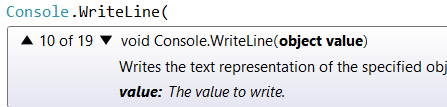
object myVariable2; // or that way

object foo = (object)10; // casting

int bar = (int)foo;

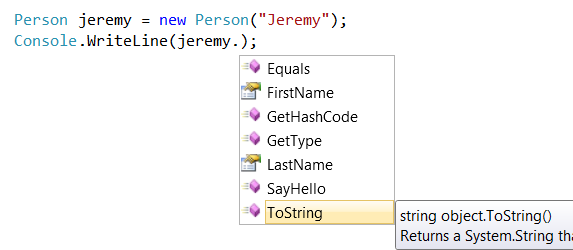
Console.WriteLine(bar);

We can pass an object to WriteLine()



If create an object Person we access a ToString() method.

But we di d not write this method. It was inherited from the Object class (automatically).



**Let’s write our own ToString method.**

We can use **OVERRIDING**.

It’s a feature of any object-oriented programming.

We can override Properties and Methods of the base class.

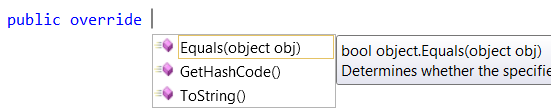
public override

Do not confuse overload and override.

Overload – we can have SayHello() method with 3 different overloads.

Overriding – we use something from the parent class.

This is what we can override in our Person class.



public override string ToString()

{

return base.ToString();

}

The key word **base** refers to the base class from which we want to inherit the method.

We change it to   
 public **override** string ToString()

{

//return base.ToString();

**return FirstName + " " + LastName;**

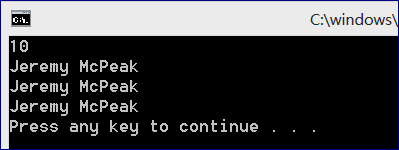
}

**Whenever we call the the ToString() method on Person object it will return the First and Last name of that person.**

Person jeremy = new Person("Jeremy", "McPeak");

Console.WriteLine(**jeremy.ToString()**);

Console.WriteLine(**jeremy**); **// the same result**



// we MAY cast jeremy as the object BUT not necesserily

// as the compiler knows that jeremy is a Person object

object foo2 = (object)jeremy;

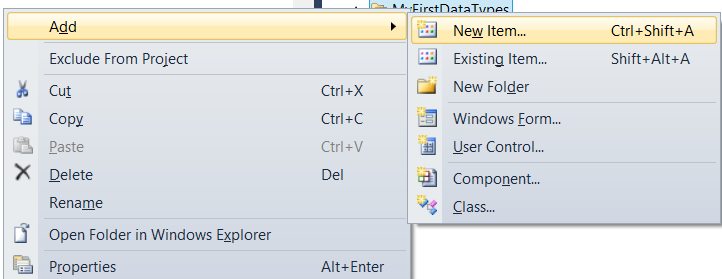
object foo1 = jeremy; // so we can skip casting

Console.WriteLine(foo1); // the same result

// However, here we need explicitly say that foo1 object is the Person object

Person jeremy2 = (Person)foo1;

**Let’s create a new class (Employee) – to represent an individual employee.**



namespace MyFirstProject.MyFirstDataTypes

{

class Employee

{

}

}

Let’s make our class public

public class Employee

{

}

…

namespace MyFirstProject.MyFirstDataTypes

{

// To inherent from Person class use : Person

// We will get Person's properties and SayHello() method

// but we don't get a Constructor

public class Employee : Person

{

// let's write the constructor for the Person class

public Employee(string firstName, string lastName, string position): base (firstName, lastName)

{

if (string.IsNullOrEmpty(lastName))

{

throw new ArgumentException("Last Name cannot be null, empty, or whitespace");

}

if (string.IsNullOrEmpty(position) || string.IsNullOrWhiteSpace(position))

{

throw new ArgumentException("Position cannot be null, empty, or whitespace");

}

Position = position; // assigned to simplified get, set methods

}

// position of an employee: owner, manager, worker, etc.

public string Position { get; private set; }

public override string ToString()

{

return base.ToString() + "; " + Position;

}

}

}

--===================

No if we call it from our main program:

using MyFirstProject.MyFirstDataTypes;

namespace MyFirstProject

{

class Program

{

static void Main(string[] args)

{

//==============================================

Employee dmitry = new Employee("Dmitry","Vakhrushev","Database Marketing Analyst");

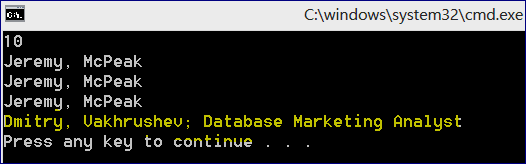
Console.WriteLine(dmitry);

//==============================================

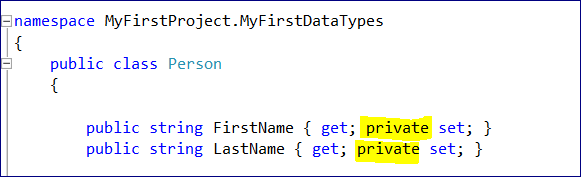
}

}

}



**However, in the Person class we cannot change the Last Name with the SET method as it’s set to Private.**



We can change it to **protected**

**Protected** is somewhat like private but it can be seen from derived classes.

public class **Person**

{

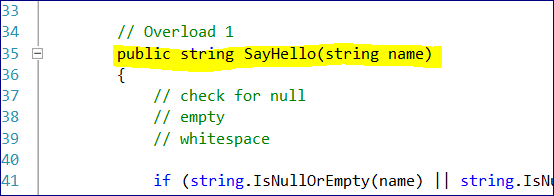
public string FirstName { get; **protected** set; }

public string LastName { get; **protected** set; }

It’s still not accessible outside of the Person class.

But we can access it from the Person class or from Employee (as it inherited from Person).

