C# Tutorial

C# is a simple, modern, general-purpose, object-oriented programming language developed by Microsoft.

The goal of this tutorial is to make learning programming easy and we intend to make each topic easy to understand and also provide complete knowledge. So on each topic, we go straight to the point for practice first, and then we support addition information for you to understand better.

Hello World Program in C#

```CSharp

using System;

namespace HelloWorldApp

{

class HelloWorld

{

static void Main(string[] args)

{

Console.WriteLine("Hello World");

Console.ReadKey();

}

}

}

```

If you don’t understand, don’t worry. We will walk you through soon.

Sections

- Basics - cover features that applies to most of programming language

- Intermediate - cover c# specific features

- Advanced - software design architecture

### Programming Basics

This section cover the basic syntax that appears to almost all modern programming languages. The topics consist of the following:

\* Basic syntax

\* Data types

\* Type Conversion

\* Many more

using Keyword

class

Comments

Let’s start by building an application for recording student grade.

#### Variable and Data type

In a real world scenario, how to define a total score of an exam to be 100?

In C#, we do like this:

int TotalScore = 100;

int is a data type, I is variable name, = is operator, 100 is value, This means an integer named TotalScore is equal to 100.

Here is how to define variables

<data\_type> <variable\_list>;

Examples:

int i, j, k;

char c, ch;

float f, salary;

double d;

In the example above, we assigned the initial value of totalScore to 100, we can also not assign value but it’s a good practice to initialize variable with a value.

What are those int, char, float? They are data types, let’s take a look.

### Data Types

There are three category of data types

\* Value types

\* Reference types

\* Pointer types

Like what we saw in variable section, int is a value type

**Value Type**

The value types directly contain data.

| Type | Represents | Range | Default Value |

| ------ | ------ | ------ | ------ |

| bool | Boolean value | True or False | False |

byte | 8-bit unsigned integer | 0 to 255 | 0

char | 16-bit Unicode character | U +0000 to U +ffff | '\0'

decimal | 128-bit precise decimal values with 28-29 significant digits | (-7.9 x 1028 to 7.9 x 1028) / 100 to 28 | 0.0M

double | 64-bit double-precision floating point type | (+/-)5.0 x 10-324 to (+/-)1.7 x 10308 | 0.0D

float | 32-bit single-precision floating point type | -3.4 x 1038 to + 3.4 x 1038 | 0.0F

int | 32-bit signed integer type | -2,147,483,648 to 2,147,483,647 | 0

long | 64-bit signed integer type | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | 0L

sbyte | 8-bit signed integer type | -128 to 127 | 0

short | 16-bit signed integer type | -32,768 to 32,767 | 0

uint | 32-bit unsigned integer type | 0 to 4,294,967,295 | 0

ulong | 64-bit unsigned integer type | 0 to 18,446,744,073,709,551,615 | 0

ushort | 16-bit unsigned integer type | 0 to 65,535 | 0

As we saw, value types can be intilized without value, but reference type cannot.

The most common reference type is string.

String str = "C# is great";

**Reference types**

The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables.

built-in reference types are: object, dynamic, and string.

object: ultimate base class for all data types, can be assigned values of any other types. type checking at compile time

dynamic: can be assigned any value. Type checking at run-time.

string: allows you to assign any string values to a variable,

Strings in C# are a built-in type, as well as a standard .NET type. The string keyword in C# is the same as the System.String type, and you can declare variables using either version without any impact on your program's behavior. When a string is instantiated in C#, a sequence of bytes in memory is allocated to it. This allocation cannot be modified after-the-fact; it can only be erased once the string is no longer in use by the program. To change a string, or to create a new string out of one or more others, a new memory allocation is required. Because string values cannot be changed once they are created, strings are said to be immutable. This aspect of strings can have performance implications in your programs, so it's helpful to understand.

**nullable** types

< data\_type> ? <variable\_name> = null;

For example

int? num1 = null;

int? num2 = 45;

double? num3 = new double?();

double? num4 = 3.14157;

Type Conversion Methods

* **Implicit type conversion** - These conversions are performed by C# in a type-safe manner. For example, are conversions from smaller to larger integral types and conversions from derived classes to base classes.
* **Explicit type conversion** - These conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.

Constants

Constants refer to fixed values that the program may not alter during its execution.

It can be initialized like this:

const <data\_type> <constant\_name> = value;

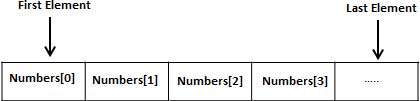
we can set constant to prevent people changing it

const int TotalScore = 100;

**Arrays and Lists**

We demonstrated how to store the score of one student but how to store the score of the whole class of students? That’s where array comes in.

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type stored at contiguous memory locations.



Declaring Arrays

datatype[] arrayName;

Assigning Values to an Array

double[] balance = { 2340.0, 4523.69, 3421.0};

You can assign values to the array at the time of declaration, as shown:

double[] balance = { 2340.0, 4523.69, 3421.0};

You can also create and initialize an array, as shown:

int [] marks = new int[5] { 99, 98, 92, 97, 95};

You may also omit the size of the array, as shown:

int [] marks = new int[] { 99, 98, 92, 97, 95};

Accessing Array Elements

double salary = balance[9];

.NET includes support for a variety of collection types, including the most commonly used List type. Typically, you'll want to work with a list of a particular type of item, so you'll declare a generic List<T>, where T in this case is the type of objects the list will hold. When you refer to a List<T> verbally, you'll say "a list of T", which makes sense since that really is what it represents. A List<int> is a list of ints.

This makes them more flexible than arrays. Generic lists are defined in the System.Collections.Generic namespace;

Declaring Lists

List<int> someInts = new List<int>(); // declares an empty list

Working with Lists

Add method

AddRange method

Structure

We talked about array which store data in the same type, but what if we want to store data of different types or just in different names? I want to put a student’s scores in every exams together, how?

Introducing structure, structure is a value type data type. It helps you to make a single variable hold related data of various data types.

For example, here is the way you can declare a student score structure:

struct StudentScore

{

public string StudentName;

public int EnglishScore;

public int MathScore;

public int ScienceScore;

};

We can use struct but class is most often used

Classes and Structures have the following basic differences:

* classes are reference types and structs are value types
* structures do not support inheritance
* structures cannot have default constructor

## Class

## Class is a blueprint for a data type, Objects are instances of a class, The methods and variables that constitute a class are called members of the class.

<access specifier> class class\_name

{

// member variables

<access specifier> <data type> variable1;

<access specifier> <data type> variable2;

...

<access specifier> <data type> variableN;

// member methods

<access specifier> <return type> method1(parameter\_list)

{

// method body

}

<access specifier> <return type> method2(parameter\_list)

{

// method body

}

...

<access specifier> <return type> methodN(parameter\_list)

{

// method body

}

}

We will introduce method later

In real scenario, sometimes we have student grade in A, B, C, D, E but not in numbers, how to address this situation?

## Declaring enum Variable

The general syntax for declaring an enumeration is:

enum <enum\_name>

{

enumeration list

};

C# enumerations are value data type. In other words, enumeration contains its own values and cannot inherit or cannot pass inheritance.

enum Grade { A, B, C, D, E, F };

Each of the symbols in the enumeration list stands for an integer value, one greater than the symbol that precedes it. By default, the value of the first enumeration symbol is 0. For example:

Enum advantages. With an enum, magic constants are separate. This modular design makes things easier to understand. Fewer bugs will be introduced.

**Method**

A method is a group of statements that together perform a task.

Defining Methods

<Access Specifier> <Return Type> <Method Name>(Parameter List)

{

//Method Body

}

A method to calculate student’s total score

Following are the various elements of a method:

* **Access Specifier**: This determines the visibility of a variable or a method from another class.
* **Return type**: A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is **void**.
* **Method name**: Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.
* **Parameter list**: Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
* **Method body**: This contains the set of instructions needed to complete the required activity.

**Access** modifiers

The access modifiers used in this example are public, private, and protected. This lesson will also explain the internal modifier, but that's more related to the [Understanding Namespaces](https://www.microsoft.com/net/tutorials/csharp/getting-started/namespaces) lesson.

Notice that the access modifiers are the first word in the declaration of the classes, methods, and properties. This holds for most cases, however, you'll notice one exception to this in the example:

protected string FirstName { get; private set; }

With properties, it's possible to further restrict the setter method, so that it's less accessible than the getter method.

### **public**

Code that's available for use by any other code should use the public access modifier. This, as its name suggests, makes it available publicly for any other code to use. It's simple to understand and work with. The IsAnAdult() method in the example above uses this access modifier, because any other code is allowed to call it:

public bool IsAnAdult()

{

var eighteenYearsAgo = DateTime.Today.AddYears(-18);

return this.DateOfBirth < eighteenYearsAgo;

}

### **private**

For code that should only be usable by other code in the same class, private is the correct access modifier. This is the most restrictive of the access modifiers, and is used to restrict this property setmethod in the example above:

public DateTime DateOfBirth { get; private set; }

Because it is private, only the Person class can set this value; the inheriting Student is not even able to modify the value.

### **Protected**

When dealing with inheritance, the protected access modifier is often useful. It allows a child class to use some of the otherwise restricted members of the parent. In the example above, FirstName and LastNameare only accessible from within Person or its child classes, as you can see in the Student class's RosterName property.

public string RosterName { get { return $"{this.LastName}, {this.FirstName}"; } }

### **internal**

Like protected, internal is more accessible than private, but less than public. When using this access modifier, the code being modified may be used by any other code in the same assembly. These keep developers without access to your assembly's source code from using the internal code.

**Note**   
The internal and protected may be used in combination. Doing this will create the union of the two allowances rather than the limitations, meaning access is provided to inheriting classes as well as within the same assembly.

**Properties and Fields**

*Fields* are types that are attached to a class. They track the state of the class, and separate instances of the same class will each track the data of their fields independently. *Properties* provide a way for other objects to access state from an object in a controlled manner. Unlike fields, which are essentially just variables, properties are methods and can add additional behavior around manipulating the state of an object.

This sample shows a Person class that has two properties: Name (string) and Age (int). Both properties are read/write.

using System;

class Person

{

private string myName ="N/A";

private int myAge = 0;

// Declare a Name property of type string:

public string Name

{

get

{

return myName;

}

set

{

myName = value;

}

}

// Declare an Age property of type int:

public int Age

{

get

{

return myAge;

}

set

{

myAge = value;

}

}

public override string ToString()

{

return "Name = " + Name + ", Age = " + Age;

}

public static void Main()

{

Console.WriteLine("Simple Properties");

// Create a new Person object:

Person person = new Person();

// Print out the name and the age associated with the person:

Console.WriteLine("Person details - {0}", person);

// Set some values on the person object:

person.Name = "Joe";

person.Age = 99;

Console.WriteLine("Person details - {0}", person);

// Increment the Age property:

person.Age += 1;

Console.WriteLine("Person details - {0}", person);

}

}

Notice the way that the properties are declared, for example, consider the Name property:

public string Name

{

get

{

return myName;

}

set

{

myName = value;

}

}

### Constructors and Property Initializers

in the above example, Person's properties all start out uninitialized. Thus, the string properties will be null, and the DateOfBirth property will initially be DateTime.MinValue, which probably isn't what's desired. You can specify default values for a class's members in its constructor. The constructor is run when an instance of the class is created, before any code can interact with the new instance. They’re a good place to ensure fields and properties are initialized. You can see an example of a Person constructor below:

Constructors can also take parameters.

public class Person

{

public Person(DateTime dateOfBirth)

{

DateOfBirth = dateOfBirth;

}

public string FirstName { get; set; } = string.Empty;

public string LastName { get; set; } = string.Empty;

public DateTime DateOfBirth { get; set; }

public string TaxPayerId { get; set; } = string.Empty;

}

Composition

Fields and properties provide a way for you to *compose* your objects from other objects. This gives you a powerful way to add and share properties and behavior between objects. For instance, let's say you needed to mail things to people and companies as part of your program. You could put all of the necessary properties on both the Person and Company classes:

public class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

public string StreetAddress { get; set; }

public string City { get; set; }

public string State { get; set; }

public string PostalCode { get; set; }

public string Country { get; set; }

}

public class Company

{

public string Name { get; set; }

public string StreetAddress { get; set; }

public string City { get; set; }

public string State { get; set; }

public string PostalCode { get; set; }

public string Country { get; set; }

}

However, a better approach would be to pull out the properties that logically can be thought of as an *address*, and compose the other classes using this new type:

public class Address

{

public string StreetAddress { get; set; }

public string City { get; set; }

public string State { get; set; }

public string PostalCode { get; set; }

public string Country { get; set; }

}

public class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

public Address ShippingAddress { get; set; }

}

public class Company

{

public string Name { get; set; }

public Address ShippingAddress { get; set; }

}

Operators

Assignment Operators

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator, Assigns values from right side operands to left side operand | C = A + B assigns value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | C += A is equivalent to C = C + A |

More assignment operator can be applied similar to +=

Arithmetic Operators

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

Logical Operators

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

Relational Operators

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

Logical and relational operators are used to compare values and return true or false (Boolean) value, example:

if (A >= B && B <10){

//Do something

}

This introduces the if else statement which is a conditional operator involved in decision making process

Others:

[**if...else statement**](https://www.tutorialspoint.com/csharp/if_else_statement_in_csharp.htm)

[**switch statement**](https://www.tutorialspoint.com/csharp/switch_statement_in_csharp.htm)

**conditional operator ? :** in previous chapter which can be used to replace **if...else** statements

Exp1 ? Exp2 : Exp3;

Loops

When you want to repeat an action for many times, this is where loops comes in.

For example, we want to calculate the total score of all students by adding up their score in math, science and English.

