A Complete C# Tutorial For Beginners to Advanced

By A.K.Raju

This C# tutorial is for students and beginners who want to learn C# programming. Basic requirement to learn C# is basic understanding of programming and some general concepts of object oriented programming. If you're new to OOP, I recommed learning basics of OOP here, Introduction to Object Oriented Programming.

In this tutorial:

1. Introduction to C#
2. Getting Ready
3. First C# Application
4. C# Types
5. Value and Reference Types
6. C# Structs
7. C# Enums
8. C# Class
9. Class Access Modifiers
10. C# Field
11. C# Constructor
12. C# Property
13. C# Method
14. C# Method Overloading
15. C# Expression
16. C# Operator
17. C# Statements
18. C# if else Statement
19. C# switch Statement
20. C# for Statement
21. C# foreach Statement
22. C# do while Statement
23. C# while Statement
24. C# go to Statement
25. C# break Statement
26. C# continue Statement
27. C# return Statement
28. C# Interface
29. C# Partial Class
30. C# Static Class
31. C# Abstract Class
32. C# Array
33. C# String
34. C# Dictionary

35. C# List
36. Summary

1. Introduction to C#

C# is a simple, modern, and object-oriented programming language developed by Microsoft. C# is an open source project managed by the .NET Foundation. C# is a fully mature object-oriented programming language and allows developers to build cross-platform applications for Windows, Web, and mobile platforms. C# apps can be deployed on Linux, Windows, iOS, and Android operating systems.

C# is a modern programming language. We can use C# to build today's modern software applications. C# can be used to develop all kind of applications including Windows client apps, components and libraries, services and APIs, Web applications, Mobile apps, cloud applications, and video games.

Here is a detailed article: What Can C# Do For You

Microsoft supports two software development frameworks, .NET Framework and .NET Core. .NET Framework was launched in 2001 to develop Windows and Web applications. But with the rise of open source trends, Microsoft open sourced language compilers and .NET and the new .NET is called .NET Core. Going forward, there is going to be only one version of .NET, that will be .NET. The next version of .NET is going to be released in 2020, called .NET 5.

Here is a detailed article: The Future of .NET

2. Getting Ready

Before starting your first C# application, you will need an editor or an Integrated Development Environment (IDE), where you can type and compile your code. Visual Studio developed by Microsoft is the best IDE out there for C# developers. The current version of Visual Studio is Visual Studio 2019.

Visual Studio 2019 comes in three different flavors – Visual Studio 2019 Enterprise, Visual Studio 2019 Professional, and Visual Studio 2019 Community.

Visual Studio 2019 Community edition is a free. We'll use Visual Studio 2017 Community edition.

Alternatively, you may also use Visual Studio Code. Visual Studio Code is a free, lightweight, open source code editor for writing and debugging code. VS Code supports major programming languages.

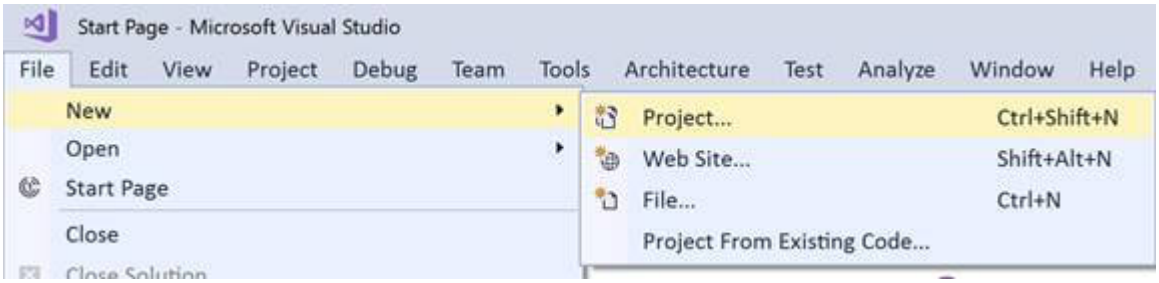
You may download Visual Studio 2017 Community here.