--=================================



**If we want OVERRIDE SayHello() method of the Person from Employee class we need to use virtual**

**Keyword when we create this method in the Person class.**

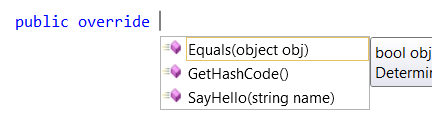
// Overload 1

public **virtual** string SayHello(string name)

{

**If something (method) is virtual it can be overridden.**

Now SayHello() method is available to us in the Person class.



public **override** string **SayHello**(string name)

{

//return base.SayHello(name);

return "Hi, how may I help you " + name + "?";

}

//==============================================

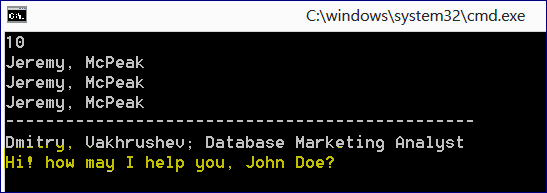
Employee dmitry = new Employee("Dmitry","Vakhrushev","Database Marketing Analyst");

Console.WriteLine(dmitry);

//==============================================

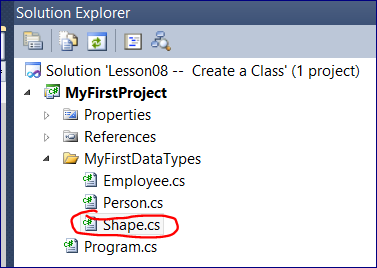
Person customer = new Person("John", "Doe");

Console.WriteLine(dmitry.SayHello(customer) + "\n");

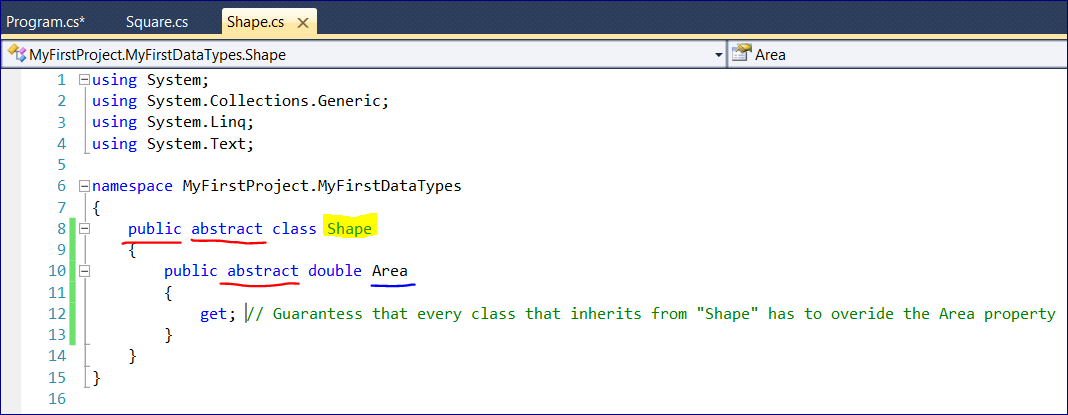


## 30 Days to Learn C# - Lesson 20 (Abstract Classes)

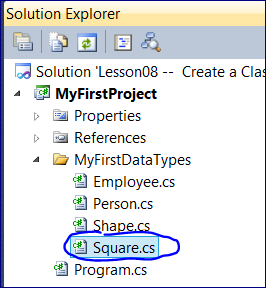
Let’s add a new regular class named “Shape”.

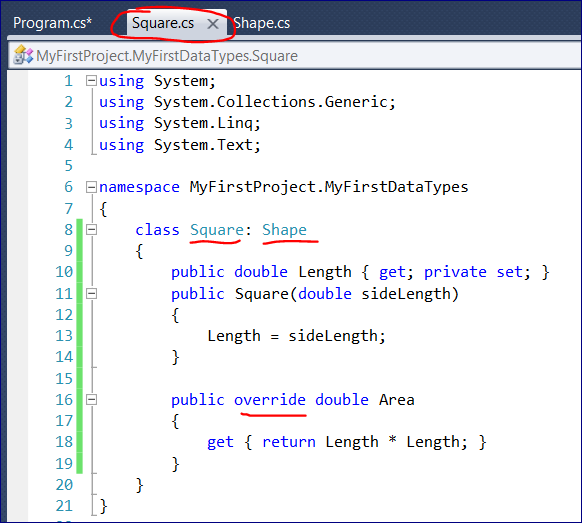


Shape Class



Add “Square” class

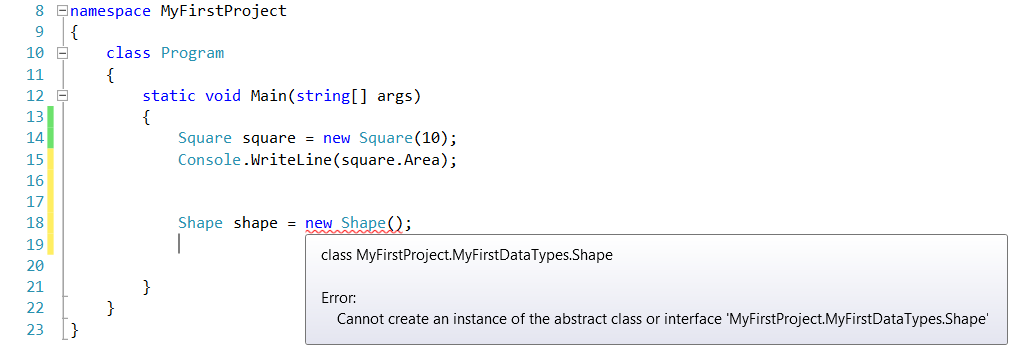




In the Main Class

We can create an instance of “Square” class

But cannot do the same for the “Shape” class. As the “Shape” is **ABSTARCT** class.



