C# Notes

C# is a simple, modern, general-purpose, object-oriented programming language developed by Microsoft within its .NET initiative led by Anders Hejlsberg. This tutorial will teach you basic C# programming and will also take you through various advanced concepts related to C# programming language.

C# is a modern, general-purpose, object-oriented programming language developed by Microsoft and approved by European Computer Manufacturers Association (ECMA) and International Standards Organization (ISO).

C# was developed by Anders Hejlsberg and his team during the development of .Net Framework.

C# is designed for Common Language Infrastructure (CLI), which consists of the executable code and runtime environment that allows use of various high-level languages on different computer platforms and architectures.

The following reasons make C# a widely used professional language −

It is a modern, general-purpose programming language

It is object oriented.

It is component oriented.

It is easy to learn.

It is a structured language.

It produces efficient programs.

It can be compiled on a variety of computer platforms.

It is a part of .Net Framework.

C# Features

C# is object oriented programming language. It provides a lot of features that are given below.

Simple

Modern programming language

Object oriented

Type safe

Interoperability

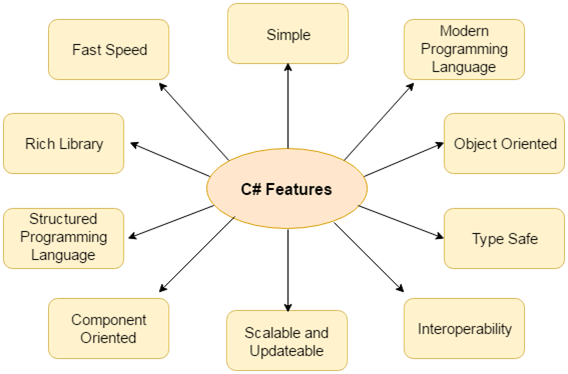
Scalable and Updateable

Component oriented

Structured programming language

Rich Library

Fast speed



1) Simple

C# is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2) Modern Programming Language

C# programming is based upon the current trend and it is very powerful and simple for building scalable, interoperable and robust applications.

3) Object Oriented

C# is object oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grow.

History of Java

4) Type Safe

C# type safe code can only access the memory location that it has permission to execute. Therefore it improves a security of the program.

5) Interoperability

Interoperability process enables the C# programs to do almost anything that a native C++ application can do.

6) Scalable and Updateable

C# is automatic scalable and updateable programming language. For updating our application we delete the old files and update them with new ones.

7) Component Oriented

C# is component oriented programming language. It is the predominant software development methodology used to develop more robust and highly scalable applications.

8) Structured Programming Language

C# is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

9) Rich Library

C# provides a lot of inbuilt functions that makes the development fast.

10) Fast Speed

The compilation and execution time of C# language is fast.

Strong Programming Features of C#

Although C# constructs closely follow traditional high-level languages, C and C++ and being an object-oriented programming language. It has strong resemblance with Java, it has numerous strong programming features that make it endearing to a number of programmers worldwide.

Following is the list of few important features of C# −

Boolean Conditions

Automatic Garbage Collection

Standard Library

Assembly Versioning

Properties and Events

Delegates and Events Management

Easy-to-use Generics

Indexers

Conditional Compilation

Simple Multithreading

LINQ and Lambda Expressions

Integration with Windows

The .Net Framework

The .Net framework is a revolutionary platform that helps you to write the following types of applications −

Windows applications

Web applications

Web services

The .Net framework applications are multi-platform applications. The framework has been designed in such a way that it can be used from any of the following languages: C#, C++, Visual Basic, Jscript, COBOL, etc. All these languages can access the framework as well as communicate with each other.

The .Net framework consists of an enormous library of codes used by the client languages such as C#. Following are some of the components of the .Net framework −

Common Language Runtime (CLR)

The .Net Framework Class Library

Common Language Specification

Common Type System

Metadata and Assemblies

Windows Forms

ASP.Net and ASP.Net AJAX

ADO.Net

Windows Workflow Foundation (WF)

Windows Presentation Foundation

Windows Communication Foundation (WCF)

LINQ

Creating Hello World Program

A C# program consists of the following parts −

Namespace declaration

A class

Class methods

Class attributes

A Main method

Statements and Expressions

Comments

Let us look at a simple code that prints the words "Hello World" −

using System;

namespace HelloWorldApplication {

class HelloWorld {

static void Main(string[] args) {

/\* my first program in C# \*/

Console.WriteLine("Hello World");

Console.ReadKey();

}

}

}

When this code is compiled and executed, it produces the following result −

Hello World

Let us look at the various parts of the given program −

The first line of the program using System; - the using keyword is used to include the System namespace in the program. A program generally has multiple using statements.

The next line has the namespace declaration. A namespace is a collection of classes. The HelloWorldApplication namespace contains the class HelloWorld.

The next line has a class declaration, the class HelloWorld contains the data and method definitions that your program uses. Classes generally contain multiple methods. Methods define the behavior of the class. However, the HelloWorld class has only one method Main.

The next line defines the Main method, which is the entry point for all C# programs. The Main method states what the class does when executed.

The next line /\*...\*/ is ignored by the compiler and it is put to add comments in the program.

The Main method specifies its behavior with the statement Console.WriteLine("Hello World");

WriteLine is a method of the Console class defined in the System namespace. This statement causes the message "Hello, World!" to be displayed on the screen.

The last line Console.ReadKey(); is for the VS.NET Users. This makes the program wait for a key press and it prevents the screen from running and closing quickly when the program is launched from Visual Studio .NET.

It is worth to note the following points −

C# is case sensitive.

All statements and expression must end with a semicolon (;).

The program execution starts at the Main method.

Unlike Java, program file name could be different from the class name.

Compiling and Executing the Program

If you are using Visual Studio.Net for compiling and executing C# programs, take the following steps −

Start Visual Studio.

On the menu bar, choose File -> New -> Project.

Choose Visual C# from templates, and then choose Windows.

Choose Console Application.

Specify a name for your project and click OK button.

This creates a new project in Solution Explorer.

Write code in the Code Editor.

Click the Run button or press F5 key to execute the project. A Command Prompt window appears that contains the line Hello World.

You can compile a C# program by using the command-line instead of the Visual Studio IDE −

Open a text editor and add the above-mentioned code.

Save the file as helloworld.cs

Open the command prompt tool and go to the directory where you saved the file.

Type csc helloworld.cs and press enter to compile your code.

If there are no errors in your code, the command prompt takes you to the next line and generates helloworld.exe executable file.

Type helloworld to execute your program.

You can see the output Hello World printed on the screen.

Identifiers

An identifier is a name used to identify a class, variable, function, or any other user-defined item. The basic rules for naming classes in C# are as follows −

A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9) or underscore. The first character in an identifier cannot be a digit.

It must not contain any embedded space or symbol such as? - + ! @ # % ^ & \* ( ) [ ] { } . ; : " ' / and \. However, an underscore ( \_ ) can be used.

It should not be a C# keyword.

C# Keywords

Keywords are reserved words predefined to the C# compiler. These keywords cannot be used as identifiers. However, if you want to use these keywords as identifiers, you may prefix the keyword with the @ character.

In C#, some identifiers have special meaning in context of code, such as get and set are called contextual keywords.

The following table lists the reserved keywords and contextual keywords in C# −

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Reserved Keywords | | | | | | |
| Abstract | As | base | bool | break | byte | case |
| Catch | char | checked | class | const | continue | decimal |
| Default | delegate | do | double | else | enum | event |
| Explicit | extern | false | finally | fixed | float | for |
| Foreach | goto | if | implicit | in | in (generic modifier) | int |
| interface | internal | is | lock | long | namespace | new |
| Null | object | operator | out | out (generic modifier) | override | params |
| Private | protected | public | readonly | ref | return | sbyte |
| Sealed | short | sizeof | stackalloc | static | string | struct |
| Switch | this | throw | true | try | typeof | uint |
| Ulong | unchecked | unsafe | ushort | using | virtual | void |
| Volatile | while |  |  |  |  |  |
| Contextual Keywords | | | | | | |
| Add | alias | ascending | descending | dynamic | from | get |
| Global | group | into | join | let | orderby | partial (type) |
| partial (method) | remove | select | set |  |  |  |

C# Variable

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

The basic variable type available in C# can be categorized as:

|  |  |
| --- | --- |
| Variable Type | Example |
| Decimal types | decimal |
| Boolean types | True or false value, as assigned |
| Integral types | int, char, byte, short, long |
| Floating point types | float and double |
| Nullable types | Nullable data types |

Let's see the syntax to declare a variable:

History of Ja

type variable\_list;

The example of declaring variable is given below:

int i, j;

double d;

float f;

char ch;

Here, i, j, d, f, ch are variables and int, double, float, char are data types.

We can also provide values while declaring the variables as given below:

int i=2,j=4;  //declaring 2 variable of integer type

float f=40.2;

char ch='B';

Rules for defining variables

A variable can have alphabets, digits and underscore.

A variable name can start with alphabet and underscore only. It can't start with digit.

No white space is allowed within variable name.

A variable name must not be any reserved word or keyword e.g. char, float etc.

Valid variable names:

int x;

int \_x;

int k20;

Invalid variable names:

int 4;

int x y;

int double;

