C# Developer Course

Tod Vachev

Codestars.com

About the course

* 22+ hours of video content
* 120+ video lectures
* 19+ sections
* 100+ exercises

Who is it for?

* Absolute beginners
* People with some experience
* People who want to refresh their c# (coding) skills

What you will learn

* Writing high quality code
* Incorporate the best coding practices in your style
* Exercises to practice in almost every lecture
* After you have tried, you will see my solution

About me

* Tod Vachev, Bestselling Udemy Instructor
* 15+ courses, 8+ on c#

Software needed

* The course provides guides on how to install required software

About the course

Variable and Data Types

* Backbone of programming
* Used to store data
* A variety of data types available in c#

The Console

* Control all aspects of the console
* Format outputs
* Change colors
* Change dimensions

Operators and Expressions conditional statements

* Logical Expressions
* True and False results
* Control the flow of code with conditional statements
* Example: logging into a website

Loops

* Re-execute a piece of code
* All application ever created have an underlying loop
* Example: keep re-entering your credentials

Methods

* Blocks of code
* Reusable
* Example: code to sort numbers

Arrays and Lists

* Essential when working with data
* Arrays and lists both are collections of data, but do it in a different way
* Example: images are 3 dimensional arrays

String Processing

* Strings are information stored as text
* Extract specific part – substrings
* Replace, Remove, Trim and more

Exceptions

* Unexpected errors while your code is running
* Can be handled

Object Oriented Programming Basics

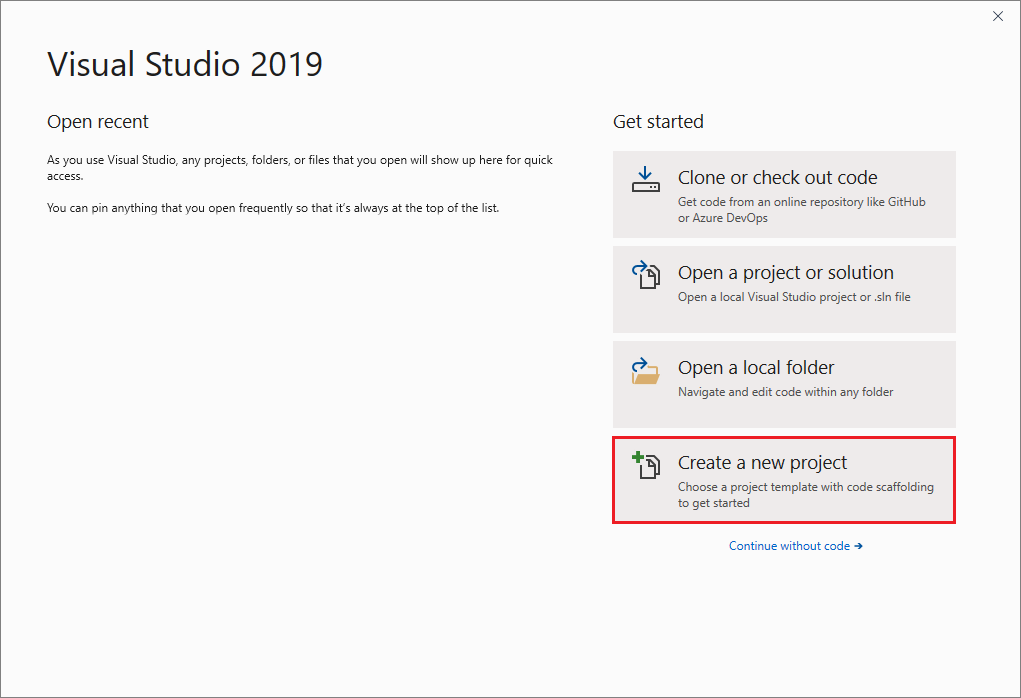
* Building blocks of a class
* Fields, Properties, Constructors, Methods, Namespaces, Access modifiers, Static members and more

4 Pillars of Object-Oriented Programming

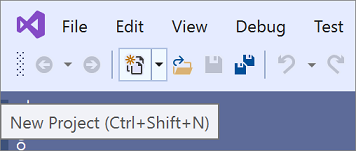
* Inheritance – eliminate redundant code
* Abstraction & Encapsulation – protect and organize
* Polymorphism – a class can take the form of another class

**Creating a Project in Visual Studio**

There are multiple ways to create a new project in visual studio 2019. When you first open visual studio, the start window appears, and from there, you can select **Create a new project**.