**We cannot create an instance of the Abstract class.**

But we can create a Square object and assign it to the Shape variable.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using MyFirstProject.MyFirstDataTypes;

namespace MyFirstProject

{

class Program

{

static void Main(string[] args)

{

Square square = new Square(10);

Console.WriteLine(square.Area);

//Shape shape = new Shape();

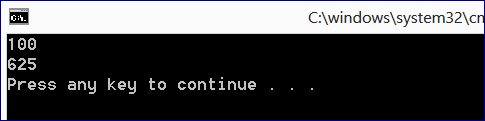
Shape shape = new Square(25);

Console.WriteLine(shape.Area);

}

}

}



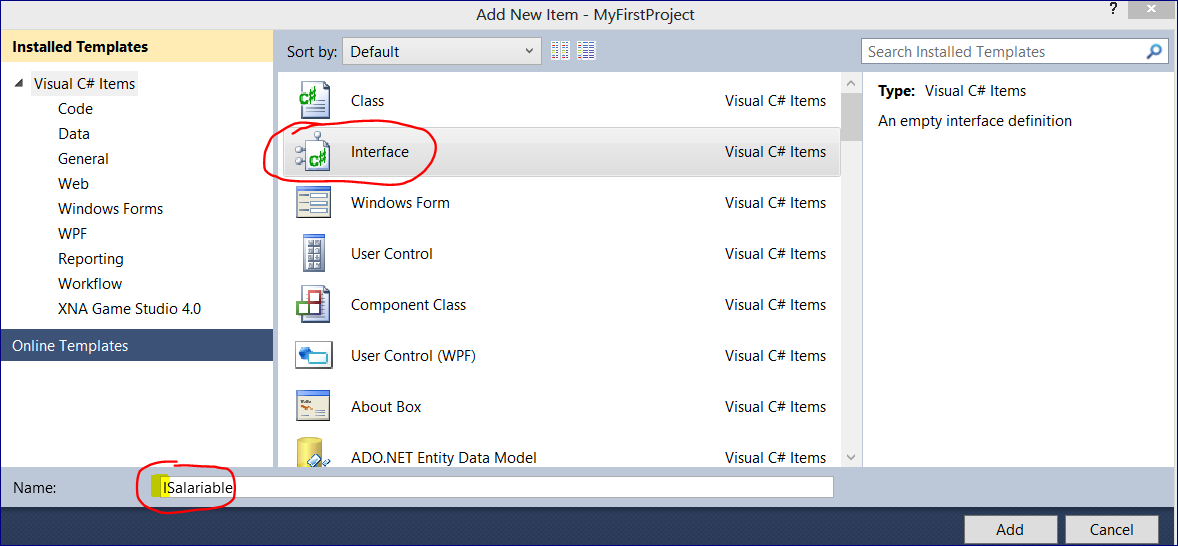
## 30 Days to Learn C# - Lesson 21 (Interfaces)

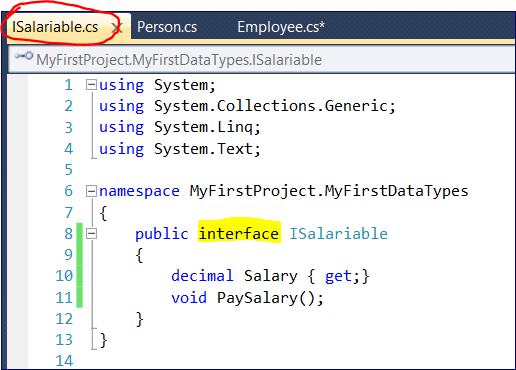
We can inherent from ONE class at a time.

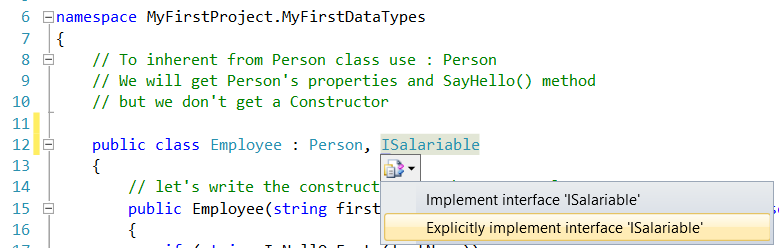
Let’s create an Interface.

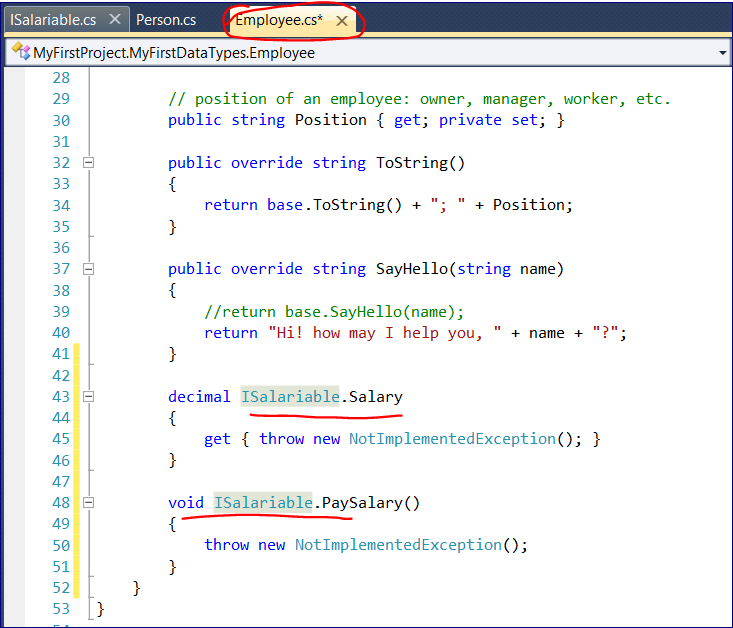
By convention it should start with a capital “I”.

