C# TUTORIAL

Simply Easy Learning by tutorialspoint.com

tutorialspoint.com

CHAPTER

3

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();
      }
   }
}
```

```
            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
```

Member Variables

Variables are attributes or data members of a class, used for storing data. In the preceding program, the *Rectangle* class has two member variables named *length* and *width*.

Member Functions

Functions are set of statements that perform a specific task. The member functions of a class are declared within the class. Our sample class Rectangle contains three member functions: *AcceptDetails*, *GetArea* and *Display*.

Instantiating a Class

In the preceding program, the class *ExecuteRectangle* is used as a class, which contains the *Main()* method and instantiates the *Rectangle* class.

Identifiers

An identifier is a name used to identify a class, variable, function, or any other user-defined item. The basic rules for naming classes in C# are as follows:

- A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9) or underscore. The first character in an identifier cannot be a digit.
- It must not contain any embedded space or symbol like ? - +! @ # % ^ & * ( ) [ ] { } . ; : " ' / and \. However, an underscore (_) can be used.
- It should not be a C# keyword.

C# Keywords

Keywords are reserved words predefined to the C# compiler. These keywords cannot be used as identifiers; however, if you want to use these keywords as identifiers, you may prefix the keyword with the @ character.

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

The following table lists the reserved keywords and contextual keywords in C#:

Reserved Keywords						
abstract	As	Base	bool	break	byte	case
catch	Char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	Goto	if	implicit	in	in (generic modifier)	int
interface	internal	is	lock	long	namespace	new
null	object	operator	out	out (generic modifier)	override	params
private	protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string	struct
switch	This	throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using	virtual	void
volatile	While					
Contextual Keywords						
add	Alias	ascending	descending	dynamic	from	get

short	16-bit signed integer type	-32,768 to 32,767	0
uint	32-bit unsigned integer type	0 to 4,294,967,295	0
ulong	64-bit unsigned integer type	0 to 18,446,744,073,709,551,615	0
ushort	16-bit unsigned integer type	0 to 65,535	0

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** method. The expression *sizeof(type)* yields the storage size of the object or type in bytes. Following is an example to get the size of *int* type on any machine:

```
namespace DataTypeApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Size of int: {0}", sizeof(int));
         Console.ReadLine();
      }
   }
}
```

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

```
Size of int: 4
```


Reference Types

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

In other words, they refer to a memory location. Using more than one variable, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value. Example of **built-in** reference types are: **object**, **dynamic** and **string**.

OBJECT TYPE

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

When a value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing**.

```
object obj;
obj = 100; // this is boxing
```

DYNAMIC TYPE

You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at runtime.

Syntax for declaring a dynamic type is:

```
dynamic <variable_name> = value;
```

```
            bool b = true;

            Console.WriteLine(i.ToString());
            Console.WriteLine(f.ToString());
            Console.WriteLine(d.ToString());
            Console.WriteLine(b.ToString());
            Console.ReadKey();

        }
    }
}
```

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

```
75
53.005
2345.7652
True
```

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
```

Misc Operators

There are few other important operators including **sizeof, typeof** and **? :** supported by C#.

Operator	Description	Example
sizeof()	Returns the size of a data type.	sizeof(int), will return 4.
typeof()	Returns the type of a class.	typeof(StreamReader);
&	Returns the address of an variable.	&a; will give actual address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y
is	Determines whether an object is of a certain type.	If(Ford is Car) // checks if Ford is an object of the Car class.
As	Cast without raising an exception if the cast fails.	Object obj = new StringReader("Hello"); StringReader r = obj as StringReader;

Example


```
using System;

namespace OperatorsAppl
{

   class Program
   {
      static void Main(string[] args)
      {

         /* example of sizeof operator */
         Console.WriteLine("The size of int is {0}", sizeof(int));
         Console.WriteLine("The size of short is {0}", sizeof(short));
         Console.WriteLine("The size of double is {0}", sizeof(double));

         /* example of ternary operator */
         int a, b;
         a = 10;
         b = (a == 1) ? 20 : 30;
         Console.WriteLine("Value of b is {0}", b);

         b = (a == 10) ? 20 : 30;
         Console.WriteLine("Value of b is {0}", b);
         Console.ReadLine();
      }
   }
}
```