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| C# Data Types  A data type specifies the type of data that a variable can store such as integer, floating, character etc.  CSHRAP Data types 1  There are 3 types of data types in C# language.   |  |  | | --- | --- | | Types | Data Types | | Value Data Type | short, int, char, float, double etc | | Reference Data Type | String, Class, Object and Interface | | Pointer Data Type | Pointers |   Value Data Type  The value data types are integer-based and floating-point based. C# language supports both signed and unsigned literals.  There are 2 types of value data type in C# language.  1) Predefined Data Types - such as Integer, Boolean, Float, etc.  2) User defined Data Types - such as Structure, Enumerations, etc.  The memory size of data types may change according to 32 or 64 bit operating system.  Let's see the value data types. It size is given according to 32 bit OS.   |  |  |  | | --- | --- | --- | | Data Types | Memory Size | Range | | Char | 1 byte | -128 to 127 | | signed char | 1 byte | -128 to 127 | | unsigned char | 1 byte | 0 to 127 | | Short | 2 byte | -32,768 to 32,767 | | signed short | 2 byte | -32,768 to 32,767 | | unsigned short | 2 byte | 0 to 65,535 | | Int | 4 byte | -2,147,483,648 to 2,147,483,647 | | signed int | 4 byte | -2,147,483,648 to -2,147,483,647 | | unsigned int | 4 byte | 0 to 4,294,967,295 | | Long | 8 byte | ?9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | | signed long | 8 byte | ?9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | | unsigned long | 8 byte | 0 - 18,446,744,073,709,551,615 | | Float | 4 byte | 1.5 \* 10-45 - 3.4 \* 1038, 7-digit precision | | Double | 8 byte | 5.0 \* 10-324 - 1.7 \* 10308, 15-digit precision | | Decimal | 16 byte | at least -7.9 \* 10?28 - 7.9 \* 1028, with at least 28-digit precision |   Reference Data Type  The reference data types do not contain the actual data stored in a variable, but they contain a reference to the variables.  If the data is changed by one of the variables, the other variable automatically reflects this change in value.  There are 2 types of reference data type in C# language.  1) Predefined Types - such as Objects, String.  2) User defined Types - such as Classes, Interface.  Pointer Data Type  The pointer in C# language is a variable, it is also known as locator or indicator that points to an address of a value.  CSHRAP Data types 2  Symbols used in pointer   |  |  |  | | --- | --- | --- | | Symbol | Name | Description | | & (ampersand sign) | Address operator | Determine the address of a variable. | | \* (asterisk sign) | Indirection operator | Access the value of an address. |   Declaring a pointer  The pointer in C# language can be declared using \* (asterisk symbol).  int \* a;  //pointer to int  char \* c; //pointer to char |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| C# Variables and (Primitive) Data Types  In this tutorial, we will learn about variables, how to create variables in C# and different data types that C# programming language supports.  A variable is a symbolic name given to a memory location. Variables are used to store data in a computer program.  How to declare variables in C#?  Here's an example to declare a variable in C#.  int age;  In this example, a variable age of type int (integer) is declared and it can only store integer values.  We can assign a value to the variable later in our program like such:  int age;  ... ... ...  age = 24;  However, the variable can also be initialized to some value during declaration. For example,  int age = 24;  Here, a variable age of type int is declared and initialized to 24 at the same time.  Since, it’s a variable, we can change the value of variables as well. For example,  int age = 24;  age = 35;  Here, the value of age is changed to 35 from 24.  Variables in C# must be declared before they can be used. This means, the name and type of variable must be known before they can be assigned a value. This is why C# is called a [statically-typed language](https://stackoverflow.com/questions/1517582/what-is-the-difference-between-statically-typed-and-dynamically-typed-languages).  Once declared, the datatype of a variable can not be changed within a scope. A scope can be thought as a block of code where the variable is visible or available to use. If you don’t understand the previous statement, don’t worry we’ll learn about scopes in the later chapters.  For now remember,we can not do the following in C#:  int age;  age = 24;  ... ... ...  float age;  Implicitly typed variables  Alternatively in C#, we can declare a variable without knowing its type using var keyword. Such variables are called implicitly typed local variables.  Variables declared using var keyword must be initialized at the time of declaration.  var value = 5;  The compiler determines the type of variable from the value that is assigned to the variable. In the above example, value is of type int. This is equivalent to:  int value;  value = 5;  You can learn more about [implicitly typed local variables](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/implicitly-typed-local-variables).  Rules for Naming Variables in C#  There are certain rules we need to follow while naming a variable. The rules for naming a variable in C# are:  The variable name can contain letters (uppercase and lowercase), underscore( \_ ) and digits only.  The variable name must start with either letter, underscore or @ symbol. For example,   | Rules for naming variables in C# | | | --- | --- | | Variable Names | Remarks | | Name | Valid | | subject101 | Valid | | \_age | Valid (Best practice for naming private member variables) | | @break | Valid (Used if name is a reserved keyword) | | 101subject | Invalid (Starts with digit) | | your\_name | Valid | | your name | Invalid (Contains whitespace) |   C# is case sensitive. It means age and Age refers to 2 different variables.  A variable name must not be a C# keyword. For example, if, for, using can not be a variable name. We will be discussing more about [C# keywords](https://www.programiz.com/csharp-programming/keywords-identifiers) in the next tutorial.  Best Practices for Naming a Variable  Choose a variable name that make sense. For example, name, age, subject makes more sense than n, a and s.  Use camelCase notation (starts with lowercase letter) for naming local variables. For example, numberOfStudents, age,  etc.Use PascalCase or CamelCase (starts with uppercase letter) for naming public member variables. For example,  FirstName, Price, etc.  Use a leading underscore (\_) followed by camelCase notation for naming private member variables. For example, \_bankBalance, \_emailAddress, etc.  You can learn more about [naming conventions in C# here](https://softwareengineering.stackexchange.com/questions/209532/naming-convention-of-variables-in-c-programming-language).  Don't worry about public and private member variables. We will learn about them in later chapters.  C# Primitive Data Types  Variables in C# are broadly classified into two types: Value types and Reference types. In this tutorial we will be discussing about primitive (simple) data types which is a subclass of Value types.  Reference types will be covered in later tutorials. However, if you want to know more about variable types, visit [C# Types and variables](https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/types-and-variables) (official C# docs).  Boolean (bool)  Boolean data type has two possible values: true or false  Default value: false  Boolean variables are generally used to check conditions such as in if statements, loops, etc.  For Example:  using System;  namespace DataType  {  class BooleanExample  {  public static void Main(string[] args)  {  bool isValid = true;  Console.WriteLine(isValid);  }  }  }  When we run the program, the output will be:  True  Signed Integral  These data types hold integer values (both positive and negative). Out of the total available bits, one bit is used for sign.  1. sbyte  Size: 8 bits  Range: -128 to 127.  Default value: 0  For example:  using System;  namespace DataType  {  class SByteExample  {  public static void Main(string[] args)  {  sbyte level = 23;  Console.WriteLine(level);  }  }  }  When we run the program, the output will be:  23  Try assigning values out of range i.e. less than -128 or greater than 127 and see what happens.  2. short  Size: 16 bits  Range: -32,768 to 32,767  Default value: 0  For example:  using System;  namespace DataType  {  class ShortExample  {  public static void Main(string[] args)  {  short value = -1109;  Console.WriteLine(value);  }  }  }  When we run the program, the output will be:  -1109  3. int  Size: 32 bits  Range: -231 to 231-1  Default value: 0  For example:  using System;  namespace DataType  {  class IntExample  {  public static void Main(string[] args)  {  int score = 51092;  Console.WriteLine(score);  }  }  }  When we run the program, the output will be:  51092  4. long  Size: 64 bits  Range: -263 to 263-1  Default value: 0L [L at the end represent the value is of long type]  For example:  using System;  namespace DataType  {  class LongExample  {  public static void Main(string[] args)  {  long range = -7091821871L;  Console.WriteLine(range);  }  }  }  When we run the program, the output will be:  -7091821871  Unsigned Integral  These data types only hold values equal to or greater than 0. We generally use these data types to store values when we are sure, we won't have negative values.  1. byte  Size: 8 bits  Range: 0 to 255.  Default value: 0  For example:  using System;  namespace DataType  {  class ByteExample  {  public static void Main(string[] args)  {  byte age = 62;  Console.WriteLine(level);  }  }  }  When we run the program, the output will be:  62  2. ushort  Size: 16 bits  Range: 0 to 65,535  Default value: 0  For example:  using System;  namespace DataType  {  class UShortExample  {  public static void Main(string[] args)  {  ushort value = 42019;  Console.WriteLine(value);  }  }  }  When we run the program, the output will be:  42019  3. uint  Size: 32 bits  Range: 0 to 232-1  Default value: 0  For example:  using System;  namespace DataType  {  class UIntExample  {  public static void Main(string[] args)  {  uint totalScore = 1151092;  Console.WriteLine(totalScore);  }  }  }  When we run the program, the output will be:  1151092  4. ulong  Size: 64 bits  Range: 0 to 264-1  Default value: 0  For example:  using System;  namespace DataType  {  class ULongExample  {  public static void Main(string[] args)  {  ulong range = 17091821871L;  Console.WriteLine(range);  }  }  }  When we run the program, the output will be:  17091821871  Floating Point  These data types hold floating point values i.e. numbers containing decimal values. For example, 12.36, -92.17, etc.  1. float  Single-precision floating point type  Size: 32 bits  Range: 1.5 × 10−45 to 3.4 × 1038  Default value: 0.0F [F at the end represent the value is of float type]  For example:  using System;  namespace DataType  {  class FloatExample  {  public static void Main(string[] args)  {  float number = 43.27F;  Console.WriteLine(number);  }  }  }  When we run the program, the output will be:  43.27  2. double  Double-precision floating point type. [What is the difference between single and double precision floating point?](https://stackoverflow.com/questions/801117/whats-the-difference-between-a-single-precision-and-double-precision-floating-p)  Size: 64 bits  Range: 5.0 × 10−324 to 1.7 × 10308  Default value: 0.0D [D at the end represent the value is of double type]  For example:  using System;  namespace DataType  {  class DoubleExample  {  public static void Main(string[] args)  {  double value = -11092.53D;  Console.WriteLine(value);  }  }  }  When we run the program, the output will be:  -11092.53  Character (char)  It represents a 16 bit unicode character.  Size: 16 bits  Default value: '\0'  Range: U+0000 ('\u0000') to U+FFFF ('\uffff')  For example:  using System;  namespace DataType  {  class CharExample  {  public static void Main(string[] args)  {  char ch1 ='\u0042';  char ch2 = 'x';  Console.WriteLine(ch1);  Console.WriteLine(ch2);  }  }  }  When we run the program, the output will be:  B  x  The unicode value of 'B' is '\u0042', hence printing ch1 will print 'B'.  Decimal  Decimal type has more precision and a smaller range as compared to floating point types (double and float). So it is appropriate for monetary calculations.  Size: 128 bits  Default value: 0.0M [M at the end represent the value is of decimal type]  Range: (-7.9 x 1028 to 7.9 x 1028) / (100 to 28)  For example:  using System;  namespace DataType  {  class DecimalExample  {  public static void Main(string[] args)  {  decimal bankBalance = 53005.25M;  Console.WriteLine(bankBalance);  }  }  }  When we run the program, the output will be:  53005.25  The suffix M or m must be added at the end otherwise the value will be treated as a double and an error will be generated.  C# Literals  Let's look at the following statement:  int number = 41;  Here,  int is a data type  number is a variable and  41 is a literal  Literals are fixed values that appear in the program. They do not require any computation. For example, 5, false, 'w' are literals that appear in a program directly without any computation.  Boolean Literals  true and false are the available boolean literals.  They are used to initialize boolean variables.  For example:  bool isValid = true;  bool isPresent = false;  Integer Literals  Integer literals are used to initialize variables of integer data types i.e. sbyte, short, int, long, byte, ushort, uint and ulong.  If an integer literal ends with L or l, it is of type long. For best practice use L (not l).  long value1 = 4200910L;  long value2 = -10928190L;  If an integer literal starts with a 0x, it represents hexadecimal value. Number with no prefixes are treated as decimal value. Octal and binary representation are not allowed in C#.  int decimalValue = 25;  int hexValue = 0x11c;// decimal value 284  Floating Point Literals  Floating point literals are used to initialize variables of float and double data types.  If a floating point literal ends with a suffix f or F, it is of type float. Similarly, if it ends with d or D, it is of type double. If neither of the suffix is present, it is of type double by default.  These literals contains e or E when expressed in scientific notation.  double number = 24.67;// double by default  float value = -12.29F;  double scientificNotation = 6.21e2;// equivalent to 6.21 x 102 i.e. 621  Character and String Literals  Character literals are used to initialize variables of char data types.  Character literals are enclosed in single quotes. For example, 'x','p', etc.  They can be represented as character, hexadecimal escape sequence, unicode representation or integral values casted to char.  char ch1 = 'R';// character  char ch2 = '\x0072';// hexadecimal  char ch3 = '\u0059';// unicode  char ch4 = (char)107;// casted from integer  String literals are the collection of character literals.  They are enclosed in double quotes. For example, "Hello", "Easy Programming", etc.  string firstName = "Richard";  string lastName = " Feynman";  C# also supports escape sequence characters such as:   | Character | Meaning | | --- | --- | | \' | Single quote | | \" | Double quote | | \\ | Backslash | | \n | Newline | | \r | Carriage return | | \t | Horizontal Tab | | \a | Alert | | \b | Backspace |   C# operators  An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise etc.  There are following types of operators to perform different types of operations in C# language.  Arithmetic Operators  Relational Operators  Logical Operators  Bitwise Operators  Assignment Operators  Unary Operators  Ternary Operators  Misc Operators  CSHARP Operators 1  Precedence of Operators in C#  The precedence of operator specifies that which operator will be evaluated first and next. The associativity specifies the operators direction to be evaluated, it may be left to right or right to left.  Let's understand the precedence by the example given below:  Features of Java - Javatpoint  int data= 10+ 5\*5  The "data" variable will contain 35 because \* (multiplicative operator) is evaluated before + (additive operator).  The precedence and   |  |  |  | | --- | --- | --- | | Category (By Precedence) | Operator(s) | Associativity | | Unary | + - ! ~ ++ -- (type)\* & sizeof | Right to Left | | Additive | + - | Left to Right | | Multiplicative | % / \* | Left to Right | | Relational | < > <= >= | Left to Right | | Shift | << >> | Left to Right | | Equality | == != | Right to Left | | Logical AND | & | Left to Right | | Logical OR | | | Left to Right | | Logical XOR | ^ | Left to Right | | Conditional OR | || | Left to Right | | Conditional AND | && | Left to Right | | Null Coalescing | ?? | Left to Right | | Ternary | ?: | Right to Left | | Assignment | = \*= /= %= += - = <<= >>= &= ^= |= => | Right to Left |   associativity of C# operators is given below: |