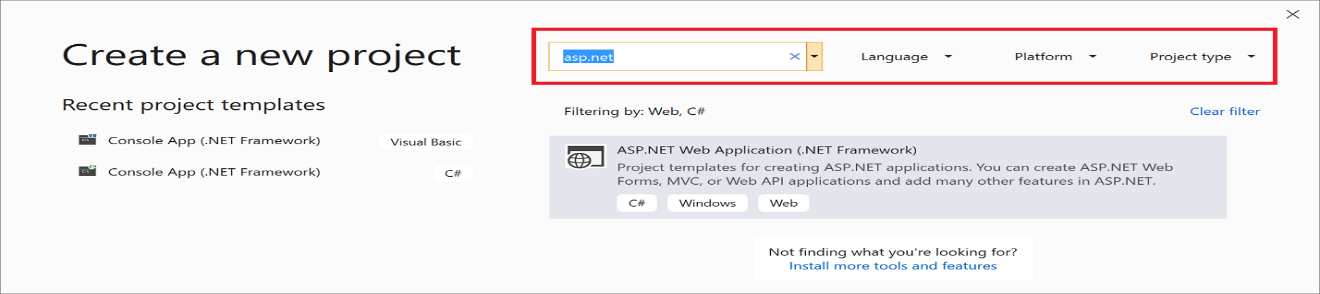
If the visual studio development environment is already open, you can create a new project by choosing **File** > **New** > **Project** on the menu bar. You can also click the **New Project** button on the toolbar, or press **Ctrl**+**Shift**+N.



**Select a template type**

On the **Create a new project** page, a list of your recently selected templates appears on the left. The templates are sorted by most recently used.

If you’re not selecting from the recently used templates, you can filter all available project templates by **Language** (e.g., C#, or C++), **Platform** (e.g., Windows, Linux or Azure), and **Project type** (e.g., Desktop or Web). You can also enter search text into the search box to further filter the templates, e.g., **asp.net**.

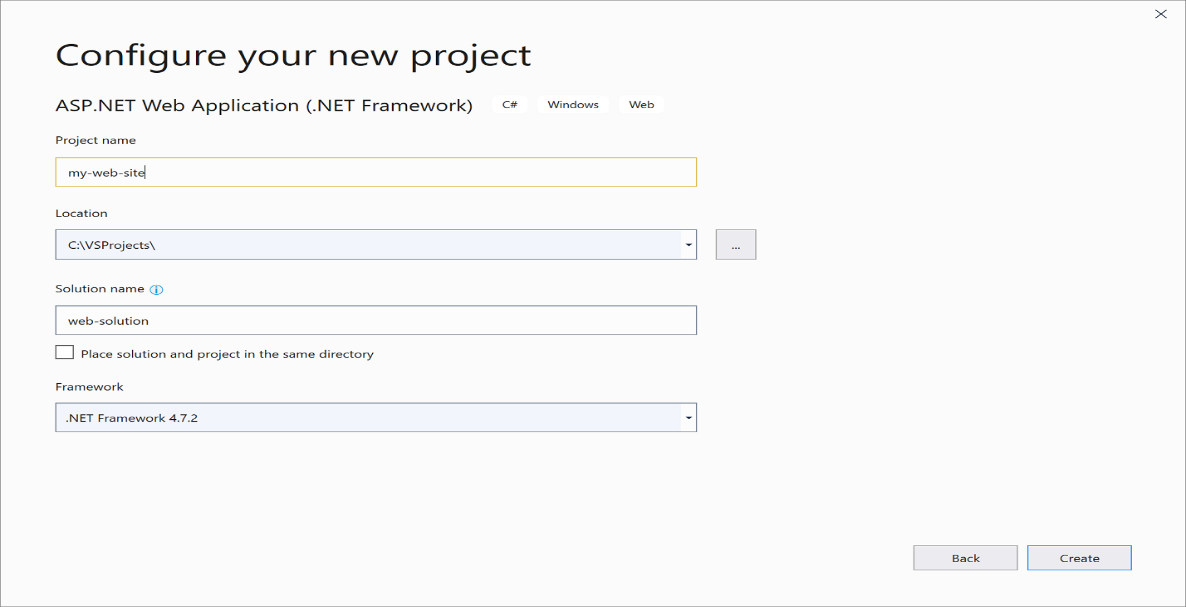


The tags that appear under each template correspond to the three-dropdown filter (Language, Platform, and Project type).

Select a template and then click **Next**.

**Configure your project**

The **Configure your new project** page has options to name your project (and solution), select a disk location, and select a Framework version (if applicable to the template you chose).



**Add additional projects to a solution**

If you want to add an additional project to a solution, right-click the solution node in **Solution Explorer** and then select **Add** > **New Project**.

**Section 2: Primitive Data types and variables**

**What is variable**

A **variable** is name given to a storage area that is used to store values of various data types. Each variable in C# needs to have a specific type, which determines the size and layout of the variable’s memory.

For example, a variable can be of the type string, which means that it will be used to store a string value. Based on the data type, specific operations can be carried out on the variable.