3. First C# Application

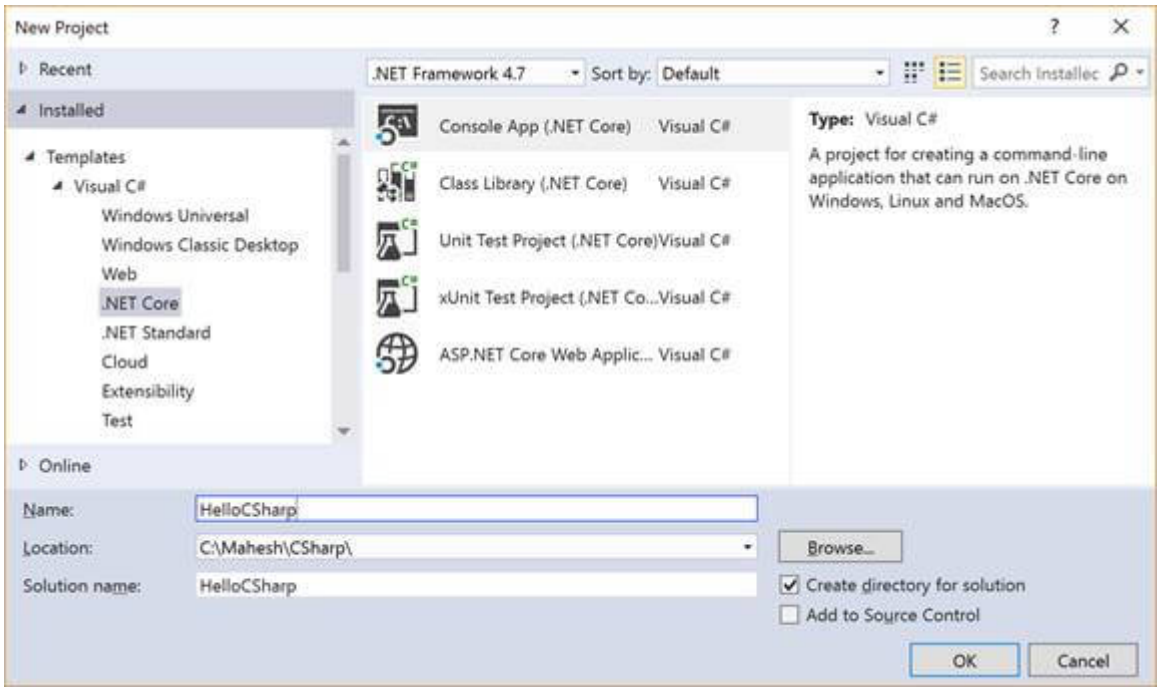
Let’s write our first simple “Hello, World!” program in C#. This is the simplest program you can write in C#. The program will write output on your console saying, “Hello, C# word!”

Open Visual Studio 2019 Community.

Select File > New > Project.



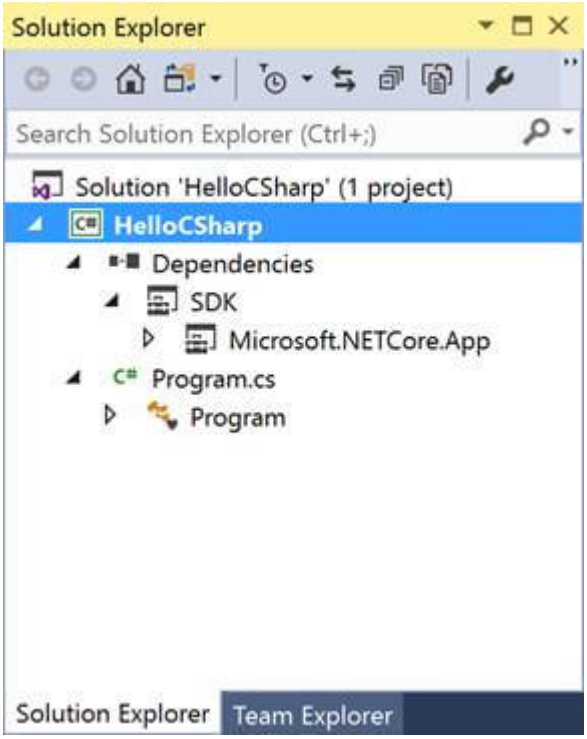
Select Templates > Visual C# > .NET Core > Console App (.NET Core).



Now, give your project a name by typing a name in the Name TextBox. I name my project, HelloCSharp.

Click OK button to create the project.

This action creates a console app.



Double click on Program.cs in Solution Explorer and delete everything in the file.

Now type the following code.

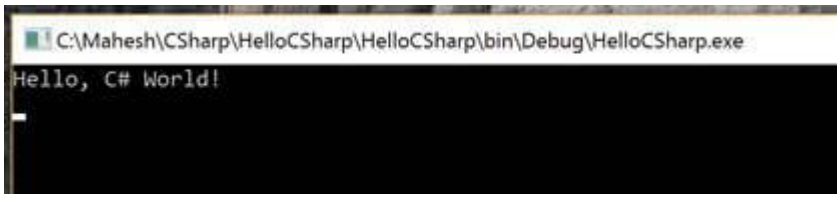
```
using System;
class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, C# World!");
        Console.ReadKey();
    }
}
```

Build and run. Hit Ctrl+F5.

The action compiles and runs your code by creating a HelloCSharp.exe file in your preselected location. The result is “Hello, C# World!” is printed on system console.

As you can see in Figure 5, HelloCSharp.exe is created in C:\Mahesh\Csharp\HelloCSharp\bin\Debug\folder.