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

```
The size of int is 4
The size of short is 2
The size of double is 8
Value of b is 30
```

The if...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax:

The syntax of an **if...else if...else** statement in C# is:

```
if(boolean_expression 1)
{
   /* Executes when the boolean expression 1 is true */
}
else if( boolean_expression 2)
{
   /* Executes when the boolean expression 2 is true */
}
else if( boolean_expression 3)
{
   /* Executes when the boolean expression 3 is true */
}
else
{
   /* executes when the none of the above condition is true */
}
```


Example:

```
using System;

namespace DecisionMaking
{

    class Program
    {
        static void Main(string[] args)
        {

            /* local variable definition */
            int a = 100;

            /* check the boolean condition */
            if (a == 10)
            {
                /* if condition is true then print the following */
                Console.WriteLine("Value of a is 10");
            }
            else if (a == 20)
            {
                /* if else if condition is true */
```

```
            /* check the boolean condition */
            if (a == 100)
            {
                /* if condition is true then check the following */
                if (b == 200)
                {
                    /* if condition is true then print the following */
                    Console.WriteLine("Value of a is 100 and b is 200");
                }
            }
            Console.WriteLine("Exact value of a is : {0}", a);
            Console.WriteLine("Exact value of b is : {0}", b);
            Console.ReadLine();
        }
    }
}
```

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

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

switch statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

Syntax:

The syntax for a **switch** statement in C# is as follows:

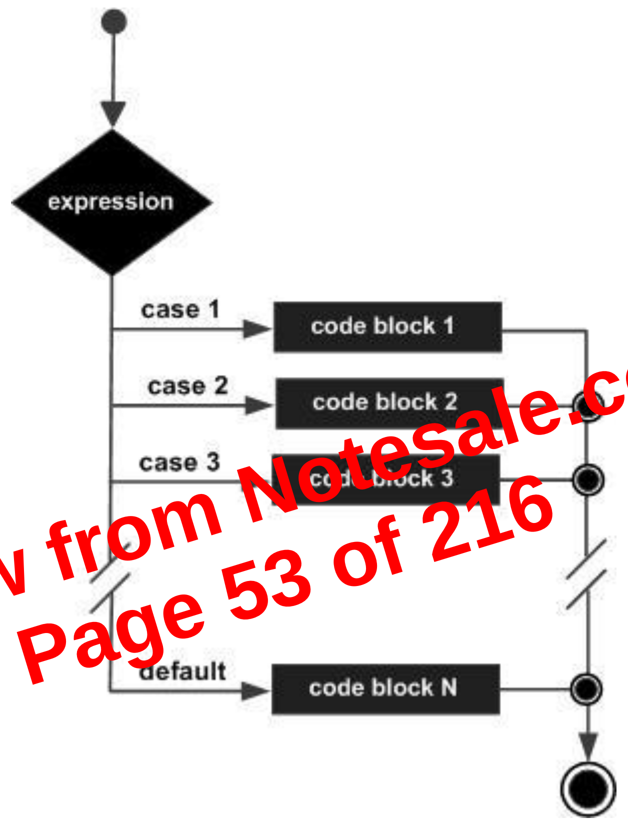
```
switch(expression){
    case constant-expression  :
       statement(s);
       break; /* optional */
    case constant-expression  :
       statement(s);
       break; /* optional */

    /* you can have any number of case statements */
    default : /* Optional */
       statement(s);
}
```

The following rules apply to a **switch** statement:

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.

- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through*to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

Flow Diagram:



Example:

```
using System;

namespace DecisionMaking
{

    class Program
    {
        static void Main(string[] args)
        {
            /* local variable definition */
            char grade = 'B';

            switch (grade)
            {
                case 'A':
                    Console.WriteLine("Excellent!");
                    break;
```

```
            for (int a = 10; a < 20; a = a + 1)
            {
                Console.WriteLine("value of a: {0}", a);
            }
            Console.ReadLine();
        }
    }
}
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

do...while loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

The syntax of a **do...while** loop in C# is:


```
do
{
   statement(s);

}while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

```
        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }//end class Rectangle
    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.Acceptdetails();
            r.Display();
            Console.ReadLine();
        }
    }
}
```

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

```
Enter Length:
4.4
Enter Width:
3.3
Length: 4.4
Width: 3.3
Area: 14.52
```


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

Protected Access Specifier

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

Internal Access Specifier

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

The following program illustrates this:

```
using System;

namespace RectangleApplication
{
    class Rectangle
    {
        //member variables
        internal double length;
        internal double width;
```

```
        double GetArea()
        {
            return length * width;
        }
       public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }//end class Rectangle
    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.length = 4.5;
            r.width = 3.5;
            r.Display();
            Console.ReadLine();
        }
    }
}
```

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

```
Length: 4.5
Width: 3.5
Area: 15.75
```


In the preceding example, notice that the member function *GetArea()* is not declared with any access specifier. Then what would be the default access specifier of a class member if we don't mention any? It is **private**.

Protected Internal Access Specifier

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

Output parameters

A return statement can be used for returning only one value from a function. However, using **output parameters**, you can return two values from a function. Output parameters are like reference parameters, except that they transfer data out of the method rather than into it.

The following example illustrates this:

```
using System;

namespace CalculatorApplication
{
   class NumberManipulator
   {
      public void getValue(out int x )
      {
         int temp = 5;
         x = temp;
      }

      static void Main(string[] args)
      {
         NumberManipulator n = new NumberManipulator();
         /* local variable definition */
         int a = 100;

         Console.WriteLine("Before method call, value of a : {0}", a);

         /* calling a function to get the value */
         n.getValue(out a);

         Console.WriteLine("After method call, value of a : {0}", a);
         Console.ReadLine();

      }
   }
}
```


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

```
Before method call, value of a : 100
After method call, value of a : 5
```

The variable supplied for the output parameter need not be assigned a value the method call. Output parameters are particularly useful when you need to return values from a method through the parameters without assigning an initial value to the parameter. Look at the following example, to understand this:

```
using System;

namespace CalculatorApplication
{
   class NumberManipulator
   {
      public void getValues(out int x, out int y )
      {
          Console.WriteLine("Enter the first value: ");
          x = Convert.ToInt32(Console.ReadLine());
          Console.WriteLine("Enter the second value: ");
          y = Convert.ToInt32(Console.ReadLine());
      }
```

CHAPTER

14

C# Nullables

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

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

```
< data_type> ? <variable_name> = null;
```

The following example demonstrates use of nullable data types:


```
using System;
namespace CalculatorApplication
{
   class NullablesAtShow
   {
      static void Main(string[] args)
      {
         int? num1 = null;
         int? num2 = 45;
         double? num3 = new double?();
         double? num4 = 3.14157;

         bool? boolval = new bool?();

         // display the values

         Console.WriteLine("Nullables at Show: {0}, {1}, {2}, {3}",
                           num1, num2, num3, num4);
         Console.WriteLine("A Nullable boolean value: {0}", boolval);
         Console.ReadLine();

      }
   }
}
```

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

```
Nullables at Show: , 45,  , 3.14157
A Nullable boolean value:
```

CHAPTER

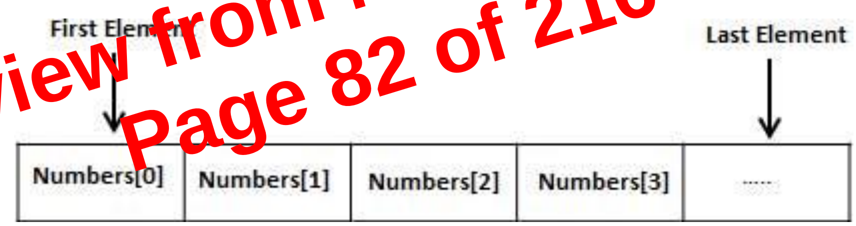
15

C# Arrays

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

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

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



Declaring Arrays

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

```
datatype[] arrayName;
```

where,

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

For example,

```
double[] balance;
```

Initializing an Array

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

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

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form a[i , j], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays

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

```
int [,] a = int [3,4] = {
 {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
 {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
 {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

Accessing Two-Dimensional Array Elements

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
int val = a[2,3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above diagram. Let us check below program where we have used nested loop to handle a two dimensional array:

```
using System;

namespace ArrayApplication
{
    class MyArray
    {
        static void Main(string[] args)
        {
            /* an array with 5 rows and 2 columns*/
            int[,] a = new int[5, 2] {{0,0}, {1,2}, {2,4}, {3,6}, {4,8} };

            int i, j;

            /* output each array element's value */
            for (i = 0; i < 5; i++)
            {
                for (j = 0; j < 2; j++)
                {
                    Console.WriteLine("a[{0},{1}] = {2}", i, j, a[i,j]);
                }
            }
           Console.ReadKey();
        }
    }
}
```

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

2	**IsReadOnly** Gets a value indicating whether the Array is read-only.
3	**Length** Gets a 32-bit integer that represents the total number of elements in all the dimensions of the Array.
4	**LongLength** Gets a 64-bit integer that represents the total number of elements in all the dimensions of the Array.
5	**Rank** Gets the rank (number of dimensions) of the Array.

Methods of the Array Class

The following table provides some of the most commonly used properties of the Array class:

S.N	Method Name & Description
1	**Clear** Sets a range of elements in the Array to zero, to false, or to null, depending on the element type.
2	**Copy(Array, Array, Int32)** Copies a range of elements from an Array starting at the first element and pastes them into another Array starting at the first element. The length is specified as a 32-bit integer.
3	**CopyTo(Array, Int32)** Copies all the elements of the current one-dimensional Array to the specified one-dimensional Array starting at the specified destination Array index. The index is specified as a 32-bit integer.
4	**GetLength** Gets a 32-bit integer that represents the number of elements in the specified dimension of the Array.
5	**GetLongLength** Gets a 64-bit integer that represents the number of elements in the specified dimension of the Array.
6	**GetLowerBound** Gets the lower bound of the specified dimension in the Array.
7	**GetType** Gets the Type of the current instance. (Inherited from Object.)
8	**GetUpperBound** Gets the upper bound of the specified dimension in the Array.
9	**GetValue(Int32)** Gets the value at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer.
10	**IndexOf(Array, Object)** Searches for the specified object and returns the index of the first occurrence within the entire one-dimensional Array.
11	**Reverse(Array)** Reverses the sequence of the elements in the entire one-dimensional Array.
12	**SetValue(Object, Int32)** Sets a value to the element at the specified position in the one-dimensional Array. The index is specified as a 32-bit integer.

CHAPTER

16

C# Strings

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

Creating a String Object

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

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

The following example demonstrates this:

```
using System;

namespace StringApplication
{
    class Program
    {
        static void Main(string[] args)
        {
           //from string literal and string concatenation
            string fname, lname;
            fname = "Rowan";
            lname = "Atkinson";

            string fullname = fname + lname;
            Console.WriteLine("Full Name: {0}", fullname);

            //by using string constructor
            char[] letters = { 'H', 'e', 'l', 'l','o' };
            string greetings = new string(letters);
            Console.WriteLine("Greetings: {0}", greetings);
```

CHAPTER

20

C# Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

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

The idea of inheritance implements the **IS-A** relationship. For example, mammal **IS A** animal, dog **IS-A**mammal hence dog **IS-A** animal as well and so on.

Base and Derived Classes

A class can be derived from more than one class or interface, which means that it can inherit data and functions from multiple base class or interface.

The syntax used in C# for creating derived classes is as follows:

```
<acess-specifier> class <base_class>
{
 ...
}
class <derived_class> : <base_class>
{
 ...
}
```

Consider a base class Shape and its derived class Rectangle:

```
using System;
namespace InheritanceApplication
{
   class Shape
   {
      public void setWidth(int w)
      {
         width = w;
      }
      public void setHeight(int h)
      {
```

