Contents

[Set-up Dev Environment 1](#_Toc509833801)

[First C# Program 1](#_Toc509833802)

[Key concepts about Class, Constructor and Methods 3](#_Toc509833803)

[C# Variables 7](#_Toc509833804)

[C# Data Types 8](#_Toc509833805)

[C# Value and Reference Types 9](#_Toc509833806)

[C# Keywords 14](#_Toc509833807)

[Interfaces 16](#_Toc509833808)

[Operators 19](#_Toc509833809)

[Encapsulation 20](#_Toc509833810)

[Type Conversion 24](#_Toc509833811)

[C# Constants 25](#_Toc509833812)

[Decision Making 28](#_Toc509833813)

[Loops 28](#_Toc509833814)

[Methods & Recursive Method Call 29](#_Toc509833815)

[Nullables 34](#_Toc509833816)

https://www.tutorialspoint.com/csharp/

http://www.tutorialsteacher.com/csharp/csharp-tutorials

# Set-up Dev Environment

C# is used for server side execution for different kind of application like web, window forms or console etc. In order to use C# with your .Net application, you need two things, .NET Framework and IDE (Integrated Development Environment).

Visual Studion IDE has Nuget support for installing thrid party API/controls in the application

With Windows OS, .NET Framework is already installed in the PC.

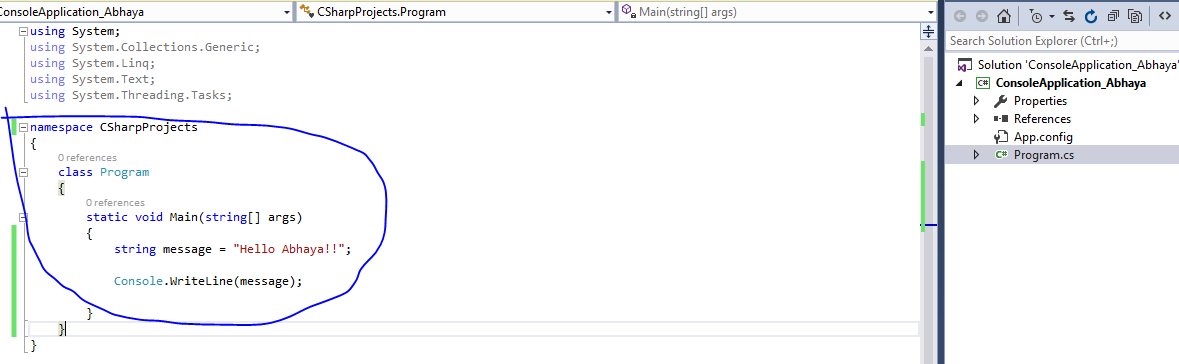
C# can be used in a window-based, web-based, or console application.

In Visual Studio File > New Project > Console Application and name the project and folder where it is to be stored. Program.cs will be created as default .cs file in Visual Studio where you can write your C# code in Program class.

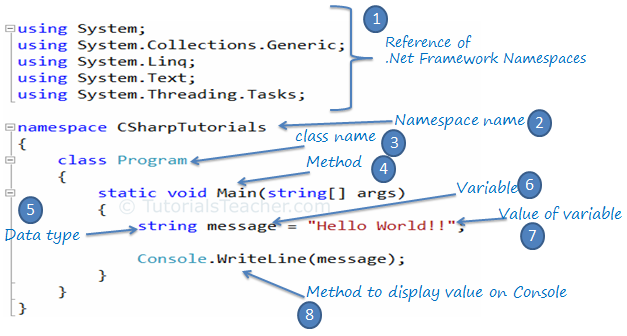
# First C# Program

Every console application starts from the Main() method of Program class. The following example code displays "Hello Abhaya!" on the console.

Edit the Program.cs file to add the code.



The following image illustrates the important parts of the above code:.

[](http://www.tutorialsteacher.com/Content/images/csharp/csharp-code-structure.png)C# Program

Explanation of above points:

1. Every .NET application takes the reference of the necessary .NET framework namespaces that it is planning to use with the "using" keyword e.g. *using System.Text*
2. Declare the namespace for the current class using the "namespace" keyword e.g. *namespace CSharpProjects*
3. We then declared a class using the "class" keyword: *class Program*
4. The Main() is a method of Program class which is the entry point of the console application.
5. String is a data type.
6. 'message' is a variable, that holds a value of a specified data type.
7. "Hello Abhaya!!" is the value of the message variable.
8. Console is a .NET framework class. WriteLine() is a method which you can use to display messages to the console.

Ctrl + F5 to build and run it.

# Key concepts about Class, Constructor and Methods

A class is like a blueprint of specific object. In the real world, every object has some color, shape and functionalities. For example, the luxury car Ferrari. Ferrari is an object of the luxury car type. The luxury car is a class that specify certain characteristic like speed, color, shape, interior etc. So any company that makes a car that meet those requirements is an object of the luxury car type. For example, every single car of BMW, lamborghini, cadillac are an object of the class called 'Luxury Car'. Here, 'Luxury Car' is a class and every single physical car is an object of the luxury car class.

Likewise, in object oriented programming, a class defines certain properties, fields, events, method etc. A class defines the kinds of data and the functionality their objects will have.

A class enables you to create your own custom types by grouping together variables of other types, methods and events. In C#, a class can be defined by using the class keyword.

Example: C# Class

public class MyClass

{

public string myField = string.Empty;

public MyClass()

{

}

public void MyMethod(int parameter1, string parameter2)

{

Console.WriteLine("First Parameter {0}, second parameter {1}",

parameter1, parameter2);

}

public int MyAutoImplementedProperty { get; set; }

private int myPropertyVar;

public int MyProperty

{

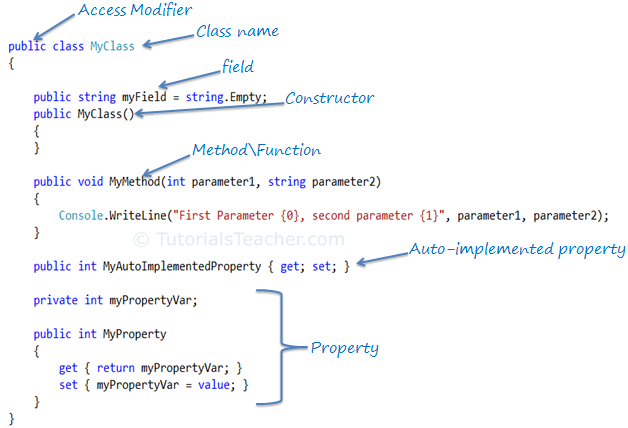
get { return myPropertyVar; }

set { myPropertyVar = value; }

}

}

The following image shows the important building blocks of C# class.