Output looks like this,



Well done!

4. C# Types

C# is a strongly typed language. Types in C# can be divided into two categories – built-in types and custom types.

Built-in types are bool, byte, sbyte, char, decimal, double, float, int, uint, long, ulong, object, short, ushort, and string.

The following code in is example of how to declare variables, assign values to them, and use them.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Types Sample");

        // Declare int variable
        int i = 34;
        // char variable
        char c = 'm';
        // float variable
        float per = 6.8f;
        // object variable
        object o = c;
        // string varialbe
        string name = "Mahesh Chand";
        // Use i in an operation
        int counter = i + 1;
        /* Use c in a condition. Check if the value of c is 'm'
        the print c and counter */
        if (c == 'm')
        {
            Console.WriteLine("i is {0}:", i);
            Console.WriteLine("counter is {0}:", counter);
            Console.WriteLine("c is {0}:", c);
            Console.WriteLine("name is {0}:", name);
            Console.WriteLine(counter);
        }

        Console.ReadKey();
    }
}
```

Learn more here: Introduction To Types In C#

5. Value and Reference Types

C# supports value and reference types.

A value type variable contains the actual data within its own memory allocation. Value types derived from System.ValueType. There are two categories of value types, structs and enum.

A reference type is a reference to a memory location where the actual data is stored. The main reference types are class, array, interface, delegate, and event. A null value is assigned to a reference type by default. A type assigned to a null value means the absence of an instance of that type.

Learn more here: C# Concepts - Value Type And Reference Type

6. C# Structs

A struct type is a value type that is typically used to encapsulate a group of variables that are similar. A struct type can declare constructors, constants, fields, methods, properties, indexers, operators, and nested types.

The following code is an example of use of a struct Book uses a record for a book with four members, Title, Author, Price, and Year. The Main program creates an object of Book struct, set its member values, and print the values.

```
using System;

namespace StructSample
{
    // Book struct
    public struct Book
    {
        public string Title;
        public string Author;
        public decimal Price;
        public short Year;
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Struct Sample!");

            // Create a Book object
            Book myBook = new Book();
            myBook.Title = "Simplified C#";
            myBook.Author = "Mahesh Chand";
            myBook.Price = 45.95M;
            myBook.Year = 2017;

            Console.WriteLine($"Book {myBook.Title} was written by {myBook.Author}
" +
               $" in {myBook.Year}. Price is {myBook.Price}");

            Console.ReadKey();
        }
    }
}
```

Learn more here: Struct In C#

7. C# Enums

Enums in C# are used to represent a set of constants as integral values. For example, to represent a week day, we know there are only seven days in a week.

Here are some common examples of struct types.

- WeekDay – Mon, Tue, Wed, Thu, Fri, Sat, Sun
- Months – Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
- Rectangle – Left, Top, Bottom, Right
- Point – X, Y

Here are couple of examples of enums.

```
// Create an enum of week days
public enum WeekDays { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

// Create an enum of RGB colors
public enum RGB { Red, Green, Blue };
```

Learn more here: Enum In C#

8. C# Class

Classes are the foundation of an object-oriented programming language such as C#. A class is a logical unit of data. Classes have members such as properties, fields, methods, and events.

The following code is a Person class with three private members, name, age, and sex. The class also has three public properties. We will discuss private and public properties shortly.

```
// Person class
class Person
{
    // Members of a class
    private string name;
    private Int16 age;
    private string sex;

    // Public properties of a class
    public string Name
    {
        get { return this.name; }
        set { this.name = value; }
    }
    public Int16 Age
    {
        get { return this.age; }
        set { this.age = value; }
    }
    public string Sex
    {
        get { return this.sex; }
        set { this.sex = value; }
    }
}
```

In C#, keyword *new* is used to create an instance of a class. The following code snippet sets the values of Name, Age, and Sex properties of object p.

```
p.Name = "Mahesh Chand";
p.Age = 40;
p.Sex = "Male";
```

The following code now reads the values of the members of object p.

```
// Print Person details
Console.WriteLine("The person {0} is {1} years old {2}.", p.Name, p.Age, p.Sex);
```

9. Class Access Modifiers

Access modifiers are accessibility levels of types and type members and set rules how a type and its members can be used within the type, current assembly, and other assemblies.

The following table describes class or struct member accessibility type and their scopes.

Access Modifier	Scope
public	The type or members can be accessed by the code in the same assembly or assemblies that reference it.
private	The type or members can be accessed only by code in the same class.
protected	The type or members can be accessed only by code in the same class, or the classes that are derived from it.
internal	The type or members can be accessed by the code in the same assembly but not from other assemblies.
protected internal	The type or member can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly.
private protected	The type of member can be accessed within the same class or the classes derived within the same assembly.

10. C# Field

A field member is a variable of a class or struct. A field can be public, private, protected, internal, or protected internal.

Fields in a class can have different accessibility levels. In the following code, fields name, age, and sex are private fields. That means they can only be accessed within the same class only. Public fields can be accessed from anywhere. Protected fields can be accessed within the Person class and any classes derived from it.

```
/// Person class
class Person
{
    // Members of a class
    private string name;
    private Int16 age;
    private string sex;
    protected string book;
    protected internal int code;
    public Int16 Year;
}
```

A field member can also be read-only. This means the field can only be assigned in the declaration or in the constructor of the class.

If the field is not static, you have to access fields from the class instance. The code snippet in Listing 20 creates an instance of the *Person* class and sets the value of *Year* field.

```
Person p = new Person();
p.Year = 2000;
```

Listing 20.

A field can also be static. That means, the fields can be available to the code without creating an instance of a class. A static field is declared using the static keyword and a readonly field is declared using the readonly keyword. These both keywords can be combined to declare a readonly static field. The following code snippet declares a readonly static variable.

```
public static readonly string author = "Mahesh Chand";
```

Fields can be variables or constants.

Learn more here: Fields and Properties in C#

11. C# Constructor

Constructors are responsible for creating an instance of a class or struct. Constructors are methods with the same name as a class or struct and must be called to create an instance of a class or struct. The constructor is called using the new operator.

Let’s look at a simple class Person that has four members.

```
/// Person class
class Person
{
    // Members of a class
    private string name;
    private int age;
    private string sex;
    private string profession;
}
```

For example, the following code snippet creates a Person object using the new operator and calls a constructor.

```
Person p = new Person();
```

Once an object is created (p in this case), now it can be used to call other members of ‘p’. For example, the following code snippet sets the value of name of ‘p’.

```
p.name = "Mahesh Chand";
```

Constructors are used to initialize the data members of new objects. A class or struct can have one or more than one constructor. A constructor does not have a return type.

Learn more here: Constructors and Its Types in C# With Examples

12. C# Property

Property members of a class provides its callers to access a class's private field members. Property members expose private fields through special methods called *accessor*. The *get* accessor is used to return the property value and the *set* accessor is used to assign a new value to the property.

Let's take a look at the following code that declares the Person class with three private fields, name, age, and sex. The private fields are exposed to the external programs through public properties, Name, Age, and Sex.

```
/// Person class
class Person
{
    // Members of a class
    private string name;
    private int age;
    private string sex;

    // Constructor
    public Person()
    {
    }

    // Public properties of a class
    public string Name
    {
        get { return this.name; }
        set { this.name = value; }
    }
    public int Age
    {
        get { return this.age; }
        set { this.age = value; }
    }
    public string Sex
    {
        get { return this.sex; }
        set { this.sex = value; }
    }
}
```

The following code creates a Person class objects and sets its properties. Once the properties are set, the actual values of the properties actually copied inside the object. The following code reads the properties values and prints them on the console.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        // Create a Person object
        Person p = new Person();
        // Set Person properties
        p.Name = "Mahesh Chand";
         = 40;
        p.Sex = "Male";

        // Get Person properties
        Console.WriteLine("Person 1 {0} is {1} years old {2}.", p.Name, p.Age, p.S
ex);

        Console.ReadKey();
    }
}
```

Learn more here: Understanding Properties In C#

13. C# Method

A method member of a class is a block of code that performs a specific task. A method signature is a combination of an access modifier followed by an additional modifier, return type, method name, and method parameters. A typical method signature looks like the following, where a public method returns a bool value and takes two parameters. The method's code block starts with an open bracket '{' and ends with a closing bracket '}.

```
// Method signature
public bool MethodName(int param1, string param2)
{
    // Implementation
    return false;
}
```

A method can return a void or a data type. Method can also take arguments and also return multiple values.

Let's add a method, SayHello to our Person class, that prints a message with the person's name to the console. The method SayHello does not return a value. The final Person class looks like the following.

```
/// Person class
class Person
{
    // Members of a class
    private string name;
    private int age;
    private string sex;
    private string profession;
    private bool retired = false;

    // Overloaded constructor
    public Person(string personName, int personAge, string personSex)
    {
        this.name = personName;
        this.age = personAge;
        this.sex = personSex;
    }

    // Public properties of a class
    public string Name
    {
        get { return this.name; }
        set { this.name = value; }
    }

    public int Age
    {
        get { return this.age; }
        set { this.age = value; }
    }
    public string Sex
    {
        get { return this.sex; }
        set { this.sex = value; }
    }

    /// <summary>
    /// A simple method prints out a message
    /// </summary>
    public void SayHello()
    {
        if (this.name.Length > 0)
            Console.WriteLine("Hello {0} from the Person class", this.name);
    }
}
```

Now let’s call this method from our Main method. As you’ve seen earlier in case of properties and fields, we call a method in a similar way we can any other members of a class or struct. The following code creates an instance of Person class, sets its Name property, and calls SayHello method.

```
using System;
class Program
{
    static void Main(string[] args)
    {
        // Create a Person object
        Person person = new Person("Mahesh Chand", 40, "male");
        // Call SayHello method
        person.SayHello();

        Console.ReadKey();
    }
}
```

Often times, a caller program needs to pass some data to a class. The passed data may be used for processing, business logic, and decision making. The last portion of a method signature is a list of method parameters that are passed by the caller program.

A method defines the type and name of parameters. Multiple parameters are separated by commas.

Here is a method that takes three parameters and returns a value.

```
public decimal CalculateSalary(int hours, decimal rate, decimal bonus)
{
    // Payment = number of hours worked * rate + bonus
    return (hours * rate) + bonus;
}
```

Now let’s call the CalculateSalary method from our Main method.

```
// Calculate salary of Person
decimal salary = person.CalculateSalary(160, 50, 3000);
Console.WriteLine("{0} gets paid ${1}.", person.Name, salary);
```

14. C# Method Overloading

C# allows you to declare same name methods differentiated by their signature. This process is called method overloading. The following class RateCalculator declares a method CalculateCost with three different signatures.

```
public class RateCalculator
{
    private decimal total;

    // Ref return method
    public decimal CalculateCost(decimal price, decimal rate)
    {
        total = price * rate;
        return total;
    }

    // Method with an out parameter
    public void CalculateCost(decimal price, decimal rate, out decimal total)
    {
        total = price * rate;
    }

    // Method with multiple return values
    public void CalculateCost(decimal price, decimal rate, out decimal total, out
decimal min, out decimal max)
    {
        min = rate * 0.5m;
        max = rate * 2;
        total = price * rate;
    }
}
```

The call of a method depends on the number and types of arguments passed to the method call. The following code calls these three difference overloaded methods.

```
// Method overloading sample
using System;
class Program
{
    static void Main(string[] args)
    {
        RateCalculator calculator = new RateCalculator();
        decimal total = calculator.CalculateCost(12.5m, 6.8m);
        calculator.CalculateCost(12.5m, 6.8m, out decimal cost);
        calculator.CalculateCost(12.5m, 6.8m, out decimal totalcost, out decimal m
inrate, out decimal maxrate);
        Console.WriteLine("Total cost {0}", totalcost);

        Console.ReadKey();
    }
}
```

Learn more here: Method Overloading In C#

15. C# Expression

An expression is a sequence of operators and operands that specify some sort of computation and the result is evaluated in a single value or object. The following code snippet is an example of two expressions.

```
Convert.ToInt32("12");
10 + 5;
```

The first expression converts a string “12” to an integer value and the second expression adds two numbers.

Typically, expressions are used in statements. For example, the following code snippet is an example of a statement. The right side of the statement is an expression that adds two numbers.

```
int a = 10 + 5;
```

An expression can consist literal values, and method invocations. The following code snippet uses literal names to add in an expression.

```
string name = "Mahesh" + "Chand";
```

An expression can also be used in method invocation. A method invocation requires the name of the method, followed by parenthesis and any method parameters. The following code snippet invokes two methods.

```
Console.WriteLine("Operators Sample!");
Console.ReadKey();
```

Learn more here: Expressions 😀and Operators in C#

16. C# Operator

An operator is responsible for an operation. For example, the operators + and - indicate adding and subtracting operands, such as values, objects, and literals. An operand can also be an expression, or any number of sub expressions.

The following code is a simple example of operators and operands.

```
using System;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Operators Sample!");

        // Declare variables
        int a = 10; int b = 20;
        decimal d = 22.50m;
        string first = "Mahesh"; string last = "Chand";

        // Use + and - operators to add and subtract values
        int total = a + b;
        decimal dTotal = a + b + d;
        int diff = b - a;
        Console.WriteLine("diff: " + diff);
        string name = first + " " + last;
        Console.WriteLine("Name: " + name);

        Console.ReadKey();
    }
}
```

The following line is a code expression. The '+' operator applies to two variables, a and b, and the result is stored in variable total.

```
int total = a + b;
```

The code also declares other variables of type decimal, and strings. The following code snippet adds a decimal value to two int values

```
decimal dTotal = a + b + d;
```

The following code snippet adds two string values. An extra space is also added during the operation.

```
string name = first + " " + last;
```

There are three types of operators, unary, binary, and ternary.

Learn more here: Operators in C#

17. C# Statements

A statement is a part of a program that represents an action such as declaring variables, assigning values, calling methods, loop through a collection, and a code block with brackets. A statement can consist of a single line ends with a semicolon, or a code block enclosed in {} brackets.

The following code snippet has two statements.

```
decimal d = 22.50m;
string first = "Mahesh";
```

Here is an if..else statement with several lines of code.

```
if (d > 22)
{
    d -= 10;
    Console.WriteLine("d is {0}", d);
}
else
{
    d += 10;
    Console.WriteLine("d is {0}", d);
}
```

18. C# if . . .else Statement

The if . . .else statement is also known as a conditional statement. Here is the syntax of if..else statements.

```
if (condition)
{
    Statement
}
else
{
    Statement
}
```

The if. . .section of the statement or statement block is executed when the condition is true; if it's false, control goes to the else statement or statement block. The 'else' portion of the statement is optional.

The following code uses if statement to check if the value of a is less than 0, then display a message, 'a is negative'.

```
int a = -1;
if (a < 0)
{
    Console.WriteLine("a is negative.");
}
```

The following code snippet uses if..else statement to check if the value of a is less than 0. If not, then display a message, 'a is 0 or positive'.

```
int a = -1;
if (a < 0)
{
    Console.WriteLine("a is negative.");
}
else
{
    Console.WriteLine("a is 0 or positive.");
}
```

The {} brackets are optional for a single line statement.

```
if (a < 0)
    Console.WriteLine("a is negative.");
else
        Console.WriteLine("a is 0 or positive.");
```

You can have a nested if . . .else statement with one of more else blocks.

You can also apply conditional or (||) and conditional and (&&) operators to combine more than one condition.

```
int a = -1; int b = 10; int c;

// Check if both numbers are negative
if (a < 0 && b < 0)
{
    Console.WriteLine("Both a and b are negative.");
}
else if (a < 0 || b < 0)
{
    Console.WriteLine("One number is negative.");
}
else
{
    Console.WriteLine("Both numbers are positive.");
}
```

The if..else statement can have other nested statements and also can have more than one if..else statements.

19. C# switch Statement

C# *switch* statement is a conditional statement that pairs with one or more *case* blocks and a default block. The case block of code is executed for the matching value of switch expression value. If the switch value doesn’t match the case value, the *default* option code is executed.

```
switch (expression)
{
    case expression_value1:
        Statement
        break;
    case expression_value2:
        Statement
        break;
    case expression_value3:
        Statement
        break;
    default:
        Statement
        break;
}
```

The expression in the above code can be any non-null expression.

The following code is a typical switch statement. The *switch* expression is a random number between 1 and 9 and based on the value of the expression, a case block is executed. If the value of switch expression is more doesn’t match with first three case values, the default block is executed.

```
// Generate a random value between 1 and 9
int caseSwitch = new Random().Next(1, 9);

switch (caseSwitch)
{
    case 1:
        Console.WriteLine("Case 1");
        break;
    case 2:
        Console.WriteLine("Case 2");
        break;
    case 3:
        Console.WriteLine("Case 3");
        break;
    default:
        Console.WriteLine("Value didn’t match earlier.");
        break;
}
```

Here is a detailed tutorial on switch statement: C# Switch Statement

20. C# for Statement

The for statement also knows as the for loop executes a block of code.

Syntax of for loop:

for (initializer; condition; iterator)

statement

This code is the simplest example of a for loop. In this code, the variable counter is initialized to value 0. The condition is until counter is less than or equal to 100, add +1 to counter. The code block displays the value of counter to the console.

```
using System;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("for loop Sample!");
        // For loop from 0 to 100
        for (int counter = 0; counter <= 100; counter++)
            Console.WriteLine(counter);

        Console.ReadKey();
    }
}
```

The initializer defines the initial value to start with and the iterator is the increment or decrement. The following code snippet starts a loop from 10 to less than 100 in increments of 10.

```
for (int counter = 10; counter <= 100; counter += 10)
{
    Console.WriteLine(counter);
}
```

The statement part of the loop has a code of block that will be executed when the condition is met. The following code blocks checks for the even numbers between 10 and 100.

```
for (int counter = 10; counter <= 100; counter += 10)
{
    if (counter % 2 != 0)
    {
        Console.WriteLine(counter);
    }
}
```

21. C# foreach Statement

The foreach statement is used to iterate through a collection of items such as an array. The foreach body must be enclosed in {} braces unless it consists of a single statement. We will discuss arrays and collections later in this book.

The following code creates an array of odd numbers and uses foreach loop to loop through the array items and read them.

```
using System;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("foreach loop Sample!");
        int[] oddArray = new int[] { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21 };
        foreach (int num in oddArray)
        {
            Console.WriteLine(num);
        }

        Console.ReadKey();
    }
}
```

Here is an example of for loop that can also be used read an array items.

```
for (int counter = 0; counter < oddArray.Length; counter++)
{
    Console.WriteLine(oddArray[counter]);
}
```

Here is a detailed tutorial on foreach: C# foreach loop

22. C# do..while Statement

The do..while statement executes a block of code until the specified while condition is false. The code body must be enclosed in {} braces unless it consists of a single statement.

The following code continues a loop until the counter is less than 20.

```
using System;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("do..while loop Sample!");
        int counter = 0;
        do
        {
            Console.WriteLine(counter);
            counter++;
        } while (counter < 20);

        Console.ReadKey();
    }
}
```

23. C# while Statement

The while statement executes a block of code until the specified while condition is false. The code body must be enclosed in {} braces unless it consists of a single statement.

The following code continues a loop until the counter is less than 20.

```
using System;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("do..while loop Sample!");
        int counter = 0;
        while (counter < 20)
        {
            Console.WriteLine(counter);
            counter++;
        }

        Console.ReadKey();
    }
}
```

24. C# go to Statement

The goto statement is used when you need to jump to a particular code segment. The following code uses a goto statement to jump from one case block to another once a condition is met. The program reads a string from the console. When you type, ‘Mahesh’, the case statement sends control to case statement, ‘Chand’.

Note that when using a go to in a case statement, you don’t have to provide a break (in all other cases, a break statement is mandatory).

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");

        Console.WriteLine("What is your name? ");
        string name = Console.ReadLine();
        switch (name)
        {
            case "Mahesh":
                Console.WriteLine("My name is Mahesh.");
                goto case "Chand";
            case "Chand":
                Console.WriteLine(@"Chand is my last name.");
                break;
            case "Neel":
                Console.WriteLine("My name is Neel. ");
                break;
            default:
                break;
        }
        Console.ReadKey();
    }
}
```

25. C# break Statement

The break statement exits from a loop or a switch immediately. The break statement is usually applicable when you need to release control of the loop after a certain condition is met, or if you want to exit from the loop without executing the rest of the loop structure. You use it in for, foreach, while, and do. . . while loop statements. The following code shows the break statement. If condition num == 15 is true, the control will exit from the loop.

```
// Array of odd numbers
int[] oddArray = new int[] { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21 };
// Loop through array items
foreach (int num in oddArray)
{
    Console.WriteLine(num);
    // Don't read any number after 15
    if (num == 15) break;
}
```

26. C# continue Statement

Similar to the break statement, the continue statement also works in for, foreach, while, and do . . . while statements. The continue statement causes the loop to exit from the current iteration and continues with the rest of the iterations in the loop.

```
for (int counter = 0; counter <= 10; counter++)
{
    if (counter == 5)
        continue;
    Console.WriteLine(counter);
}
```

In the above code, when the condition counter == 5 is true, the control exits from the current iteration and moves to the next iteration.

27. C# return Statement

The return statement returns from a method before the end of that method is reached. The return statement can either a value or not, depending on the method that calls it.

The following code is an example of a return statement. The execution goes to the caller program of the method.

```
if (counter == 9)
    return;
```

28. C# Interface

An interface is a blueprint that an implementing class or struct must implement. An interface contains only the signatures of its members. Interfaces are used by library and component developers when certain blueprint of the classes or structs must be followed by the implementing developers.

An interface has the following properties:

- Any class or struct that implements the interface must implement all its members.
- An interface can't be instantiated directly. Its members are implemented by any class or struct that implements the interface.
- Interfaces can contain events, indexers, methods, and properties.
- Interfaces contain no implementation of methods.
- A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.
- The corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member.
- Interfaces can also have base interfaces and an interface can inherited from more than one base interfaces.

Learn more here: Interface In C#

29. C# Partial Class

Partial classes were introduced in C# 2 that allows developers to split a class into multiple physical files. Partial classes come handy in large projects and a class size is too large and functionality is distributed among multiple members in a team.

In large projects, multiple team members work on same area of functionality. Let's assume that an application has a user interface (UI), a window or a page. The UI has ton of functionality where multiple developers needs to work on the same window functionality. One developer will create the user interface, change design and layout, while other two developers may work on the database and business logic functionality. We can easily separate the window code into three physical partial classes and each developer can work on a separate class.

Learn more here: Partial Classes In C# with Real Example

30. C# Static Class

Static classes don't have instance constructors and are used direct without creating an instance. Here are static classes properties.

- A static class cannot be instantiated. That means you cannot create an instance of a static class using new operator.
- A static class is a sealed class. That means you cannot inherit any class from a static class.
- A static class can have static members only. Having non-static member will generate a compiler error.
- A static class is cannot contain instance constructors.
- A static constructor is only called one time, and a static class remains in memory for the lifetime of the application domain in which your program resides.

Learn more here: Static Class In C#

31. C# Abstract Class

The purpose of an abstract class is to define a blueprint of derived classes. Abstract classes do not have any implementation. The classes that are inherited from an abstract class implements the class functionality.

Here are the key characteristics of abstract classes:

- An abstract class cannot be instantiated.
- A class may inherit one abstract class only.
- An abstract class may contain abstract methods, properties, and events.
- An abstract class can't be modified with the sealed class modifier.
- A derived from an abstract class must implement all inherited abstract methods and accessors.

Learn more here: Abstract Class In C#

32. C# array

An Array in C# is a collection of objects or types. C# Array elements can be of any type, including an array type. An array can be Single-Dimensional, Multidimensional or Jagged. A C# Array can be can be declared as fixed length or dynamic. An Array in C# can be a single dimension, multi dimension, or a jagged array. Learn how to work with arrays in C#.

In C#, an array index starts at zero. That means the first item of an array starts at the 0thposition. The position of the last item on an array will total number of items - 1. So if an array has 10 items, the last 10th item is at 9th position.

In C#, arrays can be declared as fixed length or dynamic. A *fixed length* array can store a predefined number of items. A *dynamic array* does not have a predefined size. The size of a *dynamic array* increases as you add new items to the array. You can declare an array of fixed length or dynamic. You can even change a dynamic array to static after it is defined.

The following code snippet creates an array of 3 items and values of these items are added when the array is initialized.

```
// Initialize a fixed array
int[] staticIntArray = new int[3] {1, 3, 5};
```

Here is a detailed tutorial on Arrays: Working with Arrays In C#

33. C# string

The System.String data type represents a string in .NET. A string class in C# is an object of type System.String. The String class in C# represents a string.

The following code creates three strings with a name, number, and double values.

```
// String of characters
System.String authorName = "Mahesh Chand";

// String made of an Integer
System.String age = "33";

// String made of a double
System.String numberString = "33.23";
```

Here is the complete example that shows how to use stings in C# and .NET.

```
using System;
namespace CSharpStrings
{
    class Program
    {
        static void Main(string[] args)
        {
            // Define .NET Strings
            // String of characters
            System.String authorName = "Mahesh Chand";

            // String made of an Integer
            System.String age = "33";

            // String made of a double
            System.String numberString = "33.23";

            // Write to Console.
            Console.WriteLine("Name: {0}", authorName);
            Console.WriteLine("Age: {0}", age);
            Console.WriteLine("Number: {0}", numberString);
            Console.ReadKey();
        }
    }
}
```

Here is a detailed tutorial on strings: String In C#

34. C# List

List<T> class in C# represents a strongly typed list of objects. List<T> provides functionality to create a list of objects, find list items, sort list, search list, and manipulate list items. In List<T>, T is the type of objects.

List<T> is a generic class and is defined in the System.Collections.Generic namespace. You must import this namespace in your project to access the List<T> class.

```
using System.Collections.Generic;
```

List<T> class constructor is used to create a List object of type T. It can either be empty or take an Integer value as an argument that defines the initial size of the list, also known as capacity. If there is no integer passed in the constructor, the size of the list is dynamic and grows every time an item is added to the array. You can also pass an initial collection of elements when initialize an object.

This code snippet 1 creates a List of Int16 and a list of string types. The last part of the code creates a List<T> object with an existing collection.

```
// List with default capacity
List<Int16> list = new List<Int16>();
// List with capacity = 5
List<string> authors = new List<string>(5);
string[] animals = { "Cow", "Camel", "Elephant" };
List<string> animalsList = new List<string>(animals);
```

As you can see from Listing 1, the List<string> has an initial capacity set to 5 only. However, when more than 5 elements are added to the list, it automatically expands.

Here is a detailed tutorial on list. C# List Tutorial

35. C# dictionary

The Dictionary type represents a collection of keys and values pair of data.

C# Dictionary class defined in the System.Collections.Generic namespace is a generic class and can store any data types in a form of keys and values. Each key must be unique in the collection.

Before you use the Dictionary class in your code, you must import the System.Collections.Generic namespace using the following line.

```
using System.Collections.Generic;
```

The Dictionary class constructor takes a key data type and a value data type. Both types are generic so it can be any .NET data type.

The following The Dictionary class is a generic class and can store any data types. This class is defined in the code snippet creates a dictionary where both keys and values are string types.

```
Dictionary<string, string> EmployeeList = new Dictionary<string, string>();
```

The following code snippet adds items to the dictionary.

```
EmployeeList.Add("Mahesh Chand", "Programmer");
EmployeeList.Add("Praveen Kumar", "Project Manager");
EmployeeList.Add("Raj Kumar", "Architect");
EmployeeList.Add("Nipun Tomar", "Asst. Project Manager");
EmployeeList.Add("Dinesh Beniwal", "Manager");
```

Here is a detailed tutorial on dictionary: Dictionary In C#

36. Summary

This tutorial is an introduction to C# language for beginners. In this tutorial, we learned how to write our first C# program, basics of data types, classes, objects, and class members.