**C# Tutorials**

Strong Programming Features of C#

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

**C# is part of .Net framework**

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)
* Class Library
* Common Language Specification(CLS)
* Common Type System(CTS)
* 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

Integrated Development Environment (IDE) for C#

Microsoft provides the following development tools for C# programming −

* Visual Studio 2010 (VS)
* Visual C# 2010 Express (VCE)
* Visual Web Developer

**Note: Mono** is an open-source version of the .NET Framework which includes a C# compiler and runs on several operating systems, including various flavors of Linux and Mac OS.

# **C# - Program Structure**

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 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.

# **C# - Basic Syntax**



Comments in C#

/\* Multiline comments - This program demonstrates

The basic syntax of C# programming Language \*/

Single-line comments are indicated by the '//' symbol. For example,

}//end class Rectangle

## C# Keywords

Keywords are reserved words predefined to the C# compiler.

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

# **C# - Data Types**

The variables in C#, are categorized into the following types −

* Value types
* Reference types
* Pointer types

## Value Type

Value type variables can be assigned a value directly. They are derived from the class **System.ValueType**.

The value types directly contain data. Some examples are **int, char, and float**,

## Reference Type

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

In other words, they refer to a memory location. Using multiple variables, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value.

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

### **Object Type**

The **Object Type** is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for System.Object class. The object types can be assigned values of any other types, value types, reference types, predefined or user-defined types.

**Note: 1.** When a value type is converted to object type, it is called **boxing**

int i = 10;

object o = i; //performs boxing

2. when an object type is converted to a value type, it is called **unboxing**.

object o = 10;

int i = (int)o; //performs unboxing

### **Dynamic Type**

You can store any type of value in the dynamic data type variable.

For example,

dynamic d = 20;

**Note:** Dynamic types are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run time.

### **String Type**

The **String Type** allows you to assign any string values to a variable. The string type is an alias for the System.String class. It is derived from object type.

For example,

String str = "Tutorials Point";

Note: user-defined reference types are: class, interface, or delegate

## Pointer Type

Pointer type variables store the memory address of another type.

For example,

char\* cptr;

int\* iptr;

# **C# - Type Conversion**

Type conversion is converting one type of data to another type.

* **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.

# **C# - Variables**

**Type**

1. Integral types 2. Floating point types 3. Decimal types 4. Boolean types
   1. **Nullable types**

## Accepting Values from User

The **Console** class in the **System** namespace provides a function **ReadLine()** for accepting input from the user and store it into a variable.

For example,

int num;

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

The function **Convert.ToInt32()** converts the data entered by the user to int data type, because **Console. ReadLine()** accepts the data in string format.

**Note: Constants** are treated just like regular variables except that their values cannot be modified after their definition.

const double pi = 3.14159;

**OOPS features**

**1. Encapsulation**

**2. Abstraction**

**3. Inheritance**

**4. Polymorphism**

# **C# - Encapsulation**

**Encapsulation** is defined 'as the process of enclosing one or more items within a physical or logical package'.

Abstraction and encapsulation are related features in object oriented programming.

**Note:** **Abstraction** allows making relevant information visible and **encapsulation** enables a programmer to *implement the desired level of abstraction*.

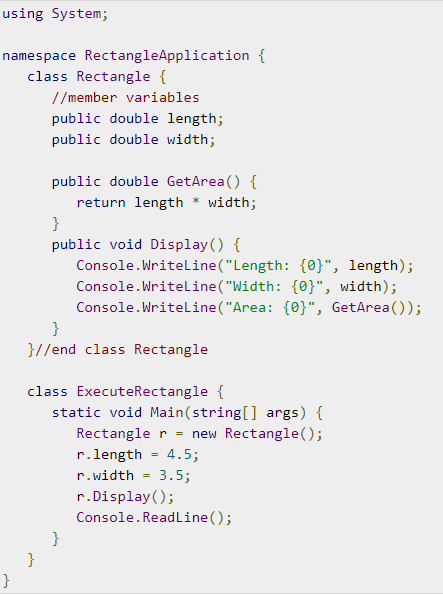
Encapsulation is implemented by using **access specifiers**.