**Concatenating Value**

Concatenating strings is appending or inserting one string to the end of another string. String in C# and .NET Core are immutable.

Here are the 6 ways to concatenate strings in C#

1. Using + operator
2. String Interpolation
3. String.Concatenate() method
4. String.Join() method
5. String.Format() method
6. stringBuilder.Append() method

**Concatenate String Using + operator**

The simplest method of adding two strings in C# is using + or += operators.

e.g.

string firstname = “john”;

string lastname =”peter”;

string freespace = “ “;

//concatenate to string variable

String name = firstname + freespace + lastname;

Console.WriteLine(name);

**String Interpolation**

String interpolation is a method to concatenate variables as a part of a string. Syntax of string interpolation starts with a ‘**$’** symbol and code variables are within a bracket {}.

e.g

//string interpolation

String author = “john”;

String book = “C# develop course”;

//use string interpolation to concatenate strings.

String bookaurthor = $”{author} is the author of {book}”;

Console.WriteLine(bookaurthor);

**Concatenate string using String.Concate method**

String.Concate() method concatenates two strings, two objects, and two arrays of strings and combinations of them.

e.g.

//string.concat method

String fName =“john”;

String lName =”peter”;

String Name = string.Concat(fName, lName);

//Concatenate Concat

String Name3 = string.Concat(string.Concat(fName, lName), “Tod”);

//Concatenate an array of strings

String[] arrstring = {“john”, “Chris love”, “peter walker”};

//Concatenate all strings of an array into one string

String arraystr = string.Concat(arresting);

Console.WriteLine(arraystr);

//concatenate a string and an array of string

String strArrstr = string.Concat(“Authors: “, authors);

**Concatenate string using String.Join method**

String.Join() method concatenates the elements of an array or the members of a collection, using the specified separator between each element or member. The array or the collection can be any data type including numbers and object. String.join method also allows you to concatenate string items of any data types with a range.

e.g.

//using string.Join(string, string[])

Int[] intarray = {1,2,3,4,5};

String separator = “, “;

String result = “Int, “;

result += String.Join(separator, intarray);

Console.WriteLine($”Result: {result}”);

**Concatenate string using String.Format method**

String.Format() method formats strings in a desired format by inserting objects and variables with specified space and alignments into another strings and literals. It is also often used to format strings into specific formats.

String.Format() method has 8 overloaded formats to provide options to format various objects and variables that allow various variables to format strings.

*Note:* in C# 6 or latest versions, string interpolation is recommended. String interpolation is more flexible and more readable and can achieve the same results without composite formatting.

* **Insert a single object in a string**

We can insert one or more objects and expressions in a string at a specified position using the String.Format method. The position in the starts at the 0th index.

For single object formatting, the first argument is the format and the second argument is the value of the object. The object is replaced at the {0} position in the string.

e.g.

//simple string.format

String date = string.Format(“Today’s date is {0}”, DateTime.Now);

* **Insert multiple objects in a string**

We can insert multiple objects or expressions in a string to give it a desired format.

e.g.

//string.Format with multiple objects

String author = “John walker”;

String book = “C# developer course”;

Int year = 2003;

Decimal price = 49.95m;

String publisher = “APress”;

//Book details

String bookDetails = string.Format(“{0} is the author of book {1} \n “ +

“published by {2} in year {3}. \n Book priceis ${4}. “, author, book, publisher, year, price);

Console.WriteLine(bookDetails);

**StringBuilder.Append method**

String object in .NET is immutable. What does that mean? It means every time you use one of the string class method, no matter if you use the same variable or a new variable, a new string object is created in memory. That means, a memory space is allocated for that new string in your computer memory. The more string methods you use to manipulate strings, the more memory space will be allocated in memory. That means in a string manipulation heavy code, if strings are not used wisely, it could lead to some serious app performance issues.

.NET provides the **System.Text.StringBuilder** class that can be used to modify strings without creating new string object. **StringBuilder** is highly recommended if you have more than a few hundred string concatenation operations in your code. **StringBuilder** is not recommended for a few string concatenation operations.

**StringBuilder** class is defined in the **System.Text** namespace.

**Using System.Text;**

**StringBuilder** constructor can take a string or can be no arguments.

StringBuilder builder = new StringBuilder(“John Walker”);

builder.Append(“, “);

builder.Append(“chris Brown”);

builder.Append(“, Daniel White”);

e.g2.

StringBuilder numbers = new StringBuilder(“John Walker”);

//create a string of 1000 number from 0 to 999

//separated by a comma and space

For(int counter = 0; counter <= 999; counter++)

{

numbers.Append(counter);

numbers.Append(“, “);

}

Console.WriteLine(numbers);

**What is primitive data type**

The **primitive data types** are number, string, Boolean, float etc.

* **Integer types -:** store whole numbers, positive or negative (such as 123 or -456), without decimal. Valid types are int and long. Even though there are many **numeric types in c#,** the most used for numbers are int (for whole numbers) and double (for floating point numbers).

**Characteristics of the integer types**

C# support the following predefined integer types:

|  |  |  |  |
| --- | --- | --- | --- |
| **C# type / keyword** | **Range** | **Size** | **.NET type** |
| sbyte | -128 to 127 | Signed 8-bit integer | System.SByte |
| byte | 0 to 255 | Unsigned 8-bit integer | System.Byte |
| Short | -32,768 to 32,767 | Signed 16-bit integer | System.int16 |
| Ushort | 0 to 65,535 | Unsigned 16-bit integer | System.uint16 |
| Int | -2,147,483,648 to 2,147,483,647 | Signed 32-bit integer | System.Int32 |
| Uint | 0 to 4,294,967,295 | Unsigned 32-bit integer | System.Uint32 |
| Long | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | Signed 64-bit integer | System.Int64 |
| Ulong | 0 to 18,446,744,073,709,551,615 | Unsigned 64-bit integer | System.Uint64 |
| Nint | Depends on platform | Signed 32-bit or 64-bit integer | System.IntPtr |
| Nuint | Depend on platform | Unsigned 32-bit or 64-bit integer | System.UIntptr |

* Byte – negative and positive, 8 bits, -128 to 127
* Sbyte – positive only, 8 bits, 0 to 255
* Int – negative and positive, 32 bits
* Uint – positive only, 32 bits
* Long – negative and positive, 64 bits
* Ulong – positive only, 64 bits

The **nint** and **nuint** types in the last two rows of the table are native-sized integers. They are represented internally by the indicated .NET types, but in each case the keyword and the .NET type are not interchangeable. The compiler provides operations and conversions for **uint** and **nuint** as integer types that it doesn’t provide for the pointer types **System.Intptr** and **System.UintPtr**.

**Floating points**

Floating-point numbers are numbers that have fractional parts (usually expressed with a decimal point).

Using a floating-point value is just as easy as using a n integer

**double** number;

just like integer, you can of course assign a value to it at the same time as declaring it;

**double** number = 70.0;

the same goes for the float and decimal types

**double** doubleeval = 70.0;

**float** floatval = 39.0f;

**decimal** decimalval = 34.0m;

**Note**

The **“f”** and **“m”** after the number – it tells the compiler that we are assigning a float and a decimal value. Without it, C# will interpret the number as double, which can’t be automatically converted to either a float or decimal

**Float, double or decimal?**

Dealing with floating point values in programming has always caused a lot of questions and concerns. For instance, C# has at least three data types for dealing with non-whole/non-integer numbers:

* Float (System.Single);
* Double (System.Double)
* Decimal (System.Decimal)

The underlying difference might be a bit difficult to understand, unless you have a lot of knowledge about how a computer works internally.

In general, the difference between the float, double and decimal data types lies in the precision and therefore also in how much memory is used to hold them. The **float** is the least expensive one – it can represent a number with up 7 digits. The **double** is more precise, with up to 16 digits, while the decimal is the most precise, with a whooping maximum of 29 digits.

When dealing with floating point values, you should use a float or a double data type when precision is less important than performance. On the other hand, if you want the maximum amount of precision and you are willing to accept a lower of performance, you should go with the decimal data type

**Boolean**

The bool (Boolean) data type is one of the simplest found in the .NET framework, because it only has two possible values: false or true. You can declare a Boolean variable like this:

**bool** isAdult;

by default, the value of a bool is false, but you can of course change that – either when you declare that variable or later on:

**bool** isAdult = true;

working with a Boolean value usually means checking its current state and then reacting to it,

e.g.

using an if-statement

**bool** isAdult = true;

if(isAdult == true)

Console.WriteLine(“An adult”);

else

Console.WriteLine(“No Record”);

The bool data type can only have two values – false or true. it’s easy to check with an if statement and is often the return type of many methods.

**Character**

The **System.Char** data type is used to hold a single, Unicode character. C# has an alias for it, called char, which you can use when declaring your char variables:

**Char** ch;

You can immediately assign a value to it, if you want to. In c#, a char is surrounded by a set of single quotes:

**Char** ch = ‘H’;

String is basically just a range of characters; .NET actually uses a list of chars to represent a string. That also means that you can pull out a single char from a string, or iterate over a string and get each character as a char data type.

e.g.

**string** helloworld = “Hello, World!”;

**foreach**(**char** c in helloworld)

{

Console.WriteLine(c);

}

A **char** data type is from microsoft **namespace** called **System.Char** is used to represent a single Unicode character like (‘a’, ‘b’, ‘!’, ‘\U0000’) ‘\U0000’ is ASCII/Unicode Codes. To represent more than one char, you use a string, which is basically just a list of chars.