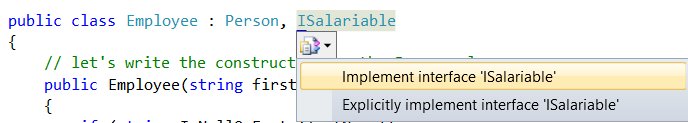
Interfaces are data types and we can use them as any other data type.











public decimal Salary

{

get { throw new NotImplementedException(); }

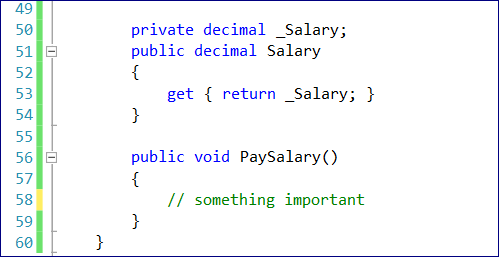
}

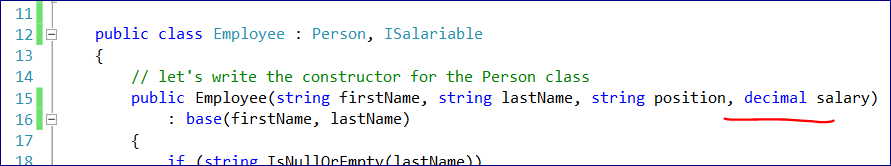
public void PaySalary()

{

throw new NotImplementedException();

}





We cannot create an ISalariable object because INTERFACE IS NOT A CLASS.

But we can create an object that implements that interface and assigns it to a variable of that type of interface.

ISalariable emp = new Employee("John", "Doe", "Sales Clerk", 2000);

**Let’s add another interface.**

namespace MyFirstProject.MyFirstDataTypes

{

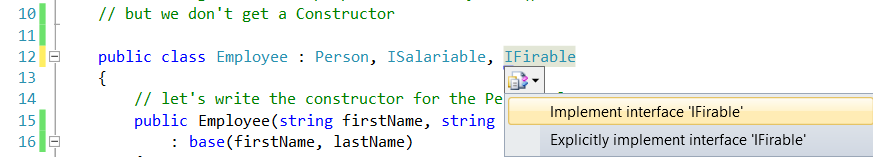
public interface IFirable

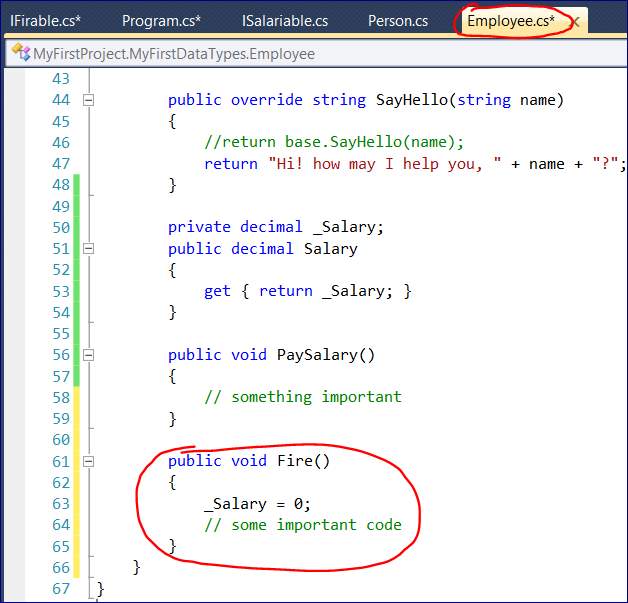
{

void Fire();

}

}





## 30 Days to Learn C# - Lesson 22 (Static Members and Classes)

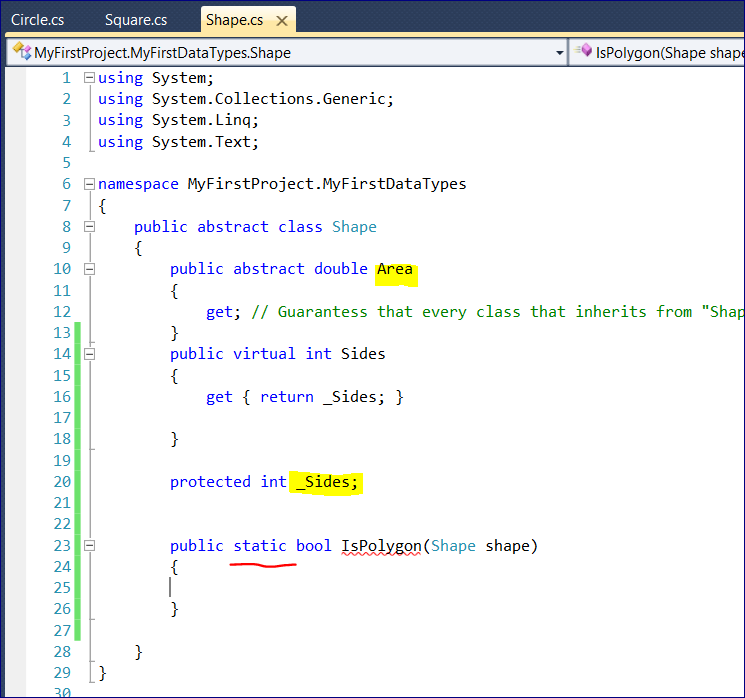
Methods we’ve written before were so called **“instance members”**.

They belong to instance of the class.

Person john = new Person("John", "Doe");

string name = john.FirstName;

We can create STATIC method that will belong to a class not to an object of that class.