namespace DataTypeApplication {

class Program {

static void Main(string[] args) {

Console.WriteLine("Size of int: {0}", sizeof(int));

Console.ReadLine();

}

}

}

**Miscellaneous Operators**

There are few other important operators including sizeof, typeof and ? : supported by C#.

[Show Examples](https://www.tutorialspoint.com/csharp/csharp_misc_operators.htm)

|  |  |  |
| --- | --- | --- |
| 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; returns actual address of the variable. |
| \* | Pointer to a 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; |

Operator Precedence in C#

Operator precedence determines the grouping of terms in an expression. This affects evaluation of an expression. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example x = 7 + 3 \* 2; here, x is assigned 13, not 20 because operator \* has higher precedence than +, so the first evaluation takes place for 3\*2 and then 7 is added into it.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators are evaluated first.

[Show Examples](https://www.tutorialspoint.com/csharp/csharp_operators_precedence.htm)

|  |  |  |
| --- | --- | --- |
| Category | Operator | Associativity |
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)\* & sizeof | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %=>>= <<= &= ^= |=  C# Operators  In this article, we will learn everything about different types of operators in C# programming language and how to use them.  Operators are symbols that are used to perform operations on operands. Operands may be variables and/or constants.  For example, in 2+3, + is an operator that is used to carry out addition operation, while 2 and 3 are operands.  Operators are used to manipulate variables and values in a program. C# supports a number of operators that are classified based on the type of operations they perform.  1. Basic Assignment Operator  Basic assignment operator (=) is used to assign values to variables. For example,  double x;  x = 50.05;  Here, 50.05 is assigned to x.  Example 1: Basic Assignment Operator  using System;  namespace Operator  {  class AssignmentOperator  {  public static void Main(string[] args)  {  int firstNumber, secondNumber;  // Assigning a constant to variable  firstNumber = 10;  Console.WriteLine("First Number = {0}", firstNumber);  // Assigning a variable to another variable  secondNumber = firstNumber;  Console.WriteLine("Second Number = {0}", secondNumber);  }  }  }  When we run the program, the output will be:  First Number = 10  Second Number = 10  This is a simple example that demonstrates the use of assignment operator.  You might have noticed the use of curly brackets { } in the example. We will discuss about them in string formatting. For now, just keep in mind that {0} is replaced by the first variable that follows the string, {1} is replaced by the second variable and so on.  2. Arithmetic Operators  Arithmetic operators are used to perform arithmetic operations such as addition, subtraction, multiplication, division, etc.  For example,  int x = 5;  int y = 10;  int z = x + y;// z = 15   | C# Arithmetic Operators | | | | --- | --- | --- | | Operator | Operator Name | Example | | + | Addition Operator | 6 + 3 evaluates to 9 | | - | Subtraction Operator | 10 - 6 evaluates to 4 | | \* | Multiplication Operator | 4 \* 2 evaluates to 8 | | / | Division Operator | 10 / 5 evaluates to 2 | | % | Modulo Operator (Remainder) | 16 % 3 evaluates to 1 |   Example 2: Arithmetic Operators  using System;    namespace Operator  {  class ArithmeticOperator  {  public static void Main(string[] args)  {  double firstNumber = 14.40, secondNumber = 4.60, result;  int num1 = 26, num2 = 4, rem;  // Addition operator  result = firstNumber + secondNumber;  Console.WriteLine("{0} + {1} = {2}", firstNumber, secondNumber, result);  // Subtraction operator  result = firstNumber - secondNumber;  Console.WriteLine("{0} - {1} = {2}", firstNumber, secondNumber, result);  // Multiplication operator  result = firstNumber \* secondNumber;  Console.WriteLine("{0} \* {1} = {2}", firstNumber, secondNumber, result);  // Division operator  result = firstNumber / secondNumber;  Console.WriteLine("{0} / {1} = {2}", firstNumber, secondNumber, result);  // Modulo operator  rem = num1 % num2;  Console.WriteLine("{0} % {1} = {2}", num1, num2, rem);  }  }  }  When we run the program, the output will be:  14.4 + 4.6 = 19  14.4 - 4.6 = 9.8  14.4 \* 4.6 = 66.24  14.4 / 4.6 = 3.1304347826087  26 % 4 = 2  Arithmetic operations are carried out in the above example. Variables can be replaced by constants in the statements. For example,  result = 4.5 + 2.7 ; // result will hold 7.2  result = firstNumber - 3.2; // result will hold 11.2  3. Relational Operators  Relational operators are used to check the relationship between two operands. If the relationship is true the result will be true, otherwise it will result in false.  Relational operators are used in decision making and loops.  Example 3: Relational Operators  using System;    namespace Operator  {  class RelationalOperator  {  public static void Main(string[] args)  {  bool result;  int firstNumber = 10, secondNumber = 20;  result = (firstNumber==secondNumber);  Console.WriteLine("{0} == {1} returns {2}",firstNumber, secondNumber, result);  result = (firstNumber > secondNumber);  Console.WriteLine("{0} > {1} returns {2}",firstNumber, secondNumber, result);  result = (firstNumber < secondNumber);  Console.WriteLine("{0} < {1} returns {2}",firstNumber, secondNumber, result);  result = (firstNumber >= secondNumber);  Console.WriteLine("{0} >= {1} returns {2}",firstNumber, secondNumber, result);  result = (firstNumber <= secondNumber);  Console.WriteLine("{0} <= {1} returns {2}",firstNumber, secondNumber, result);  result = (firstNumber != secondNumber);  Console.WriteLine("{0} != {1} returns {2}",firstNumber, secondNumber, result);  }  }  }  When we run the program, the output will be:  10 == 20 returns False  10 > 20 returns False  10 < 20 returns True  10 >= 20 returns False  10 <= 20 returns True  10 != 20 returns True  4. Logical Operators  Logical operators are used to perform logical operation such as and, or. Logical operators operates on boolean expressions (true and false) and returns boolean values. Logical operators are used in decision making and loops.  Here is how the result is evaluated for logical AND and OR operators.   | C# Logical operators | | | | | --- | --- | --- | --- | | Operand 1 | Operand 2 | OR (||) | AND (&&) | | True | true | true | true | | True | false | true | false | | False | true | true | false | | False | false | false | false |   In simple words, the table can be summarized as:  If one of the operand is true, the OR operator will evaluate it to true.  If one of the operand is false, the AND operator will evaluate it to false.  Example 4: Logical Operators  using System;    namespace Operator  {  class LogicalOperator  {  public static void Main(string[] args)  {  bool result;  int firstNumber = 10, secondNumber = 20;  // OR operator  result = (firstNumber == secondNumber) || (firstNumber > 5);  Console.WriteLine(result);  // AND operator  result = (firstNumber == secondNumber) && (firstNumber > 5);  Console.WriteLine(result);  }  }  }  When we run the program, the output will be:  True  False  5. Unary Operators  Unlike other operators, the unary operators operates on a single operand.   | C# unary operators | | | | --- | --- | --- | | Operator | Operator Name | Description | | + | Unary Plus | Leaves the sign of operand as it is | | - | Unary Minus | Inverts the sign of operand | | ++ | Increment | Increment value by 1 | | -- | Decrement | Decrement value by 1 | | ! | Logical Negation (Not) | Inverts the value of a boolean |   Example 5: Unary Operators  using System;    namespace Operator  {  class UnaryOperator  {  public static void Main(string[] args)  {  int number = 10, result;  bool flag = true;  result = +number;  Console.WriteLine("+number = " + result);  result = -number;  Console.WriteLine("-number = " + result);  result = ++number;  Console.WriteLine("++number = " + result);  result = --number;  Console.WriteLine("--number = " + result);  Console.WriteLine("!flag = " + (!flag));  }  }  }  When we run the program, the output will be:  +number = 10  -number = -10  ++number = 11  --number = 10  !flag = False  The increment (++) and decrement (--) operators can be used as prefix and postfix. If used as prefix, the change in value of variable is seen on the same line and if used as postfix, the change in value of variable is seen on the next line. This will be clear by the example below.  Example 6: Post and Pre Increment operators in C#  using System;    namespace Operator  {  class UnaryOperator  {  public static void Main(string[] args)  {  int number = 10;  Console.WriteLine((number++));  Console.WriteLine((number));  Console.WriteLine((++number));  Console.WriteLine((number));  }  }  }  When we run the program, the output will be:  10  11  12  12  We can see the effect of using ++ as prefix and postfix. When ++ is used after the operand, the value is first evaluated and then it is incremented by 1. Hence the statement  Console.WriteLine((number++));  prints 10 instead of 11. After the value is printed, the value of number is incremented by 1.  The process is opposite when ++ is used as prefix. The value is incremented before printing. Hence the statement  Console.WriteLine((++number));  prints 12.  The case is same for decrement operator (--).  6. Ternary Operator  The ternary operator ? : operates on three operands. It is a shorthand for if-then-else statement. Ternary operator can be used as follows:  variable = Condition? Expression1 : Expression2;  The ternary operator works as follows:  If the expression stated by Condition is true, the result of Expression1 is assigned to variable.  If it is false, the result of Expression2 is assigned to variable.  Example 7: Ternary Operator  using System;    namespace Operator  {  class TernaryOperator  {  public static void Main(string[] args)  {  int number = 10;  string result;  result = (number % 2 == 0)? "Even Number" : "Odd Number";  Console.WriteLine("{0} is {1}", number, result);  }  }  }  When we run the program, the output will be:  10 is Even Number  To learn more, visit C# ternary operator.  7. Bitwise and Bit Shift Operators  Bitwise and bit shift operators are used to perform bit manipulation operations.   | C# Bitwise and Bit Shift operators | | | --- | --- | | Operator | Operator Name | | ~ | Bitwise Complement | | & | Bitwise AND | | | | Bitwise OR | | ^ | Bitwise Exclusive OR | | << | Bitwise Left Shift | | >> | Bitwise Right Shift |   Example 8: Bitwise and Bit Shift Operator  using System;    namespace Operator  {  class BitOperator  {  public static void Main(string[] args)  {  int firstNumber = 10;  int secondNumber = 20;  int result;  result = ~firstNumber;  Console.WriteLine("~{0} = {1}", firstNumber, result);  result = firstNumber & secondNumber;  Console.WriteLine("{0} & {1} = {2}", firstNumber,secondNumber, result);  result = firstNumber | secondNumber;  Console.WriteLine("{0} | {1} = {2}", firstNumber,secondNumber, result);  result = firstNumber ^ secondNumber;  Console.WriteLine("{0} ^ {1} = {2}", firstNumber,secondNumber, result);  result = firstNumber << 2;  Console.WriteLine("{0} << 2 = {1}", firstNumber, result);  result = firstNumber >> 2;  Console.WriteLine("{0} >> 2 = {1}", firstNumber, result);  }  }  }  When we run the program, the output will be:  ~10 = -11  10 & 20 = 0  10 | 20 = 30  10 ^ 20 = 30  10 << 2 = 40  10 >> 2 = 2  To learn more, visit C# Bitwise and Bit Shift operator.  8. Compound Assignment Operators   | C# Compound Assignment Operators | | | | | --- | --- | --- | --- | | Operator | Operator Name | Example | Equivalent To | | | += | Addition Assignment | x += 5 | x = x + 5 | | | -= | Subtraction Assignment | x -= 5 | x = x - 5 | | | \*= | Multiplication Assignment | x \*= 5 | x = x \* 5 | | | /= | Division Assignment | x /= 5 | x = x / 5 | | | %= | Modulo Assignment | x %= 5 | x = x % 5 | | | &= | Bitwise AND Assignment | x &= 5 | x = x & 5 | | | |= | Bitwise OR Assignment | x |= 5 | x = x | 5 | | | ^= | Bitwise XOR Assignment | x ^= 5 | x = x ^ 5 | | | <<= | Left Shift Assignment | x <<= 5 | x = x << 5 | | | >>= | Right Shift Assignment | x >>= 5 | x = x >> 5 | | | => | Lambda Operator | x => x\*x | Returns x\*x | |   Example 9: Compound Assignment Operator  using System;    namespace Operator  {  class BitOperator  {  public static void Main(string[] args)  {  int number = 10;  number += 5;  Console.WriteLine(number);  number -= 3;  Console.WriteLine(number);  number \*= 2;  Console.WriteLine(number);  number /= 3;  Console.WriteLine(number);  number %= 3;  Console.WriteLine(number);  number &= 10;  Console.WriteLine(number);  number |= 14;  Console.WriteLine(number);  number ^= 12;  Console.WriteLine(number);  number <<= 2;  Console.WriteLine(number);  number >>= 3;  Console.WriteLine(number);  }  }  }  When we run the program, the output will be:  15  12  24  8  2  2  14  2  8  1 | Right to left |
| Comma | , | Left to right |