**String**

In c#, you can use strings as array of characters, however, more common practice is to use the string keyword to declare a string variable. The string is Microsoft namespace of **System.String** class.

e.g.

string username = “admin”;

System.Console.WriteLine(username[4]);

* Can be used for any kind of a value as a text
* String variable[index] to access individual characters
* Index always starts from 0
* Strings are immutable

**Arrays**

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data. Arrays are used to store multiple values in a single variable, instead of declaring separate variable for each value.

e.g.

**int[]** number = {2, 4, 5, 7,8};

to create an array you must use an indexer right after the data type

e.g.

type[] arrayname = new type[amountofitems];

arrayname = {arrayitem1, arrayitem2};

arrayname[0] = value;

arrayitem[1] = value;

string arrays -> arrayname[index][secondIndex]

strings within string arrays are still immutable

**Section 2 – The Console**

The **console** class is used to represent the standard input, output and error streams for the console applications.

A **console** is an operating system window through which a user can communicate with the operating system or a **console** is an application in which we can give text as an input from the keyword and get the text as an output from the computer end. The command prompt is an example of a console in the windows and which accept ms-dos commands.

The console contains two attributes named as screen buffer and a console window.

Console class provides different types of properties and method to perform operations.

**Properties**

|  |  |
| --- | --- |
| **Properties** | **Description** |
| BackgroundColor | Gets or sets the background color of the color |
| BufferHeight | Gets or sets the height of the buffer area |
| Bufferwidth | Gets or sets the width of the buffer area |
| Capslock | Gets a value indicating whether the CAPSLOCK keyboard toggle is turned on or turned off. |
| CursorLeft | Gets or sets the column position of the cursor within the buffer area. |
| curserSize | Gets or Sets the height of the cursor within a character cell. |
| CursorTop | Get or sets the row position of the cursor within the buffer area |
| CursorVisible | Get or Sets a value indicating whether the cursor is visible |
| Error | Gets the standard error output stream |
| ForegroundColor | Gets or Sets the foreground color of the console |
| In | Gets the standard input stream |
| inputEncoding | Gets or Sets the encoding the console uses to read input |
| IsErrorRedirected | Gets a value that indicates whether the error output stream has been redirected from the standard error stream |
| IsInputRedirected | Get a value that indicates whether input has been redirected from the standard input stream |
| IsOutputRedirected | Gets a value that indicates whether output has been redirected from the standard output stream |
| keyAvailable | Gets a value indicating whether a key press is available in the input stream |
| LargestWindowHeight | Gets the largest possible number of console window rows, based on the current font and screen resolution |
| LargestWindowWidth | Gets the largest possible number of console window columns, based on the current font and screen resolution |
| NumberLock | Gets a value indicating whether the NUM LOCK Keyboard toggle is turned on or turned off |
| Out | Gets the standard output stream |
| OutputEncoding | Gets or Sets the encoding the console uses to write output |
| Title | Gets or Sets the title to display in the console title bar |
| TreatControlCAsInput | Gets or Sets a value indicating whether the combination of the control modifier key and C console key (Ctrl+C) is treated as ordinary input or as an interruption that is handled by the operating system |
| windowHeight | Gets or Sets the height of the console window area |
| windowLeft | Gets or Sets the leftmost position of the console window area relative to the screen buffer |
| WindowTop | Get or Sets the top position of the console window area relative to the screen buffer |
| WindowWidth | Gets or Sets the width of the console window |

e.g.

console.WriteLine(“Background color :{0}”, console.BackgroundColor);

Methods

For more details <https://www.geeksforgeeks.org/console-class-in-c-sharp>

**Section 4 – operators and Expressions**

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.

**Type of operators**

* Arithmetic Operators
* Relational operations
* Bitwise operators
* Assignment operators
* Misc operators

**Arithmetic Operators**

These are the arithmetic operator that supported by C#

|  |  |  |
| --- | --- | --- |
| **Operators** | **Description** | **Example** |
| + | Add two operands | A + B = 30 |
| - | Subtracts second operand from the first | A – B = -10 |
| \* | Multiples both operands | A \* B = 200 |
| / | Divides numerator by de-numerator | B/A = 2 |
| % | Modulus operator and remainder of after an integer division | B % A = 0 |
| ++ | Increment operator increases integer value | A++ = 11 |
| -- | Decrement operator decreases integer value by one | A-- = 9 |