It’s important to know that inside of the Static class we cannot access instance data.

public static bool IsPolygon(Shape shape)

{

//we do not have access to the "instance" variables: \_Sides or Area

//but we do have access to the Shape object passed as an argument

return shape.Sides >= 3;

}

--===========

namespace MyFirstProject

{

class Program

{

static void Main(string[] args)

{

Square square = new Square(10);

Circle circle = new Circle(10);

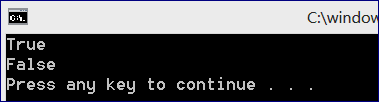
Console.WriteLine(Shape.IsPolygon(square));

Console.WriteLine(Shape.IsPolygon(circle));

}

}

}



Now let’s create a STATIC CLASS

Static classes are usually used as utility classes (e.g. Math class)

**Rule: Static cannot access Instance Data**

namespace MyFirstProject.MyFirstDataTypes

{

public static class ShapeUtility

{

public static bool IsPolygon(Shape shape)

{

**// Rule 1: anything that we defined in the static class has to be static**

**// Rule 2: we cannot create an instance pf the static class**

**// Rule 3: we cannot inherent from the static class (as we can inherent instance information)**

return shape.Sides >= 3;

}

}

}

## 30 Days to Learn C# - Lesson 23 (Extension Methods)

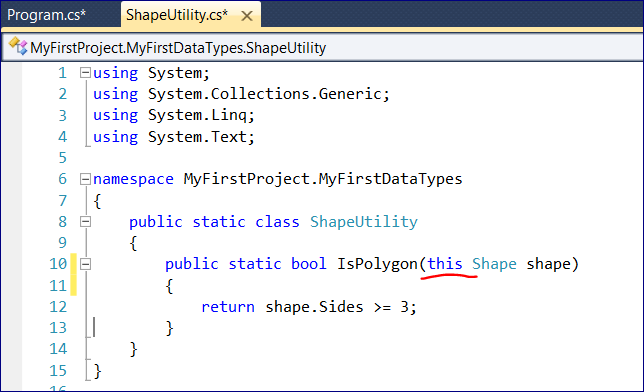
This specific feature was introduced in C# and .NET 3.5

string name = "john";

string.IsNullOrEmpty(name);

// we'd better use it in a way "name.IsNullOrEmpty()"

// extension methods allow us to do that



Adding “**this**”

In this case every Shape object is going to get **IsPolygon** method

namespace MyFirstProject

{

class Program

{

static void Main(string[] args)

{

Square square = new Square(10);

Circle circle = new Circle(10);

Console.WriteLine(ShapeUtility.IsPolygon(square));

Console.WriteLine(ShapeUtility.IsPolygon(circle));

Console.WriteLine("-----------------------------------");

// we can also use the extension method and use new syntax

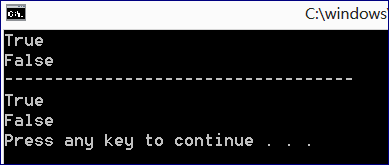
Console.WriteLine(square.IsPolygon());

Console.WriteLine(circle.IsPolygon());

}

}

}



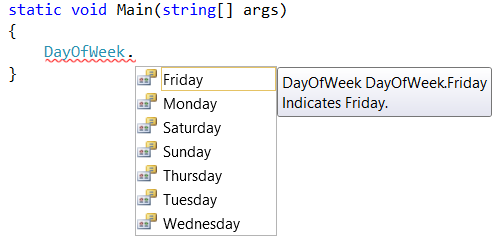
## 30 Days to Learn C# - Lesson 24 (Enums)

Allow us to assign names to CONSTANT number variables.

It resembles a static class (by syntax) BUT IT IS NOT a static class.

**Enums are Value Types.**

.NET framework has Enum already

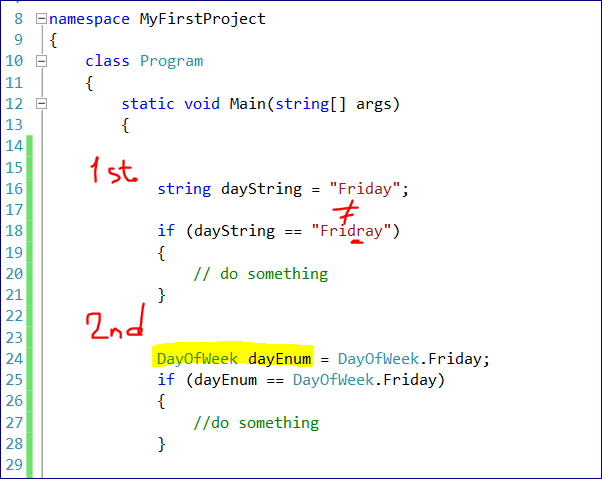


DayOfWeek day = DayOfWeek.Friday;

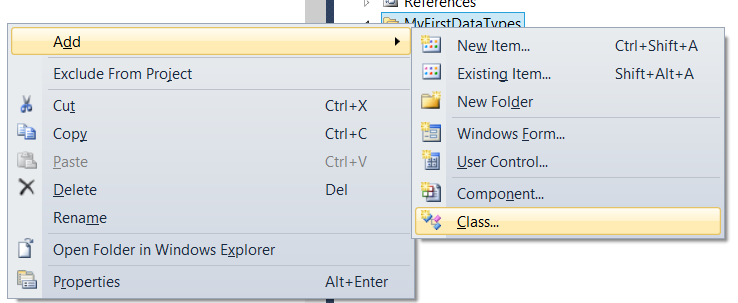
DayOfWeek day = DayOfWeek.Friday;

Why don’t we use just strings instead?

1) Enums are more reliable as you cannot make a mistake in e.g. IF statement. Because you select from the list.



**Let’s create Enum that will hold Employee Positions. Add new class**



Then we need to change the word class to **public enum**

namespace MyFirstProject.MyFirstDataTypes

{

**public enum** EmployeePosition

{

}

}

Now we need to provide possible values.

namespace MyFirstProject.MyFirstDataTypes

{

public enum EmployeePosition

{

Owner,

Manager,

SalesClerk,

Stocker

}

}

Behind the scenes the compiler assigns a numeric value to each position.

In fact, they are integers.