| C# Relational Operators | | |
| --- | --- | --- |
| Operator | Operator Name | Example |
| == | Equal to | 6 == 4 evaluates to false |
| > | Greater than | 3 > -1 evaluates to true |
| < | Less than | 5 < 3 evaluates to false |
| >= | Greater than or equal to | 4 >= 4 evaluates to true |
| <= | Less than or equal to | 5 <= 3 evaluates to false |
| != | Not equal to | 10 != 2 evaluates to true |

C# Basic Input and Output

In this tutorial, we will learn how to take input from user and and display output in C# using various methods

C# Output

In order to output something in C#, we can use

System.Console.WriteLine() OR

System.Console.Write()

Here, System is a [namespace](https://www.programiz.com/csharp-programming/namespaces), Console is a class within namespace System and WriteLine and Write are methods of class Console.

Let's look at a simple example that prints a string to output screen.

Example 1: Printing String using WriteLine()

using System;

namespace Sample

{

class Test

{

public static void Main(string[] args)

{

Console.WriteLine("C# is cool");

}

}

}

When we run the program, the output will be

C# is cool

Difference between WriteLine() and Write() method

The main difference between WriteLine() and Write() is that the Write() method only prints the string provided to it, while the WriteLine() method prints the string and moves to the start of next line as well.

Let's take at a look at the example below to understand the difference between these methods.

Example 2: How to use WriteLine() and Write() method?

using System;

namespace Sample

{

class Test

{

public static void Main(string[] args)

{

Console.WriteLine("Prints on ");

Console.WriteLine("New line");

Console.Write("Prints on ");

Console.Write("Same line");

}

}

}

When we run the program, the output will be

Prints on

New line

Prints on Same line

Printing Variables and Literals using WriteLine() and Write()

The WriteLine() and Write() method can be used to print variables and literals. Here's an example.

Example 3: Printing Variables and Literals

using System;

namespace Sample

{

class Test

{

public static void Main(string[] args)

{

int value = 10;

// Variable

Console.WriteLine(value);

// Literal

Console.WriteLine(50.05);

}

}

}

When we run the program, the output will be

10

50.05

Combining (Concatenating) two strings using + operator and printing them

Strings can be combined/concatenated using the + operator while printing.

Example 4: Printing Concatenated String using + operator

using System;

namespace Sample

{

class Test

{

public static void Main(string[] args)

{

int val = 55;

Console.WriteLine("Hello " + "World");

Console.WriteLine("Value = " + val);

}

}

}

When we run the program, the output will be

Hello World

Value = 55

Printing concatenated string using Formatted String [Better Alternative]

A better alternative for printing concatenated string is using formatted string. Formatted string allows programmer to use placeholders for variables. For example,

The following line,

Console.WriteLine("Value = " + val);

can be replaced by,

Console.WriteLine("Value = {0}", val);

{0} is the placeholder for variable val which will be replaced by value of val. Since only one variable is used so there is only one placeholder.

Multiple variables can be used in the formatted string. We will see that in the example below.

Example 5: Printing Concatenated string using String formatting

using System;

namespace Sample

{

class Test

{

public static void Main(string[] args)

{

int firstNumber = 5, secondNumber = 10, result;

result = firstNumber + secondNumber;

Console.WriteLine("{0} + {1} = {2}", firstNumber, secondNumber, result);

}

}

}

When we run the program, the output will be

5 + 10 = 15

Here, {0} is replaced by firstNumber, {1} is replaced by secondNumber and {2} is replaced by result. This approach of printing output is more readable and less error prone than using + operator.

To know more about string formatting, visit C# string formatting.

C# Input

In C#, the simplest method to get input from the user is by using the ReadLine() method of the Console class. However, Read() and ReadKey() are also available for getting input from the user. They are also included in Console class.

Example 6: Get String Input From User

using System;

namespace Sample

{

class Test

{

public static void Main(string[] args)

{

string testString;

Console.Write("Enter a string - ");

testString = Console.ReadLine();

Console.WriteLine("You entered '{0}'", testString);

}

}

}

When we run the program, the output will be:

Enter a string - Hello World

You entered 'Hello World'

Difference between ReadLine(), Read() and ReadKey() method:

The difference between ReadLine(), Read() and ReadKey() method is:

ReadLine(): The ReadLine() method reads the next line of input from the standard input stream. It returns the same string.

Read(): The Read() method reads the next character from the standard input stream. It returns the ascii value of the character.

ReadKey(): The ReadKey() method obtains the next key pressed by user. This method is usually used to hold the screen until user press a key.

If you want to know more about these methods, here is an interesting discussion on StackOverflow on: [Difference between Console.Read() and Console.ReadLine()?](https://stackoverflow.com/questions/6825943/difference-between-console-read-and-console-readline).

Example 7: Difference between Read() and ReadKey() method

using System;

namespace Sample

{

class Test

{

public static void Main(string[] args)

{

int userInput;

Console.WriteLine("Press any key to continue...");

Console.ReadKey();

Console.WriteLine();

Console.Write("Input using Read() - ");

userInput = Console.Read();

Console.WriteLine("Ascii Value = {0}",userInput);

}

}

}

When we run the program, the output will be

Press any key to continue...

x

Input using Read() - Learning C#

Ascii Value = 76

From this example, it must be clear how ReadKey() and Read() method works. While using ReadKey(), as soon as the key is pressed, it is displayed on the screen.

When Read() is used, it takes a whole line but only returns the ASCII value of first character. Hence, 76 (ASCII value of L) is printed.

Reading numeric values (integer and floating point types)

Reading a character or string is very simple in C#. All you need to do is call the corresponding methods as required.

But, reading numeric values can be slightly tricky in C#. We’ll still use the same ReadLine() method we used for getting string values. But since the ReadLine() method receives the input as string, it needs to be converted into integer or floating point type.

One simple approach for converting our input is using the methods of Convert class.

Example 8: Reading Numeric Values from User using Convert class

using System;

namespace UserInput

{

class MyClass

{

public static void Main(string[] args)

{

string userInput;

int intVal;

double doubleVal;

Console.Write("Enter integer value: ");

userInput = Console.ReadLine();

/\* Converts to integer type \*/

intVal = Convert.ToInt32(userInput);

Console.WriteLine("You entered {0}",intVal);

Console.Write("Enter double value: ");

userInput = Console.ReadLine();

/\* Converts to double type \*/

doubleVal = Convert.ToDouble(userInput);

Console.WriteLine("You entered {0}",doubleVal);

}

}

}

When we run the program, the output will be

Enter integer value: 101

You entered 101

Enter double value: 59.412

You entered 59.412

The ToInt32() and ToDouble() method of Convert class converts the string input to integer and double type respectively. Similarly we can convert the input to other types. Here is a [complete list of available methods for Convert class](https://msdn.microsoft.com/en-us/library/system.convert(v=vs.110).aspx).

There are other ways to get numeric inputs from user. To learn more, visit [Reading an integer from user input](https://stackoverflow.com/questions/24443827/reading-an-integer-from-user-input).

C# Expressions, Statements and Blocks (With Examples)

In this article, we will learn about C# expressions, C# statements, difference between expression and statement, and C# blocks.

Expressions, statements and blocks are the building block of a C# program. We have been using them since our first ["Hello World" program](https://www.programiz.com/csharp-programming/hello-world).

C# Expressions

An expression in C# is a combination of operands (variables, literals, method calls) and operators that can be evaluated to a single value. To be precise, an expression must have at least one operand but may not have any operator.

Let's look at the example below:

double temperature;

temperature = 42.05;

Here, 42.05 is an expression. Also, temperature = 42.05 is an expression too.

int a, b, c, sum;

sum = a + b + c;

Here, a + b + c is an expression.

if (age>=18 && age<58)

Console.WriteLine("Eligible to work");

Here, (age>=18 && age<58) is an expression that returns a boolean value. "Eligible to work" is also an expression.

C# Statements

A statement is a basic unit of execution of a program. A program consists of multiple statements.

For example:

int age = 21;

Int marks = 90;

In the above example, both lines above are statements.

There are different types of statements in C#. In this tutorial, we’ll mainly focus on two of them:

Declaration Statement

Expression Statement

Declaration Statement

Declaration statements are used to declare and initialize variables.

For example:

char ch;

int maxValue = 55;

Both char ch; and int maxValue = 55; are declaration statements.

Expression Statement

An expression followed by a semicolon is called an expression statement.

For example:

/\* Assignment \*/

area = 3.14 \* radius \* radius;

/\* Method call is an expression\*/

System.Console.WriteLine("Hello");

Here, 3.14 \* radius \* radius  is an expression and area = 3.14 \* radius \* radius; is an expression statement.

Likewise, System.Console.WriteLine("Hello"); is both an expression and a statement.

Beside declaration and expression statement, there are:

Selection Statements (if...else, switch)

Iteration Statements (do, while, for, foreach)

Jump Statements (break, continue, goto, return, yield)

Exception Handling Statements (throw, try-catch, try-finally, try-catch-finally)

These statements will be discussed in later tutorials.

If you want to learn more about statements, visit [C# Statements](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/statements) ( C# reference)

C# Blocks

A block is a combination of zero or more statements that is enclosed inside curly brackets { }.

For example:

Example 1: C# Blocks with statements

using System;

namespace Blocks

{

class BlockExample

{

public static void Main(string[] args)

{

double temperature = 42.05;

if (temperature > 32)

{ // Start of block

Console.WriteLine("Current temperature = {0}", temperature);

Console.WriteLine("It's hot");

} // End of block

}

}

}

When we run the program, the output will be:

Current temperature = 42.05

It's hot

Here, the two statements inside { }:

Console.WriteLine("Current temperature = {0}", temperature);

and

Console.WriteLine("It's hot");

forms a block.

Example 2: C# Blocks without statements

A block may not have any statements within it as shown in the below example.

using System;

namespace Blocks

{

class BlockExample

{

public static void Main(string[] args)

{

double temperature = 42.05;

if (temperature > 32)

{ // Start of block

// No statements

} // End of block

}

}

}

Here, the curly braces { } after if(temperature > 32) contains only comments and no statements.

C# Comments

In this article, we will learn about C# comments, different style of comments, and why and how to use them in a program.

Comments are used in a program to help us understand a piece of code. They are human readable words intended to make the code readable. Comments are completely ignored by the compiler.

In C#, there are 3 types of comments:

Single Line Comments ( // )

Multi Line Comments (/\* \*/)

XML Comments ( /// )

Single Line Comments

Single line comments start with a double slash //. The compiler ignores everything after // to the end of the line. For example,

int a = 5 + 7; // Adding 5 and 7

Here, Adding 5 and 7 is the comment.

Example 1: Using single line comment

// Hello World Program

using System;

namespace HelloWorld

{

class Program

{

public static void Main(string[] args) // Execution Starts from Main method

{

// Prints Hello World

Console.WriteLine("Hello World!");

}

}

}

The above program contains 3 single line comments:

// Hello World Program

// Execution Starts from Main method

and

// Prints Hello World

Single line comments can be written in a separate line or along with the codes in same line. However, it is recommended to use comments in a separate line.

Multi Line Comments

Multi line comments start with /\* and ends with \*/. Multi line comments can span over multiple lines.

Example 2: Using multi line comment

/\*

This is a Hello World Program in C#.

This program prints Hello World.

\*/

using System;

namespace HelloWorld

{

class Program

{

public static void Main(string[] args)

{

/\* Prints Hello World \*/

Console.WriteLine("Hello World!");

}

}

}

The above program contains 2 multi line comments:

/\*

This is a Hello World Program in C#.

This program prints Hello World.

\*/

and

/\* Prints Hello World \*/

Here, we may have noticed that it is not compulsory for a multi line comment to span over multiple lines./\* … \*/ can be used instead of single line comments.

C# conditional statements

C# if, if...else, if...else if and Nested if Statement

In this article, we will learn how to use if, if...else, if...else if statement in C# to control the flow of our program’s execution.

Testing a condition is inevitable in programming. We will often face situations where we need to test conditions (whether it is true or false) to control the flow of program. These conditions may be affected by user's input, time factor, current environment where the program is running, etc.

In this article, we'll learn to test conditions using if statement in C#.

C# if (if-then) Statement

C# if-then statement will execute a block of code if the given condition is true. The syntax of if-then statement in C# is:

if (boolean-expression)

{

// statements executed if boolean-expression is true

}

The boolean-expression will return either true or false.

If the boolean-expression returns true, the statements inside the body of if ( inside {...} ) will be executed.

If the boolean-expression returns false, the statements inside the body of if will be ignored.

For example,

if (number < 5)

{

number += 5;

}

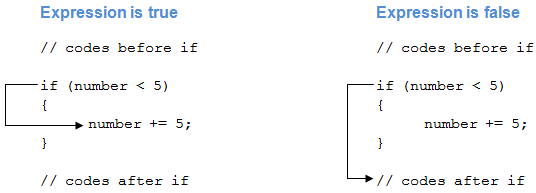
In this example, the statement

number += 5;

will be executed only if the value of number is less than 5.

Remember the [+= operator](https://www.programiz.com/csharp-programming/operators#compound-assignment)?

How if statement works?

Working of C# if Statement

Example 1: C# if Statement

using System;

namespace Conditional

{

class IfStatement

{

public static void Main(string[] args)

{

int number = 2;

if (number < 5)

{

Console.WriteLine("{0} is less than 5", number);

}

Console.WriteLine("This statement is always executed.");

}

}

}

When we run the program, the output will be:

2 is less than 5

This statement is always executed.

The value of number is initialized to 2. So the expression number < 5 is evaluated to true. Hence, the code inside the if block are executed. The code after the if statement will always be executed irrespective to the expression.

Now, change the value of number to something greater than 5, say 10. When we run the program the output will be:

This statement is always executed.

The expression number < 5 will return false, hence the code inside if block won't be executed.

C# if...else (if-then-else) Statement

The if statement in C# may have an optional else statement. The block of code inside the else statement will be executed if the expression is evaluated to false.

The syntax of if...else statement in C# is:

if (boolean-expression)

{

// statements executed if boolean-expression is true

}

else

{

// statements executed if boolean-expression is false

}

For example,

if (number < 5)

{

number += 5;

}

else

{

number -= 5;

}

In this example, the statement

number += 5;

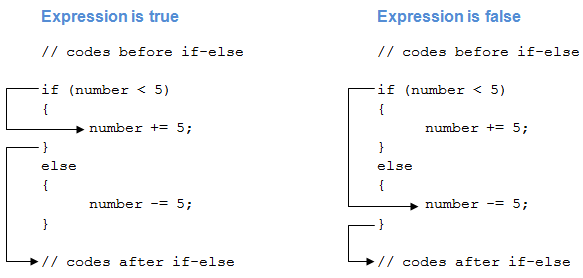
will be executed only if the value of number is less than 5.

The statement

number -= 5;

will be executed if the value of number is greater than or equal to 5.

How if...else Statement works?

Working of if...else Statement

Example 2: C# if...else Statement

using System;

namespace Conditional

{

class IfElseStatement

{

public static void Main(string[] args)

{

int number = 12;

if (number < 5)

{

Console.WriteLine("{0} is less than 5", number);

}

else

{

Console.WriteLine("{0} is greater than or equal to 5", number);

}

Console.WriteLine("This statement is always executed.");

}

}

}

When we run the program, the output will be:

12 is greater than or equal to 5

This statement is always executed.

Here, the value of number is initialized to 12. So the expression number < 5 is evaluated to false. Hence, the code inside the else block are executed. The code after the if..else statement will always be executed irrespective to the expression.

Now, change the value of number to something less than 5, say 2. When we run the program the output will be:

2 is less than 5

This statement is always executed.

The expression number < 5 will return true, hence the code inside if block will be executed.

[Ternary operator in C#](https://www.programiz.com/csharp-programming/ternary-operator) provides a shortcut for C# if...else statement.

C# if...else if (if-then-else if) Statement

When we have only one condition to test, if-then and if-then-else statement works fine. But what if we have a multiple condition to test and execute one of the many block of code.

For such case, we can use if..else if statement in C#. The syntax for if...else if statement is:

if (boolean-expression-1)

{

// statements executed if boolean-expression-1 is true

}

else if (boolean-expression-2)

{

// statements executed if boolean-expression-2 is true

}

else if (boolean-expression-3)

{

// statements executed if boolean-expression-3 is true

}

.

.

.

else

{

// statements executed if all above expressions are false

}

The if...else if statement is executed from the top to bottom. As soon as a test expression is true, the code inside of that if ( or else if ) block is executed. Then the control jumps out of the if...else if block.

If none of the expression is true, the code inside the else block is executed.

Alternatively, we can use [switch statement](https://www.programiz.com/csharp-programming/switch-statement) in such condition.

Example 3: C# if...else if Statement

using System;

namespace Conditional

{

class IfElseIfStatement

{

public static void Main(string[] args)

{

int number = 12;

if (number < 5)

{

Console.WriteLine("{0} is less than 5", number);

}

else if (number > 5)

{

Console.WriteLine("{0} is greater than 5", number);

}

else

{

Console.WriteLine("{0} is equal to 5");

}

}

}

}

When we run the program, the output will be:

12 is greater than 5

The value of number is initialized to 12. The first test expression number < 5 is false, so the control will move to the else if block. The test expression number > 5 is true hence the block of code inside else if will be executed.

Similarly, we can change the value of number to alter the flow of execution.

Nested if...else Statement

An if...else statement can exist within another if...else statement. Such statements are called nested if...else statement.

The general structure of nested if…else statement is:

if (boolean-expression)

{

if (nested-expression-1)

{

// code to be executed

}

else

{

// code to be executed

}

}

else

{

if (nested-expression-2)

{

// code to be executed

}

else

{

// code to be executed

}

}

Nested if statements are generally used when we have to test one condition followed by another. In a nested if statement, if the outer if statement returns true, it enters the body to check the inner if statement.

Example 4: Nested if...else Statement

The following program computes the largest number among 3 numbers using nested if...else statement.

using System;

namespace Conditional

{

class Nested

{

public static void Main(string[] args)

{

int first = 7, second = -23, third = 13;

if (first > second)

{

if (firstNumber > third)

{

Console.WriteLine("{0} is the largest", first);

}

else

{

Console.WriteLine("{0} is the largest", third);

}

}

else

{

if (second > third)

{

Console.WriteLine("{0} is the largest", second);

}

else

{

Console.WriteLine("{0} is the largest", third);

}

}

}

}

}

When we run the program, the output will be:

13 is the largest

C# switch Statement

In this article, we will learn about switch statement in C# and how to use them with examples.

Switch statement can be used to replace the [if...else if statement](https://www.programiz.com/csharp-programming/if-else-statement#if-else-if) in C#. The advantage of using switch over if...else if statement is the codes will look much cleaner and readable with switch.

The syntax of switch statement is:

switch (variable/expression)

{

case value1:

// Statements executed if expression(or variable) = value1

break;

case value2:

// Statements executed if expression(or variable) = value1

break;

... ... ...

... ... ...

default:

// Statements executed if no case matches

}

The switch statement evaluates the expression (or variable) and compare its value with the values (or expression) of each case (value1, value2, …). When it finds the matching value, the statements inside that case are executed.

But, if none of the above cases matches the expression, the statements inside default block is executed. The default statement at the end of switch is similar to the else block in if else statement.

However a problem with the switch statement is, when the matching value is found, it executes all statements after it until the end of switch block.

To avoid this, we use break statement at the end of each case. The break statement stops the program from executing non-matching statements by terminating the execution of switch statement.

To learn more about break statement, visit C# break statement.

Example 1: C# switch Statement

using System;

namespace Conditional

{

class SwitchCase

{

public static void Main(string[] args)

{

char ch;

Console.WriteLine("Enter an alphabet");

ch = Convert.ToChar(Console.ReadLine());

switch(Char.ToLower(ch))

{

case 'a':

Console.WriteLine("Vowel");

break;

case 'e':

Console.WriteLine("Vowel");

break;

case 'i':

Console.WriteLine("Vowel");

break;

case 'o':

Console.WriteLine("Vowel");

break;

case 'u':

Console.WriteLine("Vowel");

break;

default:

Console.WriteLine("Not a vowel");

break;

}

}

}

}

When we run the program, the output will be:

Enter an alphabet

X

Not a vowel

In this example, the user is prompted to enter an alphabet. The alphabet is converted to lowercase by using ToLower() method if it is in uppercase.

Then, the switch statement checks whether the alphabet entered by user is any of a, e, i, o or u.

If one of the case matches, Vowel is printed otherwise the control goes to default block and Not a vowel is printed as output.

Since, the output for all vowels are the same, we can join the cases as:

Example 2: C# switch Statement with grouped cases

using System;

namespace Conditional

{

class SwitchCase

{

public static void Main(string[] args)

{

char ch;

Console.WriteLine("Enter an alphabet");

ch = Convert.ToChar(Console.ReadLine());

switch(Char.ToLower(ch))

{

case 'a':

case 'e':

case 'i':

case 'o':

case 'u':

Console.WriteLine("Vowel");

break;

default:

Console.WriteLine("Not a vowel");

break;

}

}

}

}

The output of both programs is same. In the above program, all vowels print the output Vowel and breaks from the switch statement.

Although switch statement makes the code look cleaner than if...else if statement, switch is restricted to work with limited data types. Switch statement in C# only works with:

Primitive data types: bool, char and integral type

Enumerated Types (Enum)

String Class

Nullable types of above data types

Example 3: Simple calculator program using C# switch Statement

using System;

namespace Conditional

{

class SwitchCase

{

public static void Main(string[] args)

{

char op;

double first, second, result;

Console.Write("Enter first number: ");

first = Convert.ToDouble(Console.ReadLine());

Console.Write("Enter second number: ");

second = Convert.ToDouble(Console.ReadLine());

Console.Write("Enter operator (+, -, \*, /): ");

op = (char)Console.Read();

switch(op)

{

case '+':

result = first + second;

Console.WriteLine("{0} + {1} = {2}", first, second, result);

break;

case '-':

result = first - second;

Console.WriteLine("{0} - {1} = {2}", first, second, result);

break;

case '\*':

result = first \* second;

Console.WriteLine("{0} \* {1} = {2}", first, second, result);

break;

case '/':

result = first / second;

Console.WriteLine("{0} / {1} = {2}", first, second, result);

break;

default:

Console.WriteLine("Invalid Operator");

break;

}

}

}

}

When we run the program, the output will be:

Enter first number: -13.11

Enter second number: 2.41

Enter operator (+, -, \*, /): \*

-13.11 \* 2.41 = -31.5951

The above program takes two operands and an operator as input from the user and performs the operation based on the operator.

The inputs are taken from the user using the ReadLine() and Read() method. To learn more, visit [C# Basic Input and Output](https://www.programiz.com/csharp-programming/basic-input-output).

The program uses switch case statement for decision making. Alternatively, we can use if-else if ladder to perform the same operation.

C# ternary (? :) Operator

In this article, we will learn about C# ternary operator and how to use it to control the flow of program.

Ternary [operator](https://www.programiz.com/csharp-programming/operators) are a substitute for if...else statement. So before you move any further in this tutorial, go through [C# if...else statement](https://www.programiz.com/csharp-programming/if-else-statement) (if you haven't).

The syntax of ternary operator is:

Condition ? Expression1 : Expression2;

The ternary operator works as follows:

If the expression stated by Condition is true, the result of Expression1 is returned by the ternary operator.

If it is false, the result of Expression2 is returned.

For example, we can replace the following code

if (number % 2 == 0)

{

isEven = true;

}

else

{

isEven = false;

}

with

isEven = (number % 2 == 0) ? true : false ;

Why is it called ternary operator?

This operator takes 3 operand, hence called ternary operator.

Example 1: C# Ternary Operator

using System;

namespace Conditional

{

class Ternary

{

public static void Main(string[] args)

{

int number = 2;

bool isEven;

isEven = (number % 2 == 0) ? true : false ;

Console.WriteLine(isEven);

}

}

}

When we run the program, the output will be:

True

In the above program, 2 is assigned to a variable number. Then, the ternary operator is used to check if number is even or not.

Since, 2 is even, the expression (number % 2 == 0) returns true. We can also use ternary operator to return numbers, strings and characters.

Instead of storing the return value in variable isEven, we can directly print the value returned by ternary operator as,

Console.WriteLine((number % 2 == 0) ? true : false);

When to use ternary operator?

Ternary operator can be used to replace multi lines of code with a single line. However, we shouldn't overuse it.

For example, we can replace the following if..else if code

if (a > b)

{

result = "a is greater than b";

}

else if (a < b)

{

result = "b is greater than a";

}

else

{

result = "a is equal to b";

}

with a single line of code

result = a > b ? "a is greater than b" : a < b ? "b is greater than a" : "a is equal to b";

As we can see, the use of ternary operator may decrease the length of code but it makes us difficult to understand the logic of the code.

Hence, it's better to only use ternary operator to replace simple if else statements.

C# loop statements

C# for loop

In this article, we will learn about for loop in C# and different ways to use them in a program.

In programming, it is often desired to execute certain block of statements for a specified number of times. A possible solution will be to type those statements for the required number of times. However, the number of repetition may not be known in advance (during compile time) or maybe large enough (say 10000).

The best solution to such problem is loop. Loops are used in programming to repeatedly execute a certain block of statements until some condition is met.

In this article, we’ll look at for loop in C#.

C# for loop

The for keyword is used to create for loop in C#. The syntax for for loop is:

for (initialization; condition; iterator)

{

// body of for loop

}

How for loop works?

C# for loop has three statements: initialization, condition and iterator.

The initialization statement is executed at first and only once. Here, the variable is usually declared and initialized.

Then, the condition is evaluated. The condition is a boolean expression, i.e. it returns either true or false.

If the condition is evaluated to true:

The statements inside the for loop are executed.

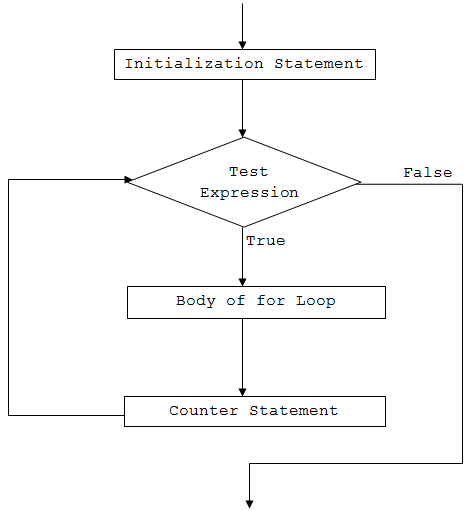
Then, the iterator statement is executed which usually changes the value of the initialized variable.

Again the condition is evaluated.

The process continues until the condition is evaluated to false.

If the condition is evaluated to false, the for loop terminates.

for Loop Flowchart

Working of C# for loop

Example 1: C# for Loop

using System;

namespace Loop

{

class ForLoop

{

public static void Main(string[] args)

{

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

{

Console.WriteLine("C# For Loop: Iteration {0}", i);

}

}

}

}

When we run the program, the output will be:

C# For Loop: Iteration 1

C# For Loop: Iteration 2

C# For Loop: Iteration 3

C# For Loop: Iteration 4

C# For Loop: Iteration 5

In this program,

initialization statement is int i=1

condition statement is i<=5

iterator statement is i++

When the program runs,

First, the variable i is declared and initialized to 1.

Then, the condition (i<=5) is evaluated.

Since, the condition returns true, the program then executes the body of the for loop. It prints the given line with Iteration 1 (Iteration simply means repetition).

Now, the iterator (i++) is evaluated. This increments the value of i to 2.

The condition (i<=5) is evaluated again and at the end, the value of i is incremented by 1. The condition will evaluate to true for the first 5 times.

When the value of i will be 6 and the condition will be false, hence the loop will terminate.

Example 2: for loop to compute sum of first n natural numbers

using System;

namespace Loop

{

class ForLoop

{

public static void Main(string[] args)

{

int n = 5,sum = 0;

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

{

// sum = sum + i;

sum += i;

}

Console.WriteLine("Sum of first {0} natural numbers = {1}", n, sum);

}

}

}

When we run the program, the output will be:

Sum of first 5 natural numbers = 15

Here, the value of sum and n are initialized to 0 and 5 respectively. The iteration variable i is initialized to 1 and incremented on each iteration.

Inside the for loop, value of sum is incremented by i i.e. sum = sum + i. The for loop continues until i is less than or equal to n (user's input).

Let's see what happens in the given program on each iteration.

Initially, i = 1, sum = 0 and n = 3

| For loop execution steps | | | |
| --- | --- | --- | --- |
| Iteration | Value of i | i<=5 | Value of sum |
| 1 | 1 | true | 0+1 = 1 |
| 2 | 2 | true | 1+2 = 3 |
| 3 | 3 | true | 3+3 = 6 |
| 4 | 4 | true | 6+4 = 10 |
| 5 | 5 | true | 10+5 = 15 |
| 6 | 6 | false | Loop terminates |

So, the final value of sum will be 15 when n = 5.

Multiple expressions inside a for loop

We can also use multiple expressions inside a for loop. It means we can have more than one initialization and/or iterator statements within a for loop. Let's see the example below.

Example 3: for loop with multiple initialization and iterator expressions

using System;

namespace Loop

{

class ForLoop

{

public static void Main(string[] args)

{

for (int i=0, j=0; i+j<=5; i++, j++)

{

Console.WriteLine("i = {0} and j = {1}", i,j);

}

}

}

}

When we run the program, the output will be:

i = 0 and j = 0

i = 1 and j = 1

i = 2 and j = 2

In this program, we have declared and initialized two variables: i and j in the initialization statement.

Also, we have two expressions in the iterator part. That means both i and j are incremented by 1 on each iteration.

For loop without initialization and iterator statements

The initialization, condition and the iterator statement are optional in a for loop. It means we can run a for loop without these statements as well.

In such cases, for loop acts as a [while loop](https://www.programiz.com/csharp-programming/do-while-loop). Let's see the example below.

Example 4: for loop without initialization and iterator statement

using System;

namespace Loop

{

class ForLoop

{

public static void Main(string[] args)

{

int i = 1;

for ( ; i<=5; )

{

Console.WriteLine("C# For Loop: Iteration {0}", i);

i++;

}

}

}

}

When we run the program, the output will be:

C# For Loop: Iteration 1

C# For Loop: Iteration 2

C# For Loop: Iteration 3

C# For Loop: Iteration 4

C# For Loop: Iteration 5

In this example, we haven't used the initialization and iterator statement.

The variable i is initialized above the for loop and its value is incremented inside the body of loop. This program is same as the one in Example 1.

Similarly, the condition is also an optional statement. However if we don't use test expression, the for loop won't test any condition and will run forever (infinite loop).

Infinite for loop

If the condition in a for loop is always true, for loop will run forever. This is called infinite for loop.

Example 5: Infinite for loop

using System;

namespace Loop

{

class ForLoop

{

public static void Main(string[] args)

{

for (int i=1 ; i>0; i++)

{

Console.WriteLine("C# For Loop: Iteration {0}", i);

}

}

}

}

Here, i is initialized to 1 and the condition is i>0. On each iteration we are incrementing the value of i by 1, so the condition will never be false. This will cause the loop to execute infinitely.

We can also create an infinite loop by replacing the condition with a blank. For example,

for ( ; ; )

{

// body of for loop

}

or

for (initialization ; ; iterator)

{

// body of for loop

}

C# while and do...while loop

In this article, we will learn about while and do...while loop in C#, how to use them and difference between them.

In programming, it is often desired to execute certain block of statements for a specified number of times. A possible solution will be to type those statements for the required number of times. However, the number of repetition may not be known in advance (during compile time) or maybe large enough (say 10000).

The best solution to such problem is loop. Loops are used in programming to repeatedly execute a certain block of statements until some condition is met.

In this article, we'll learn to use while loops in C#.

C# while loop

The while keyword is used to create while loop in C#. The syntax for while loop is:

while (test-expression)

{

// body of while

}

How while loop works?

C# while loop consists of a test-expression.

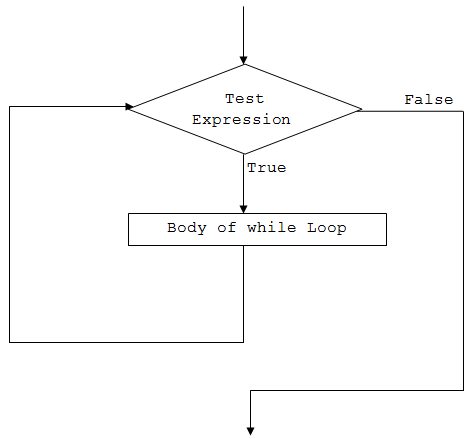
If the test-expression is evaluated to true,

statements inside the while loop are executed.

after execution, the test-expression is evaluated again.

If the test-expression is evaluated to false, the while loop terminates.

while loop Flowchart

Working of C# while loop

Example 1: while Loop

using System;

namespace Loop

{

class WhileLoop

{

public static void Main(string[] args)

{

int i=1;

while (i<=5)

{

Console.WriteLine("C# For Loop: Iteration {0}", i);

i++;

}

}

}

}

When we run the program, the output will be:

C# For Loop: Iteration 1

C# For Loop: Iteration 2

C# For Loop: Iteration 3

C# For Loop: Iteration 4

C# For Loop: Iteration 5

Initially the value of i is 1.

When the program reaches the while loop statement,

the test expression i <=5 is evaluated. Since i is 1 and 1 <= 5 is true, it executes the body of the while loop. Here, the line is printed on the screen with Iteration 1, and the value of i is increased by 1 to become 2.

Now, the test expression (i <=5) is evaluated again. This time too, the expression returns true (2 <= 5), so the line is printed on the screen and the value of i is now incremented to 3..

This goes and the while loop executes until i becomes 6. At this point, the test-expression will evaluate to false and hence the loop terminates.

Example 2: while loop to compute sum of first 5 natural numbers

using System;

namespace Loop

{

class WhileLoop

{

public static void Main(string[] args)

{

int i=1, sum=0;

while (i<=5)

{

sum += i;

i++;

}

Console.WriteLine("Sum = {0}", sum);

}

}

}

When we run the program, the output will be:

Sum = 15

This program computes the sum of first 5 natural numbers.

Initially the value of sum is initialized to 0.

On each iteration, the value of sum is updated to sum+i and the value of i is incremented by 1.

When the value of i reaches 6, the test expression i<=5 will return false and the loop terminates.

Let's see what happens in the given program on each iteration.

Initially, i = 1, sum = 0

| While loop execution steps | | | |
| --- | --- | --- | --- |
| Iteration | Value of i | i<=5 | Value of sum |
| 1 | 1 | true | 0+1 = 1 |
| 2 | 2 | true | 1+2 = 3 |
| 3 | 3 | true | 3+3 = 6 |
| 4 | 4 | true | 6+4 = 10 |
| 5 | 5 | true | 10+5 = 15 |
| 6 | 6 | false | Loop terminates |

So, the final value of sum will be 15.

C# do...while loop

The do and while keyword is used to create a do...while loop. It is similar to a while loop, however there is a major difference between them.

In while loop, the condition is checked before the body is executed. It is the exact opposite in do...while loop, i.e. condition is checked after the body is executed.

This is why, the body of do...while loop will execute at least once irrespective to the test-expression.

The syntax for do...while loop is:

do

{

// body of do while loop

} while (test-expression);

How do...while loop works?

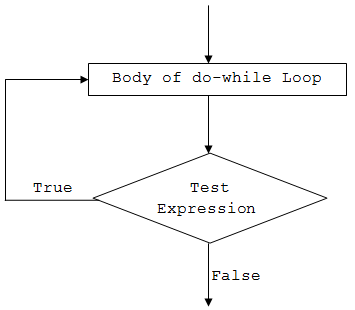
The body of do...while loop is executed at first.

Then the test-expression is evaluated.

If the test-expression is true, the body of loop is executed.

When the test-expression is false, do...while loop terminates.

do...while loop Flowchart

Working of C# do...while loop

Example 3: do...while loop

using System;

namespace Loop

{

class DoWhileLoop

{

public static void Main(string[] args)

{

int i = 1, n = 5, product;

do

{

product = n \* i;

Console.WriteLine("{0} \* {1} = {2}", n, i, product);

i++;

} while (i <= 10);

}

}

}

When we run the program, the output will be:

5 \* 1 = 5

5 \* 2 = 10

5 \* 3 = 15

5 \* 4 = 20

5 \* 5 = 25

5 \* 6 = 30

5 \* 7 = 35

5 \* 8 = 40

5 \* 9 = 45

5 \* 10 = 50

As we can see, the above program prints the multiplication table of a number (5).

Initially, the value of i is 1. The program, then enters the body of do..while loop without checking any condition (as opposed to while loop).

Inside the body, product is calculated and printed on the screen. The value of i is then incremented to 2.

After the execution of the loop’s body, the test expression i <= 10 is evaluated. In total, the do...while loop will run for 10 times.

Finally, when the value of i is 11, the test-expression evaluates to false and hence terminates the loop.

Infinite while and do...while loop

If the test expression in the while and do...while loop never evaluates to false, the body of loop will run forever. Such loops are called infinite loop.

For example:

Infinite while loop

while (true)

{

// body of while loop

}

Infinite do...while loop

do

{

// body of while loop

} while (true);

The infinite loop is useful when we need a loop to run as long as our program runs.

For example, if your program is an animation, you will need to constantly run it until it is stopped. In such cases, an infinite loop is necessary to keep running the animation repeatedly.

Nested Loops in C#: for, while, do-while

In this article, we will learn about nested loops in C#. We'll learn to use nested for, while and do-while loops in a program.

A loop within another loop is called nested loop. This is how a nested loop looks like:

Outer-Loop

{

// body of outer-loop

Inner-Loop

{

// body of inner-loop

}

... ... ...

}

As you can see, the outer loop encloses the inner loop. The inner loop is a part of the outer loop and must start and finish within the body of outer loop.

On each iteration of outer loop, the inner loop is executed completely.

Nested for loop

A for loop inside another for loop is called nested for loop.

For example:

for (int i=0; i<5; i++)

{

// body of outer for loop

for (int j=0; j<5; j++)

{

// body of inner for loop

}

// body of outer for loop

}

Example 1: Nested for Loop

using System;

namespace Loop

{

class NestedForLoop

{

public static void Main(string[] args)

{

int outerLoop = 0, innerLoop = 0;

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

{

outerLoop ++;

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

{

innerLoop++;

}

}

Console.WriteLine("Outer Loop runs {0} times", outerLoop);

Console.WriteLine("Inner Loop runs {0} times", innerLoop);

}

}

}

When we run the program, the output will be:

Outer Loop runs 5 times

Inner Loop runs 25 times

In this program, the outer loop runs for 5 times. Each time the outer loop runs, the inner loop runs for 5 times making it run 25 times altogether.

Example 2: Nested for Loop to Print Pattern

using System;

namespace Loop

{

class NestedForLoop

{

public static void Main(string[] args)

{

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

{

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

{

Console.Write(j + " ");

}

Console.WriteLine();

}

}

}

}

When we run the program, the output will be:

1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

Nested while loop

A while loop inside another while loop is called nested while loop.

For example:

while (condition-1)

{

// body of outer while loop

while (condition-2)

{

// body of inner while loop

}

// body of outer while loop

}

Example 3: Nested while Loop

using System;

namespace Loop

{

class NestedWhileLoop

{

public static void Main(string[] args)

{

int i=0;

while (i<2)

{

int j=0;

while (j<2)

{

Console.Write("({0},{1}) ", i,j);

j++;

}

i++;

Console.WriteLine();

}

}

}

}

When we run the program, the output will be:

(0,0) (0,1)

(1,0) (1,1)

Nested do-while loop

A do-while loop inside another do-while loop is called nested do-while loop.

For example:

do

{

// body of outer while loop

do

{

// body of inner while loop

} while (condition-2);

// body of outer while loop

} while (condition-1);

Example 4: Nested do-while Loop

using System;

namespace Loop

{

class NestedWhileLoop

{

public static void Main(string[] args)

{

int i=0;

do

{

int j=0;

do

{

Console.Write("({0},{1}) ", i,j);

j++;

} while (j<2);

i++;

Console.WriteLine();

} while (i<2);

}

}

}

When we run the program, the output will be:

(0,0) (0,1)

(1,0) (1,1)

Different inner and outer nested loops

It is not mandatory to nest same type of loop. We can put a for loop inside a while loop or a do-while loop inside a for loop.

Example 5: C# Nested Loop: Different inner and outer loops

using System;

namespace Loop

{

class NestedLoop

{

public static void Main(string[] args)

{

int i=1;

while (i<=5)

{

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

{

Console.Write(i + " ");

}

Console.WriteLine();

i++;

}

}

}

}

When we run the program, the output will be:

1

2 2

3 3 3

4 4 4 4

5 5 5 5 5

In the above program, a for loop is placed within a while loop. We can use different types of loop inside a loop.

C# break Statement

In this tutorial, you will learn about the working C# break statement with the help of examples.

In C#, we use the break statement to terminate the loop.

As we know, loops iterate over a block of code until the test expression is false. However, sometimes we may need to terminate the loop immediately without checking the test expression.

In such cases, the break statement is used. The syntax of break statement is,

break;

Before we learn about break, make sure to learn about

[for loop](https://www.programiz.com/csharp-programming/for-loop)

[if...else](https://www.programiz.com/csharp-programming/if-else-statement)

[while loop](https://www.programiz.com/csharp-programming/do-while-loop)

Example: C# break statement with for loop

using System;

namespace CSharpBreak {

class Program {

static void Main(string[] args) {

for (int i = 1; i <= 4; ++i) {

// terminates the loop

if (i == 3) {

break;

}

Console.WriteLine(i);

}

Console.ReadLine();

}

}

}

Output

1

2

In the above program, our for loop runs 4 times from i = 1 to 4. However, when i is equal to 3, the break statement is encountered.

if (i == 3) {

break;

}

Now, the loop is terminated suddenly. So, we only get 1 and 2 as output.

Note: The break statement is used with decision-making statements like if..else.

Example: C# break statement with while loop

using System;

namespace WhileBreak {

class Program {

static void Main(string[] args) {

int i = 1;

while (i <= 5) {

Console.WriteLine(i);

i++;

if (i == 4) {

// terminates the loop

break;

}

}

Console.ReadLine();

}

}

}

Output

1

2

3

In the above example, we have created a while loop that is supposed to run from i = 1 to 5.

However, when i is equal to 4, the break statement is encountered.

if (i == 4) {

break;

}

Now, the while loop is terminated.

Working of break statement in C#

Working of break statement

break Statement with Nested Loop

We can also use the break statement with nested loops. For example,

using System;

namespace NestedBreak {

class Program {

static void Main(string[] args) {

int sum = 0;

for(int i = 1; i <= 3; i++) { //outer loop

// inner loop

for(int j = 1; j <= 3; j++) {

if (i == 2) {

break;

}

Console.WriteLine("i = " + i + " j = " +j);

}

}

Console.ReadLine();

}

}

}

Output

i = 1 j = 1

i = 1 j = 2

i = 1 j = 3

i = 3 j = 1

i = 3 j = 2

i = 3 j = 3

In the above example, we have used the break statement inside the inner for loop. Here, the break statement is executed when i == 2.

Hence, the value of i = 2 is never printed.

Note: The break statement only terminates the inner for loop. This is because we have used the break statement inside the inner loop.

If you want to learn the working of nested loops, visit [C# Nested Loops](https://www.programiz.com/csharp-programming/nested-loops).

break with foreach Loop

We can also use the break statement with foreach loops. For example,

using System;

namespace ForEachBreak {

class Program {

static void Main(string[] args) {

int[] num = { 1, 2, 3, 4, 5 };

// use of for each loop

foreach(int number in num) {

// terminates the loop

if(number==3) {

break;

}

Console.WriteLine(number);

}

}

}

}

Output

1

2

In the above example, we have created an array with values: 1, 2, 3, 4, 5. Here, we have used the foreach loop to print each element of the array.

However, the loop only prints 1 and 2. This is because when the number is equal to 3, the break statement is executed.

if (number == 3) {

break;

}

This immediately terminates the [foreach loop](https://www.programiz.com/csharp-programming/foreach-loop).

break with Switch Statement

We can also use the break statement inside a switch case statement. For example,

using System;

namespace ConsoleApp1 {

class Program {

static void Main(string[] args) {

char ch='e';

switch (ch) {

case 'a':

Console.WriteLine("Vowel");

break;

case 'e':

Console.WriteLine("Vowel");

break;

case 'i':

Console.WriteLine("Vowel");

break;

case 'o':

Console.WriteLine("Vowel");

break;

case 'u':

Console.WriteLine("Vowel");

break;

default:

Console.WriteLine("Not a vowel");

}

}

}

}

Output

Vowel

Here, we have used the break statement inside each case. It helps us to terminate the switch statement when a matching case is found.

C# continue Statement

In this tutorial, you will learn about the working of C# continue statement with the help of examples.

In C#, we use the continue statement to skip a current iteration of a loop.

When our program encounters the continue statement, the program control moves to the end of the loop and executes the test condition (update statement in case of for loop).

The syntax for continue is:

continue;

Before we learn about continue, make sure to learn about

[for loop](https://www.programiz.com/csharp-programming/for-loop)

[while loop](https://www.programiz.com/csharp-programming/do-while-loop)

[if...else](https://www.programiz.com/csharp-programming/if-else-statement)

Example1: C# continue with for loop

using System;

namespace ContinueLoop {

class Program {

static void Main(string[] args){

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

if (i == 3) {

continue;

}

Console.WriteLine(i);

}

}

}

}

Output

1

2

4

5

In the above example, we have used the for loop to print numbers from i = 1 to 5. However, the number 3 is not printed.

This is because when the value of i is 3, the continue statement is executed.

// skips the condition

if (i == 3) {

continue;

}

This skips the current iteration of loop and moves the program control to the update statement. Hence, the value 3 is not printed.

Note: The continue statement is usually used with the if...else statement.

Example: C# continue with while loop

using System;

namespace ContinueWhile {

class Program{

static void Main(string[] args) {

int i = 0;

while (i < 5) {

i++;

if (i == 3) {

continue;

}

Console.WriteLine(i);

}

}

}

}

Output

1

2

4

5

Here, we have used the continue statement inside the while loop. Similar to the earlier program, when the value of i is 3, the continue statement is executed.

Hence, 3 is not printed on the screen.

Working of C# continue Statement

Working of continue statement

continue with Nested Loop

We use the continue statement with nested as well. For example:

using System;

namespace ContinueNested {

class Program {

static void Main(string[] args) {

int sum = 0;

// outer loop

for(int i = 1; i <= 3; i++) {

// inner loop

for(int j = 1; j <= 3; j++) {

if (j == 2) {

continue;

}

Console.WriteLine("i = " + i + " j = " +j);

}

}

}

}

}

Output

i = 1 j = 1

i = 1 j = 3

i = 2 j = 1

i = 2 j = 3

i = 3 j = 1

i = 3 j = 3

In the above example, we have used the continue statement inside the inner for loop. Here, the continue statement is executed when j == 2.

Hence, the value of j = 2 is ignored.

If you want to learn the working of nested loops, visit [C# Nested Loops](https://www.programiz.com/csharp-programming/nested-loops).

C# continue with foreach Loop

We can also use the continue statement with foreach loops. For example,

using System;

namespace ContinueForeach {

class Program {

static void Main(string[] args) {

int[] num = { 1, 2, 3, 4, 5 };

foreach(int number in num) {

// skips the iteration

if(number==3) {

continue;

}

Console.WriteLine(number);

}

}

}

}

Output

1

2

4

5

In the above example, we have created an array with values: 1, 2, 3, 4, 5. Here, we have used the foreach loop to print each element of the array.

However, the loop doesn't print the value 3. This is because when the number is equal to 3, the continue statement is executed.

if (number == 3) {

continue;

}

Hence, the print statement for this iteration is skipped.