**C# programming Manual Book**

Overview

C# is a modern, general-purpose, object-oriented programming language developed by Microsoft and approved by Ecma and ISO.

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

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

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

* Modern, general-purpose programming language
* Object oriented.
* Component oriented.
* Easy to learn.
* Structured language.
* It produces efficient programs.
* It can be compiled on a variety of computer platforms.
* Part of .Net Framework.

**Strong Programming Features of C#**

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

Following is the list of few important features:

* 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# - Program Structure

Before we study basic building blocks of the C# programming language, let us look at a bare minimum C# program structure so that we can take it as a reference in upcoming chapters.

## C# Hello World Example

A C# program basically consists of the following parts:

* Namespace declaration
* A class
* Class methods
* Class attributes
* A Main method
* Statements & Expressions
* Comments

Let us look at a simple code that would print the words "Hello World":

using System;

namespace HelloWorldApplication

{

class HelloWorld

{

static void Main(string[] args)

{

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

Console.WriteLine("Hello World");

Console.ReadKey();

}

}

}

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

Hello World

Let us look at various parts of the above 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 would contain more than one method. 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 will do when executed
* The next line /\*...\*/ will be ignored by the compiler and it has been put to add additional **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's 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, file name could be different from the class name.

**Compile & Execute a C# Program:**

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

* Start Visual Studio.
* On the menu bar, choose File, New, Project.
* Choose Visual C# from templates, and then choose Windows.
* Choose Console Application.
* Specify a name for your project, and then choose the OK button.
* The new project appears in Solution Explorer.
* Write code in the Code Editor.
* Click the Run button or the F5 key to run the project. A Command Prompt window appears that contains the line Hello World.

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

* Open a text editor and add the above-mentioned code.
* Save the file as **helloworld.cs**
* Open the command prompt tool and go to the directory where you saved the file.
* Type **csc helloworld.cs** and press enter to compile your code.
* If there are no errors in your code, the command prompt will take you to the next line and would generate **helloworld.exe** executable file.
* Next, type **helloworld** to execute your program.
* You will be able to see "Hello World" printed on the screen.

Try the Following Exercises

Exercise 1

E-Library

1] Add New Book

2] View Books

3] Search Book

4] Delete Book

5] Update Book

Enter Choice:

Aya Bank

1] Open New Account

2] Update Account

3] Deposit

4] Withdrawal

5] exit program

Enter Choice:

# C# - Basic Syntax

C# is an object-oriented programming language. In Object-Oriented Programming methodology, a program consists of various objects that interact with each other by means of actions. The actions that an object may take are called methods. Objects of the same kind are said to have the same type or, more often, are said to be in the same class.

For example, let us consider a Rectangle object. It has attributes like length and width. Depending upon the design, it may need ways for accepting the values of these attributes, calculating area and display details.

Let us look at an implementation of a Rectangle class and discuss C# basic syntax, on the basis of our observations in it:

using System;

namespace RectangleApplication

{

class Rectangle

{

// member variables

double length;

double width;

public void Acceptdetails()

{

length = 4.5;

width = 3.5;

}

public double GetArea()

{

return length \* width;

}

public void Display()

{

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

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

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

}

}

class ExecuteRectangle

{

static void Main(string[] args)

{

Rectangle r = new Rectangle();

r.Acceptdetails();

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

**The *using* Keyword**

The first statement in any C# program is

using System;

The **using** keyword is used for including the namespaces in the program. A program can include multiple using statements.

**The *class* Keyword**

The **class** keyword is used for declaring a class.

**Comments in C#**

Comments are used for explaining code. Compilers ignore the comment entries. The multiline comments in C# programs start with /\* and terminates with the characters \*/ as shown below:

/\* This program demonstrates

The basic syntax of C# programming

Language \*/

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

}//end class Rectangle

## Character Constants

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

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

|  |  |
| --- | --- |
| **Escape sequence** | **Meaning** |
| \\ | \ character |
| \' | ' character |
| \" | " character |
| \? | ? character |
| \a | Alert or bell |
| \b | Backspace |
| \f | Form feed |
| \n | Newline |
| \r | Carriage return |
| \t | Horizontal tab |
| \v | Vertical tab |
| \ooo | Octal number of one to three digits |
| \xhh . . . | Hexadecimal number of one or more digits |

Following is the example to show few escape sequence characters:

namespace EscapeChar

{

class Program

{

static void Main(string[] args)

{

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

Console.ReadLine();

}

}

}

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

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;

The following program demonstrates defining and using a constant in your program:

using System;

namespace DeclaringConstants

{

class Program

{

static void Main(string[] args)

{

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

Try the Following Exercises

Exercise3.cs

Divide Numbers

Enter Input1: 5

Enter Input2: 3

The Answer is : 1.6666666

Exercise 2.cs

Multiplying Numbers

Enter Input1: 3

Enter Input2: 3

The Answer is : 9

# C# - Decision Making

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general from of a typical decision making structure found in most of the programming languages:



C# provides following types of decision making statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Statement** | **Description** |
| [if statement](http://www.tutorialspoint.com/csharp/if_statement_in_csharp.htm) | An **if statement** consists of a boolean expression followed by one or more statements. |
| [if...else statement](http://www.tutorialspoint.com/csharp/if_else_statement_in_csharp.htm) | An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false. |
| [nested if statements](http://www.tutorialspoint.com/csharp/nested_if_statements_in_csharp.htm) | You can use one **if** or **else if** statement inside another **if** or **else if** statement(s). |
| [switch statement](http://www.tutorialspoint.com/csharp/switch_statement_in_csharp.htm) | A **switch** statement allows a variable to be tested for equality against a list of values. |
| [nested switch statements](http://www.tutorialspoint.com/csharp/nested_switch_statements_in_csharp.htm) | You can use one **switch** statement inside another **switch** statement(s). |

## The ? : Operator:

We have covered **conditional operator ? :** in previous chapter which 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 like this: 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.

Try the following Exercises

CompareExer2.cs

Comparing five Inputs

Enter input 1: **2**

Enter Input2: **15**

Enter Input 3: **82**

Enter Input 4: **12**

Enter input 5: **621**

**The Largest Number is : 621**

CompareExer1.cs

Comparing four Inputs

Enter input 1: **2**

Enter Input2: **15**

Enter Input 3: **82**

Enter Input 4: **12**

**The Largest Number is : 82**

Switch Statement

switch(condition){

Case 1: statement ;

Break;

Case 2: statement;

Break;

Case 3: statement;

Break;

.

.

.

Default: statement;

}

Sample 1

**using** System;  
 **namespace** SwitchExercise{  
       
       
   
    class SwitchTest  
    {  
        static void **Main**()  
        {  
                Console.**WriteLine**("T-Shirt Size \n  1]small   \n 2]medium  \n 3]large");  
            Console.**Write**("Please enter your selection: ");  
            **int** choice=Convert.**ToInt16**(Console.**ReadLine**());  
          
  
            **switch** (choice)  
            {  
            
                **case** 1:Console.**WriteLine**(" The price is: US$15 \t Thanks for shopping ");  
                        break;  
                **case** 2: Console.**WriteLine**(" The Price is: US$20 \t Thanks for shopping ");  
                         break;  
                                                    
                   **case** 3: Console.**WriteLine**(" The Price is: US$30 \t Thanks for shopping");  
                         break;                        
                  
                **default**:Console.**WriteLine**("Invalid selection. Please select 1, 2, or 3.");  
                break;  
            }  
              
            Console.**ReadKey**();  
          
            }  
}  
  
 }

Sample 2

**using** System;  
**namespace** DecisionMaking  
{  
   class Program  
   {  
      static void **Main**(string[] args)  
      {  
           
          Console.**Write**("Enter your score:");  
          **char** grade=Convert.**ToChar**(Console.**ReadLine**());  
           
           
         **switch** (grade)  
         {  
            **case** 'A':  
               Console.**WriteLine**("Excellent!");  
               break;  
            **case** 'B':  
            **case** 'C':  
               Console.**WriteLine**("Well done");  
               break;  
            **case** 'D':  
               Console.**WriteLine**("You passed");  
               break;  
            **case** 'F':  
               Console.**WriteLine**("Better try again");  
               break;  
               **default**:  
            Console.**WriteLine**("Invalid grade");  
               break;  
         }  
           
           
         Console.**WriteLine**("Your grade is  {0}", grade);  
         Console.**ReadLine**();  
      }  
   }  
}

Do the following exercises

Enter height: 3

Enter base: 3

The Volume of the Trianle is : 3

Enter height: 3

Enter base: 3

Formulas

Area=(b\*h)/2

vol=(b\*h)/3

Formulas

Area=pi\*r\*r;

Cir=2\*pi\*r;

Triangle choice == 2

Volume of Triangle

Triangle choice == 1

The Area of the Triangle is : 4.5

Area of Triangle

The Area of the Circle is : 28.26

Enter Radius: 3

The Area of the Circle is : 18.84

Enter Radius: 3

Circumference of Circle

Circle choice == 2

Circle choice == 1

Area of Circle

Main Menu

(My Mathematic Program)

[1] Circle

[2] Triangle

[3] Rectangle

Enter Choice:

Circle Menu

[1] Area of Circle

[2] Circumference of Circle

Enter Choice:

Main choice==1

Triangle Menu

[1] Area of Triangle

[2] Volume of Triangle

Enter Choice:

Main choice==2

Triangle Menu

[1] Area of Rectangle

[2] Volume of Rectangle

Enter Choice:

Main choice==3

Volume of Rectangle

Enter height: 3

Formulas

Area=w\*h;

Vol=b\*h\*w

Rectangle choice == 2

The Volume of the Tectangle is : 27

Rectangle choice == 1

The Area of the Rectangle is : 9

Enter width: 3

Area of Rectangle

Enter base: 3

Enter height: 3

Enter width: 3

**Control Statements - Loops**

In the last lesson, you learned how to create a simple loop by using the *goto* statement. I advised you that this is not the best way to perform loops in C#. The information in this lesson will teach you the proper way to execute iterative logic with the various C# looping statements. Its goal is to meet the following objectives:

* Learn the *while* loop.
* Learn the *do* loop.
* Learn the *for* loop.
* Learn the *foreach* loop.
* Complete your knowledge of the *break* statement.
* Teach you how to use the *continue* statement.

**The *while* Loop**

A *while* loop will check a condition and then continues to execute a block of code as long as the condition evaluates to a boolean value of *true*. Its syntax is as follows: *while (<boolean expression>) { <statements> }.* The statements can be any valid C# statements. The boolean expression is evaluated before any code in the following block has executed. When the boolean expression evaluates to *true*, the statements will execute. Once the statements have executed, control returns to the beginning of the *while* loop to check the boolean expression again.

When the boolean expression evaluates to *false*, the *while* loop statements are skipped and execution begins after the closing brace of that block of code. Before entering the loop, ensure that variables evaluated in the loop condition are set to an initial state. During execution, make sure you update variables associated with the boolean expression so that the loop will end when you want it to. Listing 4-1 shows how to implement a *while* loop.

**Listing 4-1. The While Loop: WhileLoop.cs**

using System;  
  
class WhileLoop  
{  
    public static void Main()  
    {  
        int count= 0;  
  
        while (count< 10)  
        {  
            Console.Write("{0} ", count);  
            count++;  
        }  
        Console.WriteLine();  
    }  
}

Listing 4-1 shows a simple *while* loop. It begins with the keyword *while*, followed by a boolean expression. All control statements use boolean expressions as their condition for entering/continuing the loop. This means that the expression must evaluate to either a *true* or *false* value. In this case we are checking the *myInt* variable to see if it is less than (<) 10. Since *myInt* was initialized to 0, the boolean expression will return *true* the first time it is evaluated. When the boolean expression evaluates to *true*, the block immediately following the boolean expression will be executed.

Within the *while* block we print the number and a space to the console. Then we increment (++) *myInt* to the next integer. Once the statements in the *while* block have executed, the boolean expression is evaluated again. This sequence will continue until the boolean expression evaluates to *false*. Once the boolean expression is evaluated as *false*, program control will jump to the first statement following the *while* block. In this case, we will write the numbers 0 through 9 to the console, exit the *while* block, and print a new line to the console.

**The *do* Loop**

A *do* loop is similar to the *while* loop, except that it checks its condition at the end of the loop. This means that the *do* loop is guaranteed to execute at least one time. On the other hand, a *while* loop evaluates its boolean expression at the beginning and there is generally no guarantee that the statements inside the loop will be executed, unless you program the code to explicitly do so. One reason you may want to use a *do* loop instead of a *while* loop is to present a message or menu such as the one in Listing 4-2 and then retrieve input from a user.

**Listing 4-2. The Do Loop: DoLoop.cs**

using System;  
  
class DoLoop  
{  
    public static void Main()  
    {  
        string myChoice;  
  
        do  
       {  
            // Print A Menu  
            Console.WriteLine("My Address Book\n");  
  
            Console.WriteLine("A - Add New Address");  
            Console.WriteLine("D - Delete Address");  
            Console.WriteLine("M - Modify Address");  
            Console.WriteLine("V - View Addresses");  
            Console.WriteLine("Q - Quit\n");  
  
            Console.WriteLine("Choice (A,D,M,V,or Q): ");  
  
            // Retrieve the user's choice  
            myChoice = Console.ReadLine();  
  
            // Make a decision based on the user's choice  
            switch(myChoice)  
            {  
                case "A":  
                case "a":  
                    Console.WriteLine("You wish to add an address.");  
                    break;  
                case "D":  
                case "d":  
                    Console.WriteLine("You wish to delete an address.");  
                    break;  
                case "M":  
                case "m":  
                    Console.WriteLine("You wish to modify an address.");  
                    break;  
                case "V":  
                case "v":  
                    Console.WriteLine("You wish to view the address list.");  
                    break;  
                case "Q":  
                case "q":  
                    Console.WriteLine("Bye.");  
                    break;  
                default:  
                    Console.WriteLine("{0} is not a valid choice", myChoice);  
                    break;  
            }  
  
            // Pause to allow the user to see the results  
            Console.Write("press Enter key to continue...");  
            Console.ReadLine();  
            Console.WriteLine();  
        } while (myChoice != "Q" && myChoice != "q"); // Keep going until the user wants to quit  
    }  
}

Listing 4-2 shows a *do* loop in action. The syntax of the *do* loop is *do { <statements> } while (<boolean expression>);*. The statements can be any valid C# programming statements you like. The boolean expression is the same as all others we've encountered so far. It returns either *true* or *false*.