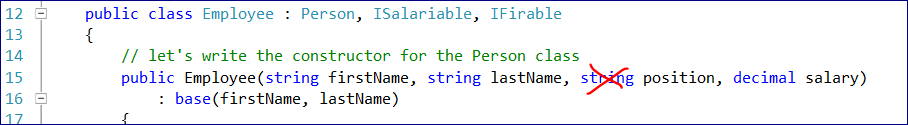
Owner = 0,

Manager = 1,

SalesClerk = 2,

Stocker = 3

Now let’s change our Employee.cs



public Employee(string firstName, string lastName, **EmployeePosition** position, decimal salary)

: base(firstName, lastName)

Here is our main program

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

**using MyFirstProject.MyFirstDataTypes;**

namespace MyFirstProject

{

class Program

{

static void Main(string[] args)

{

Employee emp = new Employee("John", "Doe", **EmployeePosition.Manager**, 200000);

}

}

}

## 30 Days to Learn C# - Lesson 25 (Structs and Memory Management)

Different types of data available to us through C#.

**They are:**

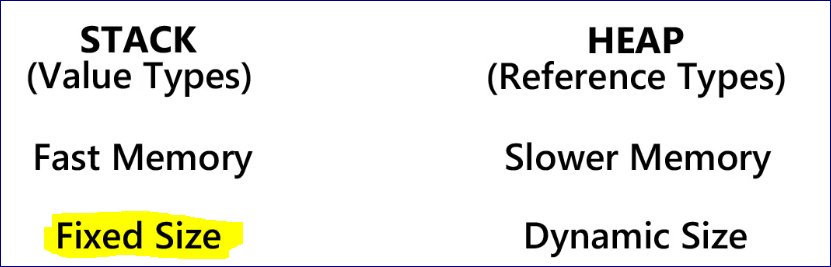
* **Classes**
* **Interfaces**
* **Enums**
* **Structs**

**Structs** are a lot like classes.

Key difference with a class: Structs are value types. They are stored by value.

As opposed to classes which are reference types.

Memory is divided into two types: STACK and HEAP



We can overflow the stack memory. This where “Stack Overflow” comes from.

Structs are very similar to Classes.

DateTime now = DateTime.Now;

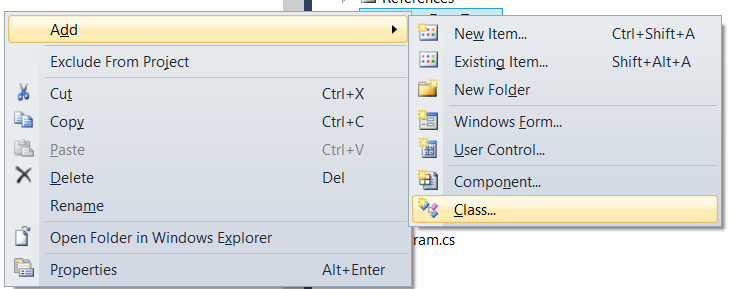
// DateTime is immutable --> we cannot modify the value of now

// We can implement a method on it an assign to another variable

DateTime foo = now.AddYears(1);

DateTime bar = new DateTime(2001, 6, 9);

**Let’s create a Struct**



namespace MyFirstProject.MyFirstDataTypes

{

class Color

{

}

}

Then we need to change the word **class** to **public struct**

namespace MyFirstProject.MyFirstDataTypes

{

**public struct** Color

{

}

}

--======================

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace MyFirstProject.MyFirstDataTypes

{

public struct Color

{

//create R(red), G(green), B(blue) properties

// the datatype is "byte"

public byte R { get; private set; }

public byte G { get; private set; }

public byte B { get; private set; }

// Constructor to set instance properties

public Color(byte red, byte green, byte blue)

{

R = red;

G = green;

B = blue;

}

// static methods

public static Color Red

{

get { return new Color(255, 0, 0); }

}

public static Color Green

{

get { return new Color(0, 255, 0); }

}

public static Color Blue

{

get { return new Color(0, 0, 255); }

}

public static Color Black

{

get { return new Color(0, 0, 0); }

}

public static Color White

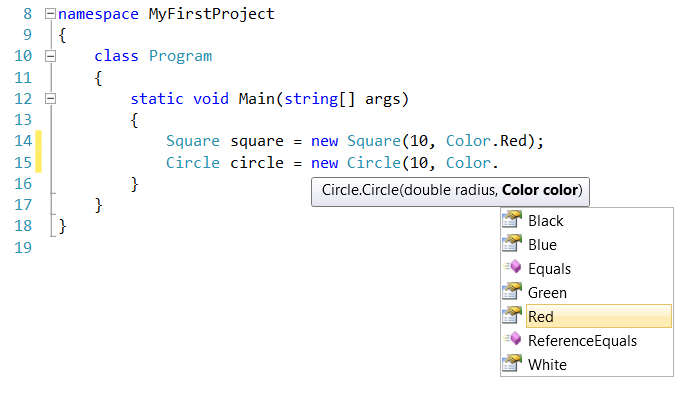
{

get { return new Color(255, 255, 255); }

}

}

}



namespace MyFirstProject

{

class Program

{

static void Main(string[] args)

{

Square square = new Square(10, Color.Red);

Circle circle = new Circle(10, Color.White);

}

}

}

**What to use Structs or Classes?**

Classes are a bit slower but Structs can crash the program if the result in stack overflow.

**The rule of thumb: when in doubt use Class.**

## 30 Days to Learn C# - Lesson 26 (Delegates)

It’s a Data Type that define a data Signature.

For example, one overload is different from another based on the signature.

