# C# Basics

# IDE Environment Introduction

In this article, we are going to talk about what the IDE is and how can we use VisualStudio to create a new project.

## Integrated Development Environment (IDE)

An IDE is an environment tool which helps us writing the code for our programs. For this complete tutorial, we are going to use VisualStudio 2017. To download it, visit [VisualStudio Download Page](https://visualstudio.microsoft.com/downloads/?utm_medium=microsoft&utm_source=docs.microsoft.com&utm_campaign=button+cta&utm_content=download+vs2017). VisualStudio has a support for different programming languages, which makes it a very popular development tool.

After the installation, we can start a new project by clicking the File menu and choosing New => Project:



For this tutorial, we will use the console application project the most, so let's choose that option:



After we click on the OK button, we are going to see our created project. The main file to work with is the Program.cs and very soon we are going to talk more about it:



## Watch Window

In some examples, we are going to use this window to examine results. To open it we need to place a breakpoint on any code line first (by clicking left mouse button):



Then start our application by pressing F5, and finally to open the watch window:



## Conclusion

Now we know the basics and how to create a new project. Very soon we are going to use this knowledge in our applications.

# Data Types, Declarations and Variable Definitions

Different data types are registered differently and different actions are allowed to execute upon them as well. For different data types, a certain amount of space is reserved on our computer.

With data type we define:

* How to register data in memory
* The possible values for that data
* Possible actions on the data

## Data Type Registration

Data types that represent the whole numbers could be expressed with a certain number of bits. For unsigned numbers, the representation is from 0 to 2N-1. But for signed number the representation is from -2N-1 to 2N-1-1. So if the data type has a size of 8 bits like the sbyte data type, we can represnt its range like this: from -27 to 27-1 => from -128 to 127.

In the next table, we will show the different data types that represent the whole numbers:

| Type | Size (bits) | Range (values) |
| --- | --- | --- |
| byte | 8 | 0 to 255 |
| sbyte | 8 | -128 to 127 |
| int | 32 | -2,147,483,648 to 2,147,483,647 |
| uint | 32 | 0 to 4294967295 |
| short | 16 | -32,768 to 32,767 |
| ushort | 16 | 0 to 65,535 |
| long | 64 | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| ulong | 64 | 0 to 18,446,744,073,709,551,615 |

Letter u in front of the type means that type can’t contain negative numbers, it is unsigned.

Above mentioned types are the whole number types. But in C#, we have the number types with the floating point.

We can present them in a table as well:

| Type | Size (bits) | Range (values) |
| --- | --- | --- |
| float | 32 | -3.402823e38 to 3.402823e38 |
| double | 64 | -1.79769313486232e308 to 1.79769313486232e308 |
| decimal | 128 | (+ or -)1.0 x 10e-28 to 7.9 x 10e28 |

Next to all of this types, in C# we have two more basic type data:

| Type | Size (bits) | Range (values) |
| --- | --- | --- |
| char | 16 | Single Unicode sign, a whole number from 0 to 65535 |
| bool | 8 | false, true |

To use char type in our code we must place it inside the single quotes: ’a’ or ’A’ or ’3’...

One more type that is often introduced as the basic data type is the **string** type. But the string is not a value type it is a reference type. To use a string in our code we must place the value inside the double quotes: „This is the string type“ or „3452“...

So, we saw we have the value types and reference types, and it is time to talk more about those types and variables as well.

## Variables in C#

Variable is a name of a memory location in which application stores values.

We should create our variables by following examples:

* studentName
* subject
* work\_day ...

The wrong examples would be

* student Name
* work-day
* 1place...

We must mention that C# is case sensitive language so the **studentName** is not the same as the **StudentName**.

The C# language has its own set of reserved words, so-called keywords. We can't use them as a name for our variables. For the list of keywords, you can visit [keyword-list](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/).

In C#, we have variables divided into two categories: **Value** type and **Reference** type. The difference is that the value type variables stores their values inside their own memory locations, but the memory location for the reference type variables contains only address to the dynamic memory location where the value is stored.

Let’s see how the value types behave in a graphic example:



Let’s do the same for the reference types:



## Variable Declarations and Expressions

We can accomplish variable declaration in following way:

<data type> <variable name> ; or <data type> <variable name>, <variable name> ... ;

So few examples would be:

class Program

{

static void Main(string[] args)

{

int age;

double temperature, change;

Student student;

}

}

Just with the declaration, we can't assign a value to a value type variables. To do that, we need to use expressions in addition:

<data type> <variable name> = <expression> ;

class Program

{

static void Main(string[] args)

{

int x = 5;

int y = 145 + x;

char p = 'p';

p = 'A';

}

}

To add a value for the reference type of variable, we need to use the **new** keyword in expression part:

class Program

{

static void Main(string[] args)

{

Student student = new Student("John", 25);

}

}

## Conclusion

Now we have learned how to declare our variables and how to assign values to them as well.

In a next post, we are going to talk about operators in C#.

# Operators in C#

The most used operators in C# are:

|  |  |
| --- | --- |
| Category | Operators |
| Aritmetic | + - \* / % |
| Relational | == != < <= > >= |
| Logical | ! && || |

## Arithmetic Operators

Arithmetic operators are defined for all the numeric data types. Operators +, -, \*, / represent the basic binary arithmetic operations (addition, subtraction, multiplication and division). Operator % is the remainder after division. The important thing to notice is that the + operator behaves differently with the number and string types. With numbers 5 + 5 is 10. But with strings „5“ + „5“ is „55“. So with the number type, it is an addition operator but with the string type, it is a concatenation operator.