```
   public class Transaction : ITransactions
   {
      private string tCode;
      private string date;
      private double amount;
      public Transaction()
      {
         tCode = " ";
         date = " ";
         amount = 0.0;
      }
      public Transaction(string c, string d, double a)
      {
         tCode = c;
         date = d;
         amount = a;
      }
      public double getAmount()
      {
         return amount;
      }
      public void showTransaction()
      {
         Console.WriteLine("Transaction: {0}", tCode);
         Console.WriteLine("Date: {0}", date);
         Console.WriteLine("Amount: {0}", getAmount());
      }
   }
   class Tester
   {
      static void Main(string[] args)
      {
         Transaction t1 = new Transaction("001", "8/10/2012", 78900.00);
         Transaction t2 = new Transaction("002", "9/10/2012", 451900.00);
         t1.showTransaction();
         t2.showTransaction();
         Console.ReadKey();
      }
   }
}
```


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

```
Transaction: 001
Date: 8/10/2012
Amount: 78900
Transaction: 002
Date: 9/10/2012
Amount: 451900
```

CHAPTER

25

C# Preprocessor Directives

The preprocessors directives give instruction to the compiler to preprocess the information before actual compilation starts.

All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not statements, so they do not end with a semicolon (;).

C# compiler does not have a separate preprocessor; however, the directives are processed as if there was one. In C# the preprocessor directives are used to help in conditional compilation. Unlike C and C++ directives, they are not used to create macros. A preprocessor directive must be the only instruction on a line.

List of Preprocessor Directives in C#

The following table lists the preprocessor directives available in C#:

Preprocessor Directive	Description.
#define	It defines a sequence of characters, called symbol.
#undef	It allows you to undefine a symbol.
#if	It allows testing a symbol or symbols to see if they evaluate to true.
#else	It allows to create a compound conditional directive, along with #if.
#elif	It allows creating a compound conditional directive.
#endif	Specifies the end of a conditional directive.
#line	It lets you modify the compiler's line number and (optionally) the file name output for errors and warnings.
#error	It allows generating an error from a specific location in your code.
#warning	It allows generating a level one warning from a specific location in your code.
#region	It lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor.
#endregion	It marks the end of a #region block.

CHAPTER

26

C# Regular Expressions

A **regular expression** is a pattern that could be matched against an input text. The .Net framework provides a regular expression engine that allows such matching. A pattern consists of one or more character literals, operators, or constructs.

Constructs for Defining Regular Expressions

There are various categories of characters, operators, and constructs that lets you to define regular expressions. Click the follwoing links to find these constructs.

- Character escapes
- Character classes
- Anchors
- Grouping constructs
- Quantifiers
- Backreference constructs
- Alternation constructs
- Substitutions
- Miscellaneous constructs

Character escapes

These are basically the special characters or escape characters. The backslash character (\) in a regular expression indicates that the character that follows it either is a special character or should be interpreted literally.

The following table lists the escape characters:

Escaped character	Description	Pattern	Matches
\a	Matches a bell character, \u0007.	\a	"\u0007" in "Warning!" + '\u0007'
\b	In a character class, matches a backspace, \u0008.	[\b]{3,}	"\b\b\b\b" in "\b\b\b\b"
\t	Matches a tab, \u0009.	(\w+)\t	"Name\t", "Addr\t" in "Name\tAddr\t"
\r	Matches a carriage return, \u000D. (\r is not equivalent to the newline character, \n.)	\r\n(\w+)	"\r\nHello" in "\r\Hello\nWorld."

$`	Substitutes all the text of the input string before the match.	B+	$`	"AABBCC"	"AAAACC"
$'	Substitutes all the text of the input string after the match.	B+	$'	"AABBCC"	"AACCCC"
$+	Substitutes the last group that was captured.	B+(C+)	$+	"AABBCCDD"	AACCDD
$_	Substitutes the entire input string.	B+	$_	"AABBCC"	"AAAABBCCCC"

Miscellaneous constructs

Following are various miscellaneous constructs:

Construct	Definition	Example
(?imnsx-imnsx)	Sets or disables options such as case insensitivity in the middle of a pattern.	\bA(?i)b\w+\b matches "ABA", "Able" in "ABA Able Act"
(?#comment)	Inline comment. The comment ends at the first closing parenthesis.	\bA(?#Matches words starting with A)\w+\b
# [to end of line]	X-mode comment. The comment starts at an unescaped # and continues to the end of the line.	(?x)\bA\w+\b#Matches words starting with A

The Regex Class

The Regex class is used for representing a regular expression.

The Regex class has the following commonly used methods:

S.N	Methods & Description
1	**public bool IsMatch(string input)** Indicates whether the regular expression specified in the Regex constructor finds a match in a specified input string.
2	**public bool IsMatch(string input, int startat)** Indicates whether the regular expression specified in the Regex constructor finds a match in the specified input string, beginning at the specified starting position in the string.
3	**public static bool IsMatch(string input, string pattern)** Indicates whether the specified regular expression finds a match in the specified input string.
4	**public MatchCollection Matches(string input)** Searches the specified input string for all occurrences of a regular expression.
5	**public string Replace(string input, string replacement)** In a specified input string, replaces all strings that match a regular expression pattern with a specified replacement string.
6	**public string[] Split(string input)** Splits an input string into an array of substrings at the positions defined by a regular expression pattern specified in the Regex constructor.

CHAPTER

28

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 classes 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.

S.N	Method Name & Purpose
1	**public override void Close()** It closes the StreamReader object and the underlying stream, and releases any system resources associated with the reader.
2	**public override int Peek()** Returns the next available character but does not consume it.
3	**public override int Read()** Reads the next character from the input stream and advances the character position by one character.

Example:

The following example demonstrates reading a text file named Jamaica.txt. The file reads:

```
Down the way where the nights are gay
And the sun shines daily on the mountain top
I took a trip on a sailing ship
And when I reached Jamaica
I made a stop
```

```
using System;
using System.IO;

namespace FileApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                // Create an instance of StreamReader to read from a file.
                // The using statement also closes the StreamReader.
                using (StreamReader sr = new StreamReader("c:/jamaica.txt"))
                {
                    string line;

                    // Read and display lines from the file until
                    // the end of the file is reached.
                    while ((line = sr.ReadLine()) != null)
                    {
                        Console.WriteLine(line);
                    }
                }
            }
            catch (Exception e)
            {
                // Let the user know what went wrong.
                Console.WriteLine("The file could not be read:");
                Console.WriteLine(e.Message);
            }
            Console.ReadKey();
        }
    }
}
```


Guess what it displays when you compile and run the program!

CHAPTER 32

C# Indexers

An **indexer** allows an object to be indexed like an array. When you define an indexer for a class, this class behaves like a **virtual array**. You can then access the instance of this class using the array access operator ([ ]).

Syntax

A one dimensional indexer has the following syntax:

```
element-type this[int index]
{
   // The get accessor.
   get
   {
      // return the value specified by index
   }

   // The set accessor.
   set
   {
      // set the value specified by index
   }
}
```


Use of Indexers

Declaration of behavior of an indexer is to some extent similar to a property. Like properties, you use **get** and **set** accessors for defining an indexer. However, properties return or set a specific data member, whereas indexers returns or sets a particular value from the object instance. In other words, it breaks the instance data into smaller parts and indexes each part, gets or sets each part.

Defining a property involves providing a property name. Indexers are not defined with names, but with the **this** keyword, which refers to the object instance. The following example demonstrates the concept:

```
using System;
namespace IndexerApplication
{
   class IndexedNames
   {
      private string[] namelist = new string[size];
      static public int size = 10;
      public IndexedNames()
```

CHAPTER

33

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
{
```

```
      public delegate void NumManipulationHandler();

      public event NumManipulationHandler ChangeNum;

      protected virtual void OnNumChanged()
      {
         if (ChangeNum != null)
         {
            ChangeNum();
         }
         else
         {
            Console.WriteLine("Event fired!");
         }

      }
      public EventTest(int n )
      {
         SetValue(n);
      }
      public void SetValue(int n)
      {
         if (value != n)
         {
            value = n;
            OnNumChanged();
         }
      }
   }
   public class MainClass
   {
      public static void Main()
      {
         EventTest e = new EventTest(5);
         e.SetValue(7);
         e.SetValue(11);
         Console.ReadKey();
      }
   }
}
```

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

```
Event Fired!
Event Fired!
Event Fired!
```

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
```

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:

The following example demonstrates use of Stack:

```
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
```

CHAPTER

36

C# Generics

Generics allow you to delay the specification of the data type of programming elements in a class or a method until it is actually used in the program. In other words, generics allow you to write a class or method that can work with any data type.

You write the specifications for the class or the method, with substitute parameters for data types. When the compiler encounters a constructor for the class or a function call for the method, it generates code to handle the specific data type. A simple example would help understanding the concept:


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

namespace GenericApplication
{
    public class MyGenericArray<T>
    {
        private T[] array;
        public MyGenericArray(int size)
        {
            array = new T[size + 1];
        }
        public T getItem(int index)
        {
            return array[index];
        }
        public void setItem(int index, T value)
        {
            array[index] = value;
        }
    }

    class Tester
    {
        static void Main(string[] args)
        {
            //declaring an int array
            MyGenericArray<int> intArray = new MyGenericArray<int>(5);
            //setting values
            for (int c = 0; c < 5; c++)
            {
                intArray.setItem(c, c*5);
            }
```

CHAPTER

38

C# Unsafe Codes

C# allows using pointer variables in a function of code block when it is marked by the **unsafe** modifier.

The **unsafe code** or the unmanaged code is a code block that uses a **pointer** variable.

Pointer Variables

A **pointer** is a variable whose value is the address of another variable i.e., the direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address.

The general form of a pointer variable declaration is:

```
type *var-name;
```

Following are valid pointer declarations:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The following example illustrates use of pointers in C#, using the unsafe modifier:

```
using System;
namespace UnsafeCodeApplication
{
    class Program
    {
        static unsafe void Main(string[] args)
        {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} ",  var);
```

```
            Console.WriteLine("Address is: {0}",  (int)p);

            Console.ReadKey();

        }

    }

}
```

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

```
Data is: 20
Address is: 99215364
```

Instead of declaring an entire method as unsafe, you can also declare a part of the code as unsafe. The example in the following section shows this.

Retrieving the Data Value Using a Pointer

You can retrieve the data stored at the located referenced by the pointer variable, using the **ToString()**method. Following example demonstrates this:


```
using System;

namespace UnsafeCodeApplication

{

   class Program

   {

      public static void Main()

      {

         unsafe

         {

            int var = 20;

            int* p = &var;

            Console.WriteLine("Data is: {0} " , var);

            Console.WriteLine("Data is: {0} " , p->ToString());

            Console.WriteLine("Address is: {0} " , (int)p);

         }

         Console.ReadKey();

      }

   }

}
```

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

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

```
Before Swap: var1: 10, var2: 20
After Swap: var1: 20, var2: 10
```

Accessing Array Elements Using a Pointer

In C#, an array name and a pointer to a data type same as the array data, are not the same variable type. For example, int *p and int[] p, are not same type. You can increment the pointer variable p because it is not fixed in memory but an array address is fixed in memory, and you can't increment that.

Therefore, if you need to access an array data using a pointer variable, as we traditionally do in C, or C++ (please check: C Pointers), you need to fix the pointer using the **fixed** keyword.

The following example demonstrates this:

```
using System;

namespace UnsafeCodeApplication

{

   class TestPointer

   {

      public unsafe static void Main()

      {

         int[]  list = {10, 100, 200};

         fixed(int *ptr = list)


         /* let us have array address in pointer */

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

         {

            Console.WriteLine("Address of list[{0}]={1}",i,(int)(ptr + i));

            Console.WriteLine("Value of list[{0}]={1}", i, *(ptr + i));

         }

         Console.ReadKey();

      }

   }

}
```


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

```
Address of list[0] = 31627168
Value of list[0] = 10
Address of list[1] = 31627172
Value of list[1] = 100
Address of list[2] = 31627176
```

```
Value of list[2] = 200
```

Compiling Unsafe Code

For compiling unsafe code, you have to specify the **/unsafe** command-line switch with command-line compiler.

For example, to compile a program named prog1.cs containing unsafe code, from command line, give the command:

```
csc /unsafe prog1.cs
```

If you are using Visual Studio IDE then you need to enable use of unsafe code in the project properties.

To do this:

- Open **project properties** by double clicking the properties node in the Solution Explorer.
- Click on the **Build** tab.
- Select the option "**Allow unsafe code**".