**So, we can pass one method as an argument to another method.**

**This is all in Main Program file**

namespace MyFirstProject

{

// Define a Delegate

public delegate void SayMessage(string message);

// Define First class

class DelegateOne

{

public void Method(string message)

{

Console.WriteLine(message);

}

}

// Define Second class

class DelegateTwo

{

public void MethodTwo(SayMessage fn)

{

fn("Hello, World! From Delegate Two");

}

}

//---------------------------------------------------------

// Main Progran

//---------------------------------------------------------

class Program

{

static void Main(string[] args)

{

DelegateOne one = new DelegateOne();

DelegateTwo two = new DelegateTwo();

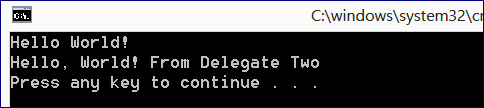
one.Method("Hello World!");

two.MethodTwo(one.Method);

}

}

}



**Another Example**

namespace MyFirstProject

{

class Program

{

static void Main(string[] args)

{

List<string> names = new List<string>

{

"Jeremy",

"Jeffrey",

"John",

"Jennifer",

"Jackie",

"Julianne"

};

var namesWithE = names.FindAll(FindNamesWithE);

foreach (var name in namesWithE)

{

Console.WriteLine(name);

}

}

static bool FindNamesWithE(string name)

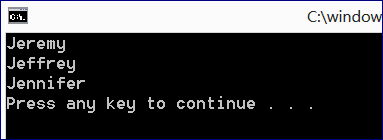
{

// select any name that has "e" as the second letter

return name.IndexOf("e") == 1; }

}

}



Or we can do it another way

namespace MyFirstProject

{

class Program

{

static void Main(string[] args)

{

List<string> names = new List<string>

{

"Jeremy",

"Jeffrey",

"John",

"Jennifer",

"Jackie",

"Julianne"

};

//-------------------------------------------------

var namesWithE = names.FindAll(

delegate(string name)

{

return name.IndexOf("e") == 1; // select any name that has "e" as the secind letter

}

);

foreach (var name in namesWithE)

{

Console.WriteLine(name);

}

}

## 30 Days to Learn C# - Lesson 27 (Functions and Lambda Expressions)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using MyFirstProject.MyFirstDataTypes;

namespace MyFirstProject

{

class Program

{

static void Main(string[] args)

{

List<string> names = new List<string>

{

"Jeremy",

"Jeffrey",

"John",

"Jennifer",

"Jackie",

"Julianne"

};

// Functions always return a value

// string as a parameter and bool as a return value. "fn" function name

Func<string, bool> fn = FindNamesWithE;

// to call the Function

bool result = fn("ae"); // this will return TRUE (as the second letter is "e")

// Lambda expressions are the shorten version of the deligate syntax

// let's use "n" for "name"

Predicate<string> pred = n => n.IndexOf("e") == 1;

var namesWithE = names.FindAll(n => n.IndexOf("e") == 1);

foreach (var name in namesWithE)

{

Console.WriteLine(name);

}

}

public static bool FindNamesWithE(string name)

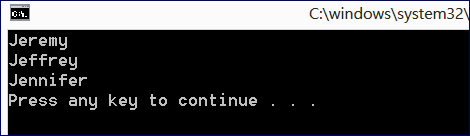
{

return name.IndexOf("e") == 1;

}

}

}



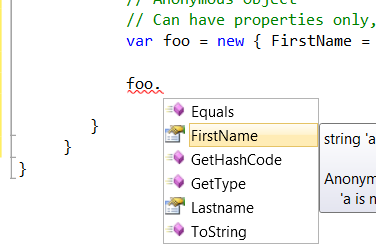
## 30 Days to Learn C# - Lesson 28 (LINQ)

We can create an Anonymous Object

// Anonymous Object

// Can have properties only, NO Methods

var foo = new { FirstName = "Jeremy", Lastname = "McPeak" };



--================

LINQ

To use LINQ we need to import it first

using System.Linq; // Language INtegrated Query (LINQ)

So now we can query almost anything: objects, XML, MS SQL databases

We can use two methods:

* We can use query syntax
* We can use extension methods

--==============================

using System;

using System.Collections.Generic;

using System.Text;

**using System.Linq; // Language INtegrated Query (LINQ)**

using MyFirstProject.MyFirstDataTypes;

namespace MyFirstProject

{

class Program

{

static void Main(string[] args)

{

IEnumerable<Employee> employees = new Employee[] {

new Employee("John","Doe",EmployeePosition.Owner, 200000),

new Employee("Jim","Donald",EmployeePosition.Manager, 100000),

new Employee("Jane","Dixie",EmployeePosition.SalesClerk, 25000),

new Employee("Jackie","Dame",EmployeePosition.SalesClerk, 25000),

new Employee("Jeff","Draper",EmployeePosition.Stocker, 18000),

new Employee("Jessica","Donroe",EmployeePosition.Stocker, 18000)

};

// Query Syntax

var query = from employee in employees

where employee.FirstName == "John"

select employee;

foreach (var x in query)

{

Console.WriteLine(x.FullName);

}

// Using extension method

var query2 = employees.Where(e => e.FirstName == "John");

// Doing it without LINQ

var emps = new List<Employee>();

foreach (var employee in employees)

{

if (employee.FirstName == "John")

{

emps.Add(employee);

}

}

//-----------------------------------

// Advanced

// Query Syntax

var query3 = from employee in employees

group employee by employee.Position;

foreach (var group in query3)

{

Console.WriteLine(group.Key);

foreach (var employee in group)

{

Console.WriteLine(employee.FullName);

}

Console.WriteLine();

}

//---------------------------

var query4 = employees.GroupBy(e => e.Position);

foreach (var group in query4)

{

Console.WriteLine(group.Key);

foreach (var employee in group)

{

Console.WriteLine(employee.FullName);

}

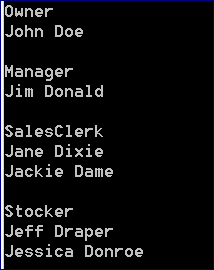
Console.WriteLine();

}

}

}

}



## 30 Days to Learn C# - Lesson 29 (Exception Handling)

**Try – Catch** statement is the heart of exception handling

For example, in the code below we can pass Shape as NULL.

We can create a Shape object but never initialize it, so the value of it will be a null object.