[](http://www.tutorialsteacher.com/Content/images/csharp/csharp-class.png)C# Class

Access Modifiers:

Access modifiers are applied on the declaration of the class, method, properties, fields and other members. They define the accessibility of the class and its members. **Public, private, protected and internal are access modifiers in C#.** We will learn about it in the [keyword](http://www.tutorialsteacher.com/csharp/csharp-keywords) section.

Field:

Field is a class level variable that can holds a value. Generally field members should have a private access modifier and used with a property.

Constructor:

A class can have parameterized or parameter less constructors. **The constructor will be called when you create an instance of a class.** Constructors can be defined by using an access modifier and class name: <access modifiers> <class name>(){ }

Example: Constructor in C#

class MyClass

{

public MyClass()

{

}

}

Method:

A method can be defined using the following template:

{access modifier} {return type} MethodName({parameterType parameterName})

Example: Method in C#

public void MyMethod(int parameter1, string parameter2)

{

// write your method code here..

}

Property:

**A property can be defined using getters and setters, as below**:

Example: Property in C#

private int \_myPropertyVar;

public int MyProperty

{

get { return \_myPropertyVar; }

set { \_myPropertyVar = value; }

}

**Property encapsulates a private field.** **It provides getters (get{}) to retrieve the value of the underlying field and setters (set{}) to set the value of the underlying field. In the above example, \_myPropertyVar is a private field which cannot be accessed directly. It will only be accessed via MyProperty. Thus, MyProperty encapsulates \_myPropertyVar.**

You can also apply some addition logic in get and set, as in the below example.

Example: Property in C#

private int \_myPropertyVar;

public int MyProperty

{

get {

return \_myPropertyVar / 2;

}

set {

if (value > 100)

\_myPropertyVar = 100;

else

\_myPropertyVar = value; ;

}

}

Auto-implemented Property:

From C# 3.0 onwards, property declaration has been made easy if you don't want to apply some logic in get or set.

The following is an example of an auto-implemented property:

Example: Auto implemented property in C#

public int MyAutoImplementedProperty { get; set; }

Notice that there is no private backing field in the above property example. The backing field will be created automatically by the compiler. You can work with an automated property as you would with a normal property of the class. Automated-implemented property is just for easy declaration of the property when no additional logic is required in the property accessors.

Namespace:

Namespace is a container for a set of related classes and namespaces. Namespace is also used to give unique names to classes within the namespace name. Namespace and classes are represented using a dot (.).

In C#, namespace can be defined using the namespace keyword.

Example: Namespace

namespace CSharpTutorials

{

class MyClass

{

}

}

In the above example, the fully qualified class name of MyClass is CSharpTutorials.MyClass.

A namespace can contain other namespaces. Inner namespaces can be separated using (.).

Example: Namespace

namespace CSharpTutorials.Examples

{

class MyClassExample

{

}

}

In the above example, the fully qualified class name of MyClassExample is CSharpTutorials.Example.MyClassExample

Points to Remember :

1. **C# Class** defines properties, fields, events, methods etc. An object is a instance of the class.
2. Access modifiers defines the accessbility of a class e.g. public, private, protected or internal.
3. **Namespace** can include one or more classes.

# C# Variables

In C#, a variable is always defined with a [data type](http://www.tutorialsteacher.com/csharp/csharp-data-types). The following is the syntax variable declaration and initialization.

Syntax:

<data type> <variable name>;

<datatype> <variable name> = <value>;

A variable can be declared and initialized later or it can be declared and initialized at the same time. In the following example, the first statement declares a variable called "message" without assigning any value to it. In the second statement, a value is assigned to the "message" variable.

Example: Variable declaration

string message;

// value can be assigned after it declared

message = "Hello World!!";

In the following example, variable is declared and initialized (a value is assigned to it) at the same time.

Example: Variable declaration & initialization

string message = "Hello World!!";

Multiple variables of the same data type can be declared and initialized in a single line separated by commas.

Example: Multiple variable declaration

int i, j, k, l = 0;

int amount, num;

When declaring multiple variables of the same data type, you can put them in multiple lines for the sake of readability; even if split across multiple lines, the compiler will consider it to be one statement, until it encounters a semicolon (;).

Example: Multi line variable declarations

int i, j,

k,

l = 0;

The value of a variable can be assigned to another variable of the same data type. However, a value must be assigned to a variable before using it.

Example: Variable assignment

int i = 100;

int j = i; // value of j will be 100

The following example would give a compile time error because string value cannot be assinged to a int type variable.

Example: Invalid Variable Assignment

string message = "Hello World!!";

int i = message; // compile time error

You must assign a value to a variable before using it otherwise the compiler will give an error. For example, in the following code, we have declared a variable called i without assigning any value to it. If we then try to display the value of the variable on the console, we will get a compile time error.

Example: Invalid Variable Assignment

int i;

//Following will give compile time error: "Use of unassigned local variable 'i'"

int j = i;

Console.WriteLine(j);

Points to Remember :

1. The variable is a name given to a data value.
2. A variable holds the value of specific data type e.g string, int, float etc.
3. A variable can be declared and initialized later or declared & initialized at the same time.
4. The value of a variable can be changed at any time throughout the program as long as it is accessible.
5. Multiple variables can be defined seperated by comma (,) in a single or multiple line till semicolon(;).
6. A value must be assigned to a variable before using it otherwise it will give compile time error.

# C# Data Types

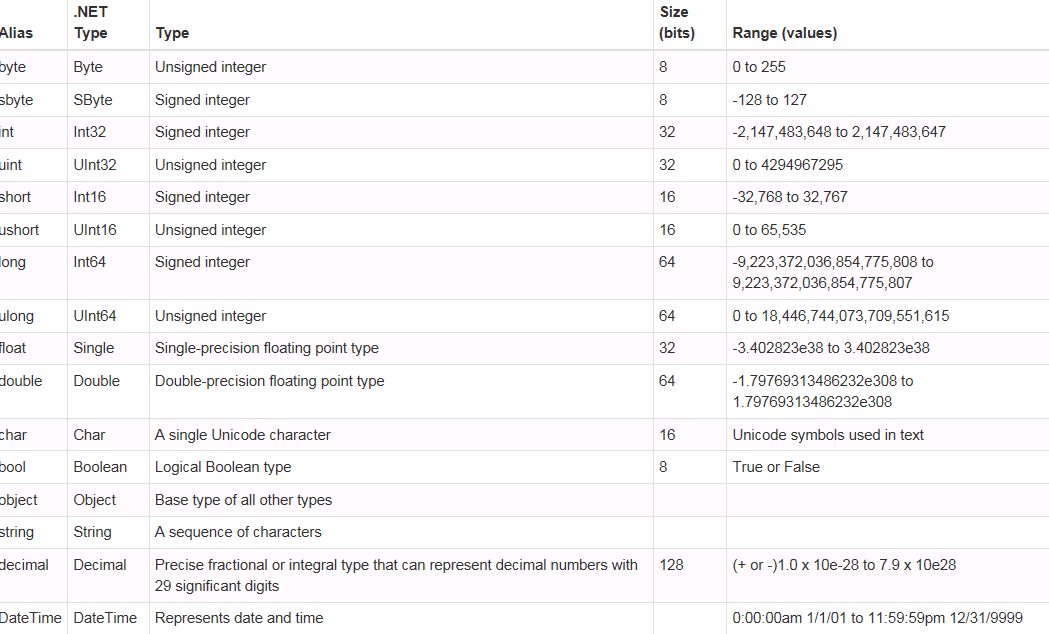
Example:

int intVar = 100;

float floatVar = 10.2f;

char charVar = 'A';

bool boolVar = true;



All data types are actually aliases referring to actual .NET types. For example, int is an alias for System.int32 type.

Data types are further classified as value type or reference type, depending on whether a variable of a particular type stores its own data or a pointer to the data in the memory.

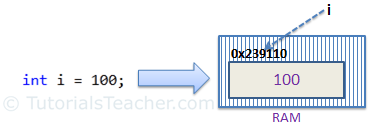
# C# Value and Reference Types

Value Type:

A data type is a value type if it holds a data value within its own memory space. It means variables of these data types directly contain their values. All the value types derive from System.ValueType, which in-turn, derives from System.Object.

For example, consider integer variable int i = 100;

The system stores 100 in the memory space allocated for the variable 'i'. The following image illustrates how 100 is stored at some hypothetical location in the memory (0x239110) for 'i':

[](http://www.tutorialsteacher.com/Content/images/csharp/value-type-memory-allocation.png)Memory allocation for Value Type

The following data types are all of value type:

* bool
* byte
* char
* decimal
* double
* enum
* float
* int
* long
* sbyte
* short
* struct
* uint
* ulong
* ushort

Passing by Value:

**When you pass a value type variable from one method to another method, the system creates a separate copy of a variable (in memory) in another method, so that if value got changed in the second method, it won't affect the variable in the first method.**

Example: Value type passes by value

static void ChangeValue(int x)

{

x = 200;

Console.WriteLine(x);

}

static void Main(string[] args)

{

int i = 100;

Console.WriteLine(i);

ChangeValue(i);

Console.WriteLine(i);

}

Output:

100   
200   
100

In the above example, variable i in Main() method remains unchanged even after we pass it to the ChangeValue() method and change its value there.

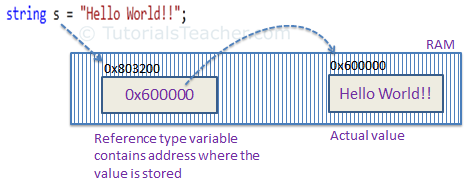
Reference type:

Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the data.

For example, consider following string variable:

string s = "Hello World!!";

The following image shows how the system allocates the memory for the above string variable.

[](http://www.tutorialsteacher.com/Content/images/csharp/raference-type-memory-allocation.png)

As you can see in the above image, the system selects a random location in memory (0x803200) for the variable 's'. The value of a variable s is 0x600000 which is the memory address of the actual data value. Thus, reference type stores the address of the location where the actual value is stored instead of value itself.

The following data types are of reference type:

* String
* All arrays, even if their elements are value types
* Class
* Delegates

Passing by Reference:

When you pass a reference type variable from one method to another, it doesn't create a new copy; instead, it passes the address of the variable. If we now change the value of the variable in a method, it will also be reflected in the calling method.

Example: Reference type variable passes by reference

static void ChangeReferenceType(Student std2)

{

std2.StudentName = "Steve";

}

static void Main(string[] args)

{

Student std1 = new Student();

std1.StudentName = "Bill";

ChangeReferenceType(std1);

Console.WriteLine(std1.StudentName);

}

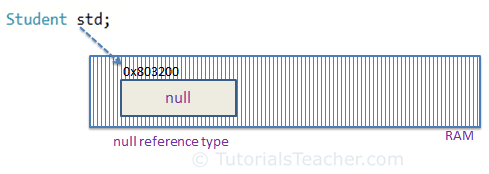
Output:

Steve

In the above example, since Student is an object, when we send the Student object std1 to the ChangeReferenceType() method, what is actually sent is the memory address of std1. Thus, when the ChangeReferenceType() method changes StudentName, it is actually changing StudentName of std1, because std1 and std2 are both pointing to the same address in memory. Therefore, the output is Steve.

null value:

Reference types have null value by default, when they are not initialized. For example, a string variable (or any other variable of reference type datatype) without a value assigned to it. In this case, it has a null value, meaning it doesn't point to any other memory location, because it has no value yet.

[](http://www.tutorialsteacher.com/Content/images/csharp/null.png)Null Reference type

A value type variable cannot be null because it holds a value not a memory address. However, value type variables must be assigned some value before use. The compiler will give an error if you try to use a local value type variable without assigning a value to it.

Example: Compile time error

void someFunction()

{

int i;

Console.WriteLine(i);

}

C# 2.0 introduced nullable types for value types so that you can assign null to a value type variable or declare a value type variable without assigning a value to it.

However, **value type field in a class can be declared without initialization** (field not a local variable in the function) . It will have a default value if not assigned any value, e.g., int will have 0, boolean will have false and so on.

Example: Value type field

class myClass

{

public int i;

}

myClass mcls = new myClass();

Console.WriteLine(mcls.i);

Output:

0

Points to Remember :

1. Value type stores the value in its memory space, whereas reference type stores the address of the value where it is stored.
2. Primitive data types and struct are of the 'Value' type. Class objects, string, array, delegates are reference types.
3. Value type passes byval by default. Reference type passes byref by default.
4. Value types and reference types stored in Stack and Heap in the memory depend on the scope of the variable.

# C# Keywords

C# contains reserved words that have special meaning for the compiler. These reserved words are called "keywords". Keywords cannot be used as a name (identifier) of a variable, class, interface, etc.