**Relational Operators**

These are the relational operator that supported by C#

|  |  |  |
| --- | --- | --- |
| **Operators** | **Description** | **Example** |
| == | Checks if the value of two operands are equal or not, if yes then condition becomes true | (A == B) is not true |
| != | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true | (A != B) is true |
| > | Checks if the values of left operand is greater than the value of right operand, if yes then condition becomes true | (A > B) is not true |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true | (A < B) is true |
| >= | Checks if the value of left operand is greater than or equal to the value of right operands, if yes then condition becomes true | (A >= B) is not true |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true | (A <= B) is true |

**Logical Operators**

These are the logical operator that supported by C#

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **&&** | Called Logical AND operator. If both the operands are non zero then conditional becomes true | (A &&) is false |
| || | Called Logical OR operator. If any of the operands is non zero the condition becomes true | (A || B) is true |
| ! | Called Logical NOT operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make | !(A && B) is true |

**Bitwise Operators**

Bitwise operator works on bits and perform bit by bit operation. These are the bitwise operator that supported by C#.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Binary AND operator copies a bit to the result if it exists in both operands | (A & B) = 12, which is 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either operands | (A | B) = 61, which is 0011 1101 |
| ^ | Binary XOR operator copies the bit if it is set in one operand but not both | (A ^ B) = 49, which ic 0011 0001 |
| ~ | Binary Ones complement operator is unary and has the effect of ‘Flipping’ bits | (~A) = -61, which is 1100 0011 in 2’s complement due to a signed binary number |
| << | Binary Left Shift operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 = 240, which is 1111 0000 |
| >> | Binary Right Shift Operator. The Left operands value is moved right by number of bits specified by the right operand | A >> 2 = 15, which is 0000 1111 |

**Assignment Operators**

These are the assignment operator that supported by C#.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator, Assigns values from right side operands to left side operand | C = A + B assigns value of A + B into C |
| += | Add AND assignment operator, it adds right operand to the left operand and assign the result to left operand | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, it subtracts right operand from the left operand and assign the result to left operand | C -= A is equivalent to c = C – A |
| \*= | Multiply AND assignment operator, it multiplies right operand from the left operand and assign the result to left operand | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator, it divides left operand with the right operand and assign the result to left operand | C /= A is equivalent to C = C / A |
| %= | Modulus AND Assignment operator, it takes modulus using two operands and assign the result to left operand | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator | C <<= 2 is Same as C = C << 2 |
| >>= | Right shift AND assignment operator | C >>= 2 is same as C = C >>2 |
| &= | Bitwise AND assignment operator | C &= 2 same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator | C ^= 2 same as C = C ^ 2 |
| |= | bitwise inclusive OR and assignment operator | C |= 2 is asme as C = C | 2 |

**Miscellaneous Operators**

These are the miscellaneous operator that supported by C#.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| sizeof() | Returns the size of a data type | sizeof(int), returns 4 |
| typeof() | Returns the type of a class | typeof(StreamReader); |
| & | Returns the address of an variable | \*a; creates pointer named ‘a’ to a variable |
| ?: | Conditional Expression | If Condition is true ? Then value X : Otherwise value Y |
| is | Determines whether an object is of a certain type | If(Ford is Car) //checks if Ford is an object of the car class |
| as | Cast without raising an exception if the cast fails | Object obj = new StringReader(“Hello”);  StringReader r = obj as StringReader; |

C# includes a decision-making operator ?: which is called the conditional operator or ternary operator.

e.g.

int x = 20, y = 10;

var result = x > y ? "x is greater than y" : "x is less than y";

Console.WriteLine(result);

The ternary operator starts with a boolean condition. If this condition evaluates to true then it will execute the first statement after ?, otherwise the second statement after : will be executed.

**Section 5: Condition statement – Flow Control**

C# provides many decision-making statements that help the flow of the C# program based on certain logical conditions.

**If” “true” do this, else if “false” do that**

In C#, as we know that if -statement is executed if the condition is true otherwise it will not execute. The if-statement contains a Boolean condition followed by a single or multi-line code block to be executed. At runtime, if a Boolean condition evaluates to true, then the code block will be executed.

Syntax

If(condition)

{

//code block to be executed when if condition evaluates to true

}

To execute only under certain circumstances, place the code in an if conditional statement block

To cover all other cases, use an else block

Only one of the two blocks will n=be executed

if(condition)

{

// code if condition is true

}

else

{

// code if condition is false

}

Else-statement can contain single or multiple statements in the curly braces{}. If the else statement only contains a single statement, then the curly braces are optional.

**Nested if Statements**

C# supports if else statements inside another if else statements. This are called nested if else statements. The nested if statements make the code more readable.

Syntax

if(condition1)

{

if(condition2)

{

// code block to be executed when

// condition1 and condition2 evaluates to true

}

else if(condition3)

{

if(condition4)

{

// code block to be executed when

// only condition1, condition3, and condition4 evaluates to true

}

else if(condition5)

{

// code block to be executed when

// only condition1, condition3, and condition5 evaluates to true

}

else

{

// code block to be executed when

// condition1, and condition3 evaluates to true

// condition4 and condition5 evaluates to false

}

}

}

Nested if add layers of conditions

Each layer is a prerequisite for the previous layer

Individual code can be running in each of the layers

**Switch Statement**

n c#, Switch is a selection statement, and it will execute a single case statement from the list of multiple case statements based on the pattern match with the defined expression. The switch statement can be used instead of if else statement when you want to test a variable against three or more conditions.

e.g.

witch(variable/expresison){

case value1:

// Statements to Execute

break;

case value2:

//Statements to Execute

break;

....

....

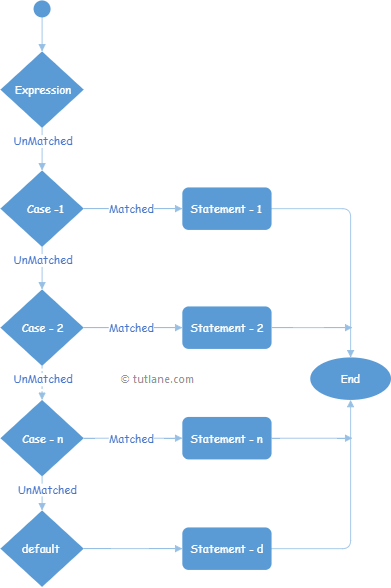
default:

// Statements to Execute if No Case Matches

break;

}

**C# Switch Statement Flow Chart Diagram**

Following is the pictorial representation of the switch case statement process flow in the C# programming language.

**Switches** on a given value and compares it against different value **cases**

The **case** that matches the **value** will be executed

The **default** case will be executed if there was no match with any of the other case

The difference with **else if** is that **else if** is based on logical operations and conditions, the **switch** is taking **one single value** and **compares** it against the **cases** of the switch

**Section 6: Loops – the backbone of every single software**

**Introduction to loops**

Looping in a programming language is a way to execute a statement or a set of statements multiple times depending on the result of the condition to be evaluated to execute statements. The result condition should be true to execute statements within loops.

**While Loops**

C# provides the while loop to continually execute a block of code or statement, when a condition is true.

string doAgain = “Y”;

int count = 0;

string[] siteName = new string[10];

while(doAgain == “Y”)

{

Console.Write(“Please Enter Site Name: “);

siteName[count++] = Console.ReadLine();

Console.Write(“Add Another”);

doAgain = Console.ReadLine();

}

**The Do-While loop and difference with while loop**

The do while loop is the same as while loop, while loops evaluate an expression before executing the statements at least once.

do

{

//code block

}while(condition);

The do-while loop starts with the **do** keyword followed by a code block and a Boolean expression with the while keyword. The code block executes, then the condition is evaluated. If condition evaluates to true.

**For Loops**

For loop execute a group of statement or block code multiple times, the for keyword indicates a loop in c#. the for loop executes a block of statement repeatedly until the specified condition returns false.

For(initializer; condition; iterator)

{

//code block

}

For is derivation of the **while** with a specific condition – to run a set amount of times

Mostly used to work with arrays

**Foreach**

The **foreach loop** is used to iterator over the elements of the collections.

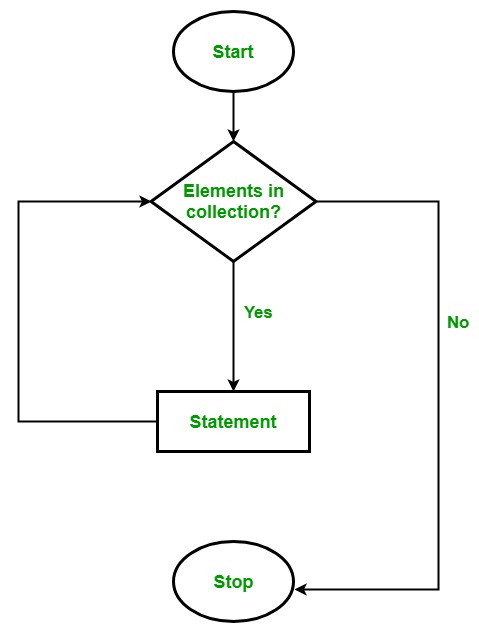
Foreach(type variableName in arrayName)

{

//code block to be executed

}

The type can be any C# or user-defined type. The collection could be any C# collection a list or array. It executes for each element present in the array.



Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array.

**Nested loops**

Nested loops are those loops that are present inside another loop. In C#, nesting of for, while, and do-while loops.

**Methods that do return values Data type**

**Method**

A method is a code that contains a series of statement. This statement in a method code block can only runs when it is called, a program causes the statement to be executed by calling the method and specifying any required method arguments. In c#, every executed instruction is performed in the context of a method.