## Relational Operators

All the relational operators return a true or false result. They are used to compare expressions or variables from both sides of the relational operator. These operators have a lower priority than arithmetic ones. So in the following example: x\*a-8\*b>y+5\*z; the left side is calculated first then the right side and then they are compared.

For the value type variables and the strings, the == (equality) operator will return true only if they are the same, otherwise, it will return false. But if variables are of reference types then the == operator will return true only if those two variables point to the same memory location, otherwise, it will return false.

So let's see this through an example:



We see that a and b are the equal as well as the s1 and s2. But the student1 and student2 are not equal because they point to different memory locations. But if we create another variable of type Student and assign the value of student1 variable to it, the == operator will return true:



## Logical Operators

The logical operators && (and) and || (or) serve to connect logical values. Expression <expression1>&&<expression2> is true only if both expressions are true. Expression <expression1>||<expression2> is false only if both expressions are false, otherwise, it is true.

The ! (negation) operator negates logical value it is applied on. It has the highest priority of all the operators mentioned. So the expression !logicalValue will be false only if logicalValue is true and vice verse.

## Increment and Decrement Operators

In the C# language, we can use operators than increments and decrements the variable value by 1. Those operators are ++ and --. So, the better way of writing this code:

static void Main(string[] args)

{

int a = 15;

a = a + 1; //now it is 16

}

Is to write it like this:

static void Main(string[] args)

{

int a = 15;

a++; //now it is 16

}

The same applies for the -- operator.

These two operators have the prefix --variable, ++variable and suffix variable--, variable++ notations. Even though both notations will change the value by 1, the result will be different. This is easier to explain through an example:



What we notice is that the prefix notation will decrement the value of "a" variable first and then assign that value to the "b" variable. But the expression with suffix notation is different. The value of the "c" variable is assigned to the "d" variable first and then decremented by 1.

The same applies for the increment operator:



Excellent. Now we have more knowledge about operators in C#. In a next part, we are going to talk about the type conversions in C#.

# Type Conversion

In C#, data can be converted from one type to another by using implicit conversion (automatic) and explicit conversion (programmers take control of this conversion).

## Implicit Conversion

Many different data could be interpreted by using different types. For example, number 74 can be interpreted as an integer but also as double (74.0). We can differentiate two situations in which implicit conversion applies.

The first one is when we calculate an expression, the compiler automatically adapt data types that we use in that expression:

class Program

{

static void Main(string[] args)

{

double b = 12.45;

int x = 10;

b = b + x;

}

}

In the code above the „b“ variable is of type double and „x“ is of type int. In the expression b + x, the compiler implicitly converts "x" from int to double and then it assigns a result to the "b".

The second situation for a conversion is when the compiler stores the result to a variable:



In this example, we see that both „x“ and „y“ are of type int, but the result is of type double.

## Explicit Conversion

For the explicit conversion, we need to write additional code to convert one type to another. We have two different ways, by using a cast operator or by using class Convert.

Let's look at the following example:



The compiler complains about an invalid conversion. What we are missing here is the cast operator, so let's use it:



By using the cast „(int)“ operator, our compiler is not complaining anymore. But what we see is that our result is not what we have expected. But this is the correct result. It is very important to understand that the cast operator can shrink data when we convert the type with the larger value scope to a type with the smaller value scope.

As we said, we can use the Convert class with its static methods, to explicitly convert one base type to another base type:



# Linear Structures

In this article, we are going to show how to solve simple problems by using linear structures. We are going to accept inputs, work with them and print the output to the console window.

So let’s start.

Example 1: We need to print out the sum of two integer values which user inputs in the console window.

Let’s create a new console application and name it SumGenerator. Then let’s type this code inside the Main method:

namespace SumGenerator

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Write the first integer:");

int first = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Write the second integer:");

int second = Convert.ToInt32(Console.ReadLine());

int result = first + second;

Console.WriteLine($"The result is {result}");

Console.ReadKey();

}

}

}

With the Console.WriteLine() statement, we display the message on the console window and move to the next line. TheConsole.ReadLine() statement will read our input, but it is of type string and what we need is an int type. So, we need to convert it with the Convert.ToInt32() statement. Finally, we calculate the sum and print it out. The Console.ReadKey() statement is here just to keep our console window open.

Let’s press F5 to start our application and enter two integer numbers:



Example 2: Write an application which for two provided inputs (name and last name), prints out the full name in a format name <space> last name.

Let’s create a new console application and to write this code:

namespace FullNameGenerator

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("What is your first name:");

string name = Console.ReadLine();

Console.WriteLine("What is your last name:");

string lastName = Console.ReadLine();

string fullName = name + " " + lastName;

Console.WriteLine($"Your full name is: {fullName}");

Console.ReadKey();

}

}

}

Excellent.

Now we know how to manipulate with inputs in our programmes and how to display the result in the console window.

# Work with Strings

The string type represents a character array which length is determined by the Length property. All the character positions inside that string are enumerated from 0 to Length-1. C# provides for us many methods to work with strings and we are going to examine the most used methods.

**Substring(int startIndex)** is the method which returns part of the string from startIndex to the end of the string.

**Substring(int startIndex, int length)** is the method which returns part of the string with defined length from the startIndex.

Let’s see this in practice:

class Program

{

static void Main(string[] args)

{

string testString = "this is some string to use it for our example.";

string partWithoutLength = testString.Substring(10);

string partWithLength = testString.Substring(5, 10);

Console.WriteLine(partWithoutLength);

Console.WriteLine(partWithLength);

Console.ReadKey();

}

}

**IndexOf()** is the method that returns the integer position of the character’s or string’s first appearance in the string. If that value doesn't exist, the method will return -1.