Keywords in C# are distributed under the following categories:

Modifier keywords:

Modifier keywords are certain keywords that indicate who can modify types and type members. Modifiers allow or prevent certain parts of programs from being modified by other parts.

| abstract,async,const,event,extern,new, override , partial , readonly ,sealed , static,unsafe ,virtual, |
| --- |

Access Modifier Keywords:

Access modifiers are applied on the declaration of the class, method, properties, fields and other members. They define the accessibility of the class and its members.

| **Access Modifiers** | **Usage** |
| --- | --- |
| public | The Public modifier allows any part of the program in the same assembly or another assembly to access the type and its members. |
| private | The Private modifier restricts other parts of the program from accessing the type and its members. Only code in the same class or struct can access it. |
| internal | The Internal modifier allows other program code in the same assembly to access the type or its members. This is default access modifiers if no modifier is specified. |
| protected | The Protected modifier allows codes in the same class or a class that derives from that class to access the type or its members. |

Statement Keywords:

Statement keywords are related to program flow.

| e.g. if ,else ,switch ,case ,do ,for ,foreach ,in, try,catch, finally ,checked ,unchecked ,fixed ,lock, while ,break ,continue ,default ,goto ,return ,yield ,throw. |
| --- |

Method parameter keywords:

These keywords are applied on the parameters of a method.

|  |
| --- |
| **params** |
| ref |
| out |

Namespace keywords:

These keywords are applied with namespace and related operators.

|  |
| --- |
| using |
| . operator |
| :: operator |
| extern alias |

Operator Keywords:

Operator keywords perform miscellaneous actions.

as ,await ,is ,new ,sizeof ,typeof ,stackalloc ,checked ,unchecked

Access keywords:

Access keywords are used to access the containing class or the base class of an object or class.

| **Access keywords** |
| --- |
| base |
| this |

Literal keywords:

Literal keywords apply to the current instance or value of an object e.g. null, false, true and void.

|  |
| --- |

Type keywords:

Type keywords are used for data types.

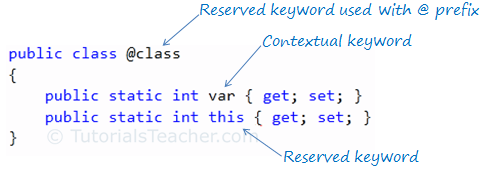
|  |
| --- |
| e.g. bool, int and datetime etc. |

Contextual Keywords:

Contextual keywords are considered as keywords, only if used in certain contexts. They are not reserved and so can be used as names or identifiers.

Examples: add, var, dynamic and global etc.

Contextual keywords are not converted into blue color (default color for keywords in visual studio) when used as an identifier in Visual Studio. For example, var in the below figure is not in blue color whereas color of this is blue color. So var is a contextual keyword.

[](http://www.tutorialsteacher.com/Content/images/csharp/keywords-in-vs.png)C# Keywords color in Visual Studio

Query keywords:

Query keywords are contextual keywords used in LINQ queries e.g. where. From, order by etc.

As mentioned above, keyword cannot be used as an identifier (name of variable, class, interface etc). However, they can be used with the prefix '@'. For example, class is a reserved keyword so it cannot be used as an identifier, but @class can be used as shown below.

Example: Keyword as identifier

public class @class

{

public static int MyProperty { get; set; }

}

public class Program

{

public static void Main()

{

@class.MyProperty = 100;

Console.WriteLine(@class.MyProperty);

}

}

Displays 100

Points to Remember :

1. Keywords are reserved words that cannot be used as name or identifier.
2. Prefix '@' with keywords if you want to use it as identifier.
3. C# includes various categories of keywords e.g. modifier keywords, access modifiers keywords, statement keywords, method param keywords etc.
4. Contextual keywords can be used as identifier.

# Interfaces

Interfaces in C# are provided as a replacement of multiple inheritance. Because C# does not support multiple inheritance, it was necessary to incorporate some other method so that the class can inherit the behavior of more than one class, avoiding the problem of name ambiguity that is found in C++. With name ambiguity, the object of a class does not know which method to call if the two base classes of that class object contain the same named method.

An interface in C# contains only the declaration of the methods, properties, and events, but not the implementation. It is left to the class that implements the interface by providing implementation for all the members of the interface. Interface makes it easy to maintain a program.

The most important thing to remember about interfaces is that the classes can only implement the methods defined in the interface because in C#, an interface is a built-in keyword that declares a reference type that includes method declarations. In addition to methods, interfaces can define properties, indexers, and events that will be discussed later in this article

In C#, an interface can be defined using the interface keyword. For example, the following is a simple interface for a logging string message:

Interface Declaration:

interface ILog

{

void Log(string msgToLog);

}

Now, different classes can implement ILog by providing an implementation of the Log() method, for example, the ConsoleLog class logs the string on the console whereas FileLog logs the string into a text file.

Implement interface using- : <interface name > syntax.

Interface implementation Example:

class ConsoleLog: ILog

{

public void Log(string msgToPrint)

{

Console.WriteLine(msgToPrint);

}

}

class FileLog :ILog

{

public void Log(string msgToPrint)

{

File.AppendText(@"C:\Log.txt").Write(msgToPrint);

}

}

Now, you can instantiate an object of either the ConsoleLog or FileLog class:

C#:

ILog log = new ConsoleLog();

//Or

ILog log = new FileLog();

Explicit Implementation:

You can implement interface explicitly by prefixing interface name with method name, as below:

C#:

class ConsoleLog: ILog

{

public void ILog.Log(string msgToPrint) // explicit implementation

{

Console.WriteLine(msgToPrint);

}

}

Explicit implementation is useful when class is implementing multiple interface thereby it is more readable and eliminates the confusion. It is also useful if interfaces have same method name coincidently.

Another Good Example with code snippets. In this example, we have taken two sub-classes of Mammal: Human and Whale. Because Human is the only subclass that has the characteristic of intelligence that distinguishes it from the other subclasses of Mammal, the Human class inherits both the class Mammal and an interface IIntelligent that selectively describes it as separated from the other classes of Mammal.



Points to Remember :

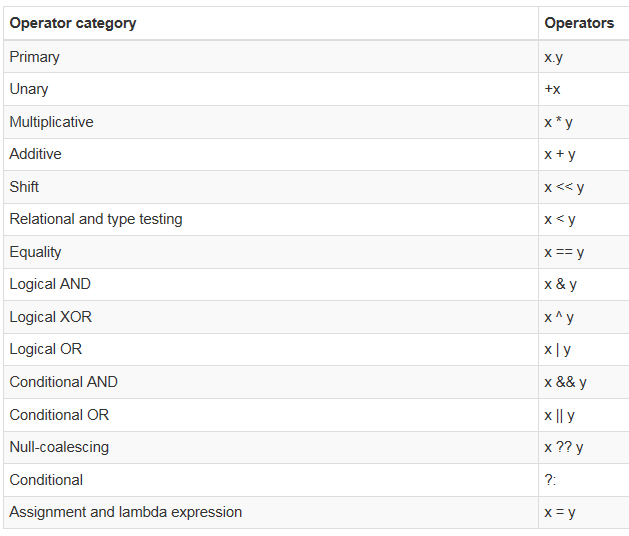
1. An **Interface** only contains declarations of method, events & properties.
2. An **Interface** can be implemented implicitly or explicitly.
3. An **Interface** cannot include private members. All the members are public by default.