An **access specifier** defines the scope and visibility of a class member. C# supports the following access specifiers −

* Public
* Private
* Protected
* Internal
* Protected internal

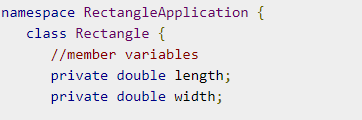
## Public Access Specifier

Public access specifier allows a class to expose its member variables and member functions to other functions and objects. Any public member can be accessed from outside the class. By creating instance of this class we can access its member.



## Private Access Specifier

Private access specifier allows a class to hide its member variables and member functions from other functions and objects. Only functions of the same class can access its private members. Even an instance of a class cannot access its private members.



## Protected Access Specifier

Protected access specifier allows a child class to access the member variables and member functions of its base class. This way it helps in implementing inheritance. We will discuss this in more details in the inheritance chapter.

## Internal Access Specifier

Internal access specifier allows a class to expose its member variables and member functions to other functions and objects in the current assembly. In other words, any member with internal access specifier can be accessed from any class or method defined within the application in which the member is defined.

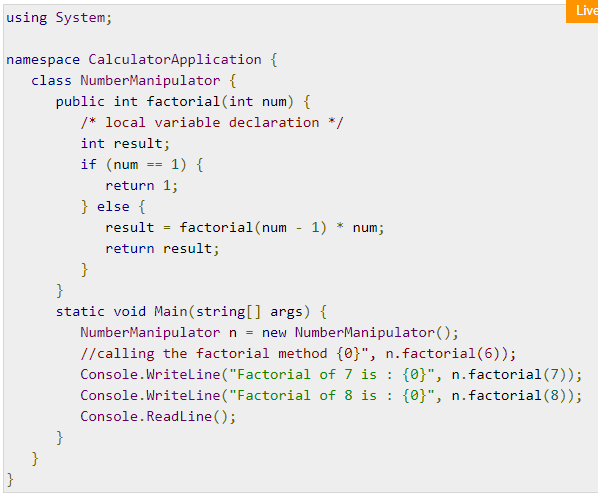
**Note:** Default access specifier of a class member is **private**.

## Protected Internal Access Specifier

The protected internal access specifier allows a class to hide its member variables and member functions from other class objects and functions, except a child class within the same application. This is also used while implementing inheritance.

## Recursive Method Call

A method can call itself. This is known as **recursion**



# **C# - Inheritance (:)**

**Note**: Provides an opportunity to reuse the code functionality and speeds up implementation time.

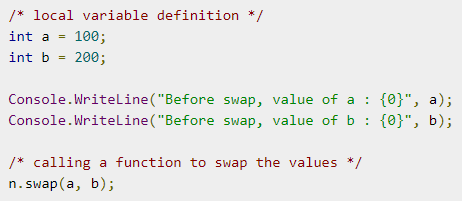
When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

Note: The idea of inheritance implements the **IS-A** relationship. For example, mammal

**IS-A** animal, dog **IS-A** mammal hence dog **IS-A** animal as well, and so on.

# **C# - Passing Parameters by Value**

This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

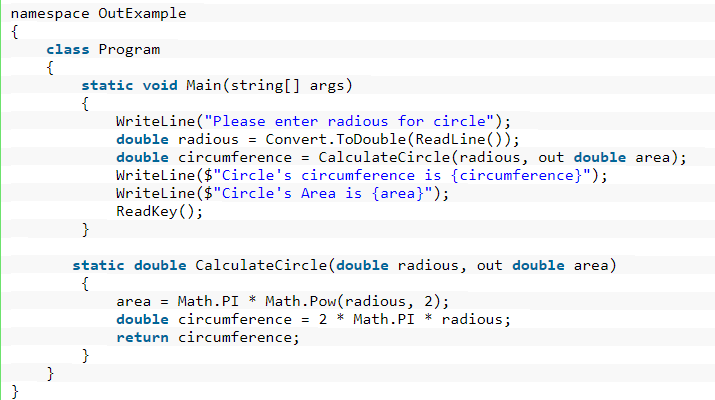


# **C# - Passing Parameters by Reference**

# This method copies the reference to the memory location of an argument into the formal parameter. This means that changes made to the parameter affect the argument.

# **C# - Passing Parameters by Output**

A return statement can be used for returning only one value from a function. However, using **output parameters**, you can return two values from a function.



# **C# - Nullables**

C# provides a special data types, the **nullable** types, to which you can assign normal range of values as well as null values.

Syntax for declaring a **nullable** type is as follows −

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

**How to access the value of Nullable type variables?**

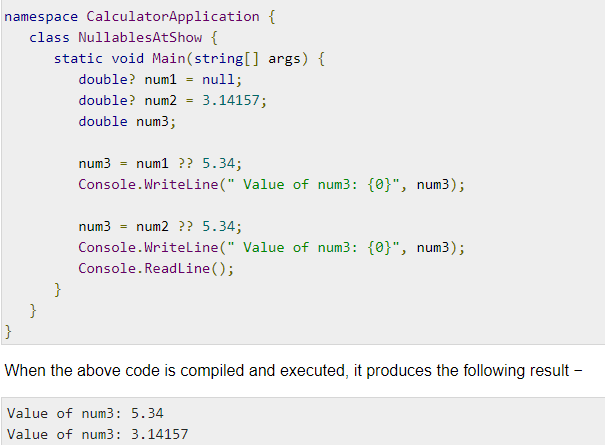
You cannot directly access the value of the Nullable type. You have to use **GetValueOrDefault() method** to get an original assigned value if it is not null. You will get the default value if it is null. The default value for null will be zero. //Console.WriteLine(n.GetValueOrDefault());

#### Characteristics of Nullable types

* With the help of nullable type you can assign a null value to a variable without creating nullable type based on the reference type.
* In Nullable types, you can also assign values to nullable type. As shown in the below example.

## The Null Coalescing Operator (??)

If the value of the first operand is null, then the operator returns the value of the second operand, otherwise it returns the value of the first operand.



# **C# - Arrays**

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data.

## Declaring Arrays

## datatype[] arrayName;

example: double[] balance;

## Initializing an Array

Array is a reference type, so you need to use the **new** keyword to create an instance of the array. For example,

double[] balance = new double[10];

## Assigning Values to an Array

double[] balance = new double[10];

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

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};

# **C# - Multidimensional Arrays**

3-dimensional array of int variables as −

int [ , , ] m;

## Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. The Following array is with 3 rows and each row has 4 columns.

int [,] a = new int [3,4] {

{0, 1, 2, 3}, /\* initializers for row indexed by 0 \*/

{4, 5, 6, 7}, /\* initializers for row indexed by 1 \*/

{8, 9, 10, 11} /\* initializers for row indexed by 2 \*/

};

## Note: A Jagged array is an array of arrays.

You can initialize a jagged array as −

int[][] scores = new int[2][]{new int[]{92,93,94},new int[]{85,66,87,88}};

Where, scores is an array of two arrays of integers - scores[0] is an array of 3 integers and scores[1] is an array of 4 integers.

# **C# - Array Class**

## The Array class is the base class for all the arrays in C#. It is defined in the System namespace.

# **C# - Strings**

## Strings as array of characters, However, more common practice is to use the string keyword to declare a string variable. The string keyword is an alias for the System.String class.

# **C# - Structures**

Structure is a value type data type.