In the *Main* method, we declare the variable *myChoice* of type *string*. Then we print a series of statements to the console. This is a menu of choices for the user. We must get input from the user, which is in the form of a *Console.ReadLine* method which returns the user's value into the *myChoice* variable. We must take the user's input and process it. A very efficient way to do this is with a *switch* statement. Notice that we've placed matching upper and lower case letters together to obtain the same functionality. This is the only legal way to have automatic fall through between cases. If you were to place any statements between two cases, you would not be able to fall through. Another point is that we used the *default:* case, which is a very good habit for the reasons stated in [Lesson 3: Control Statements - Selection](http://www.csharp-station.com/Tutorial/CSharp/Lesson03).

**Do the following Exercises using while-loop**

Exercise 1:

0

4

19

4

14

Exercise 2:

19

0

4

14

14

14

4

19

Exercise 3:

19

0

4

8

14

14

14

4

19

**The *for* Loop**

A *for* loop works like a *while* loop, except that the syntax of the *for* loop includes initialization and condition modification. *for* loops are appropriate when you know exactly how many times you want to perform the statements within the loop. The contents within the *for* loop parentheses hold three sections separated by semicolons *(<initializer list>; <boolean expression>; <iterator list>) { <statements> }*.

The initializer list is a comma separated list of expressions. These expressions are evaluated only once during the lifetime of the *for* loop. This is a one-time operation, before loop execution. This section is commonly used to initialize an integer to be used as a counter.

Once the initializer list has been evaluated, the *for* loop gives control to its second section, the boolean expression. There is only one boolean expression, but it can be as complicated as you like as long as the result evaluates to *true* or *false*. The boolean expression is commonly used to verify the status of a counter variable.

When the boolean expression evaluates to *true*, the statements within the curly braces of the *for* loop are executed. After executing *for* loop statements, control moves to the top of loop and executes the iterator list, which is normally used to increment or decrement a counter. The iterator list can contain a comma separated list of statements, but is generally only one statement. Listing 4-3 shows how to implement a *for* loop. The purpose of the program is to  print only odd numbers less than 10.

**Listing 4-3. The For Loop: ForLoop.cs**

using System;  
  
class ForLoop  
{  
    public static void Main()  
    {  
        for (int i=0; i < 20; i++)  
        {  
            if (i == 10)  
                break;  
  
            if (i % 2 == 0)  
                continue;  
  
            Console.Write("{0} ", i);  
        }  
        Console.WriteLine();  
    }  
}

Normally, *for* loop statements execute from the opening curly brace to the closing curly brace without interruption. However, in Listing 4-3, we've made a couple exceptions. There are a couple *if* statements disrupting the flow of control within the *for* block.

The first *if* statement checks to see if *i* is equal to 10. Now you see another use of the *break* statement. Its behavior is similar to the selection statements, as discussed in [Lesson 3: Control Statements - Selection](http://www.csharp-station.com/Tutorial/CSharp/Lesson03). It simply breaks out of the loop at that point and transfers control to the first statement following the end of the *for* block.

The second *if* statement uses the remainder operator to see if *i* is a multiple of 2. This will evaluate to *true* when *i* is divided by 2 with a remainder equal to zero, (0). When *true*, the *continue* statement is executed, causing control to skip over the remaining statements in the loop and transfer back to the iterator list. By arranging the statements within a block properly, you can conditionally execute them based upon whatever condition you need.

When program control reaches either a *continue* statement or end of block, it transfers to the third section within the *for* loop parentheses, the iterator list. This is a comma separated list of actions that are executed after the statements in the *for* block have been executed. Listing 4-3 is a typical action, incrementing the counter. Once this is complete, control transfers to the boolean expression for evaluation.

Similar to the *while* loop, a *for* loop will continue as long as the boolean expression is *true*. When the boolean expression becomes *false*, control is transferred to the first statement following the *for* block.

For this tutorial, I chose to implement *break* and *continue* statements in Listing 4-3 only. However, they may be used in any of the loop statements.

**The *foreach* Loop**

A *foreach* loop is used to iterate through the items in a list. It operates on arrays or collections such as ArrayList, which can be found in the System.Collections namespace. The syntax of a foreach loop is *foreach (<type> <iteration variable> in <list>) { <statements> }*. The type is the type of item contained in the list. For example, if the type of the list was *int[]* then the type would be *int*.

The iteration variable is an identifier that you choose, which could be anything but should be meaningful. For example, if the list contained an array of people's ages, then a meaningful name for item name would be age.

The *in* keyword is required.

As mentioned earlier, the list could be either an array or a collection. You learned about arrays in [Lesson 02: Operators, Types, and Variables](http://www.csharp-station.com/Tutorial/CSharp/Lesson02). You can also iterate over C# generic collections also, described in [Lesson 20: Introduction to Generic Collections](http://www.csharp-station.com/Tutorial/CSharp/Lesson20).

While iterating through the items of a list with a *foreach* loop, the list is read-only. This means that you can't modify the iteration variable within a *foreach* loop. There is a subtlety here; Later, you'll learn how to create custom types, called class and struct, that can contain multiple fields.  You can change the fields of the class or struct, but not the iteration variable for the class or struct itself in a *foreach* loop.

On each iteration through a *foreach* loop the list is queried for a new value. As long as the list can return a value, this value will be put into the read-only iteration variable, causing the statements in the *foreach* block to be executed. When the collection has been fully traversed, control will transfer to the first executable statement following the end of the *foreach* block. Listing 4-4 demonstrates how to use a *foreach* loop.

**The ForEach Loop: ForEachLoop.cs**

using System;  
  
class ForEachLoop  
{  
    public static void Main()  
    {  
        string[] names = {"Cheryl", "Joe", "Matt", "Robert"};  
  
        foreach (string person in names)  
        {  
            Console.WriteLine("{0} ", person);  
        }  
    }  
}

first thing we've done inside the *Main* method is declare and initialize the *names* array with 4 *strings*. This is the list used in the *foreach* loop.

In the *foreach* loop, we've used a *string* variable, *person*, as the item name, to hold each element of the *names* array. As long as there are names in the array that have not been returned, the *Console.WriteLine* method will print each value of the *person* variable to the screen.

**Do the following Exercises**

**\***

**\* \* \***

**\* \* \* \* \***

**\* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \***

**\* \* \* \* \***

**\* \* \***

**\***

[1] 9x9 diamond

**\* \* \* \* \* \* \* \* \* \* \***

**\* \* \***

**\* \* \* \* \***

**\* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \***

**\* \* \* \* \***

**\* \* \***

**\* \* \* \* \* \* \* \* \* \* \***

[2] 9x9 diamond with frame

**\* \* \* \* \* \* \* \* \* \* \***

**\* \* \***

**\* \* \* \* \***

**\* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \***

**\* \* \* \* \***

**\* \* \***

**\* \* \* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \* \* \***

**\* \* \***

**\* \* \* \* \***

**\* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \***

**\* \* \* \* \***

**\* \* \***

**\* \* \* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \* \* \***

**\* \* \***

**\* \* \* \* \***

**\* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \***

**\* \* \* \* \***

**\* \* \***

**\* \* \* \* \* \* \* \* \* \* \***

[3] Vertical Diamond Pattern

[4] Horizontal Diamond Pattern

**\* \* \* \* \* \* \* \* \* \* \***

**\* \* \***

**\* \* \* \* \***

**\* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \***

**\* \* \* \* \***

**\* \* \***

**\* \* \* \* \* \* \* \* \* \* \***

**\* \* \***

**\* \* \* \* \***

**\* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \***

**\* \* \* \* \***

**\* \* \***

**\* \* \* \* \* \* \* \* \* \* \***

**\* \* \***

**\* \* \* \* \***

**\* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \* \* \***

**\* \* \* \* \* \* \***

**\* \* \* \* \***

**\* \* \***

**\* \* \* \* \* \* \* \* \* \* \***

**Summary**

Loops allow you to execute a block of statements repeatedly. C# offers several statements to construct loops with, including the *while*, *do*, *for*, and *foreach* loops. *while* loops execute a block of statements as long as an expression is *true*, *do* loops execute a block of statements at least once and then keep going as long as a condition is *true*, *for* loops execute a block of statements a specified amount of times, and *foreach* loops execute a block of statements for each item in a collection. Normally a block of statements will execute from beginning to end. However, the normal flow of a loop can be changed with the *break* and *continue* statements.

So far, the only method you've seen in this tutorial is the Main method, which is the entry point of a C# application. However, you are probably wanting to write larger programs to test your new knowledge. This requires breaking up the code into methods to keep it organized and logical.

**String Manipulations**

Strings are an integral part of any programming language. In almost every programming language you would find strings of data being passed as input and being displayed as input. Strings are an important way to interact with the user.   
  
Whenever you visit a website, you are often asked to sign up into the website or login if you are already a member of the website. You enter a username and password, system performs some validation checks and you are good to go. The process might look pretty straight forward to you at that moment but behind the screen lots and lots of string handling is being done. Similarly, database applications make huge use of string and actually strings are the intermediate between the user and the database for information storage.   
  
This article is dedicated to string handling and text manipulation in C#. We will show you how actually string resides in the memory. What are the build in methods that you can call on the string class and where can they be used. After completing this article, you will have a better idea about how the login systems work and what happens to the strings that you enter there. Things you will learn in this article will help you understand and implement strings in your code in a better way. So, without wasting any further time on theory, let us get straight to the task.

* [String Construction](http://www.go4expert.com/articles/c-sharp-string-manipulation-tutorial-t30015/#string-construct)
* [Empty and Null Strings](http://www.go4expert.com/articles/c-sharp-string-manipulation-tutorial-t30015/#empty-null-strings)
* [String Searching](http://www.go4expert.com/articles/c-sharp-string-manipulation-tutorial-t30015/#string-searching)
* [String Manipulation](http://www.go4expert.com/articles/c-sharp-string-manipulation-tutorial-t30015/#string-manipulation)
* [Join and Split Strings](http://www.go4expert.com/articles/c-sharp-string-manipulation-tutorial-t30015/#join-split-strings)
* [String Comparison](http://www.go4expert.com/articles/c-sharp-string-manipulation-tutorial-t30015/#comparison)

**String Construction**

Constructing a string in C# is an extremely easy task. Strings are actually class belonging to the System namespace. So actually what you are doing is creating instances of System.String class and applying different functionalities on that class.   
  
The easiest and simplest way to declare a string in C# is to create an instance of a string class and then equate it to some literal value. A string is basically a sequence of characters. Following is the simplest way to construct a string in C#.

Code:

string name = “Expert”;

Code:

string description =”1-C#\n2-Java\n3-C++”;

Code:

string content =@”8fsf8s\\sdf9sf\_’’asaa”;

In order to create a sequence of characters, for instance some special characters you can simply use string’s constructor as

Code:

string stars = new string( ‘\*’,15);

The above line of code will generate a string with 15 asteric characters.   
  
We mentioned earlier that strings are basically sequence of characters. Therefore, we can convert a string into a character array using ToCharArray method and we can convert array of characters back to a string by passing it into the constructor of the string as follows.

Code:

char [] arr = “Expert”.ToCharArray;

string name = new string (arr);

Let us recapitulate the concepts we have studied till now with the help of some example. Have a look at Example1.  
  
**Example1**

Code:

using System;

using System.Collections;

using System.Collections.Generic;

using System.Text;

namespace CSharpTutorial {

class Program {

static void Main() {

string name = "Expert";

Console.WriteLine(name);

string description ="1-C#\n2-Java\n3-C++";

Console.WriteLine(description);

string content = @"8fsf8s\nsdf9sf\_’’asaa";

Console.WriteLine(content);

char [] arr= "Expert".ToCharArray();

string array = new string(arr);

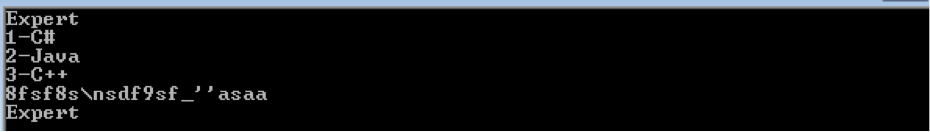
Console.WriteLine(array);

Console.ReadLine();

}

}

}

The code in Example1 is simply practical implementation of string construction concept. We have created a string name and displayed it, then we created a string description to give you an idea of how we can use \n to go to a new line within a string. And next in the content string we have displayed that if you want to includes special characters in the string as it is, for example if you want to insert \n, it will shift the control to next line but if you want to display \n in a string you can simply prefix a @ character before the string literal and it will display the string as it is. In next lines we declared an array of characters and passed it to the constructor of the string. In the next line we have displayed the value of that string. The output of code in Example1 would be as follows.  
  
**Ouput1**  
  
  
  
You can see in the fourth line that we have displayed a \n character within a string which if we don’t append @ before the string, would have shifted the control to the next line.   
  
You can see that how simple it is to create and manipulate a string in C#. Now let us get to some other important concepts.

**Empty and Null Strings**

String can be empty, which means that it doesn’t contain any character. The length of an empty string will always be zero. Usually empty strings are used for validating if user has entered any data or not. We will show that later with the help of an example, but for now there are two ways to create an empty string.

1. string name = “”;
2. string name= string.Empty;

In both of the above cases an empty string will be stored in string variable name.  
A null string is different from an empty string. String is a reference type which means that a string can be null. In order to point a string to null reference you simply have to do string name = null;  
  
There is an important and useful method associated with strings that checks if a string is null or empty. The method is IsNullOrEmpty.  
  
Now let us come towards our next example which will explain the concepts of null and empty strings.  
  
**Example2**

Code:

using System;

using System.Collections;

using System.Collections.Generic;

using System.Text;

namespace CSharpTutorial {

class Program {

static void Main() {

string name;

Console.WriteLine("Enter your name:");

name = Console.ReadLine();

if (name == string.Empty)

Console.WriteLine("The name field cannot be left empty.");

else

Console.WriteLine("There are " + name.Length + " characters in your name.");

name = null;

if(string.IsNullOrEmpty(name))

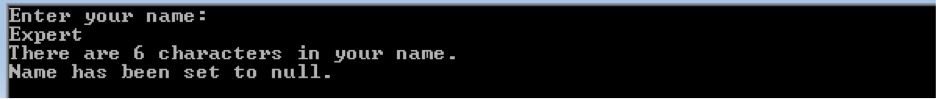
Console.WriteLine("Name has been set to null.");

Console.ReadLine();

}

}

}

Example 2 is another simple example. We simply ask the user to enter his name. If user doesn’t enter anything, it will display the message to the user that string you entered is empty which is not allowed. This is a typical login scenario where you enter your user name and password and if any of them is empty, the application prompts with an error. In those application database comparisons are being done but now for the sake of understanding we limit it to string comparison. Coming back to our problem, hand if user enters any string, we will display the length of that string. Next we will make our string null and will call IsNullOrEmpty static method on that string to verify if the string is really null or not. The output of the above code will be as follows.  
  