Why is it used?

1. To allow a class to inherit multiple behaviors from multiple interfaces.
2. To avoid name ambiguity between the methods of the different classes as was in the use of multiple inheritance in C++.
3. To combine two or more interfaces such that a class needs to implement the combined result.
4. To allow Name hiding. Name hiding is the ability to hide an inherited member name from any code outside the derived class

<https://www.codeguru.com/csharp/csharp/cs_syntax/interfaces/article.php/c7563/Interfaces-in-C.htm>

# Operators



certain operators have different meanings based on the datatype of the operand. For example, if the + operator is used with numbers, it will add the numbers but if it is used with strings, it will concatenate the two strings.

When an operator does different things based on the datatype of the operands, it is called operator over loading.

static void Main(string[] args)

{

string message1 = "Hello";

string message2 = message1 + " World!!";

Console.WriteLine(message2);

int i = 10, j = 20;

int sum = i + j;

Console.WriteLine("{0} + {1} = {2}", i, j, sum);

}

# Encapsulation

**Encapsulation** is defined 'as the process of enclosing one or more items within a physical or logical package'. Encapsulation, in object oriented programming methodology, prevents access to implementation details.

Abstraction and encapsulation are related features in object oriented programming. 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.

class Rectangle

{

//member variables

public double length;

public double width;

public double GetArea()

{

return length \* width;

}

public void Display()

{

Console.WriteLine("Length: {0}", length);

Console.WriteLine("Width: {0}", width);

Console.WriteLine("Area: {0}", GetArea());

}

}//end class Rectangle

class ExecuteRectangle

{

static void Main(string[] args)

{

Rectangle r = new Rectangle();

//can access length, width and methods because all are declared public

r.length = 4.5;

r.width = 3.5;

r.Display();

Console.ReadLine();

}

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

Length: 4.5

Width: 3.5

Area: 15.75

In the preceding example, the member variables length and width are declared **public**, so they can be accessed from the function Main() using an instance of the Rectangle class, named **r**.

The member function *Display()* and *GetArea()* can also access these variables directly without using any instance of the class. The member functions *Display()* is also declared **public**, so it can also be accessed from *Main()* using an instance of the Rectangle class, named **r**.

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.

private double length;

private double width;

public double GetArea()

{

return length \* width;

}

public void Display()

{

Console.WriteLine("Length: {0}", length);

Console.WriteLine("Width: {0}", width);

Console.WriteLine("Area: {0}", GetArea());

}

// new method created and marked public so that main() can access it and retreive the length and width. Length and width are declared as private and can't be accessed from main() method.

public void Acceptdetails()

{

Console.WriteLine("Enter Length: ");

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

Console.WriteLine("Enter Width: ");

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

}

}//end class Rectangle

class ExecuteRectangle

{

static void Main(string[] args)

{

Rectangle r = new Rectangle();

**//can't access length and width because all are declared private**

r.Acceptdetails();

r.Display();

Console.ReadLine();

}

If you make Display() declared as private and try to use within main() method, you can’t build the solution. Syntax error.

The member variables length and width are declared **private**, so they cannot be accessed from the function Main(). The member functions *AcceptDetails()* and *Display()* can access these variables since they are part of the same class rectangle. Since the member functions *AcceptDetails()* and *Display()* are declared **public**, they can be accessed from *Main()* using an instance of the Rectangle class, named **r**.

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

Length: 4.5

Width: 3.5

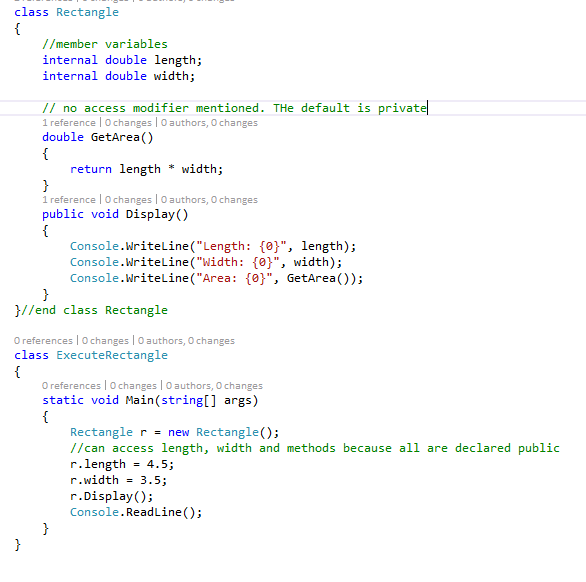
Area: 15.75

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. More on this in iNheritance.

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.



Length and width have been defined internal and thus can be accessed from within ExecuteRectangle (in the same applicn / assembly).

There is no access modifier in GetArea(); it takes the default private. Display method can still access GetArea() since it is part of same class Rectangle. Also, main() can access length and width properties since they are declared internal in Rectangle class. No separate AccessDetails() method is defined here as in private access modifier example where length and width were declared private in Rectangle class.

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

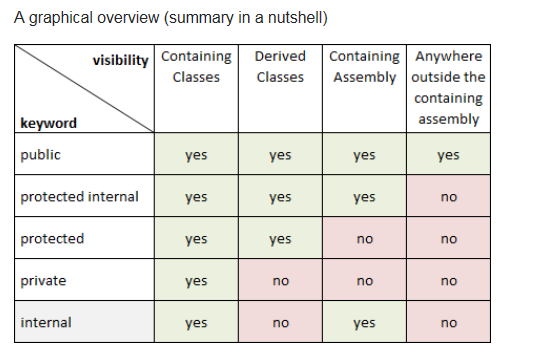
Length: 4.5

Width: 3.5

Area: 15.75

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.



# Type Conversion

Type conversion is converting one type of data to another type. It is also known as Type Casting. In C#, type casting has two forms −

* **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.
* double d = 5673.74;
* int i;
* // cast double to int.
* i = (int)d;
* Console.WriteLine(i);

The above converts double to int. When the aboce code runs, the result is 5673.