**Note:** 1. The struct statement defines a new data type, with more than one member for your program.

1. It helps you to make a single variable hold related data of various data types. The **struct** keyword is used for creating a structure.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

* Title
* Author
* Subject
* Book ID

## 

## Class versus Structure

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

# **C# - Enums**

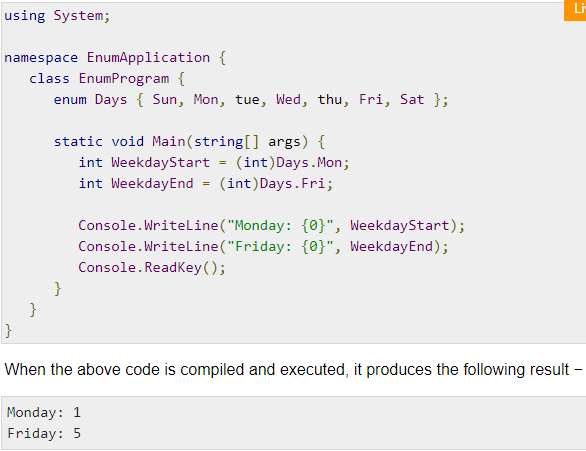
An enumeration is a set of named integer constants. An enumerated type is declared using the **enum** keyword.

Syntax:

enum <enum\_name> {

enumeration list

};

example − enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat }; 

# **C# - Classes**

Class is blueprint for a data type.

Objects are instances of a class

## C# Constructors

A class **constructor** is a special member function of a class that is executed whenever we **create new objects** of that class.

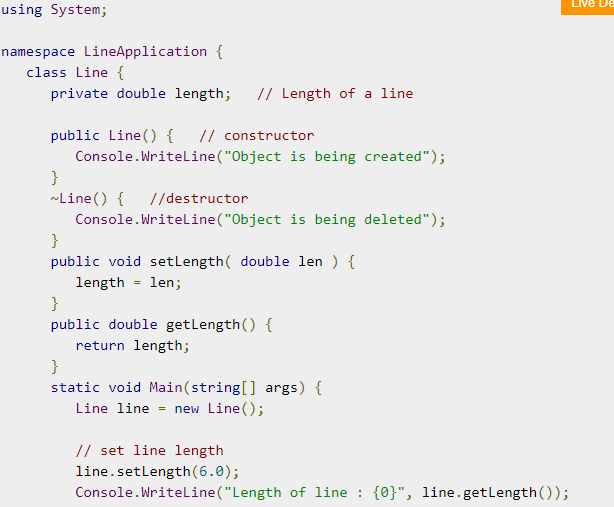
**Note**: A constructor has exactly the same name as that of class and it does not have any return type.

A **default constructor** does not have any parameter but if you need, a constructor can have parameters. Such constructors are called **parameterized constructors**. This technique helps you to assign initial value to an object at the time of its creation.

## C# Destructors

A **destructor** is a special member function of a class that is executed whenever an object of its class goes out of scope. A **destructor** has exactly the same name as that of the class with a prefixed tilde (~) and it can neither return a value nor can it take any parameters.

Note: Destructor can be very useful for releasing memory resources before exiting the program. Destructors cannot be inherited or overloaded.



## Static Members of a C# Class

When we declare a member of a class as static, it means no matter how many objects of the class are created, there is only one copy of the static member.

The keyword **static** implies that only one instance of the member exists for a class. Static variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance of it. Static variables can be initialized outside the member function or class definition.

## Multiple Inheritance in C#

**C# does not support multiple inheritance**. However, you can use interfaces to implement multiple inheritance.

# **C# - Polymorphism**

The word **polymorphism** means having many forms.

Polymorphism can be static or dynamic. In **static polymorphism**, the response to a function is determined at the compile time. In **dynamic polymorphism**, it is decided at run-time.

Static Polymorphism

The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. They are −

* Function overloading
* Operator overloading

## Function Overloading

You can have multiple definitions for the same function name in the same scope.

## Dynamic Polymorphism

**Abstract** classes contain abstract methods, which are implemented by the derived class.

**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.
* When you have a function defined in a class that you want to be implemented in an inherited class(es), you use **virtual** functions. The virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.
* Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.