--=============

namespace MyFirstProject.MyFirstDataTypes

{

public static class ShapeUtility

{

public static bool IsPolygon(Shape shape)

{

return shape.Sides >= 3;

}

}

}

--=============

class Program

{

static void Main(string[] args)

{

**Shape nullShape;**

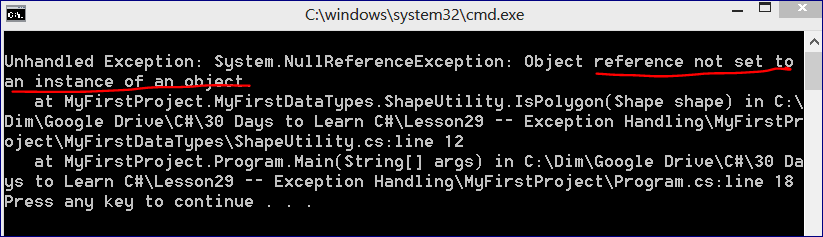
**nullShape = null;**

**Console.WriteLine(ShapeUtility.IsPolygon(nullShape));**

}

}

}



namespace MyFirstProject

{

class Program

{

static void Main(string[] args)

{

try

{

Shape nullShape = null;

Console.WriteLine(ShapeUtility.IsPolygon(nullShape));

}

// Specify the type of Exception you want to catch

// Exception is the base type of all exceptions

// We can have multiple catch

**catch (Exception ex)**

**{**

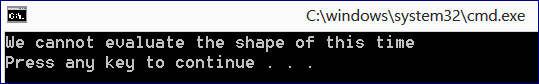
**Console.WriteLine("We cannot evaluate the shape of this time");**

**}**

}

}

}



**We can have multiple CATCH statements in one TRY-CATCH-FINALLY block.**

namespace MyFirstProject

{

class Program

{

static void Main(string[] args)

#region FirstExample

//{

// try

// {

// Shape nullShape = null;

// Console.WriteLine(ShapeUtility.IsPolygon(nullShape));

// }

// // Specify the type of Exception you want to catch

// // Exception is the base type of all exceptions

// // We can have multiple catch

// catch (Exception ex)

// {

// Console.WriteLine("We cannot evaluate the shape of this time");

// }

//}

#endregion

{

try

{

throw new System.Net.WebException(); // catch connectivity issues to the server

throw new System.IO.FileNotFoundException();

throw new OutOfMemoryException();

}

catch (System.Net.WebException webEx)

{

Console.WriteLine("We cannot retrieve the requested web resource");

}

catch (System.IO.FileNotFoundException fileEx)

{

Console.WriteLine("We cannot find the file needed to save the document");

}

catch (Exception ex)

{

Console.WriteLine("Something happened");

}

finally

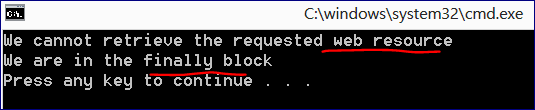
{

Console.WriteLine("We are in the finally blcock");

}

}

}



## 30 Days to Learn C# - Lesson 30 (Debugging)

This is the method we want to debus.

It’s written in the main Program

static void WriteTimestable(int multiplier)

{

for (int i = 1; i < 13; i++)

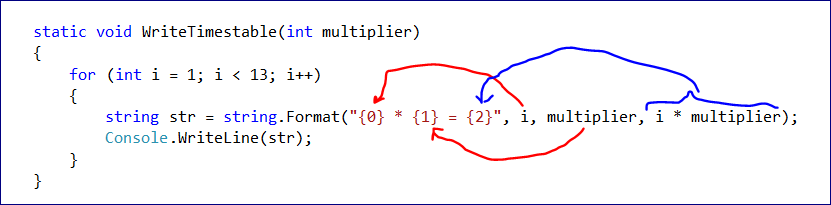
{

string str = string.Format("{0} \* {1} = {2}", i, multiplier, i \* multiplier);

Console.WriteLine(str);

}

}



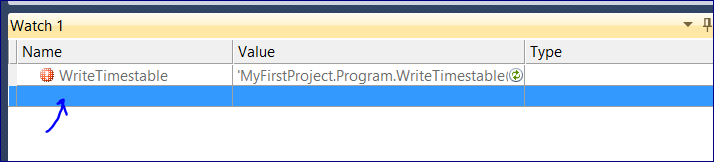
**Watch**

Add **WATCH** in debugging.

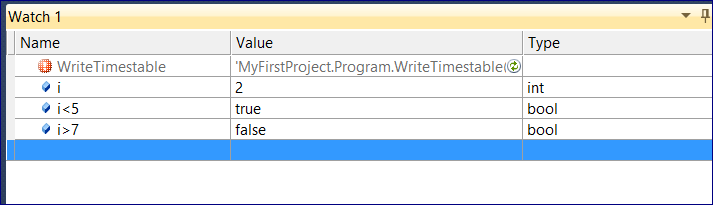
Start Debugging first 🡪 the Add Watch

Here we can watch for certain variables or expressions.

**Start printing here your variables or expressions**



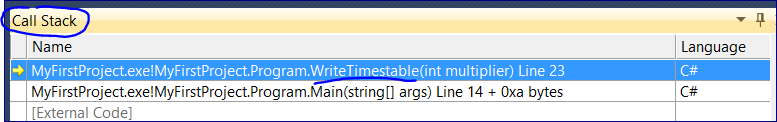
e.g.



**Call Stack**

List all of the Methods that have been called to the point on where we are in the code.

In descending order

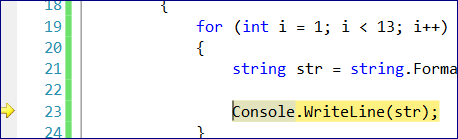


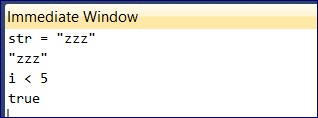
**Immediate Window**

Get immediate evaluation of any declared variables

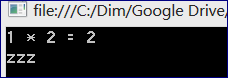
Print i<5 and press enter

We can change the value of the particular variable, e.g. **str = “zzz”**





Then press F11 to execute this line



Or we can evaluate an expression one time here