**Output2**  
  


**String Searching**

Searching characters or substrings within a string is another extremely important task while dealing with strings. For example, if you want to find out that if a string contains a specific words or specific character or if you want to find out if a string ends with a particular word or not you can make use of string searching functions. Although, string type contains numerous searching functions along with their overloads, we will study the ones that are most commonly used.  
  
In order to access a character within a string you can simply use array style index searching. Strings are basically sequence of characters; therefore you can access character within a string by its index. For example

Code:

string name = “Expert”;

char c = name[1] // This will return ‘x’, because x is located at first index of Expert

Simply, in order to search strings within a string or more commonly known as substrings. You can make use of three basic functions.  
  
**1. Contains**  
  
This function is used to search a substring in a string. For example

Code:

string name = “I am a CSharp Expert”;

Console.WriteLine (name.Contains(“CSharp”));

This will evaluate to true because string name contains substring “CSharp”  
  
**2. StartsWith**  
  
This function is used to search a substring in at the start of a string. For example

Code:

string name = “James is a CSharp Expert”;

Console.WriteLine (name.StartsWith(“James”));

This will evaluate to true because string name starts with “James”  
  
**3. EndsWith**   
  
This function is used to search a substring in at the end of a string. For example

Code:

string name = “James is a CSharp Expert”;

Console.WriteLine (name.EndsWith(“Expert”));

This will evaluate to true because string name ends with “Expert”  
  
These are the three basic methods for finding a substring. Another extremely important method is that of IndexOf method which is used to find the starting index of a substring. For example, if you have a string named

Code:

string name = “expert”;

Console.WriteLine(name.IndexOf(“per”));

This will display the value 2, because the starting index of substring per is 2 in the string expert.   
  
Now, as usual, we will explain all of the string searching concepts that we have studied till now with the help of an example. Have a look at Example3 to further understand the aforementioned string searching functions.  
  
**Example3**

Code:

using System;

using System.Collections;

using System.Collections.Generic;

using System.Text;

namespace CSharpTutorial

{

class Program

{

static void Main()

{

string name = "I am a CSharp Expert";

Console.WriteLine("Calling Contains Function ...");

Console.WriteLine(name.Contains("CSharp"));

name = "James is a CSharp Expert";

Console.WriteLine("Calling StartsWith Function ...");

Console.WriteLine(name.StartsWith("James"));

Console.WriteLine("Calling EndsWith Function ...");

Console.WriteLine(name.EndsWith("Expert"));

name = "Expert";

Console.WriteLine("Calling IndexOf Function ...");

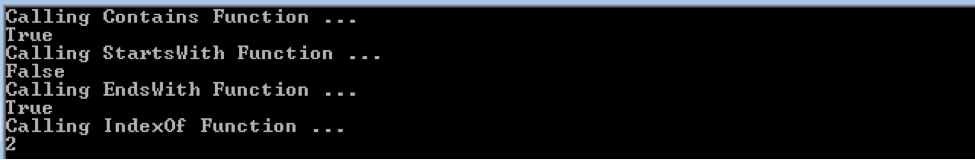
Console.WriteLine(name.IndexOf("per"));

Console.ReadLine();

}

}

}

The code in Example3 is the implementation of what we studied in theory. We have simply created a string name and have then called corresponding Contains, StartsWith and EndsWith function on that string. Finally we called IndexOf function. You will see in the output that all of the functions were evaluated to true in the Console.WriteLine statement because Contains found Charp, StartsWith found James and EndsWith found Expert in the string name. In the end, the Index of method will fetch the starting index of the string ‘per’ and will display that on the output screen. The output of the above code is as follows.  
  
**Output3**  
  
  
  
An extremely important thing to note here is that if you change the case of the string while calling Contains, StartsWith and EndsWith method.  
  
For instance in the Example3, if you make following changes while calling the StartsWith method

Code:

string name = "James is a CSharp Expert";

Console.WriteLine("Calling StartsWith Function ...");

Console.WriteLine(name.StartsWith("JAMES"));

Now this method will evaluate to false. Though string ‘name’ starts with ‘James’ but in the StartsWith method we are comparing the string ‘JAMES’ all uppercase to the actual string. Therefore, this will evaluate to false. But there is a very simple solution to this problem. Contains, StartsWith and EndsWith methods come with overload using which you can ignore the case while comparing the string.   
  
If you use following overload of these methods, you can compare the strings irrespective of the case.

Code:

string name = "James is a CSharp Expert";

Console.WriteLine("Calling StartsWith Function ...");

Console.WriteLine(name.StartsWith("JAMES",true,System.Globalization.CultureInfo.InvariantCulture));

Basically, the first argument is the string to compare; the second argument is Boolean variable ignorecase, if true, the case of the string being compared will be ignored, if false, the case of the string will be taken into account.   
  
Pay particular attention to the third argument, CulturalInfo is a type in System.Globalization namespace. We have passed the InvariantCulture property of this type as a third argument in StartsWith method. It will actually make the method insensitive to the region or location in which the StartsWith method is being used. This is particularly important, because English language strings might work differently in different languages and regions. Therefore, if you want your application strings to behave uniformly in different regions, you can use this overload.   
  
Another way to control the case of the comparison is to use another overload of these methods. Make the following changes in your code.

Code:

string name = "Expert";

Console.WriteLine("Calling IndexOf Function ...");

Console.WriteLine(name.IndexOf("PER"));

Now, the index returned would be some negative value, because IndexOf method is not able to find ‘PER’ substring inside the “Expert” string due to case sensitivity. There is another way to control case sensitivity while comparison. Make following changes in the code.

Code:

string name = "Expert";

Console.WriteLine("Calling IndexOf Function ...");

Console.WriteLine(name.IndexOf("PER",StringComparison.InvariantCultureIgnoreCase));

This code will again ignore the case of the string passed inside the IndexOf method.

**String Manipulation**

String manipulation is a process of calling a method on a string instance and retrieving another string. Strings are immutable in nature which means that whenever a string is manipulated a new string instance is created in the memory and returned back. The actual string instance stays as it is. StringBuilder class is a string variant that is mutable; we will see that later. For now, let us discuss some string manipulation functions.  
  
**1. Substring**  
  
Substring, method returns portion of string based on the indexes passed to it. For example

Code:

string name = "Expert";

string sub = name.Substring(1, 3);

This function will fetch the substring from index 1 of string ‘Expert’ to index ‘3’. The output of this will be ‘xpe’.  
  
If you pass only one parameter to Substring function of the string, it will omit those number of characters from the string starting from the first index. Consider the following lines of code.

Code:

string name = "Expert";

string sub = name.Substring(2);

The string sub will contain ‘pert’, because the first two characters ‘Ex’ would be omitted by the substring function when we pass single integer parameter ‘2’ to it.  
  
**2. Insert**  
  
Another important function which is used in string manipulation is the Insert function. This function is used to insert string inside another string, starting from the specified index. Have a look at the example.

Code:

string name = "Expert";

string sub = name.Insert(2, "--");

Here we have a string ‘Expert’ stored in string instance name. We call insert function on this string and pass it parameters 2 and ‘–‘, respectively. It means that starting from the index 2, insert string ‘--’ into it. The output string ‘sub’ will be “Ex--pert”.  
  
**3.Remove**  
  
Remove is another important, yet simple to use string manipulation function. Remove basically allows you to remove a substring from main string. The first parameter is the starting index from where you want to remove the string and the second parameter is the number of characters you want to remove. Consider following lines of code.

Code:

string name = "Ex--pert";

string sub = name.Remove(2, 2);

Here we have an input string ‘name’ that contains string “Ex--pert”. We want to remove two dashes between the Ex and pert. We can simply pass 2 as a first parameter telling the compiler that we want to start removing string from index 2 and then 2, telling that 2 characters from that index.  
  
**4. PadLeft and PadRight**  
  
PadLeft and PadRight are used to pad certain character on the left and right of a string respectively. The first parameter is the number of character to pad and the second parameter is the character to pad. If you pass only one parameter i.e. number of characters, spaces would be padded. Consider following code.  
  
This code will append 5 sterics on the right of string name.

Code:

string name = "Expert";

string sub = name.PadRight(5,'\*');

And this code will append 5 spaces on the left of the string name.

Code:

string name = "Expert";

string sub = name.PadLeft(5);

**5. Trim**  
  
Trim function is used to trim specified characters from the start and end of the string. If characters are not specified, it will trim white spaces from the start and end. TrimStart and TrimEnd method trim white spaces from only start and end of the string respectively. Have a look at following code.

Code:

string name = " I am a CSharp expert ";

string sub = name.Trim();

If we display the ‘name’ string, the white spaces at the beginning and end would not be there.  
  
**6. ToUpper and ToLower**  
  
These functions are used to convert a string into upper and lower case respectively. We will see the usage of this method in our detailed example.  
  
Now, have a look at the Exampl1, we have implemented all the functions that we have studied till now in this Example1.  
  
**Example4**

Code:

using System.Collections;

using System.Collections.Generic;

using System.Text;

namespace CSharpTutorial

{

class Program

{

static void Main()

{

Console.WriteLine("Using Substring function ...");

string name = "Expert";

string sub = name.Substring(1, 3);

Console.WriteLine(sub);

Console.WriteLine("===========\n");

name = "Expert";

sub = name.Substring(2);

Console.WriteLine(sub);

Console.WriteLine("===========\n");

Console.WriteLine("Using Insert function ...");

name = "Expert";

sub = name.Insert(2, "--");

Console.WriteLine(sub);

Console.WriteLine("===========\n");

Console.WriteLine("Using Remove function ...");

name = "Ex--pert";

sub = name.Remove(2, 2);

Console.WriteLine(sub);

Console.WriteLine("===========\n");

Console.WriteLine("Using Trim function ...");

name = " I am a CSharp expert ";

sub = name.Trim();

Console.WriteLine(sub);

Console.WriteLine("===========\n");

Console.WriteLine("Using ToUpper function ...");

name = "Expert";

sub = name.ToUpper();

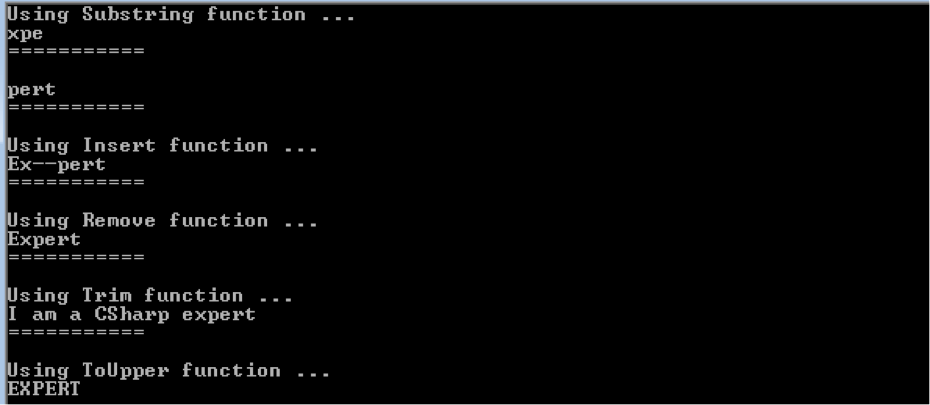
Console.WriteLine(sub);

Console.ReadLine();

}

}

}

The code in Example1 is extremely simple. We have simply copy pasted the function examples into a working example to see how the program works and is the output of the program similar to what we studied in the examples. When you run this program you will see that all the outputs are according to our expectations and functions do their job as we discussed earlier. The output of this code is as follows.  
  
**Output4**  
  


**Join and Split Strings**

String joining and splitting are also handy. Often times you want to break a sentence into words or you want to join words to form a single string. String joining and splitting function basically allow you to do this. There are many important string manipulation function but we will study most important of them.  
  
**1. Split**  
  
Split method is used to split the strings. By default, it uses whitespaces as the parameter to split strings. Have a look at the following example.

Code:

string content = "I am a CSharp Expert";

string[] str = content.Split();

Now in the above case, ‘str’ array will contain words of the string ‘content’ at separate indexes.   
  
This split method can be overloaded as well, for instance. You have a string string content = "I,am,a,CSharp,Expert";  
  
And you want to split this string into words based on the comma, which means that you want that whenever a compiler reaches a comma it split the string and returns the number of words in the form of strings. This can be achieved by overloading the Split method and passing it the delimiting character which is comma in the above case. You can use Split function in following way in your code string[] str = content.Split(',');  
  
**2. Join**  
  
The Join function does exactly opposite of Split function. Join function takes a delimiting character and array of strings as an input and then join the words to form a string. Have a look at the following example.

Code:

string[] str = "I am a CSharp Expert".Split(' ');

string content = string.Join(",", str);

In the above code, basically we are splitting the string on the bases of a space. And then using Join function we are joining these words by inserting a comma between the words.  
  
**3. Concat and '+'**  
  
Concat is a static string method. It is similar to Join in functionality but It doesn’t take any delimiting character as a parameter. Its functionality is as follows.

Code:

string name = string.Concat("I ", "am ", " a", " CSharp", " Expert");

Plus operator “+” is also used to Join strings in a similar manner as illustrated by the following code snippet.

Code:

string name = "I " + "am " + " a" + " CSharp" + " Expert";

Again we will put all these concepts in a working example and will see what the output of this code is. Have a look at Example2.  
  
**Example5**

Code:

using System;

using System.Collections;

using System.Collections.Generic;

using System.Text;

namespace CSharpTutorial

{

class Program

{

static void Main()

{

Console.WriteLine("Using Split functions");

string content = "I am a CSharp Expert";

string[] str = content.Split();

foreach (string word in str)

{

Console.Write(word + "|");

}

Console.WriteLine();

content = "I,am,a,CSharp,Expert";

str = content.Split(',');

foreach (string word in str)

{

Console.Write(word + " ");

}

Console.WriteLine("\n==========\n");

Console.WriteLine("Using Join functions");

str = "I am a CSharp Expert".Split(' ');

content = string.Join(",", str);

Console.WriteLine(content);

Console.WriteLine("==========\n");

Console.WriteLine("Using Concatenate functions");

string name = string.Concat("I ", "am ", " a", " CSharp", " Expert");

Console.WriteLine(name);

name = "I " + "am " + " a" + " CSharp" + " Expert";

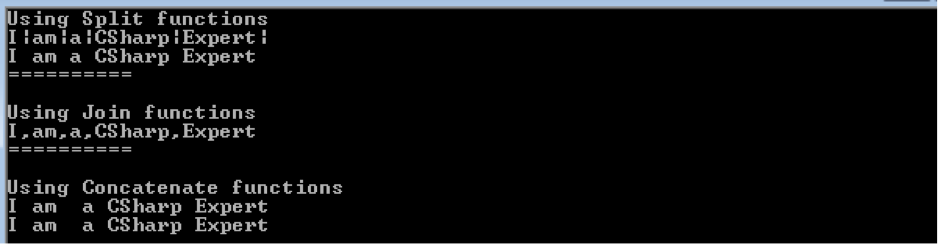
Console.WriteLine(name);

Console.ReadLine();

}

}

}

In the main method we have first split the string and then have displayed each element of the string array using foreach loop. Next we have called Join and Concat methods respectively to display their functionality as discussed earlier. The output of this code would be as follows.  
  
**Output5**  
  


**String Comparison**

String comparison is probably the most frequent task that you will do while programming. Whether it is a signup process or shopping cart application, string comparison lays at the heart of such applications. The simplest example of string comparison can be.

Code:

if(name == “Expert”)

{

Do something

}

Here we are saying that if the string stored in string instance ‘name’ is equal to string literal “Expert”, this statement should evaluate to true otherwise false. This method of comparing two strings is absolutely fine but it has some limitations. First is that this comparison is case sensitive. And secondly, a string might contain characters that do not belong to English language or you application might be running in different cultures, in such scenario the above method is futile. But doesn’t worry, .NET framework provide solution for both of the above problems.  
  
The .NET framework has an ‘Equals’ method that takes two strings to compare along with a third argument that is used to pass some additional information regarding the case and cultural sensitivity. There are three major variations of System.StringComparison enum.

* InvariantCulture - If this enum is passed as a third argument to the Equals method, the culture insensitivity is enforced and the ordering of the string is done according to standard English en-US as the culture. This should be used if you are developing a standard application for different regions.
* CurrentCulture - This enum will take the region in which application is being used while comparing the strings. Different cultures have different alphabets and also the ordering of those alphabets might be different to that used in en-US standard. Therefore, if you are developing separate version of applications for different regions, you should pass this enum.
* Ordinal - Ordinal is the third comparison type. It simply compares and orders the string based on the ASCII/Unicode values. This should be used in scenarios where you want to compare strings based on the ASCII values.

You must be wondering, where is the case insensitivity feature in all of the above mentioned string. The answer is simple; you just have to append the string ‘IgnoreCase’ with these enums to ignore the case. The new values are also actually enums of the String.Comparison type. So, if you want to compare the strings in region independent application and you want to ignore the case as well you can pass the enum ‘’InvariantCultureIgnoreCase”, similarly you can use “CurrentCultureIgnoreCase” and OrdinalIgnoreCase to ignore the case of the string where you want to use current culture and ordinal comparisons respectively.   
  
The concepts explained in String Comparison section have been implemented in Example3. Have a look at Example3 to understand the usage, functionality and variations of the Equals method for string comparison.  
  
**Example6**

Code:

using System;

using System.Collections;

using System.Collections.Generic;

using System.Text;

namespace CSharpTutorial {

class Program {

static void Main() {

string name = "Expert";

if (name == "EXPERT")

Console.WriteLine("The strings are equal");

else

Console.WriteLine("The strings are not equal.");

if (string.Equals(name, "EXPERT",StringComparison.CurrentCultureIgnoreCase))

Console.WriteLine("The strings are equal");

else

Console.WriteLine("The strings are not equal.");

if (string.Equals(name, "EXPERT", StringComparison.InvariantCultureIgnoreCase))

Console.WriteLine("The strings are equal");

else

Console.WriteLine("The strings are not equal.");

if (string.Equals(name, "EXPERT", StringComparison.OrdinalIgnoreCase))

Console.WriteLine("The strings are equal");

else

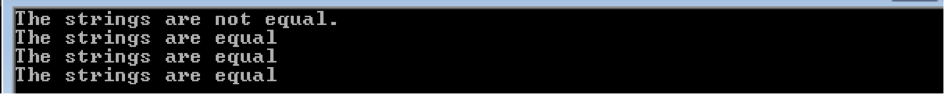
Console.WriteLine("The strings are not equal.");

Console.ReadLine();

}

}

}

The example is simple to understand, we have implemented the Equals method along with its variations to explain the string comparison.   
  
The output of this code is as follows  
  


Do the following exercises

**StringExer1.cs**

String input: hello how is valuestar

Output is: Hello How Is Valuestar

**StringExer2.cs**

String input: hello how is valuestar

Output is: HelloHowIsValuestar

**StringExer3.cs**

String input: hello how is life

Output is:

a: 0 b: 0 c: 0 d: 0 e: 2 f: 1 g: 0 h: 2 i: 2 j: 0 k:0

l:1 m: 0 n: 0 o:2 p: 0 q: 0 r: 0 s: 1 t: 0 u: 0 v:0

w: 1 x: 0 y: 0 z:0

**StringExer4.cs**

String input: hello friend, how is life, I hope you are fine, God BLess

Output is:

hello friend

How is life

I hope you are fine

God Bless

Enter name: maung maung

Enter age: 23

Enter address: Hle Dan, yangon

Enter Phone: 09411361

Enter Email: mgmg@gmail.com

Output is: mang maung | 23| Hle Dan,Yangon| 09411361| mgmg@gmail.com|

**StringExer5.cs**

**Array in C#**

**One Dimensional Array in C#**

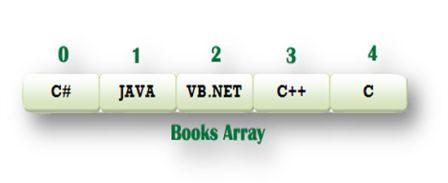
* What is single dimensional array?
* How to declare one dimensional array?
* How to use single dimensional array in C# programming?

The one dimensional array or single dimensional array in C# is the simplest type of array that contains only one row for storing data. It has single set of square bracket (“[]”). To declare single dimensional array in C#, you can write the following code.

string[] Books = new string[5];

The array age is a one dimensional array that contains only 5 elements in a single row.

**Programming Example of One Dimensional Array:**



Example1

using System;

namespace One\_Dimensional\_Array

{

class Program

{

static void Main(string[] args)

{

//Declaring single dimensional array

string[] Books = new string[5];

Books[0] = "C#";

Books[1] = "Java";

Books[2] = "VB.NET";

Books[3] = "C++";

Books[4] = "C";

Console.WriteLine("All the element of Books array is:\n\n");

int i = 0;

//Formatting Output

Console.Write("\t1\t2\t3\t4\t5\n\n\t");

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

{

Console.Write("{0}\t", Books[i]);

}

Console.ReadLine();

}

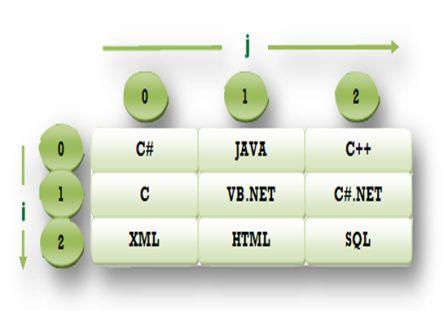
}

}

**Multi Dimensional array in C#**

* What is multi dimensional array?
* How to declare and initialize multi dimensional array in C#?
* How to use multi dimensional array in C# programming?

The multi dimensional array in C# is such type of array that contains more than one row to store data on it. The multi dimensional array is also known as rectangular array in c sharp because it has same length of each row. It can be two dimensional array or three dimensional array or more. It contains more than one coma (,) within single rectangular brackets (“[ , , ,]”). To storing and accessing the elements from multidimensional array, you need to use nested loop in program. The following example will help you to figure out the concept of multidimensional array.



using System;

namespace multi\_dimensional\_array

{

class Program

{

static void Main(string[] args)

{

int i, j;

//Declaring multi dimensional array

string[,] Books = new string[3, 3];

for (i = 0; i < 3; i++)

{

for (j = 0; j < 3; j++)

{

Console.Write("\nEnter Book Name for {0}. Row and {1}. column:\t", i + 1, j + 1);

Books[i, j] = Console.ReadLine();

}

}

Console.WriteLine("\n\n=========================");

Console.WriteLine("All the element of Books array is:\n\n");

//Formatting Output

Console.Write("\t1\t2\t3\n\n");

//outer loop for accessing rows

for (i = 0; i < 3; i++)

{

Console.Write("{0}.\t", i + 1);

//inner or nested loop for accessing column of each row

for (j = 0; j < 3; j++)

{

Console.Write("{0}\t", Books[i, j]);

}

Console.Write("\n");

}

Console.WriteLine("\n\n=========================");

Console.ReadLine();

}

}

}

**Three DimensionalArray**

2D3

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

2D2

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

2D1

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

**using** System;  
  
**namespace** ThreeDArray  
{  
    class Program  
    {  
        **public** static void **Main**(string[] args)  
        {  
              
              
            **int**[,,] num={  
                        {{1,5,2},  
                         {3,7,6},  
                         {14,9,8}      
                          
                        },  
                        {{15,28,49},  
                         {33,17,29},  
                         {102,48,25}  
                              
  
                        },  
                        {{114,64,71},  
                         {32,22,84},  
                         {26,12,58}  
  
                        }  
  
      
                };  
                **for**(**int** i=0;i<3;i++){  
                    **for**(**int** j=0;j<3;j++){  
                        **for**(**int** k=0;k<3;k++){  
                          
                            Console.**Write**("inputs["+i+"]["+j+"]["+k+"]is:"+num[i,j,k]);  
  
  
                        }  
                        Console.**WriteLine**();  
  
  
                    }  
  
                }  
  
  
  
              
              
              
              
            Console.**Write**("Press any key to continue . . . ");  
            Console.**ReadKey**(**true**);  
        }  
    }  
}

**Sorting Algorithms**

**Bubble Sort**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | http://mathbits.com/MathBits/Java/arrays/bubleglass.gif | In the **bubble sort,** as elements are sorted they gradually "bubble" (or rise) to their proper location in the array, like bubbles rising in a glass of soda. The bubble sort repeatedly compares **adjacent elements** of an array. The first and second elements are compared and swapped if out of order.  Then the second and third elements are compared and swapped if out of order.  This sorting process continues until the last two elements of the array are compared and swapped if out of order.  http://mathbits.com/MathBits/Java/arrays/BubbleChart.gif | | When this first pass through the array is complete, the bubble sort returns to elements one and two and starts the process all over again.  So, when does it stop?  **The bubble sort knows that it is finished when it examines the entire array and no "swaps" are needed (thus the list is in proper order).** The bubble sort keeps track of the occurring swaps by the use of a flag.   The table below follows an array of numbers before, during, and after a bubble sort for *descending* order.  A "pass" is defined as one full trip through the array comparing and if necessary, swapping, **adjacent** elements.  Several passes have to be made through the array before it is finally sorted. | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **Array at beginning:** | 84 | 69 | 76 | 86 | 94 | 91 | | **After Pass #1:** | 84 | 76 | 86 | 94 | 91 | 69 | | **After Pass #2:** | 84 | 86 | 94 | 91 | 76 | 69 | | **After Pass #3:** | 86 | 94 | 91 | 84 | 76 | 69 | | **After Pass #4:** | 94 | 91 | 86 | 84 | 76 | 69 | | **After Pass #5 (done):** | 94 | 91 | 86 | 84 | 76 | 69 |   The bubble sort is an easy algorithm to program, but it is slower than many other sorts.  With a bubble sort, it is always necessary to make one final "pass" through the array to check to see that no swaps are made to ensure that the process is finished.  In actuality, the process is finished before this last pass is made.    **Using system;**  **using** System;  **namespace** Bubble\_Sort {     class Program{                   **public** static void **Main**( String[] args)             {                       **int** j;                     **int**[] num={22,33,44,62,16,26};                                                **bool** flag = **true**;   // set flag to true to begin first pass                        **int** temp;   //holding variable                         **while** ( flag )                        {                                   flag= **false**;    //set flag to false awaiting a possible swap                                   **for**( j=0;  j <num.Length -1;  j++ )                                   {                                         **if** ( num[ j ] <num[j+1] )   // change to > for ascending sort                                         {                                                temp = num[ j ];                //swap elements                                                num[ j ] = num[ j+1 ];                                                 num[ j+1 ] = temp;                                                 flag = **true**;              //shows a swap occurred                                             }                              }                        }                                                                         Console.**WriteLine**("The Sorted Output is:");                                                **for**(**int** k=0;k<num.Length;k++){                                                                            Console.**Write**("     {0}",num[k]);                        }                                                Console.**WriteLine**("\n\npress any key....");                                                    Console.**ReadKey**();                                                                } }      }  } |
|  |

**Exchange Sort**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | http://mathbits.com/MathBits/Java/arrays/exchangearrows.gif | The **exchange sort** is similar to its cousin, the bubble sort, in that it compares elements of the array and swaps those that are out of order.  (Some people refer to the "exchange sort" as a "bubble sort".)  The difference between these two sorts is the manner in which they compare the elements. **The exchange sort compares the first element with each following element of the array, making any necessary swaps.**  http://mathbits.com/MathBits/Java/arrays/ExchangeChart.jpg |   When the first pass through the array is complete, the exchange sort then takes the second element and compares it with each following element of the array swapping elements that are out of order.  This sorting process continues until the entire array is ordered.  Let's examine our same table of elements again using an exchange sort for descending order.  Remember, a "pass" is defined as one full trip through the array comparing and if necessary, swapping elements   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **Array at beginning:** | 84 | 69 | 76 | 86 | 94 | 91 | | **After Pass #1:** | 94 | 69 | 76 | 84 | 86 | 91 | | **After Pass #2:** | 94 | 91 | 69 | 76 | 84 | 86 | | **After Pass #3:** | 94 | 91 | 86 | 69 | 76 | 84 | | **After Pass #4:** | 94 | 91 | 86 | 84 | 69 | 76 | | **After Pass #5 (done):** | 94 | 91 | 86 | 84 | 76 | 69 |   The exchange sort, in some situations, is slightly more efficient than the bubble sort.  It is not necessary for the exchange sort to make that final complete pass needed by the bubble sort to determine that it is finished.  **using** System;  **namespace** Exchange\_Sort {     class Program{                   **public** static void **Main**( String[] args)             {                                            **int**[] num={22,33,44,62,16,31,26,87};                                              **int** temp;                            **for**(**int** i=0;i<num.Length-1;i++)                        {                                                                     **for**(**int** j=i+1;  j <num.Length;  j++ )                                   {                                         **if** ( num[ j ] <num[i] )                                            {                                                temp = num[ i ];                                                                num[ i ] = num[ j ];                                                 num[ j ] = temp;                                                                                                        }                              }                        }                                                                      Console.**WriteLine**("The Sorted Output is:");                                               **for**(**int** k=0;k<num.Length;k++){                                                                            Console.**Write**("     {0}",num[k]);                       }                                                                                          Console.**WriteLine**("\n\n\npress any key....");                                                    Console.**ReadKey**();                                                                } }      } |

# Sorting Arrays [C#]

This example shows how to sort arrays in C#. Array can be sorted using static method [**Array.Sort**](http://msdn2.microsoft.com/en-us/library/system.array.sort.aspx) which internally use Quicksort algorithm.

## Sorting array of primitive types

To sort array of primitive types such as **int**, **double** or **string** use method [Array.Sort(Array)](http://msdn2.microsoft.com/en-us/library/6tf1f0bc.aspx) with the array as a paramater. The primitive types implements interface [IComparable](http://msdn2.microsoft.com/en-us/library/system.icomparable.aspx), which is internally used by the Sort method (it calls IComparable.Com­pareTo method). See example how to sort int array:

[C#]

// **sort int array**

int[] intArray = new int[5] { 8, 10, 2, 6, 3 };

**Array.Sort**(intArray);

// write array

foreach (int i in intArray) Console.Write(i + " "); // output: 2 3 6 8 10

or how to sort string array:

[C#]

// **sort string array**

string[] stringArray = new string[5] { "X", "B", "Z", "Y", "A" };

**Array.Sort**(stringArray);

// write array

foreach (string str in stringArray) Console.Write(str + " "); // output: A B X Y Z

## Sorting array of custom type using delegate

To sort your own types or to sort by more sophisticated rules, you can use **delegate to anonymous method**. The generic delegate [Comparison<T>](http://msdn2.microsoft.com/en-us/library/tfakywbh.aspx) is declared as public delegate int Comparison<T> (T x, T y). It points to a method that compares two objects of the same type. It should return less then 0 when X < Y, zero when X = Y and greater then 0 when X > Y. The method (to which the delegate points) can be also an anonymous method (written inline).

Following example demonstrates how to **sort an array of custom type** using the delegate to anonynous comparison method. The custom type in this case is a **class User** with properties Name and Age.

[C#]

// array of custom type

User[] users = new User[3] { new User("Betty", 23), // name, age

new User("Susan", 20),

new User("Lisa", 25) };

[C#]

// **sort array by name**

**Array.Sort**(users, **delegate**(User user1, User user2) **{**

return user1.**Name**.CompareTo(user2.**Name**);

**}**);

// write array (output: **Betty**23 **Lisa**25 **Susan**20)

foreach (User user in users) Console.Write(user.Name + user.Age + " ");

[C#]

// **sort array by age**

**Array.Sort**(users, **delegate**(User user1, User user2) **{**

return user1.**Age**.CompareTo(user2.**Age**); // (user1.Age - user2.Age)

**}**);

// write array (output: Susan**20** Betty**23** Lisa**25**)

foreach (User user in users) Console.Write(user.Name + user.Age + " ");

## Sorting array using IComparable

If you implement [IComparable](http://msdn2.microsoft.com/en-us/library/system.icomparable.aspx) interface in your custom type, you can sort array easily like in the case of primitive types. The Sort method calls internally [IComparable.Com­pareTo](http://msdn2.microsoft.com/en-us/library/system.icomparable.compareto.aspx) method.

[C#]

// custom type

public class User : **IComparable**

{

// ...

// implement IComparable interface

public int **CompareTo**(object obj)

{

if (obj is User) {

return this.Name.CompareTo((obj as User).Name); // compare user names

}

throw new ArgumentException("Object is not a User");

}

}

Use it as you sorted the primitive types in the previous examples.

[C#]

// sort using IComparable implemented by User class

**Array.Sort**(users); // sort array of User objects

String Sorting Example

**using** System;  
  
**namespace** One\_Dimensional\_Array  
{  
    class Program{   
          
        **public** static void **Main**( String[] args)  
            {    
                      
            String []names=**new** string[6]{"aung aung","ko ko gyi","may thu","lwan pyine","bo bo tun","boby soxer"};  
                          
            Array.**Sort**(names);  
                         
            Console.**WriteLine**("The Sorted Output is:");  
                         
                      **foreach**(String name **in** names){  
                         
                         
                           Console.**Write**("     {0}",name);  
                      }  
                         
                      
                      
                       Console.**WriteLine**("\n\n\npress any key....");      
                         
                       Console.**ReadKey**();  
                         
                         
               }  
}  
      
}

Do the Following Exercises

**Do the following Exercises**

MENU

[1] ADD NAME

[2] VIEW NAMES

[3] SORT NAMES

[4] DELETE NAME

[5]SEARCH NAME

[5] EXIT PROGRAM

**ENTER CHOICE:**

MENU

[1] ADD A NUMBER

[2] VIEW NUMBERS

[3] SORT NUMBERS

[4] DELETE NUMBERS

[5] EXIT PROGRAM

**ENTER CHOICE:**

STUDENT RECORDS

[1] ADD RECORD

[2] VIEW RECORDS

[4] DELETE REOCRD

[5]SEARCH RECORD

[5] EXIT PROGRAM

**ENTER CHOICE:**

**C# List**

**List < T > Class**

The Collection classes are a group of classes designed specifically for grouping together objects and performing tasks on them. List class is a collection and defined in the System.Collections.Generic namespace and it provides the methods and properties like other Collection classes such as add, insert, remove, search etc.

The C# List < T > class represents a strongly typed list of objects that can be accessed by index and it supports storing values of a specific type without casting to or from object.

**List < T >**

The parameter T is the type of elements in the list.

**Add Integer values in the List collection**

List<int> iList = new List<int>();

iList.Add(2);

iList.Add(3);

iList.Add(5);

iList.Add(7);

**Add String values in the List**

List<string> colors = new List<string>();

colors.Add("Red");

colors.Add("Blue");

colors.Add("Green");

c# count items in a list

How to find the size of a list?

You can use count property to know the number of items in the List collection or the length of a C# List

colors.Count

c# for each loop in a list

You can retrieve items from List collection by using for loops.

foreach loop

foreach (string color in colors)

{

Console.WriteLine(color);

}

for loop

for (int i = 0; i < colors.Count; i++)

{

Console.WriteLine (colors[i]);

}

c# insert items in a list

You can use insert(index,item) method to insert an in the specified index.

colors.Insert(1, "violet");

In the above code the color "violet" inserted in the index position 1.

**How to sort a C# List**

You can use the sort() method of C# List for ordering items in the List.

colors.Sort();

**How to remove an item from List collection ?**

Remove() can use to remove item from List collection.

colors.Remove("violet");

**How to check if an item exist in the List collection ?**

You can use List.Contains() methods to check an item exists in the List

if (colors.Contains("Blue"))

{

Console.WriteLine ("Blue color exist in the list");

}

**How to copy an Array to a List collection ?**

string[] strArr = new string[3];

strArr[0] = "Red";

strArr[1] = "Blue";

strArr[2] = "Green";

//here to copy array to List

List<string> arrlist = new List<string>(strArr);

**How to Convert List to String in C#**

You can convert a C# List to a string use the following method.

string combindedString = string.Join(",", colors);

The output look like "Red,Blue,Green"

You can replace the seperator to any character instead of ","

**Convert List to Array in C#**

You can convert a C# List to an Array using toArray() method.

string[] arr = colors.ToArray();

**How to empty a list in C#?**

Finally clear method remove all the items from List collection.

arrlist.Clear ();

c# List Vs Arraylist

In C# List is depend by array so the theoretical limit of size would be the limit of the array's capacity. Appending elements is efficient because we are using the free slots at the end, but inserting elements can be slow because all elements in the List after the insertion point have to be shifted to make a free slot. In case of searching, it is is efficient if the BinarySearch method is used on a list that has been sorted, if you use any other search algoritham is inefficient because each item must be individually checked.

**List Example 1**

/\*  
 ValueStar Computer Training Center

Composer: Andrew Tin Mai Zaw  
 Sample Simulation Program for " National Book Store"   
   
 book properties  
 ----------------------  
 book id (string)   
 book title(string)  
 book author(string)  
 price(string)  
 status(string): new/sold  
  
 \*/  
**using** System;  
**using** System.Collections.Generic;  
  
**namespace** ListSample1  
{  
    class Program  
    {  
        **public** static void **Main**(string[] args)  
        {  
              
              
            List<string> books=**new** List<string>();  
              
            string[] info=**new** string[5]{"Book ID:","Book Title:","Book Author:","Book Price:","Selling Status:"};  
              
            string[] str;  
              
            **char** deli=',';  
              
            **int** i=0,k=0,book\_index=0;  
              
            **bool** bookfound=**false**;  
              
            **for**(;;){  
              
            Console.**WriteLine**("\n\n\t\*\*\*\*National Book Store\*\*\*\*\*\n\n");  
            Console.**WriteLine**("1] Add New Book to my Stock");  
            Console.**WriteLine**("2] View Books From the Stock");  
            Console.**WriteLine**("3] Sell a Book");  
            Console.**WriteLine**("4] Search Book");  
            Console.**WriteLine**("5] Exit Program");  
              
            Console.**Write**("Ente Choice");  
            **int** choice=Convert.**ToInt32**(Console.**ReadLine**());  
              
            **switch**(choice){  
              
                    **case** 1: Console.**WriteLine**(" adding a new book to my stock");  
                    Console.**Write**("Enter Book ID:");  
                    String id=Console.**ReadLine**();  
                      
                    Console.**Write**("Enter Book Title:");  
                    String title=Console.**ReadLine**();  
                      
                    Console.**Write**("Enter Book Author:");  
                    String author=Console.**ReadLine**();  
                      
                    Console.**Write**("Enter Book Price:");  
                    String price=Console.**ReadLine**();  
                      
                    String status="new";  
                      
                    String book=string.**Concat**(id,deli,title,deli,author,deli,price,deli,status,deli);  
                      
                    books.**Insert**(i,book);  
                    i++;  
                      
                    Console.**WriteLine**("Successfully added a new book to your stock");  
                      
                    break;  
                      
                    **case** 2: Console.**WriteLine**(" View Books From the Stock");  
                      
                            **int** bookcount=1;  
                      
                    **foreach**(string mybook **in** books){  
                          
                          
                         str=mybook.**Split**(deli);  
                           
                         Console.**WriteLine**(" Book # :{0}",bookcount++);  
                         Console.**WriteLine**("--------------------------------");  
                         **for**(**int** info\_index=0;info\_index<5;info\_index++){  
                               
                                       
                                       
                             Console.**WriteLine**(" {0}  {1}",info[info\_index],str[info\_index]);  
                                      
                                       
                                      
                                      
                                      
                            }  
                           
                          
                          Console.**WriteLine**("--------------------------------");  
                    }  
                      
                    break;  
                      
                    **case** 3:Console.**WriteLine**(" Sell a Book");  
                            Console.**Write**("Enter Book ID to sell:");  
                            String bookid=Console.**ReadLine**();  
                              
                            **foreach**(string sellbook **in** books){  
                                  
                                Console.**WriteLine**("Flag1");  
                                  
                                str=sellbook.**Split**(deli);  
                                **for**(**int** search\_index=0;search\_index<5;search\_index++){  
                                  
                                        bookfound=**false**;  
                                        Console.**WriteLine**("Flag2");  
                                        **if**(string.**Compare**(bookid,str[search\_index],**true**)==0){  
                                      
                                            bookfound=**true**;  
                                            book\_index=k;  
                                            Console.**WriteLine**("Flag3");  
                                        }  
                                          
                                      
                                    **if**(bookfound==**true**) break;  
                                      
                                }  
                                  
                                  
                                **if**(bookfound==**true**) break;  
                                  
                                k++;  
                            }  
                              
                            **if**(bookfound==**true**){  
                              
                                string book\_to\_sell=books[book\_index];  
                                str=book\_to\_sell.**Split**(deli);  
                                str[4]="sold";  
                                string updated\_book=string.**Concat**(str[0],deli,str[1],deli,str[2],deli,str[3],deli,str[4],deli);  
                                  
                                  
                                  
                                books.**Remove**(book\_to\_sell);  
                                books.**Insert**(book\_index,updated\_book);  
                                  
                                  
                                  
                                Console.**WriteLine**("Successfully updated!!! this book is already Sold");  
                                  
                                  
                            }  
                            **else**{  
                                  
                                Console.**WriteLine**("Sorry Book ID not Found ...try again!!!");  
                            }  
                      
                        break;  
                    **case** 4: Console.**WriteLine**("Goodbye");  
                      
                            Console.**Write**("Press any key to exit program . . . ");  
                            Console.**ReadKey**(**true**);  
                           return;  
                              
            }  
              
              
              
            }  
              
              
        }  
    }  
}

**Do the following Exercise using List**

Specifications:

1. Book id - string
2. Title -string
3. Author -string
4. Subject -string
5. Published Date-string
6. Borrow fee:string
7. Fine(penalty) -string

My E-Library

1] Add New Book

2] View Books

3] Search Book ( 1] by Author 2] by Title)

4] Delete Book

5] Borrow Book

6] Exit program

Enter Choice:

**C# ArrayList**

ArrayList is one of the most flexible data structure from [CSharp](http://csharp.net-informations.com) Collections. ArrayList contains a simple list of values. ArrayList implements the IList interface using an array and very easily we can add , insert , delete , view etc. It is very flexible because we can add without any size information , that is it will grow dynamically and also shrink .

c-sharp-arraylist

**Add : Add an Item in an ArrayList**

**Insert : Insert an Item in a specified position in an ArrayList**

**Remove : Remove an Item from ArrayList**

**RemoveAt: remove an item from a specified position**

**Sort : Sort Items in an ArrayList**

**How to add an Item in an ArrayList ?**

**Syntax : ArrayList.add(object)**

**object : The Item to be add the ArrayList**

**ArrayList arr;**

**arr.Add("Item1");**

**How to Insert an Item in an ArrayList ?**

**Syntax : ArrayList.insert(index,object)**

**index : The position of the item in an ArrayList**

**object : The Item to be add the ArrayList**

**ArrayList arr;**

**arr.Insert(3, "Item3");**

**How to remove an item from arrayList ?**

**Syntax : ArrayList.Remove(object)**

**object : The Item to be add the ArrayList**

**arr.Remove("item2")**

**How to remove an item in a specified position from an ArrayList ?**

**Syntax : ArrayList.RemoveAt(index)**

**index : the position of an item to remove from an ArrayList**

**ItemList.RemoveAt(2)**

**How to sort ArrayList ?**

**Syntax : ArrayList.Sort()**

**Array list Example 1**

**/\***

**Study of capacity, count, and sort**

**\*/**

using System;

using System.Collections;

namespace CollectionApplication

{

class Program

{

static void Main(string[] args)

{

ArrayList al = new ArrayList();

Console.WriteLine("Adding some numbers:");

al.Add(45);

al.Add(78);

al.Add(33);

al.Add(56);

al.Add(12);

al.Add(23);

al.Add(9);

Console.WriteLine("Capacity: {0} ", al.Capacity);

Console.WriteLine("Count: {0}", al.Count);

Console.Write("Content: ");

foreach (int i in al)

{

Console.Write(i + " ");

}

Console.WriteLine();

Console.Write("Sorted Content: ");

al.Sort();

foreach (int i in al)

{

Console.Write(i + " ");

}

Console.WriteLine();

Console.ReadKey();

}

}

}

**Array List Example 2**

**/\***

**Study of Contains(), IndexOf() methods and return values**

**\*/**

namespace ArrayList2

{

    class Program

    {

        static void Main(string[] args)

        {

            ArrayList arr = new ArrayList(7);

            arr.Add("Sunday");

            arr.Add("Monday");

            arr.Add("Tuesday");

            arr.Add("Wednesday");

            arr.Add("Thusday");

            arr.Add("Friday");

            arr.Add("Saturday");

            Console.WriteLine("The elements in the arraylist are:");

            foreach (object obj in arr)

            {

                Console.WriteLine(obj);

            }

            Console.WriteLine();

            Console.WriteLine("The element Saturday contain in the arraylist is:" + arr.Contains("Saturday"));

            Console.WriteLine("The element Monday contain in the arraylist is:" + arr.Contains("Monday"));

            Console.WriteLine("The element Tuesday contain in the arraylist is:" + arr.Contains("Tuesday"));

            Console.WriteLine("The element May contain in the arraylist is:" + arr.Contains("May"));

            Console.WriteLine("The element Hello contain in the arraylist is:" + arr.Contains("Hello"));

  Console.WriteLine("The index value of Sunday is:" + arr.IndexOf("Sunday"));

            Console.WriteLine("The index value of Monday is:" + arr.IndexOf("Monday"));

            Console.WriteLine("The index value of Tuesday is:" + arr.IndexOf("Tuesday"));

arr.Remove("Monday");

           arr.Remove("Tuesday");

  foreach (object obj in arr)

            {

                Console.WriteLine(obj);

            }

        }

    }

}

**Array List Example 3**

namespace ArrayList3

{

    class Program

    {

        static void Main(string[] args)

        {

            ArrayList arr = new ArrayList(7);

            arr.Add("Sunday");

            arr.Add("Monday");

            arr.Add("Tuesday");

            arr.Add("Wednesday");

            arr.Add("Thusday");

            arr.Add("Friday");

            arr.Add("Saturday");

            Console.WriteLine("The elements in the arraylist are:");

            foreach (object obj in arr)

            {

                Console.WriteLine(obj);

            }

            arr.Insert(0, "\*\*\*\*\*\*\*\*Names of the days\*\*\*\*\*\*\*");

            arr.Insert(8, "\*\*\*\*\*\*\*\*\*Names of the months\*\*\*\*\*\*\*\*\*\*");

            arr.Insert(9, "Janurary");

            arr.Insert(10, "Feburary");

            arr.Insert(11, "March");

            arr.Insert(12, "April");

            arr.Insert(13, "May");

            Console.WriteLine("The elements in the arraylist are after the insert operation:");

            Console.WriteLine();

            foreach (object obj in arr)

            {

                Console.WriteLine(obj);

            }

        }

    }

}

1] Add New Car

2] View Car

( 1] by Brand

2] by Model

3] Sold Car

4] Remain Items)

3] Car Sale

5] Update Car Record

6] Delete Record

Enter Choice:

Specifications:

1. CarID

2. Brand (Toyota/KIA/Honda/Ford)

3. Model (year)

4. Price

5. Date

6. Frame No

Yuzana caR sale service

**Do the Following Exercise using ArrayList**

E

D

C

B

A

Peek

***List in First out***

Pop

Push

**C# - Stack**

It represents a ***last-in, first out*** collection of object. It is used when you need a last-in, first-out access of items. When you add an item in the list, it is called pushing the item and when you remove it, it is called popping the item.

**Methods and Properties of the Stack Class**

The following table lists some commonly used properties of the Stack class:

Property Description

Count Gets the number of elements contained in the Stack.

The following table lists some of the commonly used methods of the Stack class:

Sr.No. Methods

1 ***public virtual void Clear();***

Removes all elements from the Stack.

2 ***public virtual bool Contains(object obj);***

Determines whether an element is in the Stack.

3 ***public virtual object Peek();***

Returns the object at the top of the Stack without removing it.

4 ***public virtual object Pop();***

Removes and returns the object at the top of the Stack.

5 ***public virtual void Push(object obj);***

Inserts an object at the top of the Stack.

6 ***public virtual object[] ToArray();***

Copies the Stack to a new array.

Example2

The following example demonstrates use of Stack:

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

*// An example string array.*

string[] values = { "Dot", "Net", "Perls" };

*// Copy an array into a Stack.*

var stack = new Stack<string>(values);

*// Display the Stack.*

Console.WriteLine("--- Stack contents ---");

foreach (string value in stack)

{

Console.WriteLine(value);

}

*// See if the stack contains "Perls"*

Console.WriteLine("--- Stack Contains method result ---");

bool contains = stack.Contains("Perls");

Console.WriteLine(contains);

}

}

**Output**

--- Stack contents ---

Perls

Net

Dot

--- Stack Contains method result ---

True

Example2

The following example demonstrates use of Stack:

using System;

using System.Collections;

namespace CollectionsApplication

{

class Program

{

static void Main(string[] args)

{

Stack st = new Stack();

st.Push('A');

st.Push('M');

st.Push('G');

st.Push('W');

Console.WriteLine("Current stack: ");

foreach (char c in st)

{

Console.Write(c + " ");

}

Console.WriteLine();

st.Push('V');

st.Push('H');

Console.WriteLine("The next poppable value in stack: {0}", st.Peek());

Console.WriteLine("Current stack: ");

foreach (char c in st)

{

Console.Write(c + " ");

}

Console.WriteLine();

Console.WriteLine("Removing values ");

st.Pop();

st.Pop();

st.Pop();

Console.WriteLine("Current stack: ");

foreach (char c in st)

{

Console.Write(c + " ");

}

}

}

}

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

Current stack:

W G M A

The next poppable value in stack: H

Current stack:

H V W G M A

Removing values

Current stack:

G M A

/\*

The following Example shows

The usage of Stack

Push(), Pop() and Peek() methods

\*/

using System;  
  
using System.Collections;  
  
  
  
namespace CollectionsApplication  
  
{  
  
    class Program  
  
    {  
  
        static void Main(string[] args)  
  
        {           
  
            Stack st = new Stack();  
  
          while (true)  
  
            {  
  
                Console.Clear();  
  
                Console.WriteLine("\n            Stack MENU");  
  
                Console.WriteLine("1. Add an element");  
  
                Console.WriteLine("2. See the Top element.");  
  
                Console.WriteLine("3. Remove top element.");  
  
                Console.WriteLine("4. Display stack elements.");  
  
                Console.WriteLine("5. Exit");  
  
                Console.Write("Select your choice: ");  
  
                int choice = Convert.ToInt32(Console.ReadLine());  
  
                switch (choice)  
  
                {  
  
                    case 1:  
  
                        Console.WriteLine("Enter an Element : ");  
  
                        st.Push(Console.ReadLine());  
  
                        break;  
  
  
                    case 2: Console.WriteLine("Top element is: {0}", st.Peek());  
  
                        break;  
  
   
  
                    case 3: Console.WriteLine("Element removed: {0}", st.Pop());  
  
                        break;  
  
                         case 4:   Console.WriteLine("Display Stack Elements ");  
  
                        foreach(Object obj in st){  
                            Console.WriteLine(" {0}  ",obj);  
                        }  
                        break;  
  
                     case 5: System.Environment.Exit(1);  
  
                        break;  
  
                }  
  
                Console.ReadKey();  
  
            }  
  
        }  
  
          
    }  
    }

Do the following Exercises

Exercise 1

**Student record:**

1. Name
2. Entry\_Date
3. Age
4. Course title
5. Phone
6. Address
7. Course\_Fee

Northern City Computer Training Center

Yangon, Myanmar

-----------------------------------------------------------

1] Add new Student

2] View Students

3] Search Student

4] Delete Student

5]exit program

-----------------------------------------------------------

Enter Choice:\_

Enqueue

E

D

C

B

A

**C# - Queue Class**

***First-In, First Out***

Dequeue

It represents a ***first-in, first out*** collection of object. It is used when you need a first-in, first-out access of items. When you add an item in the list, it is called enqueue, and when you remove an item, it is called deque.

Methods and Properties of the Queue Class

The following table lists some of the commonly used properties of the Queue class:

Property Description

*Count Gets the number of elements contained in the Queue.*

The following table lists some of the commonly used methods of the Queue class:

Sr.No. Methods

1 public virtual void Clear();

*Removes all elements from the Queue.*

2 public virtual bool Contains(object obj);

*Determines whether an element is in the Queue.*

3 public virtual object Dequeue();

*Removes and returns the object at the beginning of the Queue*

*.*

4 public virtual void Enqueue(object obj);

Adds an object to the end of the Queue.

5 public virtual object[] ToArray();

*Copies the Queue to a new array.*

6 public virtual void TrimToSize();

Sets the capacity to the actual number of elements in the Queue.

**Example 1**

The following example demonstrates use of Queue:

using System;

using System.Collections;

namespace CollectionsApplication

{

class Program

{

static void Main(string[] args)

{

Queue q = new Queue();

q.Enqueue('A');

q.Enqueue('M');

q.Enqueue('G');

q.Enqueue('W');

Console.WriteLine("Current queue: ");

foreach (char c in q) Console.Write(c + " ");

Console.WriteLine();

q.Enqueue('V');

q.Enqueue('H');

Console.WriteLine("Current queue: ");

foreach (char c in q) Console.Write(c + " ");

Console.WriteLine();

Console.WriteLine("Removing some values ");

char ch = (char)q.Dequeue();

Console.WriteLine("The removed value: {0}", ch);

ch = (char)q.Dequeue();

Console.WriteLine("The removed value: {0}", ch);

Console.ReadKey();

}

}

}

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

Current queue:

A M G W

Current queue:

A M G W V H

Removing values

The removed value: A

The removed value: M

**Example 2**

C# program that copies Queue

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

// New Queue of integers.

Queue<int> queue = new Queue<int>();

queue.Enqueue(5);

queue.Enqueue(10);

queue.Enqueue(15);

queue.Enqueue(20);

// Create new array with Length equal to Queue's element count.

int[] array = new int[queue.Count];

// Copy the Queue to the int array.

queue.CopyTo(array, 0);

// Loop through and display int[] in order.

Console.WriteLine("Array:");

for (int i = 0; i < array.Length; i++)

{

Console.WriteLine(array[i]);

}

// Loop through int array in reverse order.

Console.WriteLine("Array reverse order:");

for (int i = array.Length - 1; i >= 0; i--)

{

Console.WriteLine(array[i]);

}

}

}

Output

Array:

5

10

15

20

Array reverse order:

20

15

10

5

DO the Following Exercise

Exercise 2

City Express

Hle Dan, Yangon, Myanmar

-----------------------------------------------------------

1] Add new product

2] View Product

3] Search Product

4] Delete Product

5] Update Product

6] Exit program

-----------------------------------------------------------

Enter Choice:\_

**Product record:**

1. Name
2. Price
3. Manufactured\_Date
4. Expired\_Date
5. Items

**C# HashTable**

Hashtable in C# represents a collection of key/value pairs which maps keys to value. Any non-null object can be used as a key but a value can. We can retrieve items from hashTable to provide the key . Both keys and values are Objects.

The commonly used functions in Hashtable are :

**Add : To add a pair of value in HashTable**

**ContainsKey : Check if a specified key exist or not**

**ContainsValue : Check the specified Value exist in HashTable**

**Remove : Remove the specified Key and corresponding Value**

**Add : To add a pair of value in HashTable**

**Syntax : HashTable.Add(Key,Value)**

**Key : The Key value**

**Value : The value of corresponding key**

**Hashtable ht;**

**ht.Add("1", "Sunday");**

**ContainsKey : Check if a specified key exist or not**

**Synatx : bool HashTable.ContainsKey(key)**

**Key : The Key value for search in HahTable**

**Returns : return true if item exist else false**

**ht.Contains("1");**

**ContainsValue : Check the specified Value exist in HashTable**

**Synatx : bool HashTable.ContainsValue(Value)**

**Value : Search the specified Value in HashTable**

**Returns : return true if item exist else false**

**ht.ContainsValue("Sunday")**

**Remove : Remove the specified Key and corresponding Value**

**Syntax : HashTable.Remove(Key)**

**Key : The key of the element to remove**

**ht.Remove("1");**

**Methods and Properties of the Hashtable Class**

The following table lists some of the commonly used properties of the Hashtable class:

Property Description

**Count**  Gets the number of key-and-value pairs contained in the Hashtable.

**IsFixedSize**  Gets a value indicating whether the Hashtable has a fixed size.

**IsReadOnly**  Gets a value indicating whether the Hashtable is read-only.

**Item**  Gets or sets the value associated with the specified key.

**Keys**  Gets an ICollection containing the keys in the Hashtable.

**Values**  Gets an ICollection containing the values in the Hashtable.

**The following table lists some of the commonly used methods of the Hashtable class:**

Sr.No. Method

1 public virtual void Add(object key, object value);

Adds an element with the specified key and value into the Hashtable.

2 public virtual void Clear();

Removes all elements from the Hashtable.

3 public virtual bool ContainsKey(object key);

Determines whether the Hashtable contains a specific key.

4 public virtual bool ContainsValue(object value);

Determines whether the Hashtable contains a specific value.

5 public virtual void Remove(object key);

Removes the element with the specified key from the Hashtable.

**Hash Table Example 1**

C# program that loops over Keys, Values

**using** System;  
**using** System.Collections;  
  
**namespace** HashTable  
{  
    class Program  
    {  
        **public** static void **Main**(string[] args)  
        {  
    Hashtable hashtable = **new** **Hashtable**();  
    hashtable.**Add**(400, "Blaze");  
    hashtable.**Add**(500, "Fiery");  
    hashtable.**Add**(600, "Fire");  
    hashtable.**Add**(800, "Immolate");  
  
    // Display the keys.  
    **foreach** (**int** key **in** hashtable.Keys)  
    {  
        Console.**WriteLine**(key);  
    }  
  
    // Display the values.  
    **foreach** (string **value** **in** hashtable.Values)  
    {  
        Console.**WriteLine**(**value**);  
    }  
  
    // Put keys in an ArrayList.  
    ArrayList arrayList = **new** **ArrayList**(hashtable.Keys);  
    **foreach** (**int** key **in** arrayList)  
    {  
        Console.**WriteLine**(key);  
    }  
  
              
            Console.**Write**("Press any key to continue . . . ");  
            Console.**ReadKey**(**true**);  
        }  
    }  
}

**Hash Table Example 2**

**using** System;  
**using** System.Collections;  
  
**namespace** HashTable  
{  
    class Program  
    {  
        **public** static void **Main**(string[] args)  
        {  
    Hashtable ht = **new** **Hashtable**();  
           
         ht.**Add**("001", "Zara Ali");  
         ht.**Add**("002", "Abida Rehman");  
         ht.**Add**("003", "Joe Holzner");  
         ht.**Add**("004", "Mausam Benazir Nur");  
         ht.**Add**("005", "M. Amlan");  
         ht.**Add**("006", "M. Arif");  
         ht.**Add**("007", "Ritesh Saikia");  
           
         **if** (ht.**ContainsValue**("Nuha Ali"))  
         {  
            Console.**WriteLine**("This student name is already in the list");  
         }  
         **else**  
         {  
            ht.**Add**("008", "Nuha Ali");  
         }  
           
         // Get a collection of the keys.  
         ICollection key = ht.Keys;  
           
         **foreach** (string k **in** key)  
         {  
            Console.**WriteLine**(k + ": " + ht[k]);  
         }  
           
         Console.**ReadKey**();  
  
        }  
    }  
}When the above code is compiled and executed, it produces the following result:

001: Zara Ali

002: Abida Rehman

003: Joe Holzner

004: Mausam Benazir Nur

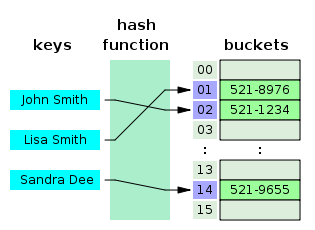
005: M. Amlan

006: M. Arif

007: Ritesh Saikia

008: Nuha Ali

**Do the following Exercise**

****

\*\*\*\*\*\*\*\*\*\*\* My Phone Book \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1] Add new Entry

2] View Entries

3] Update Entry

4] Search Entry

5] Delete Entry

6] Exit program

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter Choice:

Hash table:

Key= name

Value=Phone number

# C# Dictionary

A Dictionary class is a data structure that represents a collection of keys and values pair of data. The key is identical in a key-value pair and it can have at most one value in the dictionary, but a value can be associated with many different keys.

This class is defined in the System.Collections.Generic namespace, so you should import or using System.Collections.Generic namespace.

using System.Collections.Generic

Syntax:

c# dictionary sample

Parameters :

TKey - The type of the keys in the dictionary.

TValue - The type of the values in the dictionary.

e.g.

Dictionary<string, string>

Dictionary<string, int>

Methods in the class **Dictionary<K, V>**:

-     **void Add(K, V)** adds a new pair (key and a value) to the hash-table. Throws an exception in the case that the key exists. This operation is extremely fast.

-     **bool TryGetValue(K, out V)** returns an element of type V via the **out** parameter for the given key or **null**, if there is no such key. The result of this operation will be **true** if such an element is found. The operation is very fast, because the algorithm for searching an element by key in the hash-table is with complexity about O(1)

-     **bool Remove(K)** removes the element with this key. This operation works very fast.

-     **void Clear()** removes all the elements from the dictionary.

-     **bool ContainsKey(K)** check if there is an ordered pair with this key in the dictionary. This operation works extremely fast.

-     **bool ContainsValue(V)** checks if there is one or more ordered pairs with this value. This operation is slow because it checks every element of the hash-table (like searching in a list).

-     **int Count** returns the number of ordered pairs within the dictionary.

-     Other operations – extracting all the keys, values or ordered pairs into a structure that could be iterated through using a loop.

**Dictionary Example 1**

*WordCountingWithSortedDictionary.cs*

using System;

using System.Collections.Generic;

namespace DictionarySample

{

class Program

{

public static void Main(string[] args)

{

string text ="Mary had a little lamblittle Lamb, little Lamb, Mary had a Little lamb,whose fleece were white as snow.";

string[] tokens = text.Split(' ', '.', ',', '-', '?', '!');

Dictionary<string, int> words = new Dictionary<string, int>();

foreach (string word in tokens)

{

if (!string.IsNullOrEmpty(word.Trim()))

{

int count;

if (!words.TryGetValue(word, out count))

{

count = 0;

}

words[word] = count + 1;

}

}

foreach (var wordEntry in words)

{

Console.WriteLine("Word '{0}' occurs {1} time(s) in the text", wordEntry.Key, wordEntry.Value);

}

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

Console.ReadKey(true);

}

}

}

**The output from executing this code is the following:**

Word 'a' occurs 2 time(s) in the text

Word 'as' occurs 1 time(s) in the text

Word 'fleece' occurs 1 time(s) in the text

Word 'had' occurs 2 time(s) in the text

Word 'lamb' occurs 2 time(s) in the text

Word 'Lamb' occurs 2 time(s) in the text

Word 'little' occurs 3 time(s) in the text

Word 'Little' occurs 1 time(s) in the text

Word 'mary' occurs 2 time(s) in the text

Word 'snow' occurs 1 time(s) in the text

Word 'was' occurs 1 time(s) in the text

Word 'white' occurs 1 time(s) in the text

Word 'whose' occurs 1 time(s) in the text

**Dictionary Example 2**

using System;

using System.Collections.Generic;

namespace DictionarySample

{

class Program

{

public static void Main(string[] args)

{

IDictionary<string, double> studentMarks = new Dictionary<string, double>(6);

studentMarks["Alan"] = 3.00;

studentMarks["Helen"] = 4.50;

studentMarks["Tom"] = 5.50;

studentMarks["James"] = 3.50;

studentMarks["Mary"] = 4.00;

studentMarks["Nerdy"] = 6.00;

double marysMark = studentMarks["Mary"];

Console.WriteLine("Mary's mark: {0:0.00}", marysMark);

studentMarks.Remove("Mary");

Console.WriteLine("Mary's mark removed.");

Console.WriteLine("Is Mary in the dictionary: {0}", studentMarks.ContainsKey("Mary") ? "Yes!": "No!");

Console.WriteLine("Nerdy's mark is {0:0.00}.", studentMarks["Nerdy"]);

studentMarks["Nerdy"] = 3.25;

Console.WriteLine( "But we all know he deserves no more than {0:0.00}.", studentMarks["Nerdy"]);

double annasMark;

bool findAnna = studentMarks.TryGetValue("Anna",out annasMark);

Console.WriteLine( "Is Anna's mark in the dictionary? {0}", findAnna ? "Yes!": "No!");

studentMarks["Anna"] = 6.00;

findAnna = studentMarks.TryGetValue("Anna", out annasMark);

Console.WriteLine( "Let's try again: {0}. Anna's mark is {1}", findAnna ? "Yes!" : "No!", annasMark);

Console.WriteLine("Students and marks:");

foreach (KeyValuePair<string, double> studentMark in studentMarks)

{

Console.WriteLine("{0} has {1:0.00}",studentMark.Key, studentMark.Value);

}

Console.WriteLine( "There are {0} students in the dictionary",studentMarks.Count);

studentMarks.Clear();

Console.WriteLine("Students dictionary cleared.");

Console.WriteLine("Is dictionary empty: {0}",

studentMarks.Count == 0);

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

Console.ReadKey(true);

}

}

}

**The output of the program execution will be:**

Mary's mark: 4.00

Mary's mark removed.

Is Mary in the dictionary: No!

Nerdy's mark is 6.00.

But we all know he deserves no more than 3.25.

Is Anna's mark in the dictionary? No!

Let's try again: Yes!. Anna's mark is 6

Students and marks:

Alan has 3.00

Helen has 4.50

Tom has 5.50

James has 3.50

Anna has 6.00

Nerdy has 3.25

There are 6 students in the dictionary

Students dictionary cleared.

Is dictionary empty: True

**Do the following Exercise**

Dictionary:

Key=word

Value=meaning

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*My Dictionary\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1] Add new vocabulary

2] Search Vocabularies

3] Update Vocabulary

4] Delete Vocabulary

5]Exit Program

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Enter Choice:

# C# - Methods

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

**Defining Methods in C#**

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**: This contains the set of instructions needed to complete the required activity.

**Example:**

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;

}

...

}

**Calling Methods in C#**

You can call a method using the name of the method. The following example illustrates this:

using System;

namespace CalculatorApplication

{

class NumberManipulator

{

public int FindMax(int num1, int num2)

{

/\* local variable declaration \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

static void Main(string[] args)

{

/\* local variable definition \*/

int a = 100;

int b = 200;

int ret;

NumberManipulator n = new NumberManipulator();

//calling the FindMax method

ret = n.FindMax(a, b);

Console.WriteLine("Max value is : {0}", ret );

Console.ReadLine();

}

}

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

Max value is : 200

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*.

using System;

namespace CalculatorApplication

{

class NumberManipulator

{

public int FindMax(int num1, int num2)

{

/\* local variable declaration \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

}

class Test

{

static void Main(string[] args)

{

/\* local variable definition \*/

int a = 100;

int b = 200;

int ret;

NumberManipulator n = new NumberManipulator();

//calling the FindMax method

ret = n.FindMax(a, b);

Console.WriteLine("Max value is : {0}", ret );

Console.ReadLine();

}

}

}

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

Max value is : 200

**Recursive Method Call**

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:

using System;

namespace CalculatorApplication

{

class NumberManipulator

{

public int factorial(int num)

{

/\* local variable declaration \*/

int result;

if (num == 1)

{

return 1;

}

else

{

result = factorial(num - 1) \* num;

return result;

}

}

static void Main(string[] args)

{

NumberManipulator n = new NumberManipulator();

//calling the factorial method

Console.WriteLine("Factorial of 6 is : {0}", n.factorial(6));

Console.WriteLine("Factorial of 7 is : {0}", n.factorial(7));

Console.WriteLine("Factorial of 8 is : {0}", n.factorial(8));

Console.ReadLine();

}

}

}

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

Factorial of 6 is: 720

Factorial of 7 is: 5040

Factorial of 8 is: 40320

# C# - Structures

n C#, a structure is a value type data type. 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

## 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 would declare the Book structure:

struct Books

{

public string title;

public string author;

public string subject;

public int book\_id;

};

The following program shows the use of the structure:

using System;

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();

}

}

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

Book 1 title : C Programming

Book 1 author : Nuha Ali

Book 1 subject : C Programming Tutorial

Book 1 book\_id : 6495407

Book 2 title : Telecom Billing

Book 2 author : Zara Ali

Book 2 subject : Telecom Billing Tutorial

Book 2 book\_id : 6495700

Structure MINI PROJECT

View Disks by Category

1] Action Movies

2] Drama Movies

3] Comedy Movies

4] Scary Movies

Enter Choice:

View Disks Sub Menu

1] View Disk By Type

2] View Disk by Category

Enter Choice:

View Disks by Type

1] CD

2] DVD

Enter Choice:

Specifications:

1. Disc id
2. Title
3. Type (CD/DVD)
4. Category (Action/Drama/Comedy/Scary)
5. Rental price
6. Rental Date
7. Return Date
8. Penalty

Main MENU

Movie Disc Rental Service

1] Add New Disk Record

2] View Disks ( 1] by type 2]by Category)

3] Search Disk ( 1] by ID 2] by Title)

4] Delete Record

5] Update Record

6] Exit program

Enter Choice:

TIC TAC TOE GAME

Tic Tac Toe Game 2

Player 1

Player 2

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |

## Features of C# Structures

You have already used a simple structure named Books. 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 can't 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 will remain unassigned and the object cannot be used until all the fields are initialized.

## Class vs 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

In the light of the above discussions, let us rewrite the previous example:

using System;

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();

}

}

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

Title : C Programming

Author : Nuha Ali

Subject : C Programming Tutorial

Book\_id : 6495407

Title : Telecom Billing

Author : Zara Ali

Subject : Telecom Billing Tutorial

Book\_id : 6495700

# C# - Classes

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object. Objects are instances of a class. The methods and variables that constitute a class are called members of the class.

**Class Definition**

A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. Following is the general form of a class definition:

<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

}

}

Please note that,

* Access specifiers specify the access rules for the members as well as the class itself, if not mentioned then the default access specifier for a class type is **internal**. Default access for the members is **private**.
* Data type specifies the type of variable, and return type specifies the data type of the data, the method returns, if any.
* To access the class members, you will use the dot (.) operator.
* The dot operator links the name of an object with the name of a member.

The following example illustrates the concepts discussed so far:

using System;

namespace BoxApplication

{

class Box

{

public double length; // Length of a box

public double breadth; // Breadth of a box

public double height; // Height of a box

}

class Boxtester

{

static void Main(string[] args)

{

Box Box1 = new Box(); // Declare Box1 of type Box

Box Box2 = new Box(); // Declare Box2 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.height = 5.0;

Box1.length = 6.0;

Box1.breadth = 7.0;

// box 2 specification

Box2.height = 10.0;

Box2.length = 12.0;

Box2.breadth = 13.0;

// volume of box 1

volume = Box1.height \* Box1.length \* Box1.breadth;

Console.WriteLine("Volume of Box1 : {0}", volume);

// volume of box 2

volume = Box2.height \* Box2.length \* Box2.breadth;

Console.WriteLine("Volume of Box2 : {0}", volume);

Console.ReadKey();

}

}

}

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

Volume of Box1 : 210

Volume of Box2 : 1560

**Member Functions and Encapsulation**

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Member variables are attributes of an object (from design perspective) and they are kept private to implement encapsulation. These variables can only be accessed using the public member functions.

Let us put above concepts to set and get the value of different class members in a class:

using System;

namespace BoxApplication

{

class Box

{

private double length; // Length of a box

private double breadth; // Breadth of a box

private double height; // Height of a box

public void setLength( double len )

{

length = len;

}

public void setBreadth( double bre )

{

breadth = bre;

}

public void setHeight( double hei )

{

height = hei;

}

public double getVolume()

{

return length \* breadth \* height;

}

}

class Boxtester

{

static void Main(string[] args)

{

Box Box1 = new Box(); // Declare Box1 of type Box

Box Box2 = new Box();

double volume;

// Declare Box2 of type Box

// box 1 specification

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

// box 2 specification

Box2.setLength(12.0);

Box2.setBreadth(13.0);

Box2.setHeight(10.0);

// volume of box 1

volume = Box1.getVolume();

Console.WriteLine("Volume of Box1 : {0}" ,volume);

// volume of box 2

volume = Box2.getVolume();

Console.WriteLine("Volume of Box2 : {0}", volume);

Console.ReadKey();

}

}

}

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

Volume of Box1 : 210

Volume of Box2 : 1560

**Constructors in C#**

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

A constructor will have exact same name as the class and it does not have any return type. Following example explains the concept of constructor:

using System;

namespace LineApplication

{

class Line

{

private double length; // Length of a line

public Line()

{

Console.WriteLine("Object is being created");

}

public void setLength( double len )

{

length = len;

}

public double getLength()

{

return length;

}

static void Main(string[] args)

{

Line line = new Line();

// set line length

line.setLength(6.0);

Console.WriteLine("Length of line : {0}", line.getLength());

Console.ReadKey();

}

}

}

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

Object is being created

Length of line : 6

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 as shown in the following example:

using System;

namespace LineApplication

{

class Line

{

private double length; // Length of a line

public Line(double len) //Parameterized constructor

{

Console.WriteLine("Object is being created, length = {0}", len);

length = len;

}

public void setLength( double len )

{

length = len;

}

public double getLength()

{

return length;

}

static void Main(string[] args)

{

Line line = new Line(10.0);

Console.WriteLine("Length of line : {0}", line.getLength());

// set line length

line.setLength(6.0);

Console.WriteLine("Length of line : {0}", line.getLength());

Console.ReadKey();

}

}

}

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

Object is being created, length = 10

Length of line : 10

Length of line : 6

Do the Following Exercises

Do the following Exercise

Myanmar SBT Car sale Center

1] add new car

2] view cars

3] update car inforamation

4] Delete car

Enter Choice:

Classes: Car, SBT\_Center

Car Specifications:

ID

Brand

Color

Model

Windows

Price

**C# - File I/O**

A **file** is a collection of data stored in a disk with a specific name and a directory path. When a file is opened for reading or writing, it becomes a **stream**.

The stream is basically the sequence of bytes passing through the communication path. There are two main streams: the **input stream** and the **output stream**. The **input stream** is used for reading data from file (read operation) and the **output stream** is used for writing into the file (write operation).

**C# I/O Classes**

The System.IO namespace has various class that are used for performing various operation with files, like creating and deleting files, reading from or writing to a file, closing a file etc.

The following table shows some commonly used non-abstract classes in the System.IO namespace:

|  |  |
| --- | --- |
| **I/O Class** | **Description** |
| BinaryReader | Reads primitive data from a binary stream. |
| BinaryWriter | Writes primitive data in binary format. |
| BufferedStream | A temporary storage for a stream of bytes. |
| Directory | Helps in manipulating a directory structure. |
| DirectoryInfo | Used for performing operations on directories. |
| DriveInfo | Provides information for the drives. |
| File | Helps in manipulating files. |
| FileInfo | Used for performing operations on files. |
| FileStream | Used to read from and write to any location in a file. |
| MemoryStream | Used for random access to streamed data stored in memory. |
| Path | Performs operations on path information. |
| StreamReader | Used for reading characters from a byte stream. |
| StreamWriter | Is used for writing characters to a stream. |
| StringReader | Is used for reading from a string buffer. |
| StringWriter | Is used for writing into a string buffer. |

**The FileStream Class**

The **FileStream** class in the System.IO namespace helps in reading from, writing to and closing files. This class derives from the abstract class Stream.

You need to create a **FileStream** object to create a new file or open an existing file. The syntax for creating a **FileStream** object is as follows:

FileStream <object\_name> = new FileStream( <file\_name>,

<FileMode Enumerator>, <FileAccess Enumerator>, <FileShare Enumerator>);

For example, for creating a FileStream object **F** for reading a file named **sample.txt**:

FileStream F = new FileStream("sample.txt", FileMode.Open, FileAccess.Read, FileShare.Read);

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| FileMode | The **FileMode** enumerator defines various methods for opening files. The members of the FileMode enumerator are:   * **Append**: It opens an existing file and puts cursor at the end of file, or creates the file, if the file does not exist. * **Create**: It creates a new file. * **CreateNew**: It specifies to the operating system, that it should create a new file. * **Open**: It opens an existing file. * **OpenOrCreate**: It specifies to the operating system that it should open a file if it exists, otherwise it should create a new file. * **Truncate**: It opens an existing file and truncates its size to zero bytes. |
| FileAccess | **FileAccess** enumerators have members: **Read**, **ReadWrite** and **Write**. |
| FileShare | **FileShare** enumerators have the following members:   * **Inheritable**: It allows a file handle to pass inheritance to the child processes * **None**: It declines sharing of the current file * **Read**: It allows opening the file for reading * **ReadWrite**: It allows opening the file for reading and writing * **Write**: It allows opening the file for writing |

**Example:**

The following program demonstrates use of the **FileStream** class:

using System;

using System.IO;

namespace FileIOApplication

{

class Program

{

static void Main(string[] args)

{

FileStream F = new FileStream("test.dat",

FileMode.OpenOrCreate, FileAccess.ReadWrite);

for (int i = 1; i <= 20; i++)

{

F.WriteByte((byte)i);

}

F.Position = 0;

for (int i = 0; i <= 20; i++)

{

Console.Write(F.ReadByte() + " ");

}

F.Close();

Console.ReadKey();

}

}

}

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

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 -1

**Advanced File Operations in C#**

The preceding example provides simple file operations in C#. However, to utilize the immense powers of C# System.IO classes, you need to know the commonly used properties and methods of these classes.

We will discuss these classes and the operations they perform, in the following sections. Please click the links provided to get to the individual sections:

|  |
| --- |
| **Topic and Description** |
| [Reading from and Writing into Text files](http://www.tutorialspoint.com/csharp/csharp_text_files.htm) It involves reading from and writing into text files. The **StreamReader** and **StreamWriter** class helps to accomplish it. |
| [Reading from and Writing into Binary files](http://www.tutorialspoint.com/csharp/csharp_binary_files.htm) It involves reading from and writing into binary files. The **BinaryReader** and **BinaryWriter** class helps to accomplish this. |
| [Manipulating the Windows file system](http://www.tutorialspoint.com/csharp/csharp_windows_file_system.htm) It gives a C# programamer the ability to browse and locate Windows files and directories. |

**C# - Delegates**

C# delegates are similar to pointers to functions, in C or C++. A **delegate** is a reference type variable that holds the reference to a method. The reference can be changed at runtime.

Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the **System.Delegate** class.

**Declaring Delegates**

Delegate declaration determines the methods that can be referenced by the delegate. A delegate can refer to a method, which have the same signature as that of the delegate.

For example, consider a delegate:

public delegate int MyDelegate (string s);

The preceding delegate can be used to reference any method that has a single *string* parameter and returns an *int* type variable.

Syntax for delegate declaration is:

delegate <return type> <delegate-name> <parameter list>

**Instantiating Delegates**

Once a delegate type has been declared, a delegate object must be created with the **new** keyword and be associated with a particular method. When creating a delegate, the argument passed to the **new** expression is written like a method call, but without the arguments to the method. For example:

public delegate void printString(string s);

...

printString ps1 = new printString(WriteToScreen);

printString ps2 = new printString(WriteToFile);

Following example demonstrates declaration, instantiation and use of a delegate that can be used to reference methods that take an integer parameter and returns an integer value.

using System;

delegate int NumberChanger(int n);

namespace DelegateAppl

{

class TestDelegate

{

static int num = 10;

public static int AddNum(int p)

{

num += p;

return num;

}

public static int MultNum(int q)

{

num \*= q;

return num;

}

public static int getNum()

{

return num;

}

static void Main(string[] args)

{

//create delegate instances

NumberChanger nc1 = new NumberChanger(AddNum);

NumberChanger nc2 = new NumberChanger(MultNum);

//calling the methods using the delegate objects

nc1(25);

Console.WriteLine("Value of Num: {0}", getNum());

nc2(5);

Console.WriteLine("Value of Num: {0}", getNum());

Console.ReadKey();

}

}

}

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

Value of Num: 35

Value of Num: 175

**Multicasting of a Delegate**

Delegate objects can be composed using the "+" operator. A composed delegate calls the two delegates it was composed from. Only delegates of the same type can be composed. The "-" operator can be used to remove a component delegate from a composed delegate.

Using this useful property of delegates you can create an invocation list of methods that will be called when a delegate is invoked. This is called **multicasting** of a delegate. The following program demonstrates multicasting of a delegate:

using System;

delegate int NumberChanger(int n);

namespace DelegateAppl

{

class TestDelegate

{

static int num = 10;

public static int AddNum(int p)

{

num += p;

return num;

}

public static int MultNum(int q)

{

num \*= q;

return num;

}

public static int getNum()

{

return num;

}

static void Main(string[] args)

{

//create delegate instances

NumberChanger nc;

NumberChanger nc1 = new NumberChanger(AddNum);

NumberChanger nc2 = new NumberChanger(MultNum);

nc = nc1;

nc += nc2;

//calling multicast

nc(5);

Console.WriteLine("Value of Num: {0}", getNum());

Console.ReadKey();

}

}

}

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

Value of Num: 75

**Use of Delegate**

The following example demonstrates the use of delegate. The delegate *printString* can be used to reference methods that take a string as input and return nothing.

We use this delegate to call two methods, the first prints the string to the console, and the second one prints it to a file:

using System;

using System.IO;

namespace DelegateAppl

{

class PrintString

{

static FileStream fs;

static StreamWriter sw;

// delegate declaration

public delegate void printString(string s);

// this method prints to the console

public static void WriteToScreen(string str)

{

Console.WriteLine("The String is: {0}", str);

}

//this method prints to a file

public static void WriteToFile(string s)

{

fs = new FileStream("c:\\message.txt",

FileMode.Append, FileAccess.Write);

sw = new StreamWriter(fs);

sw.WriteLine(s);

sw.Flush();

sw.Close();

fs.Close();

}

// this method takes the delegate as parameter and uses it to

// call the methods as required

public static void sendString(printString ps)

{

ps("Hello World");

}

static void Main(string[] args)

{

printString ps1 = new printString(WriteToScreen);

printString ps2 = new printString(WriteToFile);

sendString(ps1);

sendString(ps2);

Console.ReadKey();

}

}

}

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

The String is: Hello World

## Example 2:

This example provides a simple application for troubleshooting for a hot water boiler system. When the maintenance engineer inspects the boiler, the boiler temperature and pressure is automatically recorded into a log file along with the remarks of the maintenance engineer.

using System;

using System.IO;

namespace BoilerEventAppl

{

// boiler class

class Boiler

{

private int temp;

private int pressure;

public Boiler(int t, int p)

{

temp = t;

pressure = p;

}

public int getTemp()

{

return temp;

}

public int getPressure()

{

return pressure;

}

}

// event publisher

class DelegateBoilerEvent

{

public delegate void BoilerLogHandler(string status);

//Defining event based on the above delegate

public event BoilerLogHandler BoilerEventLog;

public void LogProcess()

{

string remarks = "O. K";

Boiler b = new Boiler(100, 12);

int t = b.getTemp();

int p = b.getPressure();

if(t > 150 || t < 80 || p < 12 || p > 15)

{

remarks = "Need Maintenance";

}

OnBoilerEventLog("Logging Info:\n");

OnBoilerEventLog("Temparature " + t + "\nPressure: " + p);

OnBoilerEventLog("\nMessage: " + remarks);

}

protected void OnBoilerEventLog(string message)

{

if (BoilerEventLog != null)

{

BoilerEventLog(message);

}

}

}

// this class keeps a provision for writing into the log file

class BoilerInfoLogger

{

FileStream fs;

StreamWriter sw;

public BoilerInfoLogger(string filename)

{

fs = new FileStream(filename, FileMode.Append, FileAccess.Write);

sw = new StreamWriter(fs);

}

public void Logger(string info)

{

sw.WriteLine(info);

}

public void Close()

{

sw.Close();

fs.Close();

}

}

// The event subscriber

public class RecordBoilerInfo

{

static void Logger(string info)

{

Console.WriteLine(info);

}//end of Logger

static void Main(string[] args)

{

BoilerInfoLogger filelog = new BoilerInfoLogger("e:\\boiler.txt");

DelegateBoilerEvent boilerEvent = new DelegateBoilerEvent();

boilerEvent.BoilerEventLog += new

DelegateBoilerEvent.BoilerLogHandler(Logger);

boilerEvent.BoilerEventLog += new

DelegateBoilerEvent.BoilerLogHandler(filelog.Logger);

boilerEvent.LogProcess();

Console.ReadLine();

filelog.Close();

}//end of main

}//end of RecordBoilerInfo

}

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

Logging info:

Temperature 100

Pressure 12

Message: O. K

Do the Following Exercise

YOMA BANK

(WELCOME TO YAMA BANK)

[1] ACCOUNT \*

[2] UPDATE

[3]SEARCH

[4] EXIT PROGRAM

**ENTER CHOICE:**

3] SEARCH SUBMENU

[1] BY NAME

[2] BY ACCOUNT NUMBER

ENTER CHOICE:

\*(ACCCOUNT NO, ACCOUNT NAME, NRC, GENGER, PHONE, EMAIL, ADDRESS, DATE, INITIAL AMOUNT)

\*\*( PHONE, EMAIL, ADDRESS, DATE,INITIAL AMOUNT)

[1] DEPOSIT

[2] WIDTHDRAW

[3] PERSONAL INFORMATION UPDATE\*\*

ENTER CHOICE:

[1] BY NAME

[2] BY ACCUNT NO

ENTER CHOICE:

2] UPDATE SUBMENU

1] ACCOUNT SUBMENU