There are different overloads of this method: IndexOf(char value), IndexOf(string value), IndexOf(char value, int startIndex), IndexOf(string value, int startIndex) etc. If we use this method with the startIndex parameter, we will not search from the beginning of the string but from that position to the end:

int charPosition = testString.IndexOf('i');

int stringPosition = testString.IndexOf("some");

int charPosWithStartIndex = testString.IndexOf('s', 10);

int stringPosWithStartIndex = testString.IndexOf("some", 10);

**LastIndexOf()** is the method that returns the position of the last appearance of character or string value. This method has the same overloads as the IndexOf method:

int lastPosition = testString.LastIndexOf('o');

int stringLastPosition = testString.LastIndexOf("is");

**Contains(string value)** is the method that returns true if a string contains the value, otherwise, it will return false:

bool containsResult = testString.Contains("for");

**StartsWith(string value)** is the method which returns true if a string starts with the value, otherwise, returns false. As opposed to this method the **EndsWith(string value)** method returns true if a string ends with the value, otherwise, returns false.

bool startsWithResult = testString.StartsWith("bad");

bool endsWithResult = testString.EndsWith("example");

**Remove(int startIndex)** method removes characters from the string from the startIndex position to the end of the string and returns that new string. There is an overloaded method **Remove(int startIndex, int count)** which removes a specified number of characters from the string from the starting index position. With the count parameter we deciede how many characters we want to delete:

string loweredString = testString.Remove(10);

string loweredStringWithCount = testString.Remove(10, 9);

**Insert(int startIndex, string value)** is the method that inserts the value into the string from the startIndex position and returns a modified string:

string stringWithInsert = testString.Insert(13, "UPDATED ");

**ToLower()** returns a new string with all the lower case letters:

string lowerCaseString = testString.ToLower();

**ToUpper()** returns a new string with all the upper case letters:

string upperCaseString = testString.ToUpper();

Example 1: Create an application in which accepts as input name and last name space separated, and then prints out name in one row and last name in second row:

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter your full name, blank space separated");

string fullName = Console.ReadLine();

int blankPosition = fullName.IndexOf(' ');

string name = fullName.Substring(0, blankPosition);

string lastName = fullName.Substring(blankPosition + 1);

Console.WriteLine(name);

Console.WriteLine(lastName);

Console.ReadKey();

}

}

The result:



Example 2: Create an application that accepts as input a sentence and removes the first and last word of that sentence:

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter your sentence: ");

string sentence = Console.ReadLine();

int firstBlankPosition = sentence.IndexOf(' ');

string withoutFirstWord = sentence.Remove(0, firstBlankPosition + 1);

int lastBlankPosition = withoutFirstWord.LastIndexOf(' ');

string withoutFirstAndLast = withoutFirstWord.Remove(lastBlankPosition);

Console.WriteLine(withoutFirstAndLast);

Console.ReadKey();

}

}



# Conditions in C#

In this article, we are going to talk about condition statements in C#. We will learn how to write simple condition statements, nested condition statements, and multiple condition statements.

## Basic Condition Statements

If we want to execute some expression but only if some condition is met, then we need to use conditional statements. To create such a statement we need to use if and else keywords.

We can create a conditional statement like this:

if (condition)

{

< expression1 > ;

}

else

{

< expression2 > ;

}

The condition is a logical expression which can result in true or false. If it is true then the <expression1> will be executed, otherwise, <expression2> will be executed. Behind every expression, we need to place the ";" sign.

We can execute more expressions if the condition is true or false:

if (condition)

{

< expression1 > ;

< expression2 > ;

}

else

{

< expression3 > ;

< expression4 > ;

}

Example 1: Create an application which determines the grater number of two integer inputs:

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter the first number: ");

int first = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter the second number: ");

int second = Convert.ToInt32(Console.ReadLine());

if(first > second)

{

Console.WriteLine($"The greater number is {first}");

}

else

{

Console.WriteLine($"The greater number is {second}");

}

Console.ReadKey();

}

}



We don’t have to use only if and else keywords in conditional statements, we can add another condition by adding „else if“ block part:

if(condition1)

{

< expression 1 > ;

}

else if(condition 2)

{

< expression 2 > ;

}

.

else if(condition n)

{

<expression n>

}

.

else

{

< expression k > ;

}

Example 2: Create an application which takes any string and the font color (r for red, g for green, o for other) as inputs. Then it needs to print out that string with the selected color:

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter your random string: ");

string sentence = Console.ReadLine();

Console.WriteLine("Choose your color: r for Red, g for Green, o for Other");

char color = Convert.ToChar(Console.ReadLine());

if(color == 'r')

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine(sentence);

}

else if(color == 'g')

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine(sentence);

}

else

{

Console.ForegroundColor = ConsoleColor.Blue;

Console.WriteLine(sentence);

}

Console.ReadKey();

}

}



## Nested Conditional Statements

In C# we can write conditional statement inside conditional statement if our project requires that from us. So, the base syntax looks like this:

if (condition)

{

if (condition2)

{

< expression1 > ;

}

else

{

< expression2 > ;

}

}

else

{

< expression3 > ;

}

Example 3: Create an application in which the user enters a number between 1 and 100. If the number is lower then 50, our application will output multiplication by 5. But if a number is greater then 50 then for even number application will output multiplication by 2 and for an odd number application will output multiplication by 3:

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter your number: ");

int number = Convert.ToInt32(Console.ReadLine());

if (number > 50)

{

if(number % 2 == 0) //reminder in division with two for even numbers is always a zero.