For loop and foreach loop

Other loops

|  |  |
| --- | --- |
| **Loop Type** | **Description** |
| [**while loop**](https://www.tutorialspoint.com/csharp/csharp_while_loop.htm) | It repeats a statement or a group of statements while a given condition is true. It tests the condition before executing the loop body. |
| [**for loop**](https://www.tutorialspoint.com/csharp/csharp_for_loop.htm) | It executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| [**do...while loop**](https://www.tutorialspoint.com/csharp/csharp_do_while_loop.htm) | It is similar to a while statement, except that it tests the condition at the end of the loop body |
| [**nested loops**](https://www.tutorialspoint.com/csharp/csharp_nested_loops.htm) | You can use one or more loop inside any another while, for or do..while loop. |

Loop control statements change execution from its normal sequence.

|  |  |
| --- | --- |
| **Control Statement** | **Description** |
| [**break statement**](https://www.tutorialspoint.com/csharp/csharp_break_statement.htm) | Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch. |
| [**continue statement**](https://www.tutorialspoint.com/csharp/csharp_continue_statement.htm) | Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |

A loop becomes infinite loop if a condition never becomes false.

Data conversion

Strings

String Operations

Formatting Dates and Times

Intermediate Section

Static method

Examine Console.WriteLine() Method

Object-oriented programming

**Inheritance**

Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.

## Base and Derived Classes

<acess-specifier> class <base\_class>

{

...

}

class <derived\_class> : <base\_class>

{

...

}

In some languages, classes can inherit from multiple base classes, pulling in behavior from multiple parents. However, C# supports single inheritance, meaning that a class can only inherit from one base class.

A simple example to demonstrate inheritance is one that uses geometric shapes. You can define a class, Shape, that includes methods for calculating values like Perimeter. Then, you can inherit from Shape with various specific kinds of shapes, implementing the methods as you do so.

public class Shape

{

public virtual int Perimeter()

{

return 0;

}

}

public class Rectangle : Shape

{

public int Height { get; set; }

public int Width { get; set; }

public override int Perimeter()

{

return (Height + Width) \* 2;

}

}

public class Triangle : Shape

{

public int Side1 { get; set; }

public int Side2 { get; set; }

public int Side3 { get; set; }

public override int Perimeter()

{

return Side1 + Side2 + Side3;

}

}

You'll notice two new keywords in the example above: virtual and override, as well as some new syntax in the class definition lines. You specify the class your class inherits from by specifying it after a colon (:) following the class's name.

By declaring that method as virtual, child classes can modify the behavior of the method; by default methods cannot be changed by child class implementations.

When changing base class behavior, the child class defines the method with the same return type, name, and signature, as well as the override keyword. Requiring the use of the override keyword ensures that developers do not accidentally override base class behavior.

### **Abstract class**

Sometimes, you may define a class that should never actually be instantiated as an object - it should only be used as a base class for others. These are referred to as *abstract* classes. In the example above, it doesn't really make a lot of sense to create an instance of Shape, which has no sides and which has a Perimetermethod that always returns 0. It would probably make more sense to define the Shape class as abstract. You can also define methods as abstract, so that their implementation is required in child classes, but unnecessary in the base class. An abstract version of the Shape class would be:

public abstract class Shape

{

public abstract int Perimeter();

}

Note that abstract methods do not require the virtual keyword, since they *must* be overridden in child class implementations (the code will not compile otherwise).

**Polymorphism**

Polymorphism means, "occurring in several different forms". In programming, polymorphism refers to the ability to have code that can work with objects of different forms as if they were the same.

**polymorphism** means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'.

**Static polymorphism**, the response to a function is determined at the compile time. In **dynamic polymorphism**, it is decided at run-time.

## Function Overloading

## In a scenario we want to output different data types or have different number of input for a function, this is called function overloading

## Example:

using System;

namespace PolymorphismApplication

{

class Printdata

{

void print(int i)

{

Console.WriteLine("Printing int: {0}", i );

}

void print(double f)

{

Console.WriteLine("Printing float: {0}" , f);

}

void print(string s)

{

Console.WriteLine("Printing string: {0}", s);

}

static void Main(string[] args)

{

Printdata p = new Printdata();

// Call print to print integer

p.print(5);

// Call print to print float

p.print(500.263);

// Call print to print string

p.print("Hello C++");

Console.ReadKey();

}

}

}

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

## Dynamic Polymorphism

C# allows you to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it. **Abstract** classes contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.

Here are the rules about abstract classes:

* You cannot create an instance of an abstract class
* You cannot declare an abstract method outside an abstract class
* When a class is declared **sealed**, it cannot be inherited, abstract classes cannot be declared sealed.

*Encapsulation* is a fundamental concept in computer science and programming. At its core, encapsulation is simply "information hiding", but that doesn't convey the reasoning behind the practice. By hiding information about the inner workings of a software construct, you force collaborators to work only with the construct's exposed interface. How work is done within the construct is a "black box", and as a result, the inner workings are free to change without disrupting collaborators, provided the external interface (and associated behavior) is not changed.