Examples of Type Conversion Methods: ToBoolean (converts a type to Boolean if possible)

ToString (Converts a type to String), ToDateTime etc.

int i = 75;

float f = 53.005f;

double d = 2345.7652;

bool b = true;

Console.WriteLine(i.ToString());

Console.WriteLine(f.ToString());

Console.WriteLine(d.ToString());

Console.WriteLine(b.ToString());

The above produces the result below:

75

53.005

2345.7652

True

# C# Constants

The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals. Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are also enumeration constants as well. The constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

An integer literal can be a decimal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, and there is no prefix id for decimal. An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals −

212 /\* Legal \*/

215u /\* Legal \*/

0xFeeL /\* Legal \*/

Following are other examples of various types of Integer literals −

85 /\* decimal \*/

0x4b /\* hexadecimal \*/

30 /\* int \*/

30u /\* unsigned int \*/

30l /\* long \*/

30ul /\* unsigned long \*/

Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

Here are some examples of floating-point literals −

3.14159 /\* Legal \*/

314159E-5F /\* Legal \*/

Charatcer Constants

Character literals are enclosed in single quotes. For example, 'x' and can be stored in a simple variable of char type. A character literal can be a plain character (such as 'x'), an escape sequence (such as '\t'), or a universal character (such as '\u02C0').

There are certain characters in C# when they are preceded by a backslash. They have special meaning and they are used to represent like newline (\n) or tab (\t). Here, is a list of some of such escape sequence codes −

|  |  |
| --- | --- |
| \n | Newline |
| \r | Carriage return |
| \t | Horizontal tab |
| \v | Vertical tab |
| \xhh . . . | Hexadecimal number of one or more digits |

e.g. Console.WriteLine("Hello\tWorld\n\n");

returns Hello World

String Literals

String literals or constants are enclosed in double quotes "" or with @"". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters. You can break a long line into multiple lines using string literals and separating the parts using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"

@"hello dear"

Defining Constants

Constants are defined using the const keyword. Syntax for defining a constant is −

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

Example:

const double pi = 3.14159;

// constant declaration

double r;

Console.WriteLine("Enter Radius: ");

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

double areaCircle = pi \* r \* r;

Console.WriteLine("Radius: {0}, Area: {1}", r, areaCircle);

Console.ReadLine();

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

Enter Radius:

3

Radius: 3, Area: 28.27431

Any attempt to change pi to some other value throws build errors.

# Decision Making

If

If-else

Nested if

Switch (case statements) and nested swith

?:

**conditional operator ? :** can be used to replace **if...else** statements. It has the following general form −

Exp1 ? Exp2 : Exp3;

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined as follows: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

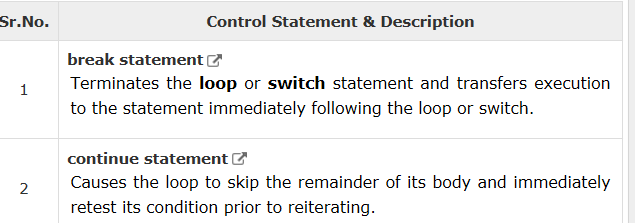
# Loops

While

Do .. while

For

Break and continue can be used to terminate a loop or go to next iteration of the loop



# Methods & Recursive Method Call

A method is a group of statements that together perform a task. Every C# program has at least one class with a method named Main.

To use a method, you need to −

* Define the method
* Call the method

When you define a method, you basically declare the elements of its structure. The syntax for defining a method in C# is as follows −

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

Method Body

}

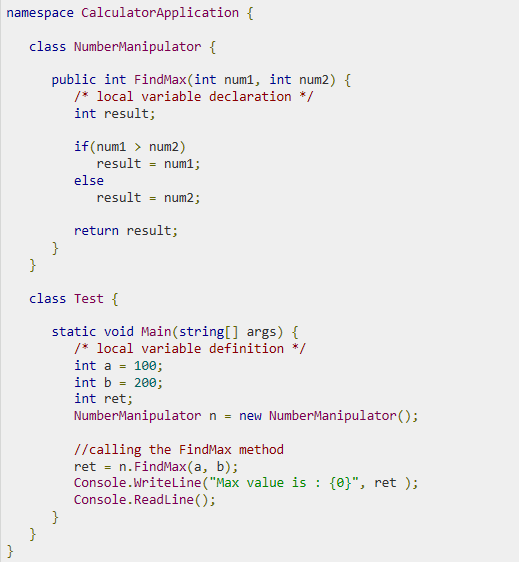
Following are the various elements of a method −

* 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 − It contains the set of instructions needed to complete the required activity.

Following code snippet shows a function FindMax that takes two integer values and returns the larger of the two. It has public access specifier, so it can be accessed from outside the class using an instance of the class.

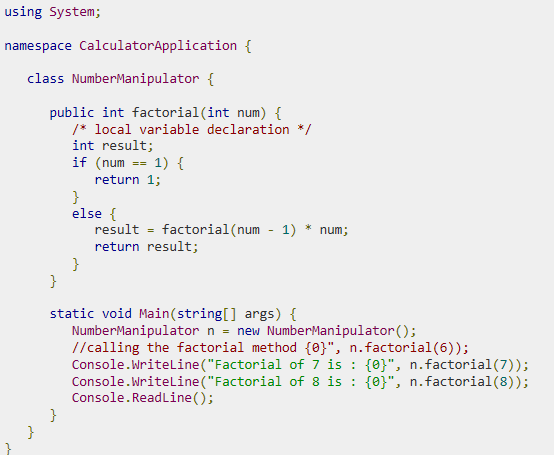
* class NumberManipulator {
* public int FindMax(int num1, int num2) {
* /\* local variable declaration \*/
* int result;
* if (num1 > num2)
* result = num1;
* else
* result = num2;
* return result;
* }
* ...
* }

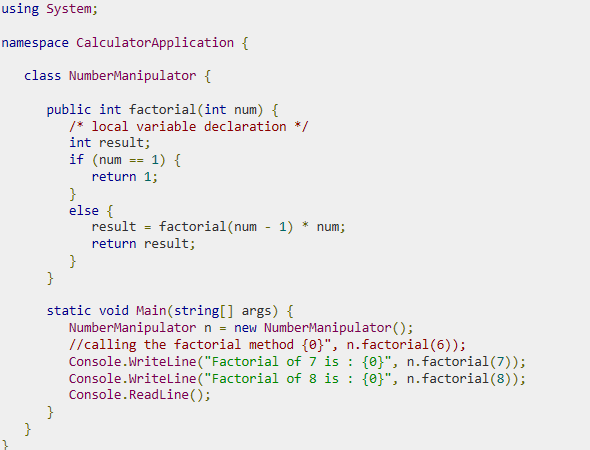
You can also call public method from other classes by using the instance of the class. For example, the method FindMax belongs to the NumberManipulator class, you can call it from another class Test.