{

Console.WriteLine(number \* 2);

}

else

{

Console.WriteLine(number \* 3);

}

}

else

{

Console.WriteLine(number \* 5);

}

Console.ReadKey();

}

}



## Multiple Branching

In a situation where we need more than one or two conditions to execute some expression, using multiple branching could be an advance. To use multiple branching in C# we need to use switch and case keywords:

switch (expression)

{

case value1:

<expression 1> ;

break;

case value2:

<expression 2> ;

break;

default:

< expression3>;

break;

}

Example 4: Create an application which accepts month number as an input and prints out the number of days in that month:



# While Loop

While loop is a loop with a precondition. This means that we are checking a condition first and then if a condition returns true, we execute our expression:

while(condition)

{

< expression > ;

}

Example 1: Create an application which calculates the sum of all the numbers from n to m (inputs from a user):

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter the integer n number:");

int n = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter the integer m number");

int m = Convert.ToInt32(Console.ReadLine());

int sum = 0;

while(n <= m)

{

sum += n;

n++;

}

Console.WriteLine($"Sum from n to me is {sum}");

Console.ReadKey();

}

}



So let's explain the code above.

Because we calculate the sum of all numbers from "n" to "m", we need to have a variable to store that value. It needs to be initialized with a zero at a beginning. Without that, our app will fail to build due to the sum variable being unassigned.

In a while loop, we are going through all the numbers from n to m and adding every number to the sum variable. We are using this expression: sum += n; which is a shorter for sum = sum + n;

Finally, we need to increment the n variable by 1. Without that, we would have an infinite loop because the value of the n variable would always be lesser than the value of the „m“ variable.

# For Loop

For loop is another loop with a precondition. We use the following syntax to write it in the C#:

for (initialization; condition; progression;)

{

<loop body > ;

}

We use initialization on the beginning of the loop and it serves the purpose of initializing the variable with a value. The condition is used to determine when the loop is completed. Progression is a part in which we increment or decrement our variable initialized in the initialization part. The body consists of all the expressions we need to execute as long as the condition is true.

It is important to know that the order of execution is: Initialization, Condition, Loop Body, Progression.

Example 1: Create an application which calculates the sum of all the numbers from n to m (inputs from a user):

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter the integer n number:");

int n = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter the integer m number");

int m = Convert.ToInt32(Console.ReadLine());

int sum = 0;

for(int i = n; i <= m; i++)

{

sum += i;

}

Console.WriteLine($"Sum from n to m is {sum}");

Console.ReadKey();

}

}



Example 2: Create an application that prints out all the integer numbers from n to 1:

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter number n that is greater than 1: ");

int n = Convert.ToInt32(Console.ReadLine());

for (int i = n; i >= 1; i--)

{

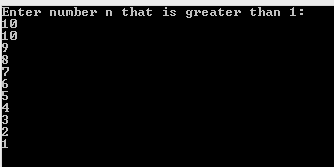
Console.WriteLine(i);

}

Console.ReadKey();

}

}



# Do-While Loop

The do-while loop is a loop with postcondition. What this means is that the loop body is executed first and the condition is checked after, totally opposite from the previous loop examples.

We can implement this loop in the following way:

do

{

< expression > ;

} while (condition);

Example 1: Create an application which calculates the sum of all the numbers from n to m (inputs from a user):

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter the integer n number:");

int n = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter the integer m number");

int m = Convert.ToInt32(Console.ReadLine());

int sum = 0;

do

{

sum += n;

n++;

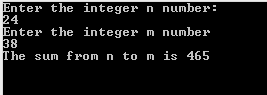
} while (n <= m);

Console.WriteLine($"The sum from n to m is {sum}");

Console.ReadKey();

}

}



Example 2: Create an application which prints out the summary of all the even numbers to n:

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter the upper border number n: ");

int n = Convert.ToInt32(Console.ReadLine());

int sum = 2;

int startingNumber = 4;

do

{

sum += startingNumber;

startingNumber += 2;

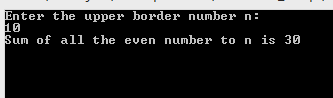
}while (startingNumber <= n);

Console.WriteLine($"Sum of all the even number to n is {sum}");

Console.ReadKey();

}

}



# Handling Exceptions

Exceptions are the problems that could appear unplanned in our code while we develop our project. That’s why we call these exceptions unhandled exceptions. If they are not handled, they will cause our application to stop working and will throw one of the exception messages. That is not something we want in our code.

## Try-Catch Block

C# provides to us a built-in support to handle those unhandled exceptions by using a try-catch block of code:

try

{

// expressions that could cause an exception

}

catch(Exception ex)

{

// handle exception

}

In the try block, we write our code and the catch block will handle all the exceptions that could arise in the try block. This way our program won’t stop at all and we can show some meaningful message to the user.

Let’s see how our program works without and with handling exceptions.