**Type of methods**

* Pure virtual method
* Virtual method
* Abstract method
* Partial method
* Extension method
* Instance method
* Static method

**Pure Virtual Method**

Pure virtual method is the term that programmers use in C++. There is a term “abstract” in place of “pure virtual method” in C#.

For example

public void abstract DoSomething();

**Virtual Method**

Virtual method makes some default functionality. In other words, virtual methods are being implemented in the base class and can be overridden in the derived class.

For example

**public** **class** A

{

**public** **virtual** **int** Calculate(**int** a, **int** b)

    {

**return** a + b;

    }

}

**public** **class** B: A

{

**public** **override** **int** Calculate(**int** a, **int** b)

    {

**return** a + b + 1;

    }

}

**Abstract Method**

Abstract method is the method with no implement and is implicitly virtual. You can make abstract method only in abstract class.

For example

1. **public** **void** **abstract** DoSomething(**int** a);

**Partial Method**

A partial method has its signature defined in one part of a partial type, and its implementation defined in another part of the type.

For example

**namespace** PM

{

    partial **class** A

    {

        partial **void** OnSomethingHappened(**string** s);

    }

    // This part can be in a separate file.

    partial **class** A

    {

        // Comment out this method and the program

        // will still compile.

        partial **void** OnSomethingHappened(String s)

        {

            Console.WriteLine("Something happened: {0}", s);

        }

    }

}

**Extension Method**

Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type.

Extension methods are used to add some functionality to given type.

**Note**

* For making Extension methods, you need to have a static class with static method.
* Do not make Extension method for one or two lines of code, write Extension method for logic.

e.g.

**public** **static** **class** ExtensionMethods

{

**public** **static** **string** UppercaseFirstLetter(**this** **string** value)

    {

        //

        // Uppercase the first letter in the string.

        //

**if** (value.Length > 0)

        {

**char**[] array = value.ToCharArray();

            array[0] = **char**.ToUpper(array[0]);

**return** **new** **string**(array);

        }

**return** value;

    }

}

**class** Program

{

**static** **void** Main()

    {

        //

        // Use the string extension method on this value.

        //

**string** value = "deeksha sharma";

        value = value.UppercaseFirstLetter();

        Console.WriteLine(value);

    }

}

**Instance Method**

An instance method operates on a given instance of a class, and that instance can be accessed as this.

e.g.

**public** **class** PropertyUtil

{

**public** **void** DoSomething()

    {

        // Do Something.

    }

**public** **void** DoSomethingElse()

    {

**this**.DoSomething(); // Calling to instance method.

    }

}

**public** **class** Program

{

**static** **void** Main()

    {

        PropertyUtil util = **new** PropertyUtil();

        util.DoSomething(); // Calling to instance method.

    }

}

**Static Method**

This belongs to the type; it does not belong to instance of the type. You can access static methods by class name.

You can put static method in static or non-static classes.

The only difference is that static methods in a non-static class cannot be extension methods.

e.g.

**public** **static** **class** StaticClass

{

**public** **static** **int** DoSomething()

    {

**return** 2;

    }

}

**public** **class** Program

{

**static** **void** Main()

    {

        StaticClass.DoSomething();

    }

}

**Method Overloading**

Method overloading is the common way of implementing polymorphism. It is the ability to redefine a function in more than one form. A user can implement function overloading by defining two or more functions in a class sharing the same name. C# can distinguish the methods with **different method signatures**. The methods can have the same name but with different parameters, and data types of the parameters within the same class.

e.g.

// C# program to demonstrate the function

// overloading by changing the Number

// of parameters

using System;

class GFG {

// adding two integer values.

public int Add(int a, int b)

{

int sum = a + b;

return sum;

}

// adding three integer values.

public int Add(int a, int b, int c)

{

int sum = a + b + c;

return sum;

}

// Main Method

public static void Main(String[] args)

{

// Creating Object

GFG ob = new GFG();

int sum1 = ob.Add(1, 2);

Console.WriteLine("sum of the two "

+ "integer value : " + sum1);

int sum2 = ob.Add(1, 2, 3);

Console.WriteLine("sum of the three "

+ "integer value : " + sum2);

}

}

**Ref**

Ref Will make any variable behave as a reference type

**Out**

Out Is essentially another return, you can use it to return additional value from a method

Out can also directly create the variable without it being explicitly created before

**Introduction to one Dimensional Arrays**

An array is a collection of value, it a container for set amount of value. An array is a collection of the variable of same data type. The value of array can be accessed using index position of array.

**Bubble Sort**

Bubble sort is a simple sorting algorithm. This algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order. Bubble sort algorithm finds the biggest elements and pushes them as further to the right as possible. Bubble sort is slow for big arrays

**Selection Sort Algorithm**