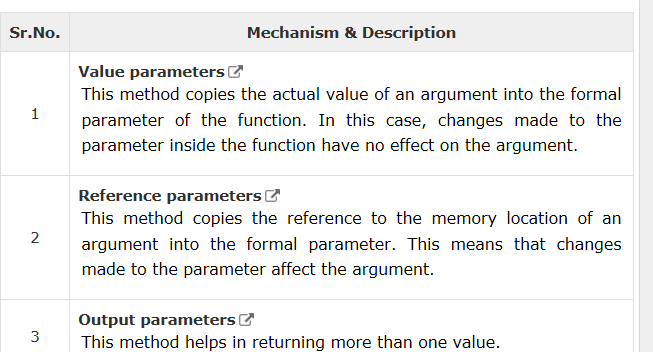
Recursive Method Calls

A method can call itself. This is known as recursion. Following is an example that calculates factorial for a given number using a recursive function –





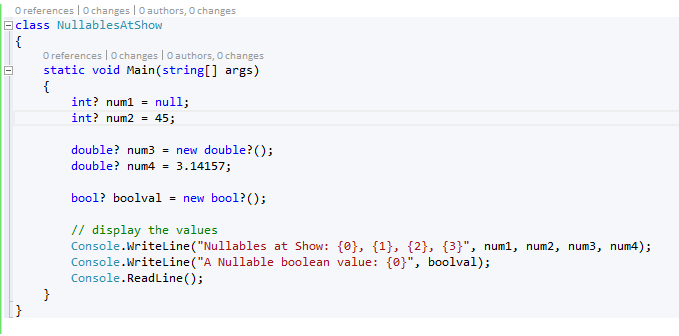
When method with parameters is called, you need to pass the parameters to the method. There are three ways that parameters can be passed to a method



# Nullables

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

For example, you can store any value from -2,147,483,648 to 2,147,483,647 or null in a Nullable<Int32> variable. Similarly, you can assign true, false, or null in a Nullable<bool> variable. Syntax for declaring a nullable type is as follows –



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

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

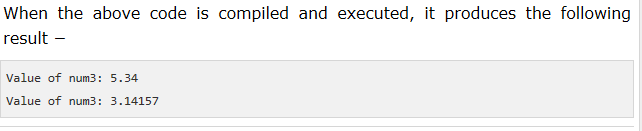
Nullables at Show: , 45, , 3.14157

A Nullable boolean value:

Null Coalescing Operator (??)

The null coalescing operator is used with the nullable value types and reference types. It is used for converting an operand to the type of another nullable (or not) value type operand, where an implicit conversion is possible.

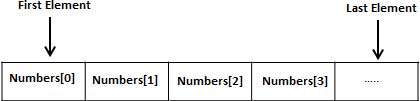
If the value of the first operand is null, then the operator returns the value of the second operand, otherwise it returns the value of the first operand. The following example explains this –



# 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, 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.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in C#, you can use the following syntax −

datatype[] arrayName;

where,

* datatype is used to specify the type of elements in the array.
* [ ] specifies the rank of the array. The rank specifies the size of the array.
* arrayName specifies the name of the array.

For example,

int[] SSN;

Initializing Arrays

Declaring an array does not initialize the array in the memory. When the array variable is initialized, you can assign values to the 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

You can assign values to individual array elements, by using the index number, like −

double[] balance = new double[10];

balance[0] = 4500.0;

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};

You can copy an array variable into another target array variable. In such case, both the target and source point to the same memory location −

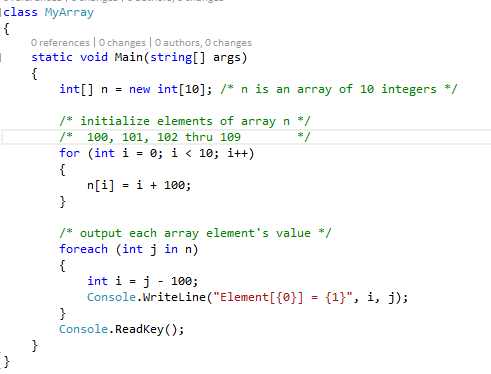
int [] marks = new int[] { 99, 98, 92, 97, 95};

int[] score = marks;

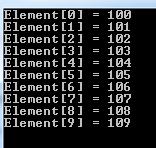
When you create an array, C# compiler implicitly initializes each array element to a default value depending on the array type. For example, for an int array all elements are initialized to 0.

Accessing array elements

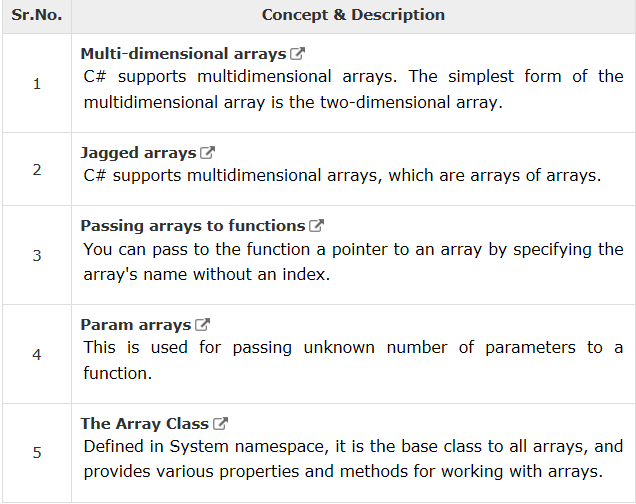
You can also use a foreach statement to iterate through an array.



Output



Important concepts about arrays:



# C# Strings

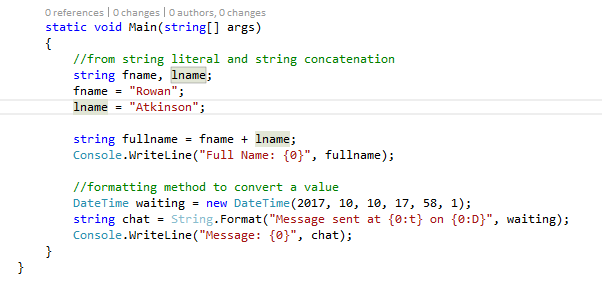
In C#, you can use strings as array of characters, However, more common practice is to use the string keyword to declare a string variable. The string keyword is an alias for the System.String class.