Example 1: Create an application which prints out the square root of the integer number entered by the user:

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Enter your number: ");

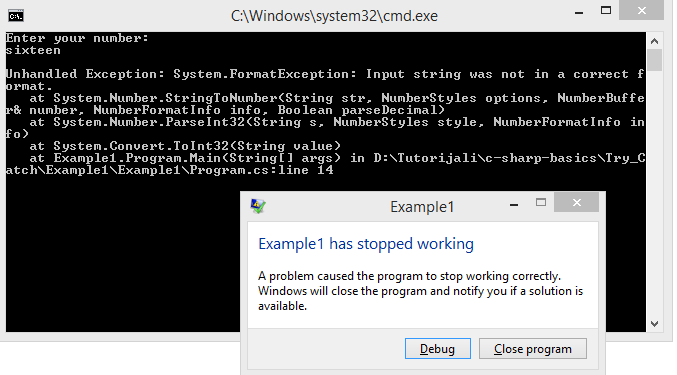
int num = Convert.ToInt32(Console.ReadLine());

Console.WriteLine(Math.Sqrt(num));

}

}

This code is going to work just fine if a user enters an integer number, but look at what is going to happen if a user enters a string:



We see that our application has stopped working. This is very bad for a user. So, let’s implement the same code but with the try-catch block:

class Program

{

static void Main(string[] args)

{

try

{

Console.WriteLine("Enter your number: ");

int num = Convert.ToInt32(Console.ReadLine());

Console.WriteLine(Math.Sqrt(num));

Console.ReadKey();

}

catch (Exception ex)

{

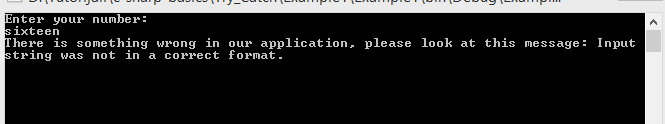
Console.WriteLine($"There is something wrong in our application, please look at this message: {ex.Message}");

Console.ReadKey();

}

}

}



As we can see, our app didn’t stop and we have a nice readable message for our user, which is much better then the previous example.

C# has its own set of specific exceptions which we can use in our application. Some of them are: NullReferenceException, ArgumentOutOfRangeException, InvalidCastException, FileNotFoundException, DevideByZeroException, FormatException, InvalidOperationException etc.

We can use them in this way:

public class Program

{

public static void Main()

{

try

{

//Code in here that could cause an exception

}

catch (DivideByZeroException ex)

{

Console.Write("Cannot divide by zero. Please try again.");

}

catch (InvalidOperationException ex)

{

Console.Write("Not a valid number. Please try again.");

}

catch (FormatException ex)

{

Console.Write("Not a valid number. Please try again.");

}

catch(Exception ex)

{

Console.Write("Any exception that previous catch blocks didn’t handle.");

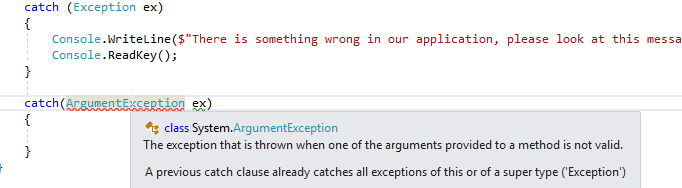
}

Console.ReadKey();

}

}

It is very important to place the specific catch blocks before the global catch block, otherwise, our compiler will complain:



Excellent.

Now we know how to write a safe code and how to handle errors in our app.

# Access Modifiers

In this article we are going to explain different types of access modifiers in C# and what is their purpose. It is important to have this knowledge in order to work easier with methods in a next article.

Access modifiers specify the accessibility of an object and all of its members in the C# project. All the C# types have access modifiers implemented, even if they are not mentioned (default access modifier is applied then).

## Access Modifiers Types

C# provides five types of access modifiers: private, public, protected, internal, protected-internal.

Objects that implement **private** access modifier are accessible only inside a class or a structure. We can't access them outside the class they are created.

Objects that implement **public** access modifier are accessible from everywhere in our project. There are no accessibility restrictions.

The **protected** keyword implies that the object is accessible inside the class and in all classes that derive from that class.

The **internal** keyword specifies that the object is accessible only inside its own assembly but not in other assemblies.

The protected **internal access** modifier is a combination of protected and internal.

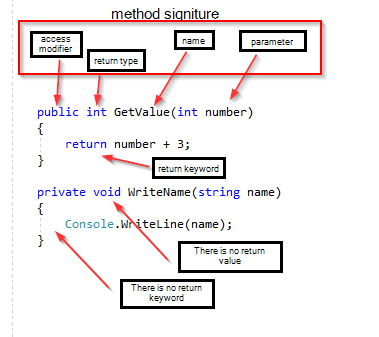
# Methods

A method is a code block that we can use to extract part of our code to reuse them thus making our classes more readable and easier to maintain. We can execute all that code inside a method once we call that method by using its name and specifying required arguments.

## Method Signatures

We can declare our methods by specifying the method signature that consists of the **access modifier** (public, private...), a **return value** (void, int, double...), a **name of a method**, and **method parameters**. If we want our method to have an implementation, it needs to have two curly brackets to specifies the body of the method. We place our code between those curly brackets.

If our method returns any value (for example an int) it needs to have as a return type an int keyword and inside curly brackets the return keyword. If it doesn’t return anything, then as a return value we use void keyword and without return keyword in a method body:



In our project, we **can** have two different methods with the same name, but we **can’t** have two different methods with the same method signature. At least one part of the method signature needs to be different. When we have two or more methods with the same name but different signature, that’s called **Method Overloading.**

## Parameters and Arguments

In our previous example, we have seen that our methods accept only one parameter, but that doesn't have to be the case. We can create a method signature with as many parameters as we need:

public void WriteAllNumbers(int a, int b, int c)

{

Console.WriteLine($"{a} {b} {c}");

}

It is important that every parameter has its own type, name and that they are comma separated.

When we create a method in the signature, we create parameters (imagine them as the placeholders for the value of the same type). But, when we call that method we are passing real values (arguments) for those parameters:

WriteAllNumbers(15, 16, 67);

Example 1: Create an application which prints out the sum, subtraction, and multiplication of the two inputs:

class Program

{

public static void Sum(int first, int second) //method needs to be static because we are calling it in a static Main method.

{

int result = first + second;

Console.WriteLine($"Sum result: {result}");

}

public static void Subtract(int first, int second)

{

int result = first - second;

Console.WriteLine($"Substraction result: {result}");

}

public static void Multiplication(int first, int second)

{

int result = first \* second;

Console.WriteLine($"Multiplication result: {result}");

}

static void Main(string[] args)

{

Console.WriteLine("Enter the first number: ");

int firstArgument = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter the second number: ");

int secondArgument = Convert.ToInt32(Console.ReadLine());

Sum(firstArgument, secondArgument);

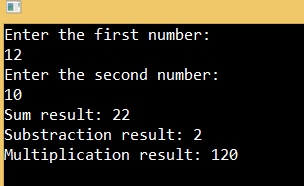
Subtract(firstArgument, secondArgument);

Multiplication(firstArgument, secondArgument);

Console.ReadKey();

}

}



# Ref and Out Keywords

In a previous post, we were sending a value type arguments while calling our methods. Why is the type of the argument important here? Well, when we send the argument of type int, double, decimal etc (basic value types), we are not sending the actual value but its copy. This means that our original values are not changed inside the methods, because we send a completely new copy of a original value and all the operations inside a method are executed upon the copy value.

We can show this in an example:

class Program

{

public static void ChangeAndWrite(int number)

{

number = 10;

Console.WriteLine($"Inside ChangeAndWrite method, number value is: {number}");

}

static void Main(string[] args)

{

int number = 5;

Console.WriteLine($"Value of the number prior to ChangeAndWrite call is: {number}");

ChangeAndWrite(number);

Console.WriteLine($"Value of the number after the ChangeAndWrite call is: {number}");

Console.ReadKey();

}

}



As we can see, the value of the number variable change only inside the method but the original value is the same as before calling the ChangeAndWrite method. And again, this is because we are sending the exact copy of the original value.

## Using Ref and Out Keywords

But we can change this. If we want to change the original values inside our methods, we can do that by using ref and out keywords inside the method signature and inside the method call as well.

We can use the ref keyword only if the variable which we use as an argument is initialized before calling a method. It is a different situation with the out keyword, we don't have to initialize it before calling a method but inside a method, it needs to receive a value.

So, to make thins simple. If we want to change the existing value of a variable inside a method, we are going to use the ref keyword. But, if we want to assign a completely new value to the variable inside a method, then we use the out keyword.

## Example1 for the Value Type

In the previous example, we saw how the value type variables behave if we don’t use the ref or out keywords. In this one, we are going to see the behavior of value type variables when we use those keywords:

class Program

{

public static void ChangeRef(ref int numberRef)

{

numberRef = 25;

Console.WriteLine($"Inside the ChangeRef method the numberRef is {numberRef}");

}

public static void ChangeOut( out int numberOut)

{

numberOut = 60;

Console.WriteLine($"Inside the ChangeOut method the numberOut is {numberOut}");

}

static void Main(string[] args)

{

int numberRef = 15;

Console.WriteLine($"Before calling the ChangeRef method the numberRef is {numberRef}");

ChangeRef(ref numberRef);

Console.WriteLine($"After calling the ChangeRef method the numberRef is {numberRef}");

Console.WriteLine();

int numberOut;

Console.WriteLine("Before calling the ChangeOut method the numberOut is unassigned");

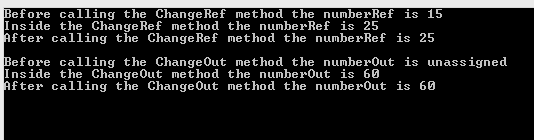
ChangeOut(out numberOut);

Console.WriteLine($"After calling the ChangeOut method the numberOut is {numberOut}");

Console.ReadKey();

}

}



## Example 2 for the Reference Type

We know from the previous post about types, that the reference types don’t store their value inside their own memory location but the address towards the memory location where the value is stored. Therefore when we send an argument as a reference type to the method and change that parameter, the original value is changed. This is because we are not sending the copy of the value but the copy of the address that points to the original value. This is the same thing as when we use the ref keyword with the value types.

Still, we can use the ref keyword with the reference types if we want to create a new object with the same address.

Let’s see all of this in an example:

class Program

{

public static void ChangeColor(Pen pen)

{

pen.Color = Color.Green;

Console.WriteLine($"Inside the ChangeColor method the color is {pen.Color}");

}

public static void CreateNewObjectWithoutRef(Pen pen)

{

pen = new Pen(Color.Red);

Console.WriteLine($"Inside the CreateNewObjectWithoutRef method the color of new pen object is {pen.Color}");

}

public static void CreateNewObjectWithRef(ref Pen pen)

{

pen = new Pen(Color.Yellow);

Console.WriteLine($"Inside the CreateNewObjectWithRef method the color of new pen object is {pen.Color}");

}

static void Main(string[] args)

{

Pen pen = new Pen(Color.Blue);

Console.WriteLine($"Before ChangeColor method: {pen.Color}");

ChangeColor(pen);

Console.WriteLine($"After the ChangeColor method: {pen.Color}");

Console.WriteLine();

Console.WriteLine($"Before CreateNewObjectWithoutRef method: {pen.Color}");

CreateNewObjectWithoutRef(pen);

Console.WriteLine($"After CreateNewObjectWithoutRef method: {pen.Color}");

Console.WriteLine();

Console.WriteLine($"Before CreateNewObjectWithRef method: {pen.Color}");

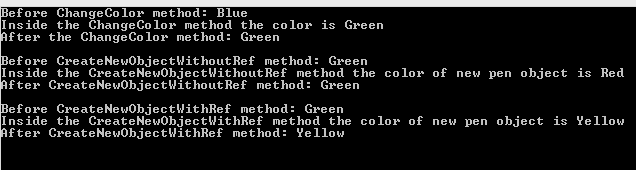
CreateNewObjectWithRef(ref pen);

Console.WriteLine($"After CreateNewObjectWithRef method: {pen.Color}");

Console.ReadKey();

}

}



In the first method, we are not using the ref keyword but the value is changed because we send the copy of the address where the original value is stored. In the second method, the original value stays the same because we create a new object inside method thus the new address. But in the third method, we are using the ref keyword and the original value changes because with the ref keyword we are copying the same address to the new object.

## Conclusion

Now we know how to use ref and out keywords with the value and reference types. This is quite a useful feature in C#, thus knowing how to work with those keywords is an advantage for the developers.

# Working with Files, StreamWriter and StreamReader

The StreamReader and StreamWriter classes enable the reading and writing actions to a file. Both of these classes exist in the System.IO namespace as well as many other classes for working with files and directories.

## Creating Objects for StreamWriter and StreamReader

To create objects for the StreamReader and StreamWriter classes we need to use the standard initialization for the reference data types. We can execute this initialization in a couple of ways but the most common is by only providing an address to the file:

StreamReader readerRelativePath = new StreamReader("test.txt");

StreamReader readerAbsolutePath = new StreamReader("C:\\MyProject\\test.txt");

StreamWriter writerRelativePath = new StreamWriter("test.txt");

StreamWriter writerAbsolutePath = new StreamWriter("C:\\MyProject\\test.txt");

As we can see from the code above, we can provide the relative or absolute path to our file. If we provide a relative path (just a name and extension) Visual Studio will place a file inside the projectName/bin/debug folder.

## StreamReader Methods

StreamReader contains many different methods to work with files but we are going to mention few of those.

The Read() method will return next sign as an integer number or -1 if we reached the end of the file. We can use explicit conversion (cast) to convert that integer into a char type:

static void Main(string[] args)

{

StreamReader sr = new StreamReader("test.txt");

int x;

char ch;

x = sr.Read();

while(x != -1)

{

ch = (char)x;

//do stuff here

x = sr.Read();

}

}

The ReadLine() method will return a whole line as a string. If we reached the end of the file it will return null:

static void Main(string[] args)

{

StreamReader sr = new StreamReader("test.txt");

string line = sr.ReadLine();

while(line != null)

{

//some coding

line = sr.ReadLine();

}

}

The ReadToEnd() method returns a whole file in one string. If there is nothing more to read it will return an empty string ("").

The Peek() method checks the next character in the file or if it finds nothing it will return -1:

static void Main(string[] args)

{

StreamReader sr = new StreamReader("test.txt");

string line;

while(sr.Peek() != -1)

{

line = sr.ReadLine();

//some coding

}

}

## StreamWriter Methods

The two most important methods for the StreamWriter class is the Write() and WriteLine(). With the Write() method we write a line inside a file but without moving to another line after. But with the WriteLine() method we write a line inside a file and moving to another line.

When we finish with the StreamWriter class, we need to close it with the Close() method.

Example1: Create an application that writes five random numbers from 1 to 100 to a file named numbers.txt. Then it will read all the numbers from that file, print them out and print the maximum number:

class Program

{

public static void WriteToFile(string path)

{

StreamWriter sw = new StreamWriter(path);

Random r = new Random(); //class to generate random numbers

for(int i = 1; i <= 5; i++)

{

sw.WriteLine(r.Next(1,101));

}

sw.Close();

}

public static void PrintNumbersAndMax(string path)

{

StreamReader sr = new StreamReader(path);

string line = sr.ReadLine();

Console.WriteLine(line);

int max = Convert.ToInt32(line);

while ((line = sr.ReadLine()) != null)

{

Console.WriteLine(line);

int temp = Convert.ToInt32(line);

if(temp > max)

{

max = temp;

}

}

Console.WriteLine($"Max number is: {max}");

}

static void Main(string[] args)

{

WriteToFile("numbers.txt");

PrintNumbersAndMax("numbers.txt");

Console.ReadLine();

}

}

# Recursion and Recursive Methods

Recursion is a concept in which method call itself. Every recursive method needs to be terminated, therefore, we need to write a condition in which we check is the termination condition satisfied. If we don't do that, a recursive method will create endless calls of itself.

Example 1: Create an application which calculates the sum of all the numbers from n to m recursively:

class Program

{

public static int CalculateSumRecursively(int n, int m)

{

int sum = n;

if(n < m)

{

n++;

return sum += CalculateSumRecursively(n, m);

}

return sum;

}

static void Main(string[] args)

{

Console.WriteLine("Enter number n: ");

int n = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter number m: ");

int m = Convert.ToInt32(Console.ReadLine());

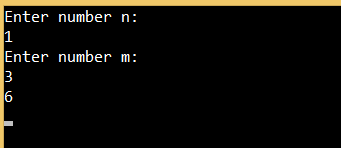
int sum = CalculateSumRecursively(n, m);

Console.WriteLine(sum);

Console.ReadKey();

}

}



## Code Explanation

The method CalculateSumRecursively is our recursive method that calculates the sum of the numbers from "n" to "m". The first thing we do is to set our sum to the value of n. Then, we check if the value of n is less then the value of m. If it is we increase the value of n by 1 and add to our sum a result of the same method but with the increased n. If it is not, we just return the value of the sum variable.

The C# will reserve a memory storage for every recursive method so that the values from the previous method are not overridden.

So let's see our example through the diagram:



## Additional Example

Let’s practice some more with the Example2: Create an application which prints out how many times the number can be divided by 2:

class Program

{

public static int CountDivisions(double number)

{

int count = 0;

if(number % 2 == 0)

{

count++;

number /= 2;

return count += CountDivisions(number);

}

return count;

}

static void Main(string[] args)

{

Console.WriteLine("Enter your number: ");

double number = Convert.ToDouble(Console.ReadLine());

int count = CountDivisions(number);

Console.WriteLine($"Total number of divisions: {count}");

Console.ReadKey();

}

}

Excellent.

Now we have a good knowledge of recursion and recursive methods.

# Arrays

We often have a situation where we need to use a couple variables of the same type and to execute the same operation on each of them. Imagine if we need hundreds of them or even more, well we will concur that creating a hundred variables of an int type, for example, is not a solution.

Luckily, C# provides us with the complex type named array.

Arrays are the reference data types that consist of data of the same type, arranged in the known order. We can access any information inside an array by stating the name of the array and the position of the data. The data position inside an array is called **Index.** In arrays indexes are zero-based. That means that the first element is stored on the zero index and the last element is on the array. Length – 1 index. So, if our array has 5 elements, indexes are addressed from 0 to 4.

## Array Declaration and Initialization

To declare an array we state the type of that array then the square brackets and finally the name of that array:

int[] numbers;

Pen[] pens;

**The important thing to know is that no matter whether we store the reference type or the value type data inside an array, the array is always a reference data type.**

To initialize our arrays we need to write a new keyword then the type and finally the square brackets with the array capacity inside:

numbers = new int[5];

pens = new Pen[5];

In a first example, we store the int type (value type) inside the numbers array thus reserving the space in our memory for five integers. But in the second example, we are reserving the space in our memory for five Pen types (reference types) so we are not storing their values bat their references. All the Pen values are null for now.

Until now, we have just allocated the memory for our values, we didn’t actually add those values at all. So, to finish the initialization process we need to add values to our arrays. The most common way is to declare, allocate and initialize an array in one line of code:

int[] arrayExample = new int[5] { 4, 5, 7, 8, 3};

Pen[] penArrayExample = new Pen[3] { new Pen(Color.Red), new Pen(Color.Green), new Pen(Color.Blue) };

We can use the indexes as well, to populate an array:

int[] numbers = new int[2];

numbers[0] = 5; numbers[1] = 7;

## Array Manipulation

To manipulate with an array, we can use a for loop. With the for loop we are using indexes to access each element of an array:

static void Main(string[] args)

{

int[] numbers = new int[5] { 4, 5, 7, 8, 3};

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

{

Console.WriteLine(numbers[i]);

}

}

We can do the same thing but with the foreach loop. Difference between these two approaches is because with the for loop we are using indexes to access elements (variable i), but with the foreach loop we are not using indexes but the actual values:

static void Main(string[] args)

{

int[] numbers = new int[5] { 4, 5, 7, 8, 3};

foreach (int i in numbers)

{

Console.WriteLine(i);

}

}

## Examples

Example1: Create an application in which we will create an array of n elements, populate that array with the random integer numbers, print out all those numbers and the sum:

class Program

{

//array is a reference type so every action in this method will affect original array

public static void PopulateArray(int[] numbers)

{

Random r = new Random();

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

{

numbers[i] = r.Next(1, 101);

Console.WriteLine($"The {i+1}. element is {numbers[i]}");

}

}

public static void CalculateSum(int[] numbers)

{

int sum = 0;

foreach (int i in numbers)

{

sum += i;

}

Console.WriteLine($"The sum of all the elements is {sum}");

}

static void Main(string[] args)

{

Console.WriteLine("Enter an array capacity: ");

int capacity = Convert.ToInt32(Console.ReadLine());

int[] numbers = new int[capacity];

PopulateArray(numbers);

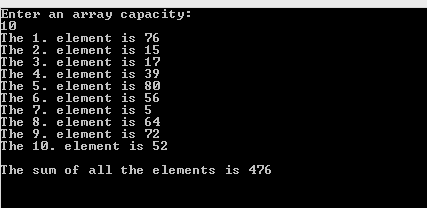
Console.WriteLine();

CalculateSum(numbers);

Console.ReadKey();

}

}



## Parameter Arrays

A params array enables us to pass a variable number of arguments to a method. To create a params array we must specify the params keyword when we specify the parameters for our method:

public static void TestMethod(params int[] numbers)

{

//method body

}

The effect of the params keyword is that it allows us to send any number of arguments to the method’s parameter without creating an array:

static void Main(string[] args)

{

TestMethod(1, 2, 3);

}

Even though a params array is very useful, we still have some limitations while working with them:

* We can’t use params keyword to work with two-dimensional arrays
* Method overloading is not possible solely with the params keyword
* We can’t specify ref or out keywords with params arrays
* A params array has to be the last parameter in our method
* A non-params method always take priority against the params methods

Example2: Create an application that prints out the minimum of all the numbers sent to the PrintMin method:

class Program

{

public static void PrintMin(params int[] numbers)

{

int min = numbers[0];

for(int i=1; i < numbers.Length; i++)

{

if(min > numbers[i])

{

min = numbers[i];

}

}

Console.WriteLine(min);

}

static void Main(string[] args)

{

PrintMin(49, 58, 12, 98, 47, 13);

Console.ReadKey();

}

}