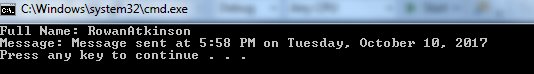
Creating a String Object

You can create string object using one of the following methods −

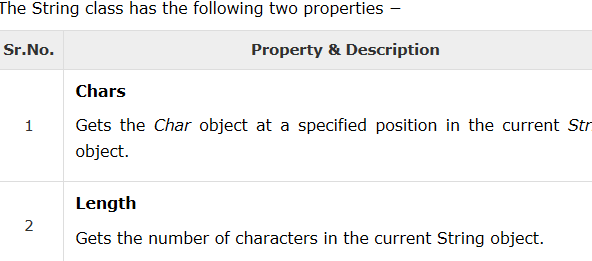
* By assigning a string literal to a String variable
* By using a String class constructor
* By using the string concatenation operator (+)
* By retrieving a property or calling a method that returns a string
* By calling a formatting method to convert a value or an object to its string representation

Example:





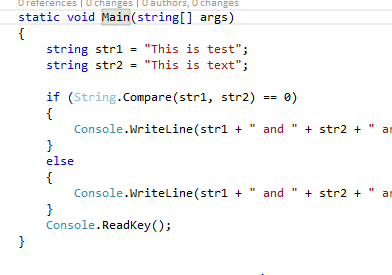
Properties of String Class



Methods of the String Class

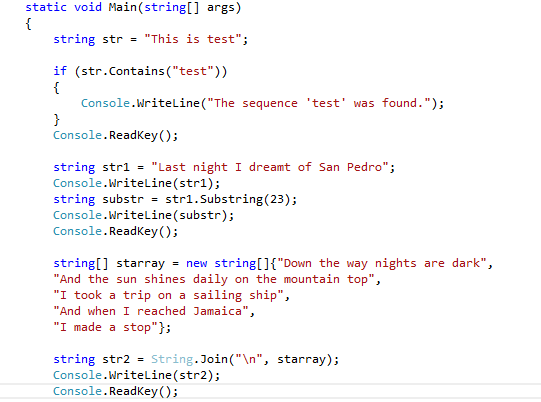
The String class has numerous methods that help you in working with the string objects. The following table provides some of the most commonly used methods –

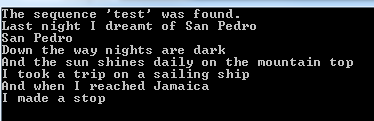
Compare method example:



Output: 

Substring, Contains and Join functions:



Output: 

# C# Structures

Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member for your program.

For example, here is the way you can declare the Book structure −

The following program shows the use of the structure −

struct Books {

public string title;

public string author;

public string subject;

public int book\_id;

};

public class testStructure {

public static void Main(string[] args) {

Books Book1; /\* Declare Book1 of type Book \*/

Books Book2; /\* Declare Book2 of type Book \*/

/\* book 1 specification \*/

Book1.title = "C Programming";

Book1.author = "Nuha Ali";

Book1.subject = "C Programming Tutorial";

Book1.book\_id = 6495407;

/\* book 2 specification \*/

Book2.title = "Telecom Billing";

Book2.author = "Zara Ali";

Book2.subject = "Telecom Billing Tutorial";

Book2.book\_id = 6495700;

/\* print Book1 info \*/

Console.WriteLine( "Book 1 title : {0}", Book1.title);

Console.WriteLine("Book 1 author : {0}", Book1.author);

Console.WriteLine("Book 1 subject : {0}", Book1.subject);

Console.WriteLine("Book 1 book\_id :{0}", Book1.book\_id);

/\* print Book2 info \*/

Console.WriteLine("Book 2 title : {0}", Book2.title);

Console.WriteLine("Book 2 author : {0}", Book2.author);

Console.WriteLine("Book 2 subject : {0}", Book2.subject);

Console.WriteLine("Book 2 book\_id : {0}", Book2.book\_id);

Console.ReadKey();

}

}

Output: 

Features of C# Structures

Structures in C# are quite different from that in traditional C or C++. The C# structures have the following features −

* Structures can have methods, fields, indexers, properties, operator methods, and events.
* Structures can have defined constructors, but not destructors. However, you cannot define a default constructor for a structure. The default constructor is automatically defined and cannot be changed.
* Unlike classes, structures cannot inherit other structures or classes.
* Structures cannot be used as a base for other structures or classes.
* A structure can implement one or more interfaces.
* Structure members cannot be specified as abstract, virtual, or protected.
* When you create a struct object using the **New** operator, it gets created and the appropriate constructor is called. Unlike classes, structs can be instantiated without using the New operator.
* If the New operator is not used, the fields remain unassigned and the object cannot be used until all the fields are initialized.

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

Now, the example can be rewritten as :

struct Books {

private string title;

private string author;

private string subject;

private int book\_id;

public void getValues(string t, string a, string s, int id) {

title = t;

author = a;

subject = s;

book\_id = id;

}

public void display() {

Console.WriteLine("Title : {0}", title);

Console.WriteLine("Author : {0}", author);

Console.WriteLine("Subject : {0}", subject);

Console.WriteLine("Book\_id :{0}", book\_id);

}

};

public class testStructure {

public static void Main(string[] args) {

Books Book1 = new Books(); /\* Declare Book1 of type Book \*/

Books Book2 = new Books(); /\* Declare Book2 of type Book \*/

/\* book 1 specification \*/

Book1.getValues("C Programming",

"Nuha Ali", "C Programming Tutorial",6495407);

/\* book 2 specification \*/

Book2.getValues("Telecom Billing",

"Zara Ali", "Telecom Billing Tutorial", 6495700);

/\* print Book1 info \*/

Book1.display();

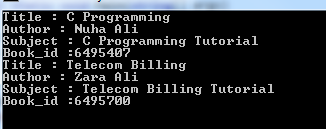
/\* print Book2 info \*/

Book2.display();

Console.ReadKey();

}

}

Output: 

# C# Enums

An enumeration is a set of named integer constants. An enumerated type is declared using the enum keyword. C# enumerations are value data type. In other words, enumeration contains its own values and cannot inherit or cannot pass inheritance.

Declaring Enum variable

The general syntax for declaring an enumeration is −

enum <enum\_name> {

enumeration list

};

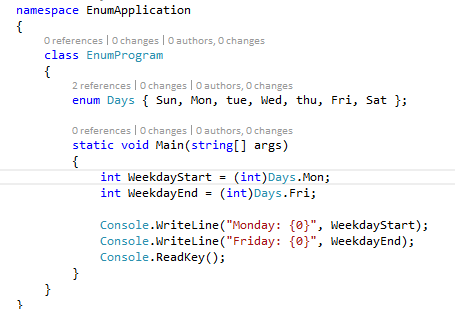
Where,

* The *enum\_name* specifies the enumeration type name.
* The *enumeration list* is a comma-separated list of identifiers.

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

Example:



Output:

