# UNIT - I

# C# - Overview

C# is a modern, general-purpose, object-oriented programming language developed by Microsoft and approved by European Computer Manufacturers Association (ECMA) and International Standards Organization (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 on different computer platforms and architectures.

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

* It is a modern, general-purpose programming language
* It is object oriented.
* It is component oriented.
* It is easy to learn.
* It is a structured language.
* It produces efficient programs.
* It can be compiled on a variety of computer platforms.
* It is a 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 a number of programmers worldwide.

Following is the list of few important features of C# −

* Boolean Conditions
* Automatic Garbage Collection
* Standard Library
* Assembly Versioning
* Properties and Events
* Delegates and Events Management
* Easy-to-use Generics
* Indexers
* Conditional Compilation
* Simple Multithreading
* LINQ and Lambda Expressions
* Integration with Windows

# C# - Environment

In this chapter, we will discuss the tools required for creating C# programming. We have already mentioned that C# is part of .Net framework and is used for writing .Net applications. Therefore, before discussing the available tools for running a C# program, let us understand how C# relates to the .Net framework.

**The .Net Framework**

The .Net framework is a revolutionary platform that helps you to write the following types of applications −

* Windows applications
* Web applications
* Web services

The .Net framework applications are multi-platform applications. The framework has been designed in such a way that it can be used from any of the following languages: C#, C++, Visual Basic, Jscript, COBOL, etc. All these languages can access the framework as well as communicate with each other.

The .Net framework consists of an enormous library of codes used by the client languages such as C#. Following are some of the components of the .Net framework −

* Common Language Runtime (CLR)
* The .Net Framework Class Library
* Common Language Specification
* Common Type System
* Metadata and Assemblies
* Windows Forms
* ASP.Net and ASP.Net AJAX
* ADO.Net
* Windows Workflow Foundation (WF)
* Windows Presentation Foundation
* Windows Communication Foundation (WCF)
* LINQ

**Integrated Development Environment (IDE) for C#**

Microsoft provides the following development tools for C# programming −

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

The last two are freely available from Microsoft official website. Using these tools, you can write all kinds of C# programs from simple command-line applications to more complex applications. You can also write C# source code files using a basic text editor, like Notepad, and compile the code into assemblies using the command-line compiler, which is again a part of the .NET Framework.

Visual C# Express and Visual Web Developer Express edition are trimmed down versions of Visual Studio and has the same appearance. They retain most features of Visual Studio. In this tutorial, we have used Visual C# 2010 Express.

# 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.

**Creating Hello World Program**

A C# program consists of the following parts −

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

Let us look at a simple code that prints 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 this code is compiled and executed, it produces the following result −

Hello World

Let us look at the various parts of the given program −

* The first line of the program **using System;** - the **using** keyword is used to include the **System** namespace in the program. A program generally has multiple **using** statements.
* The next line has the **namespace** declaration. A **namespace** is a collection of classes. The *HelloWorldApplication* namespace contains the class *HelloWorld*.
* The next line has a **class** declaration, the class *HelloWorld* contains the data and method definitions that your program uses. Classes generally contain multiple methods. Methods define the behavior of the class. However, the *HelloWorld* class has only one method **Main**.
* The next line defines the **Main** method, which is the **entry point** for all C# programs. The **Main** method states what the class does when executed.
* The next line /\*...\*/ is ignored by the compiler and it is put to add **comments** in the program.
* The Main method specifies its behavior with the statement **Console.WriteLine("Hello World");**

*WriteLine* is a method of the *Console* class defined in the *System* namespace. This statement causes the message "Hello, World!" to be displayed on the screen.

* The last line **Console.ReadKey();** is for the VS.NET Users. This makes the program wait for a key press and it prevents the screen from running and closing quickly when the program is launched from Visual Studio .NET.

It is worth to note the following points −

* C# is case sensitive.
* All statements and expression must end with a semicolon (;).
* The program execution starts at the Main method.
* Unlike Java, program file name could be different from the class name.

**Compiling and Executing the 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 click OK button.
* This creates a new project in Solution Explorer.
* Write code in the Code Editor.
* Click the Run button or press F5 key to execute 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 takes you to the next line and generates **helloworld.exe** executable file.
* Type **helloworld** to execute your program.
* You can see the output Hello World printed on the screen.

# C# - Data Types

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

* Value types
* Reference types
* Pointer types

## Value Type

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

The value types directly contain data. Some examples are **int, char, and float**, which stores numbers, alphabets, and floating point numbers, respectively. When you declare an **int** type, the system allocates memory to store the value.

The following table lists the available value types in C# 2010 −

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Represents** | **Range** | **Default Value** |
| bool | Boolean value | True or False | False |
| byte | 8-bit unsigned integer | 0 to 255 | 0 |
| char | 16-bit Unicode character | U +0000 to U +ffff | '\0' |
| decimal | 128-bit precise decimal values with 28-29 significant digits | (-7.9 x 1028 to 7.9 x 1028) / 100 to 28 | 0.0M |
| double | 64-bit double-precision floating point type | (+/-)5.0 x 10-324 to (+/-)1.7 x 10308 | 0.0D |
| float | 32-bit single-precision floating point type | -3.4 x 1038 to + 3.4 x 1038 | 0.0F |
| int | 32-bit signed integer type | -2,147,483,648 to 2,147,483,647 | 0 |
| long | 64-bit signed integer type | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | 0L |
| sbyte | 8-bit signed integer type | -128 to 127 | 0 |
| short | 16-bit signed integer type | -32,768 to 32,767 | 0 |
| uint | 32-bit unsigned integer type | 0 to 4,294,967,295 | 0 |
| ulong | 64-bit unsigned integer type | 0 to 18,446,744,073,709,551,615 | 0 |
| ushort | 16-bit unsigned integer type | 0 to 65,535 | 0 |

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 −

using System;

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 Type

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

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

### Object Type

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

Syntax for declaring a dynamic type is −

dynamic <variable\_name> = value;

For example,

dynamic d = 20;

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

### String Type

The **String Type** allows you to assign any string values to a variable. The string type is an alias for the System.String class. It is derived from object type. The value for a string type can be assigned using string literals in two forms: quoted and @quoted.

For example,

String str = "Tutorials Point";

A @quoted string literal looks as follows −

@"Tutorials Point";

The user-defined reference types are: class, interface, or delegate. We will discuss these types in later chapter.

## Pointer Type

Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as the pointers in C or C++.

Syntax for declaring a pointer type is −

type\* identifier;

For example,

char\* cptr;

int\* iptr;

We will discuss pointer types in the chapter 'Unsafe Codes'.

# C# - Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C# has a specific type, which determines the size and layout of the variable's memory the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

The basic value types provided in C# can be categorized as –

|  |  |
| --- | --- |
| **Type** | **Example** |
| Integral types | sbyte, byte, short, ushort, int, uint, long, ulong, and char |
| Floating point types | float and double |
| Decimal types | decimal |
| Boolean types | true or false values, as assigned |
| Nullable types | Nullable data types |

C# also allows defining other value types of variable such as **enum** and reference types of variables such as **class**, which we will cover in subsequent chapters.

**Defining Variables**

Syntax for variable definition in C# is −

<data\_type> <variable\_list>;

Here, data\_type must be a valid C# data type including char, int, float, double, or any user-defined data type, and variable\_list may consist of one or more identifier names separated by commas.

Some valid variable definitions are shown here −

int i, j, k;

char c, ch;

float f, salary;

double d;

You can initialize a variable at the time of definition as −

int i = 100;

**Initializing Variables**

Variables are initialized (assigned a value) with an equal sign followed by a constant expression. The general form of initialization is −

variable\_name = value;

Variables can be initialized in their declaration. The initializer consists of an equal sign followed by a constant expression as −

<data\_type> <variable\_name> = value;

Some examples are −

int d = 3, f = 5; /\* initializing d and f. \*/

byte z = 22; /\* initializes z. \*/

double pi = 3.14159; /\* declares an approximation of pi. \*/

char x = 'x'; /\* the variable x has the value 'x'. \*/

It is a good programming practice to initialize variables properly, otherwise sometimes program may produce unexpected result.

The following example uses various types of variables −

using System;

namespace VariableDefinition {

class Program {

static void Main(string[] args) {

short a;

int b ;

double c;

/\* actual initialization \*/

a = 10;

b = 20;

c = a + b;

Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);

Console.ReadLine();

}

}

}

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

a = 10, b = 20, c = 30

**Accepting Values from User**

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

For example,

int num;

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

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

**Lvalue and Rvalue Expressions in C#**

There are two kinds of expressions in C# −

* **lvalue** − An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.
* **rvalue** − An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and hence they may appear on the left-hand side of an assignment. Numeric literals are rvalues and hence they may not be assigned and can not appear on the left-hand side. Following is a valid C# statement −

int g = 20;

But following is not a valid statement and would generate compile-time error −

10 = 20;

# C# - Constants and Literals

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

The constants are treated just like regular variables except that their values cannot be modified after their definition.

## Integer Literals

An integer literal can be a decimal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, and there is no prefix id for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals −

212 /\* Legal \*/

215u /\* Legal \*/

0xFeeL /\* Legal \*/

Following are other examples of various types of Integer literals −

85 /\* decimal \*/

0x4b /\* hexadecimal \*/

30 /\* int \*/

30u /\* unsigned int \*/

30l /\* long \*/

30ul /\* unsigned long \*/

## Floating-point Literals

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

Here are some examples of floating-point literals −

3.14159 /\* Legal \*/

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

510E /\* Illegal: incomplete exponent \*/

210f /\* Illegal: no decimal or exponent \*/

.e55 /\* Illegal: missing integer or fraction \*/

While representing in decimal form, you must include the decimal point, the exponent, or both; and while representing using exponential form you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

## Character Constants

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

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

|  |  |
| --- | --- |
| **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 |
| \xhh . . . | Hexadecimal number of one or more digits |

Following is the example to show few escape sequence characters −

using System;

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

# C# - Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C# has rich set of built-in operators and provides the following type of operators −

* Arithmetic Operators
* Relational Operators
* Logical Operators
* Bitwise Operators
* Assignment Operators
* Misc Operators

This tutorial explains the arithmetic, relational, logical, bitwise, assignment, and other operators one by one.

**Arithmetic Operators**

Following table shows all the arithmetic operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20 then −

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

**Relational Operators**

Following table shows all the relational operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20, then −

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

**Logical Operators**

Following table shows all the logical operators supported by C#. Assume variable **A** holds Boolean value true and variable **B** holds Boolean value false, then −

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

**Bitwise Operators**

Bitwise operator works on bits and perform bit by bit operation. The truth tables for &, |, and ^ are as follows −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **q** | **p & q** | **p | q** | **p ^ q** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

Assume if A = 60; and B = 13; then in the binary format they are as follows −

A = 0011 1100

B = 0000 1101

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

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011

The Bitwise operators supported by C# are listed in the following table. Assume variable A holds 60 and variable B holds 13, then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) = 12, which is 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either operand. | (A | B) = 61, which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) = 49, which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) = 61, which is 1100 0011 in 2's complement due to a signed binary number. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 = 240, which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 = 15, which is 0000 1111 |

**Assignment Operators**

There are following assignment operators supported by C# −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator, Assigns values from right side operands to left side operand | C = A + B assigns value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand | C -= A is equivalent to C = C - A |
| \*= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator | C &= 2 is same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator | C ^= 2 is same as C = C ^ 2 |
| |= | bitwise inclusive OR and assignment operator | C |= 2 is same as C = C | 2 |

**Miscellaneous 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), returns 4. |
| typeof() | Returns the type of a class. | typeof(StreamReader); |
| & | Returns the address of an variable. | &a; returns actual address of the variable. |
| \* | Pointer to a variable. | \*a; creates pointer named 'a' 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; |

**Operator Precedence in C#**

Operator precedence determines the grouping of terms in an expression. This affects evaluation of an expression. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example x = 7 + 3 \* 2; here, x is assigned 13, not 20 because operator \* has higher precedence than +, so the first evaluation takes place for 3\*2 and then 7 is added into it.

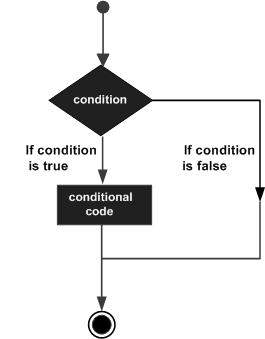
Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators are evaluated first.

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)\* & sizeof | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %=>>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

# C# - Decision Making

Decision making structures requires the programmer to 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 form 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.

|  |  |
| --- | --- |
| **Sr.No.** | **Statement & Description** |
| 1 | [if statement](https://www.tutorialspoint.com/csharp/if_statement_in_csharp.htm)  An **if statement** consists of a boolean expression followed by one or more statements. |
| 2 | [if...else statement](https://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. |
| 3 | [nested if statements](https://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). |
| 4 | [switch statement](https://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. |
| 5 | [nested switch statements](https://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.

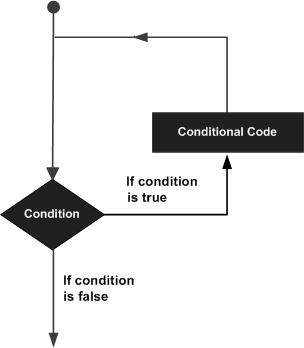
The value of a ? expression is determined as follows: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

# C# - Loops

There may be a situation, when you need to execute a block of code several number of times. In general, the statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or a group of statements multiple times and following is the general from of a loop statement in most of the programming languages −



C# provides following types of loop to handle looping requirements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Sr.No.** | **Loop Type & Description** |
| 1 | [while loop](https://www.tutorialspoint.com/csharp/csharp_while_loop.htm)  It repeats a statement or a group of statements while a given condition is true. It tests the condition before executing the loop body. |
| 2 | [for loop](https://www.tutorialspoint.com/csharp/csharp_for_loop.htm)  It executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 3 | [do...while loop](https://www.tutorialspoint.com/csharp/csharp_do_while_loop.htm)  It is similar to a while statement, except that it tests the condition at the end of the loop body |
| 4 | [nested loops](https://www.tutorialspoint.com/csharp/csharp_nested_loops.htm)  You can use one or more loop inside any another while, for or do..while loop. |

## Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C# provides the following control statements. Click the following links to check their details.

|  |  |
| --- | --- |
| **Sr.No.** | **Control Statement & Description** |
| 1 | [break statement](https://www.tutorialspoint.com/csharp/csharp_break_statement.htm)  Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch. |
| 2 | [continue statement](https://www.tutorialspoint.com/csharp/csharp_continue_statement.htm)  Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |

## Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

### Example

using System;

namespace Loops {

class Program {

static void Main(string[] args) {

for (; ; ) {

Console.WriteLine("Hey! I am Trapped");

}

}

}

}

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but programmers more commonly use the for(;;) construct to signify an infinite loop.

# C# - Encapsulation

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

Abstraction and encapsulation are related features in object oriented programming. Abstraction allows making relevant information visible and encapsulation enables a programmer to *implement the desired level of abstraction*.

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

* Public
* Private
* Protected
* Internal
* Protected internal

**Public Access Specifier**

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

**Private Access Specifier**

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

**Protected Access Specifier**

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

**Internal Access Specifier**

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

**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.

# 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

**Passing Parameters to a Method**

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

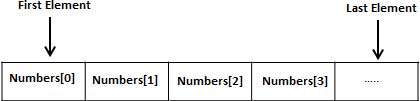
|  |  |
| --- | --- |
| **Sr.No.** | **Mechanism & Description** |
| 1 | [Value parameters](https://www.tutorialspoint.com/csharp/csharp_value_parameters.htm)  This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. |
| 2 | [Reference parameters](https://www.tutorialspoint.com/csharp/csharp_reference_parameters.htm)  This method copies the reference to the memory location of an argument into the formal parameter. This means that changes made to the parameter affect the argument. |
| 3 | [Output parameters](https://www.tutorialspoint.com/csharp/csharp_output_parameters.htm)  This method helps in returning more than one value. |

# 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 stored at contiguous memory locations.

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 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. For example,

double[] balance = new double[10];

**Assigning Values to an Array**

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

double[] balance = new double[10];

balance[0] = 4500.0;

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

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

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

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

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

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

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

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

int[] score = marks;

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

**Accessing Array Elements**

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example,

double salary = balance[9];

The following example, demonstrates the above-mentioned concepts declaration, assignment, and accessing arrays −

using System;

namespace ArrayApplication {

class MyArray {

static void Main(string[] args) {

int [] n = new int[10]; /\* n is an array of 10 integers \*/

int i,j;

/\* initialize elements of array n \*/

for ( i = 0; i < 10; i++ ) {

n[ i ] = i + 100;

}

/\* output each array element's value \*/

for (j = 0; j < 10; j++ ) {

Console.WriteLine("Element[{0}] = {1}", j, n[j]);

}

Console.ReadKey();

}

}

}

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

Element[0] = 100

Element[1] = 101

Element[2] = 102

Element[3] = 103

Element[4] = 104

Element[5] = 105

Element[6] = 106

Element[7] = 107

Element[8] = 108

Element[9] = 109

**Using the *foreach* Loop**

In the previous example, we used a for loop for accessing each array element. You can also use a **foreach** statement to iterate through an array.

using System;

namespace ArrayApplication {

class MyArray {

static void Main(string[] args) {

int [] n = new int[10]; /\* n is an array of 10 integers \*/

/\* initialize elements of array n \*/

for ( int i = 0; i < 10; i++ ) {

n[i] = i + 100;

}

/\* output each array element's value \*/

foreach (int j in n ) {

int i = j-100;

Console.WriteLine("Element[{0}] = {1}", i, j);

}

Console.ReadKey();

}

}

}

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

Element[0] = 100

Element[1] = 101

Element[2] = 102

Element[3] = 103

Element[4] = 104

Element[5] = 105

Element[6] = 106

Element[7] = 107

Element[8] = 108

Element[9] = 109

**C# Arrays**

There are following few important concepts related to array which should be clear to a C# programmer −

|  |  |
| --- | --- |
| **Sr.No.** | **Concept & Description** |
| 1 | [Multi-dimensional arrays](https://www.tutorialspoint.com/csharp/csharp_multi_dimensional_arrays.htm)  C# supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. |
| 2 | [Jagged arrays](https://www.tutorialspoint.com/csharp/csharp_jagged_arrays.htm)  C# supports multidimensional arrays, which are arrays of arrays. |
| 3 | [Passing arrays to functions](https://www.tutorialspoint.com/csharp/csharp_passing_arrays_to_functions.htm)  You can pass to the function a pointer to an array by specifying the array's name without an index. |
| 4 | [Param arrays](https://www.tutorialspoint.com/csharp/csharp_param_arrays.htm)  This is used for passing unknown number of parameters to a function. |
| 5 | [The Array Class](https://www.tutorialspoint.com/csharp/csharp_array_class.htm)  Defined in System namespace, it is the base class to all arrays, and provides various properties and methods for working with arrays. |

# C# - Strings

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

## Creating a String Object

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

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

## Properties of the String Class

The String class has the following two properties −

|  |  |
| --- | --- |
| **Sr.No.** | **Property & Description** |
| 1 | **Chars**  Gets the *Char* object at a specified position in the current *String* object. |
| 2 | **Length**  Gets the number of characters in the current String object. |

## Methods of the String Class

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

Given below is the list of methods of the String class.

You can visit MSDN library for the complete list of methods and String class constructors.

## Examples

The following example demonstrates some of the methods mentioned above −

### Comparing Strings

using System;

namespace StringApplication {

class StringProg {

static void Main(string[] args) {

string str1 = "This is test";

string str2 = "This is text";

if (String.Compare(str1, str2) == 0) {

Console.WriteLine(str1 + " and " + str2 + " are equal.");

} else {

Console.WriteLine(str1 + " and " + str2 + " are not equal.");

}

Console.ReadKey() ;

}

}

}

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

This is test and This is text are not equal.

### String Contains String

using System;

namespace StringApplication {

class StringProg {

static void Main(string[] args) {

string str = "This is test";

if (str.Contains("test")) {

Console.WriteLine("The sequence 'test' was found.");

}

Console.ReadKey() ;

}

}

}

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

The sequence 'test' was found.

### Getting a Substring

using System;

namespace StringApplication {

class StringProg {

static void Main(string[] args) {

string str = "Last night I dreamt of San Pedro";

Console.WriteLine(str);

string substr = str.Substring(23);

Console.WriteLine(substr);

}

}

}

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

San Pedro

### Joining Strings

using System;

namespace StringApplication {

class StringProg {

static void Main(string[] args) {

string[] starray = new string[]{"Down the way nights are dark",

"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"};

string str = String.Join("\n", starray);

Console.WriteLine(str);

}

}

}

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

Down the way nights are dark

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

# C# - Structures

In 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 can 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

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

**Class versus Structure**

Classes and Structures have the following basic differences −

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

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# - Enums

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

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

**Declaring *enum* Variable**

The general syntax for declaring an enumeration is −

enum <enum\_name> {

enumeration list

};

Where,

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

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

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

**Example**

The following example demonstrates use of enum variable −

using System;

namespace EnumApplication {

class EnumProgram {

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

static void Main(string[] args) {

int WeekdayStart = (int)Days.Mon;

int WeekdayEnd = (int)Days.Fri;

Console.WriteLine("Monday: {0}", WeekdayStart);

Console.WriteLine("Friday: {0}", WeekdayEnd);

Console.ReadKey();

}

}

}

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

Monday: 1

Friday: 5

# UNIT-II

# C# - Classes

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

**Defining a Class**

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

}

}

Note −

* 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 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 similar to 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 the 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

**C# Constructors**

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

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

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

**C# Destructors**

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

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

Following example explains the concept of destructor −

using System;

namespace LineApplication {

class Line {

private double length; // Length of a line

public Line() { // constructor

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

}

~Line() { //destructor

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

}

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());

}

}

}

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

Object is being created

Length of line : 6

Object is being deleted

**Static Members of a C# Class**

We can define class members as static using the **static** keyword. When we declare a member of a class as static, it means no matter how many objects of the class are created, there is only one copy of the static member.

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

The following example demonstrates the use of **static variables** −

using System;

namespace StaticVarApplication {

class StaticVar {

public static int num;

public void count() {

num++;

}

public int getNum() {

return num;

}

}

class StaticTester {

static void Main(string[] args) {

StaticVar s1 = new StaticVar();

StaticVar s2 = new StaticVar();

s1.count();

s1.count();

s1.count();

s2.count();

s2.count();

s2.count();

Console.WriteLine("Variable num for s1: {0}", s1.getNum());

Console.WriteLine("Variable num for s2: {0}", s2.getNum());

Console.ReadKey();

}

}

}

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

Variable num for s1: 6

Variable num for s2: 6

You can also declare a **member function** as **static**. Such functions can access only static variables. The static functions exist even before the object is created. The following example demonstrates the use of **static functions** −

using System;

namespace StaticVarApplication {

class StaticVar {

public static int num;

public void count() {

num++;

}

public static int getNum() {

return num;

}

}

class StaticTester {

static void Main(string[] args) {

StaticVar s = new StaticVar();

s.count();

s.count();

s.count();

Console.WriteLine("Variable num: {0}", StaticVar.getNum());

Console.ReadKey();

}

}

}

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

Variable num: 3

# C# - Inheritance

One of the most important concepts in object-oriented programming is 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 speeds up implementation time.

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

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 classes or interfaces.

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

<acess-specifier> class <base\_class> {

...

}

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

...

}

## Initializing Base Class

The derived class inherits the base class member variables and member methods. Therefore the super class object should be created before the subclass is created. You can give instructions for superclass initialization in the member initialization list.

The following program demonstrates this −

using System;

namespace RectangleApplication {

class Rectangle {

//member variables

protected double length;

protected double width;

public Rectangle(double l, double w) {

length = l;

width = w;

}

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 Tabletop : Rectangle {

private double cost;

public Tabletop(double l, double w) : base(l, w) { }

public double GetCost() {

double cost;

cost = GetArea() \* 70;

return cost;

}

public void Display() {

base.Display();

Console.WriteLine("Cost: {0}", GetCost());

}

}

class ExecuteRectangle {

static void Main(string[] args) {

Tabletop t = new Tabletop(4.5, 7.5);

t.Display();

Console.ReadLine();

}

}

}

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

Length: 4.5

Width: 7.5

Area: 33.75

Cost: 2362.5

## Multiple Inheritance in C#

**C# does not support multiple inheritance**. However, you can use interfaces to implement multiple inheritance. The following program demonstrates this −

using System;

namespace InheritanceApplication {

class Shape {

public void setWidth(int w) {

width = w;

}

public void setHeight(int h) {

height = h;

}

protected int width;

protected int height;

}

// Base class PaintCost

public interface PaintCost {

int getCost(int area);

}

// Derived class

class Rectangle : Shape, PaintCost {

public int getArea() {

return (width \* height);

}

public int getCost(int area) {

return area \* 70;

}

}

class RectangleTester {

static void Main(string[] args) {

Rectangle Rect = new Rectangle();

int area;

Rect.setWidth(5);

Rect.setHeight(7);

area = Rect.getArea();

// Print the area of the object.

Console.WriteLine("Total area: {0}", Rect.getArea());

Console.WriteLine("Total paint cost: ${0}" , Rect.getCost(area));

Console.ReadKey();

}

}

}

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

Total area: 35

Total paint cost: $2450

# C# - Polymorphism

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

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

**Static Polymorphism**

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

* Function overloading
* Operator overloading

We discuss operator overloading in next chapter.

**Function Overloading**

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

The following example shows using function **print()** to print different data types −

using System;

namespace PolymorphismApplication {

class Printdata {

void print(int i) {

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

}

void print(double f) {

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

}

void print(string s) {

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

}

static void Main(string[] args) {

Printdata p = new Printdata();

// Call print to print integer

p.print(5);

// Call print to print float

p.print(500.263);

// Call print to print string

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

Console.ReadKey();

}

}

}

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

Printing int: 5

Printing float: 500.263

Printing string: Hello C++

**Dynamic Polymorphism**

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

Here are the rules about abstract classes −

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

The following program demonstrates an abstract class −

using System;

namespace PolymorphismApplication {

abstract class Shape {

public abstract int area();

}

class Rectangle: Shape {

private int length;

private int width;

public Rectangle( int a = 0, int b = 0) {

length = a;

width = b;

}

public override int area () {

Console.WriteLine("Rectangle class area :");

return (width \* length);

}

}

class RectangleTester {

static void Main(string[] args) {

Rectangle r = new Rectangle(10, 7);

double a = r.area();

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

Console.ReadKey();

}

}

}

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

Rectangle class area :

Area: 70

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

Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.

The following program demonstrates this −

using System;

namespace PolymorphismApplication {

class Shape {

protected int width, height;

public Shape( int a = 0, int b = 0) {

width = a;

height = b;

}

public virtual int area() {

Console.WriteLine("Parent class area :");

return 0;

}

}

class Rectangle: Shape {

public Rectangle( int a = 0, int b = 0): base(a, b) {

}

public override int area () {

Console.WriteLine("Rectangle class area :");

return (width \* height);

}

}

class Triangle: Shape {

public Triangle(int a = 0, int b = 0): base(a, b) {

}

public override int area() {

Console.WriteLine("Triangle class area :");

return (width \* height / 2);

}

}

class Caller {

public void CallArea(Shape sh) {

int a;

a = sh.area();

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

}

}

class Tester {

static void Main(string[] args) {

Caller c = new Caller();

Rectangle r = new Rectangle(10, 7);

Triangle t = new Triangle(10, 5);

c.CallArea(r);

c.CallArea(t);

Console.ReadKey();

}

}

}

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

Rectangle class area:

Area: 70

Triangle class area:

Area: 25

# C# - Operator Overloading

You can redefine or overload most of the built-in operators available in C#. Thus a programmer can use operators with user-defined types as well. Overloaded operators are functions with special names the keyword **operator** followed by the symbol for the operator being defined. similar to any other function, an overloaded operator has a return type and a parameter list.

For example, go through the following function −

public static Box operator+ (Box b, Box c) {

Box box = new Box();

box.length = b.length + c.length;

box.breadth = b.breadth + c.breadth;

box.height = b.height + c.height;

return box;

}

The above function implements the addition operator (+) for a user-defined class Box. It adds the attributes of two Box objects and returns the resultant Box object.

## Implementing the Operator Overloading

The following program shows the complete implementation −

using System;

namespace OperatorOvlApplication {

class Box {

private double length; // Length of a box

private double breadth; // Breadth of a box

private double height; // Height of a box

public double getVolume() {

return length \* breadth \* height;

}

public void setLength( double len ) {

length = len;

}

public void setBreadth( double bre ) {

breadth = bre;

}

public void setHeight( double hei ) {

height = hei;

}

// Overload + operator to add two Box objects.

public static Box operator+ (Box b, Box c) {

Box box = new Box();

box.length = b.length + c.length;

box.breadth = b.breadth + c.breadth;

box.height = b.height + c.height;

return box;

}

}

class Tester {

static void Main(string[] args) {

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

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

Box Box3 = new Box(); // Declare Box3 of type Box

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

// 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);

// Add two object as follows:

Box3 = Box1 + Box2;

// volume of box 3

volume = Box3.getVolume();

Console.WriteLine("Volume of Box3 : {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

Volume of Box3 : 5400

## Overloadable and Non-Overloadable Operators

The following table describes the overload ability of the operators in C# −

|  |  |
| --- | --- |
| **Sr.No.** | **Operators & Description** |
| 1 | **+, -, !, ~, ++, --**  These unary operators take one operand and can be overloaded. |
| 2 | **+, -, \*, /, %**  These binary operators take one operand and can be overloaded. |
| 3 | **==, !=, <, >, <=, >=**  The comparison operators can be overloaded. |
| 4 | **&&, ||**  The conditional logical operators cannot be overloaded directly. |
| 5 | **+=, -=, \*=, /=, %=**  The assignment operators cannot be overloaded. |
| 6 | **=, ., ?:, ->, new, is, sizeof, typeof**  These operators cannot be overloaded. |

# C# - Interfaces

An interface is defined as a syntactical contract that all the classes inheriting the interface should follow. The interface defines the **'what'** part of the syntactical contract and the deriving classes define the **'how'** part of the syntactical contract.

Interfaces define properties, methods, and events, which are the members of the interface. Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members. It often helps in providing a standard structure that the deriving classes would follow.

Abstract classes to some extent serve the same purpose, however, they are mostly used when only few methods are to be declared by the base class and the deriving class implements the functionalities.

## Declaring Interfaces

Interfaces are declared using the interface keyword. It is similar to class declaration. Interface statements are public by default. Following is an example of an interface declaration −

public interface ITransactions {

// interface members

void showTransaction();

double getAmount();

}

## Example

The following example demonstrates implementation of the above interface −

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System;

namespace InterfaceApplication {

public interface ITransactions {

// interface members

void showTransaction();

double getAmount();

}

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

# C# - Exception Handling

An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.

* **try** − A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.
* **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
* **finally** − The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
* **throw** − A program throws an exception when a problem shows up. This is done using a throw keyword.

**Syntax**

Assuming a block raises an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following −

try {

// statements causing exception

} catch( ExceptionName e1 ) {

// error handling code

} catch( ExceptionName e2 ) {

// error handling code

} catch( ExceptionName eN ) {

// error handling code

} finally {

// statements to be executed

}

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

**Exception Classes in C#**

C# exceptions are represented by classes. The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class. Some of the exception classes derived from the System.Exception class are the **System.ApplicationException** and **System.SystemException** classes.

The **System.ApplicationException** class supports exceptions generated by application programs. Hence the exceptions defined by the programmers should derive from this class.

The **System.SystemException** class is the base class for all predefined system exception.

The following table provides some of the predefined exception classes derived from the Sytem.SystemException class −

|  |  |
| --- | --- |
| **Sr.No.** | **Exception Class & Description** |
| 1 | **System.IO.IOException**  Handles I/O errors. |
| 2 | **System.IndexOutOfRangeException**  Handles errors generated when a method refers to an array index out of range. |
| 3 | **System.ArrayTypeMismatchException**  Handles errors generated when type is mismatched with the array type. |
| 4 | **System.NullReferenceException**  Handles errors generated from referencing a null object. |
| 5 | **System.DivideByZeroException**  Handles errors generated from dividing a dividend with zero. |
| 6 | **System.InvalidCastException**  Handles errors generated during typecasting. |
| 7 | **System.OutOfMemoryException**  Handles errors generated from insufficient free memory. |
| 8 | **System.StackOverflowException**  Handles errors generated from stack overflow. |

**Handling Exceptions**

C# provides a structured solution to the exception handling in the form of try and catch blocks. Using these blocks the core program statements are separated from the error-handling statements.

These error handling blocks are implemented using the **try**, **catch**, and **finally** keywords. Following is an example of throwing an exception when dividing by zero condition occurs −

using System;

namespace ErrorHandlingApplication {

class DivNumbers {

int result;

DivNumbers() {

result = 0;

}

public void division(int num1, int num2) {

try {

result = num1 / num2;

} catch (DivideByZeroException e) {

Console.WriteLine("Exception caught: {0}", e);

} finally {

Console.WriteLine("Result: {0}", result);

}

}

static void Main(string[] args) {

DivNumbers d = new DivNumbers();

d.division(25, 0);

Console.ReadKey();

}

}

}

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

Exception caught: System.DivideByZeroException: Attempted to divide by zero.

at ...

Result: 0

**Creating User-Defined Exceptions**

You can also define your own exception. User-defined exception classes are derived from the **Exception** class. The following example demonstrates this −

using System;

namespace UserDefinedException {

class TestTemperature {

static void Main(string[] args) {

Temperature temp = new Temperature();

try {

temp.showTemp();

} catch(TempIsZeroException e) {

Console.WriteLine("TempIsZeroException: {0}", e.Message);

}

Console.ReadKey();

}

}

}

public class TempIsZeroException: Exception {

public TempIsZeroException(string message): base(message) {

}

}

public class Temperature {

int temperature = 0;

public void showTemp() {

if(temperature == 0) {

throw (new TempIsZeroException("Zero Temperature found"));

} else {

Console.WriteLine("Temperature: {0}", temperature);

}

}

}

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

TempIsZeroException: Zero Temperature found

**Throwing Objects**

You can throw an object if it is either directly or indirectly derived from the **System.Exception** class. You can use a throw statement in the catch block to throw the present object as −

Catch(Exception e) {

...

Throw e

}

# C# - Properties

**Properties** are named members of classes, structures, and interfaces. Member variables or methods in a class or structures are called **Fields**. Properties are an extension of fields and are accessed using the same syntax. They use **accessors** through which the values of the private fields can be read, written or manipulated.

Properties do not name the storage locations. Instead, they have **accessors** that read, write, or compute their values.

For example, let us have a class named Student, with private fields for age, name, and code. We cannot directly access these fields from outside the class scope, but we can have properties for accessing these private fields.

## Accessors

The **accessor** of a property contains the executable statements that helps in getting (reading or computing) or setting (writing) the property. The accessor declarations can contain a get accessor, a set accessor, or both. For example −

// Declare a Code property of type string:

public string Code {

get {

return code;

}

set {

code = value;

}

}

// Declare a Name property of type string:

public string Name {

get {

return name;

}

set {

name = value;

}

}

// Declare a Age property of type int:

public int Age {

get {

return age;

}

set {

age = value;

}

}

## Example

The following example demonstrates use of properties −

using System;

namespace tutorialspoint {

class Student {

private string code = "N.A";

private string name = "not known";

private int age = 0;

// Declare a Code property of type string:

public string Code {

get {

return code;

}

set {

code = value;

}

}

// Declare a Name property of type string:

public string Name {

get {

return name;

}

set {

name = value;

}

}

// Declare a Age property of type int:

public int Age {

get {

return age;

}

set {

age = value;

}

}

public override string ToString() {

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

}

}

class ExampleDemo {

public static void Main() {

// Create a new Student object:

Student s = new Student();

// Setting code, name and the age of the student

s.Code = "001";

s.Name = "Zara";

s.Age = 9;

Console.WriteLine("Student Info: {0}", s);

//let us increase age

s.Age += 1;

Console.WriteLine("Student Info: {0}", s);

Console.ReadKey();

}

}

}

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

Student Info: Code = 001, Name = Zara, Age = 9

Student Info: Code = 001, Name = Zara, Age = 10

# C# - Indexers

An **indexer** allows an object to be indexed such as an array. When you define an indexer for a class, this class behaves similar to 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. similar to the 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() {

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

namelist[i] = "N. A.";

}

public string this[int index] {

get {

string tmp;

if( index >= 0 && index <= size-1 ) {

tmp = namelist[index];

} else {

tmp = "";

}

return ( tmp );

}

set {

if( index >= 0 && index <= size-1 ) {

namelist[index] = value;

}

}

}

static void Main(string[] args) {

IndexedNames names = new IndexedNames();

names[0] = "Zara";

names[1] = "Riz";

names[2] = "Nuha";

names[3] = "Asif";

names[4] = "Davinder";

names[5] = "Sunil";

names[6] = "Rubic";

for ( int i = 0; i < IndexedNames.size; i++ ) {

Console.WriteLine(names[i]);

}

Console.ReadKey();

}

}

}

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

Zara

Riz

Nuha

Asif

Davinder

Sunil

Rubic

N. A.

N. A.

N. A.

# 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 has 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 is 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 similar to 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 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

## Using Delegates

The following example demonstrates the use of delegate. The delegate *printString* can be used to reference method that takes a string as input and returns 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

# C# - Events

**Events** are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.

## Using Delegates with Events

The events are declared and raised in a class and associated with the event handlers using delegates within the same class or some other class. The class containing the event is used to publish the event. This is called the **publisher** class. Some other class that accepts this event is called the **subscriber** class. Events use the **publisher-subscriber** model.

A **publisher** is an object that contains the definition of the event and the delegate. The event-delegate association is also defined in this object. A publisher class object invokes the event and it is notified to other objects.

A **subscriber** is an object that accepts the event and provides an event handler. The delegate in the publisher class invokes the method (event handler) of the subscriber class.

## Declaring Events

To declare an event inside a class, first a delegate type for the event must be declared. For example,

public delegate string MyDel(string str);

Next, the event itself is declared, using the **event** keyword −

event MyDel MyEvent;

The preceding code defines a delegate named *BoilerLogHandler* and an event named *BoilerEventLog*, which invokes the delegate when it is raised.

## Example

using System;

namespace SampleApp {

public delegate string MyDel(string str);

class EventProgram {

event MyDel MyEvent;

public EventProgram() {

this.MyEvent += new MyDel(this.WelcomeUser);

}

public string WelcomeUser(string username) {

return "Welcome " + username;

}

static void Main(string[] args) {

EventProgram obj1 = new EventProgram();

string result = obj1.MyEvent("Tutorials Point");

Console.WriteLine(result);

}

}

}

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

Welcome Tutorials Point

**UNIT – III**

## Using Windows Form Controls

In recent years, Visual Basic has won great acclaim for granting programmers the tools for creating highly detailed user interfaces via an intuitive form designer, along with an easy to learn programming language that together produced probably the best environment for rapid application development out there. One of the things that Visual Basic does, and other rapid application development tools, such as Delphi, also does, is provide access to a number of prefabricated controls that the developer can use to quickly build the user interface (UI) for an application.

At the center of most Visual Basic Windows applications stands the form designer. You create a user interface by dragging and dropping controls from a toolbox to your form, placing them where you want them to be when you run the program, and then double-clicking the control to add handlers for the control. The controls provided out of the box by Microsoft along with custom controls that can be bought at reasonable prices, have supplied programmers with an unprecedented pool of reusable, thoroughly tested code that is no further away than a click with the mouse. What was central to Visual Basic is now, through Visual Studio.NET, available to C# programmers.

Most of the controls used before .NET were, and still are, special COM objects, known as ActiveX controls. These are usually able to render themselves at both design and runtime. Each control has a number of properties allowing the programmer to do a certain amount of customization, such as setting the background color, caption, and its position on the form. The controls that we'll see in this chapter have the same look and feel as ActiveX controls, but they are not – they are .NET assemblies. However, it is still possible to use the controls that have been designed for older versions of Visual Studio but there is a small performance overhead because .NET has to wrap the control when you do so. For obvious reasons, when they designed .NET, Microsoft did not want to render the immense pool of existing controls redundant, and so have provided us with the means to use the old controls, even if future controls are built as pure .NET components.

These .NET assemblies can be designed in such a way that you will be able to use them in any of the Visual Studio languages, and the hope and belief is that the growing component industry will latch on, and start producing pure .NET components. We'll look at creating control ourselves in the next chapter.

An in depth explanation of .NET assemblies is provided in Chapter 21. Please refer to it if you want to know more about what an assembly is.

We have already seen the form designer in action, if only briefly, in the examples provided earlier in this book. In this chapter, we'll take a closer look at it, and especially how we use a number of controls, all of which come out of the box with Visual Studio.NET. Presenting all of the controls present in Visual Studio.NET will be an impossible task within the scope of this book, and so we'll be presenting the most commonly used controls, ranging from labelsand text boxes, to list views and status bars.

## The Windows Form Designer

We'll start out by taking a brief tour of the Windows Form Designer. This is the main playing ground when you are laying out your user interface. It is perfectly possible to design forms without using Visual Studio.NET, but designing an interface in Notepad can be a quite painful experience.

Let's look at the environment we'll be using. Start Visual Studio.NET and create a new C# Windows Application project by selecting File | New | Project. In the dialog that appears, click Visual C# Projects in the tree to the left and then select Windows Application in the list to the right. For now, simply use the default name suggested by Visual Studio and click OK. This should bring up a window much like the one shown below:

|  |
| --- |
| https://www.codeproject.com/KB/books/1861004982/image003.jpg |

If you are famliar with the forms designer found in Visual Basic you will notice the similarities – obviously someone desided that the designer was a winner and desided to allow it to be used it in other Visual Studio languages as well. If you are not familiar with the Visual Basic designer, then quite a few things are going on in the above screenshot, so let's take a moment and go through the panels one by one.

In the center of the screen is the form that you are designing. You can drag and drop controls from the toolbox onto the form. The toolbox is collapsed in the picture above, but if you move the mouse pointer to the far left of the screen over the Toolbox tab, it will unfold. You can then click the pin at the top right of the panel to pin it down. This will rearrange the work area so that the toolbox is now always on top, and isn't obscuring the form. We'll take a closer look at the toolbox and what it contains shortly.

Also collapsed on the left hand bar is the Server Explorer – represented by the computers icon on top of the toolbox tab. You can think of this as a small version of the Windows Control Panel. From here, you can browse computers on a network, add and remove database connections, and much more.

To the right of the window are two panels. The top-right one is the Solution Explorer and the class view. In the Solution Explorer, you can see all open projects and their associated files. By clicking the tab at the bottom of the Solution Explorer, you activate the Class Viewer. In this, you can browse all of the classes in your projects and all of the classes that they are derived from.

At the bottom right of the screen, is the Properties panel. This panel will contain all of the properties of the selected item for easy reference and editing. We'll be using this panel quite a bit in this chapter.

Also in this panel, the Dynamic Help tab is visible. This panel will show help tips to you for any selected objects and code even while you type. If your computer uses one of the older microprocessors or has a small amount of RAM, then I suggest that you remove this from the panel when it is not needed, as all that searching for help can make performance rather sluggish.

## The Toolbox

Let's have a closer look at the toolbox. If you haven't already, move your mouse pointer over the toolbox on the left of the screen, and pin it to the foreground by clicking the pin at the top right of the panel that unfolds:

If you accidentally remove the toolbox by clicking the X instead, you can make it reappear by selecting Toolbox from the View menu, or by pressing Ctrl-Alt-X.

|  |
| --- |
| https://www.codeproject.com/KB/books/1861004982/image005.jpg |

The toolbox contains a selection of all the controls available to you as a .NET developer. In particular, it provides the selection that is of importance to you as a Windows Application developer. If you had chosen to create a Web Forms project, rather than a Windows Application, you would have been given a different toolbox to use. You are not limited to use this selection. You can customize the toolbox to fit your needs, but in this chapter, we'll be focusing on the controls found in the selection that is shown in the picture above – in fact, we'll look at most of the controls that are shown here.

Now that we know where we'll be doing the work, let's look at controls in general.

## Controls

Most controls in .NET derive from the System.Windows.Forms.Control class. This class defines the basic functionality of the controls, which is why many properties and events in the controls we'll see are identical. Many of these classes are themselves base classes for other controls, as is the case with the Label and TextBoxBase classes in the diagram below:

|  |
| --- |
| https://www.codeproject.com/KB/books/1861004982/image007.gif |

Some controls, named custom or user controls, derive from another class: System.Windows.Forms.UserControl. This class is itself derived from the Control class and provides the functionality we need to create controls ourselves. We'll cover this class in Chapter 14. Incidentally, controls used for designing Web user interfaces derive from yet another class, System.Web.UI.Control.

### Properties

All controls have a number of properties that are used to manipulate the behavior of the control. The base class of most controls, Control, has a number of properties that other controls either inherit directly or override to provide some kind of custom behavior.

The table below shows some of the most common properties of the Control class. These properties will be present in most of the controls we'll visit in this chapter, and they will therefore, not be explained in detail again, unless the behavior of the properties is changed for the control in question. Note that this table is not meant to be exhaustive; if you want to see all f the properties in the class, please refer to the MSDN library:

|  |  |  |
| --- | --- | --- |
| Name | Availability | Description |
| Anchor | Read/Write | Using this property, you can specify how the control behaves when its container is resized. See below for a detailed explanation of this property. |
| BackColor | Read/Write | The background color of a control. |
| Bottom | Read/Write | By setting this property, you specify the distance from the top of the window to the bottom of the control. This is not the same as specifying the height of the control. |
| Dock | Read/Write | Allows you to make a control dock to the edges of a window. See below for a more detailed explanation of this property. |
| Enabled | Read/Write | Setting Enabled to true usually means that the control can receive input from the user. Setting Enabled to false usually means that it cannot. |
| ForeColor | Read/Write | The foreground color of the control. |
| Height | Read/Write | The distance from the top to the bottom of the control. |
| Left | Read/Write | The left edge of the control relative to the left edge of the window. |
| Name | Read/Write | The name of the control. This name can be used to reference the control in code. |
| Parent | Read/Write | The parent of the control. |
| Right | Read/Write | The right edge of the control relative to the left edge of the window. |
| TabIndex | Read/Write | The number the control has in the tab order of its container. |
| TabStop | Read/Write | Specifies whether the control can be accessed by the Tab key. |
| Tag | Read/Write | This value is usually not used by the control itself, and is there for you to store information about the control on the control itself. When this property is assigned a value through the Windows Form designer, you can only assign a string to it. |
| Top | Read/Write | The top edge of the control relative to the top of the window. |
| Visible | Read/Write | Specifies whether or not the control is visible  at runtime. |
| Width | Read/Write | The width of the control. |

### Anchor and Dock Properties

These two properties are especially useful when you are designing your form. Ensuring that a window doesn't become a mess to look at if the user decides to resize the window is far from trivial, and numerous lines of code have been written to achieve this. Many programs solve the problem by simply disallowing the window from being resized, which is clearly the easiest way around the problem, but not the best. The Anchor and Dock properties that have been introduced with .NET lets you solve this problem without writing a single line of code.

The Anchor property is used to to specify how the control behaves when a user resizes the window. You can specify if the control should resize itself, anchoring itself in proportion to its own edges, or stay the same size, anchoring its position relative to the window's edges.

The Dock property is related to the Anchor property. You can use it to specify that a control should dock to an edge of its container. If a user resizes the window, the control will continue to be docked to the edge of the window. If, for instance, you specify that a control should dock with the bottom of its container, the control will resize itself to always occupy the bottom part of the screen, no matter how the window is resized. The control will not be resized in the process; it simply stays docked to the edge of the window.

See the text box example later in this chapter for the exact use of the Anchor property.

### Events

When a user clicks a button or presses a button, you as the programmer of the application, want to be told that this has happened. To do so, controls use events. The Control class defines a number of events that are common to the controls we'll use in this chapter. The table below describes a number of these events. Once again, this is just a selection of the most common events; if you need to see the entire list, please refer to the MSDN library:

|  |  |
| --- | --- |
| Name | Description |
| Click | Occurs when a control is clicked. In some cases, this event will also occur when a user presses Enter. |
| DoubleClick | Occurs when a control is double-clicked. Handling the Click event on some controls, such as the Button control will mean that the DoubleClick event can never be called. |
| DragDrop | Occurs when a drag-and-drop operation is completed, in other words, when an object has been dragged over the control, and the user releases the mouse button. |
| DragEnter | Occurs when an object being dragged enters the bounds of the control. |
| DragLeave | Occurs when an object being dragged leaves the bounds of the control. |
| DragOver | Occurs when an object has been dragged over the control. |
| KeyDown | Occurs when a key becomes pressed while the control has focus. This event always occurs before KeyPress and KeyUp. |
| Name | Description |
| KeyPress | Occurs when a key becomes pressed, while a control has focus. This event always occurs after KeyDown and before KeyUp. The difference between KeyDown and KeyPress is that KeyDown passes the keyboard code of the key that has been pressed, while KeyPress passes the corresponding char value for the key. |
| KeyUp | Occurs when a key is released while a control has focus. This event always occurs after KeyDown and KeyPress. |
| GotFocus | Occurs when a control receives focus. Do not use this event to perform validation of controls. Use Validating and Validated instead. |
| LostFocus | Occurs when a control looses focus. Do not use this event to perform validation of controls. Use Validating and Validated instead. |
| MouseDown | Occurs when the mouse pointer is over a control and a mouse button is pressed. This is not the same as a Click event because MouseDown occurs as soon as the button is pressed and before it is released. |
| MouseMove | Occurs continually as the mouse travels over the control. |
| MouseUp | Occurs when the mouse pointer is over a control and a mouse button  is released. |
| Paint | Occurs when the control is drawn. |
| Validated | This event is fired when a control with the CausesValidation property set to true is about to receive focus. It fires after the Validating event finishes and indicates that validation is complete. |
| Validating | Fires when a control with the CausesValidation property set to true is about to receive focus. Note that the control which is to be validated is the control which is losing focus, not the one that is receiving it. |

We will see many of these events in the examples in the rest of the chapter.

We are now ready to start looking at the controls themselves, and we'll start with one that we've seen in previous chapters, the Button control.

## The Button Control

When you think of a button, you are probably thinking of a rectangular button that can be clicked to perform some task. However, technically there are three buttons in Visual Studio.NET. This is because radio buttons (as the name implies) and check boxes are also buttons. Because of this, the Button class is not derived directly from Control, but from another class called ButtonBase, which is derived from Control. We'll focus on the Button control in this section and leave radio buttons and check boxes for later in the chapter.

The button control exists on just about any Windows dialog you can think of. A button is primarily used to perform three kinds of tasks:

* To close a dialog with a state (for example, OK and Cancel buttons)
* To perform an action on data entered on a dialog (for example clicking Search efter entering some search criteria)
* To open another dialog or application (for example, Help buttons)

Working with the button control is very straightforward. It usually consists of adding the control to your form and double-clicking it to add the code to the Click event, which will probably be enough for most applications you'll work on.

Let's look at some of the commonly used properties and events of the control. This will give you an idea what can be done with it. After that, we'll create a small example that demonstrates some of the basic properties and events of a button.

### Button Properties

We'll list the properties as members of the button base, even if technically they are defined in the ButtonBase base class. Only the most commonly used properties are explained here. Please refer to MSDN for a complete listing:

|  |  |  |
| --- | --- | --- |
| Name | Availability | Description |
| FlatStyle | Read/Write | The style of the button can be changed with this property. If you set the style to PopUp, the button will appear flat until the user moves the mouse pointer over it. When that happens, the button pops up to its normal 3D look. |
| Enabled | Read/Write | We'll mention this here even though it is derived from Control, because it's a very important property for a button. Setting the Enabled property to false means that the button becomes grayed out and nothing happens when you click it. |
| Image | Read/Write | Allow you to specify an image (bitmap, icon etc.), which will be displayed on the button. |
| ImageAlign | Read/Write | With this property, you can set where the image on the button should appear. |

### Button Events

By far the most used event of a button is the Click event. This happens whenever a user clicks the button, by which we mean pressing the left mouse button and releasing it again while over the button. This means that if you left-click on the button and then draw the mouse away from the button before releasing it the Click event will not be raised. Also, the Click event is raised when the button has focus and the user press Enter. If you have a button on a form, you should always handle this event.

**The Label and LinkLabel Controls**

The Label control is probably the most used control of them all. Look at any Windows application and you'll see them on just about any dialog you can find. The label is a simple control with one purpose only: to present a caption or short hint to explain something on the form to the user.

Out of the box, Visual Studio.NET includes two label controls that are able to present them selves to the user in two distinct ways:

* Label, the standard Windows label
* LinkLabel, a label like the standard one (and derived from it), but presents itself as an internet link (a hyperlink)

The two controls are found at the top of the control panel on the Window Forms tab. In the picture below, one of each of the two types of Label have been dragged to a to illustrate the difference in appearance between the two:

|  |
| --- |
| https://www.codeproject.com/KB/books/1861004982/image015.jpg |

## The TextBox Control

Text boxes should be used when you want the user to enter text that you have no knowledge of at design time (for example the name of the user). The primary function of a text box is for the user to enter text, but any characters can be entered, and it is quite possible to force the user to enter numeric values only.

Out of the box .NET comes with two basic controls to take text input from the user: TextBox and RichTextBox (we'll discuss RichTextBox later in this chapter). Both controls are derived from a base class called TextBoxBase which itself is derived from Control.

TextBoxBase provides the base functionality for text manipulation in a text box, such as selecting text, cutting to and pasting from the Clipboard, and a wide range of events. We'll not focus so much now on what is derived from where, but instead look at the simpler of the two controls first – TextBox. We'll build one example that demonstrates the TextBox properties and build on that to demonstrate the RichTextBox control later.

### TextBox Properties

As has been stated earlier in this chapter, there are simply too many properties for us to describe them all, and so this listing includes only the most common ones:

|  |  |  |
| --- | --- | --- |
| Name | Availability | Description |
| CausesValidation | Read/Write | When a control that has this property set to true is about to receive focus, two events are fired: Validating and Validated. You can handle these events in order to validate data in the control that is losing focus. This may cause the control never to receive focus. The related events are discussed below. |
| CharacterCasing | Read/Write | A value indicating if the TextBox changes the case of the text entered. The possible  values are:  q        Lower: All text entered into the text box is converted lower case.  q        Normal: No changes are made to  the text.  q        Upper: All text entered into the text box is converted to upper case. |
| MaxLength | Read/Write | A value that specifies the maximum length in characters of any text, entered into the TextBox. Set this value to zero it the maximum limit is limited only by available memory. |
| Multiline | Read/Write | Indicates if this is a multiline control. A multiline control is able to show multiple lines of text. |
| PasswordChar | Read/Write | Specifies if a password character should replace the actual characters entered into a single line textbox. If the Multiline property is true then this has no effect. |
| ReadOnly | Read/Write | A Boolean indicating if the text is read only. |
| ScrollBars | Read/Write | Specifies if a multilane text box should display scrollbars. |
| SelectedText | Read/Write | The text that is selected in the text box. |
| SelectionLength | Read/Write | The number of characters selected in the text. If this value is set to be larger than the total number of characters in the text, it is reset by the control to be the total number of characters minus the value of SelectionStart. |
| SelectionStart | Read/Write | The start of the selected text in a text box. |
| WordWrap | Read/Write | Specifies if a multiline text box should automatically wrap words if a line exceeds the width of the control. |

### TextBox Events

Careful validation of the text in the TextBox controls on a form can make the difference between happy users and very angry ones.

You have probably experienced how annoying it is, when a dialog only validates its contents when you click OK. This approach to validating the data usually results in a message box being displayed informing you that the data in "TextBox number three" is incorrect. You can then continue to click OK until all the data is correct. Clearly this is not a good approach to validating data, so what can we do instead?

The answer lies in handling the validation events a TextBox control provides. If you want to make sure that invalid characters are not entered in the text box or only values within a certain range are allowed, then you will want to indocate to the user of the control whether the value engtered is valid or not.

The TextBox control provides these events (all of which are inherited from Control):

|  |  |
| --- | --- |
| Name | Description |
| Enter  GotFocus  Leave  Validating  Validated  LostFocus | These six events occur in the order they are listed here. They are known as "Focus Events" and are fired whenever a controls focus changes, with two exceptions. Validating and Validated are only fired if the control that receives focus has the CausesValidation property set to true. The reason why it's the receiving control that fires the event is that there are times where you do not want to validate the control, even if focus changes. An example of this is if the user clicks a Help button. |
| KeyDown  KeyPress  KeyUp | These three events are known as "Key Events". They allow you to monitor and change what is entered into your controls.  KeyDown and KeyUp receive the key code corresponding to the key that was pressed. This allows you to determine if special keys such as Shift or Control and F1 were pressed.  KeyPress, on the otherhand, receives the character corresponding to a keyboard key. This means that the value for the letter "a" is not the same as the letter "A". It is useful if you want to exclude a range of characters, for example, only allowing numeric values to be entered. |
| Change | Occurs whenever the text in the textbox is changed, no matter what  the change. |

**Example: Using Multiple Forms**

Write a program that uses the 2 forms shown to the right, and performs the following actions:

* + When it starts, it shows **Form1**
  + After clicking on **Go!** on**Form1**, it hides **Form1**, and displays **FormLogin**
  + The user enters his name and password into the appropriate textboxes
  + Afterwards, when the user clicks on the **Log In** button:
    - * If the name is “joe”, it checks if the entered password is “xyz”

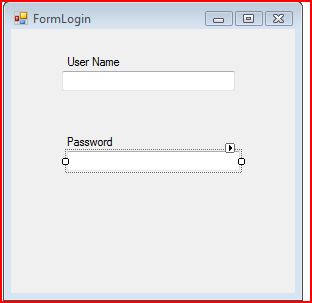
If the password is “xyz”, it hides **FormLogin** and displays **Form1**

If the password is *not* “xyz”, it ignores input (no change of display)

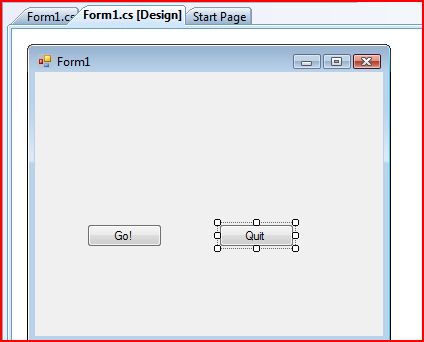
* + - * If the name is *not* “joe“, it clears both *User Name* and *Password*
  + After user clicks on **Quit** on **Form1**, it displays a message box with a “Thank you!” message.
  + It quits when the user closes the message box.

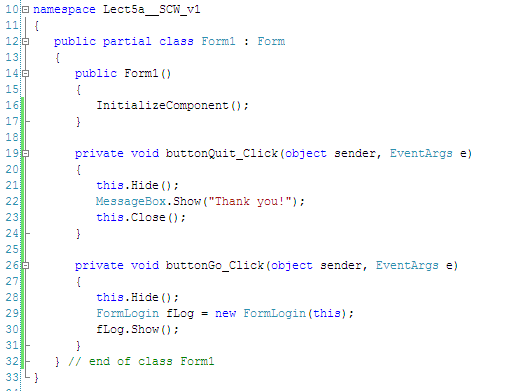
NOTE: 1) Code for the program example with 2 forms.txt

2) We will use the password textbox for “Password” (so the password will not be visible on screen; instead, a dot will be shown for each password character typed.)



**Log In**



**RECALL:**

* Afteruser clicks on **Quit** on **Form1**, the program:

- Hides **Form1**

**-** Displays a message box with a “Thank you!” message.

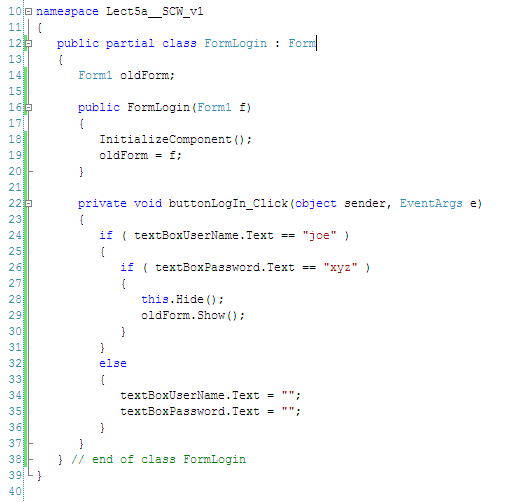
- Quits (closes) after the user closes the message box.

* After clicking on **Go!** on**Form1**, the program:

- Hides **Form1**

- Creates **FormLogin**

- Displays **FormLogin**

 **RECALL:**

The user enters his name and password into the appropriate textboxes on **FormLogin**

Afterwards, when the user clicks on the **Log In** button:

- If the name is “joe”, it checks if the entered password is “xyz”

- If the password is “xyz”, it hides **FormLogin** and displays **Form1**

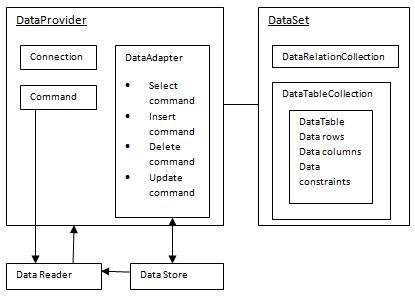
- Else (if the password is *not* “xyz”), it ignores input (no change of display)

- Else (if the name is *not* “joe“), it clears both *User Name* and *Password* textboxes.

# ADO.NET

ADO.NET provides a bridge between the front end controls and the back end database. The ADO.NET objects encapsulate all the data access operations and the controls interact with these objects to display data, thus hiding the details of movement of data.

The following figure shows the ADO.NET objects at a glance:



## The DataSet Class

The dataset represents a subset of the database. It does not have a continuous connection to the database. To update the database a reconnection is required. The DataSet contains DataTable objects and DataRelation objects. The DataRelation objects represent the relationship between two tables.

## he DataTable Class

The DataTable class represents the tables in the database. It has the following important properties; most of these properties are read only properties except the PrimaryKey property:

|  |  |
| --- | --- |
| **Properties** | **Description** |
| ChildRelations | Returns the collection of child relationship. |
| Columns | Returns the Columns collection. |
| Constraints | Returns the Constraints collection. |
| DataSet | Returns the parent DataSet. |
| DefaultView | Returns a view of the table. |
| ParentRelations | Returns the ParentRelations collection. |
| PrimaryKey | Gets or sets an array of columns as the primary key for the table. |
| Rows | Returns the Rows collection. |

## The DataAdapter Object

The DataAdapter object acts as a mediator between the DataSet object and the database. This helps the Dataset to contain data from multiple databases or other data source.

## The DataReader Object

The DataReader object is an alternative to the DataSet and DataAdapter combination. This object provides a connection oriented access to the data records in the database. These objects are suitable for read-only access, such as populating a list and then breaking the connection.

## DbCommand and DbConnection Objects

The DbConnection object represents a connection to the data source. The connection could be shared among different command objects.

The DbCommand object represents the command or a stored procedure sent to the database from retrieving or manipulating data.

//Example Program

amespace createdatabase

{

public partial class \_Default : System.Web.UI.Page

{

protected void Page\_Load(object sender, EventArgs e)

{

if (!IsPostBack)

{

DataSet ds = CreateDataSet();

GridView1.DataSource = ds.Tables["Student"];

GridView1.DataBind();

}

}

private DataSet CreateDataSet()

{

//creating a DataSet object for tables

DataSet dataset = new DataSet();

// creating the student table

DataTable Students = CreateStudentTable();

dataset.Tables.Add(Students);

return dataset;

}

private DataTable CreateStudentTable()

{

DataTable Students = new DataTable("Student");

// adding columns

AddNewColumn(Students, "System.Int32", "StudentID");

AddNewColumn(Students, "System.String", "StudentName");

AddNewColumn(Students, "System.String", "StudentCity");

// adding rows

AddNewRow(Students, 1, "M H Kabir", "Kolkata");

AddNewRow(Students, 1, "Shreya Sharma", "Delhi");

AddNewRow(Students, 1, "Rini Mukherjee", "Hyderabad");

AddNewRow(Students, 1, "Sunil Dubey", "Bikaner");

AddNewRow(Students, 1, "Rajat Mishra", "Patna");

return Students;

}

private void AddNewColumn(DataTable table, string columnType, string columnName)

{

DataColumn column = table.Columns.Add(columnName, Type.GetType(columnType));

}

//adding data into the table

private void AddNewRow(DataTable table, int id, string name, string city)

{

DataRow newrow = table.NewRow();

newrow["StudentID"] = id;

newrow["StudentName"] = name;

newrow["StudentCity"] = city;

table.Rows.Add(newrow);

}

}

}

**UNIT – IV**

ASP.NET is a web development platform, which provides a programming model, a comprehensive software infrastructure and various services required to build up robust web applications for PC, as well as mobile devices.

ASP.NET works on top of the HTTP protocol, and uses the HTTP commands and policies to set a browser-to-server bilateral communication and cooperation.

ASP.NET is a part of Microsoft .Net platform. ASP.NET applications are compiled codes, written using the extensible and reusable components or objects present in .Net framework. These codes can use the entire hierarchy of classes in .Net framework.

The ASP.NET application codes can be written in any of the following languages:

* C#
* Visual Basic.Net
* Jscript
* J#

ASP.NET is used to produce interactive, data-driven web applications over the internet. It consists of a large number of controls such as text boxes, buttons, and labels for assembling, configuring, and manipulating code to create HTML pages.

**ASP.NET Web Forms Model**

ASP.NET web forms extend the event-driven model of interaction to the web applications. The browser submits a web form to the web server and the server returns a full markup page or HTML page in response.

All client side user activities are forwarded to the server for stateful processing. The server processes the output of the client actions and triggers the reactions.

Now, HTTP is a stateless protocol. ASP.NET framework helps in storing the information regarding the state of the application, which consists of:

* Page state
* Session state

|  |
| --- |
| **Components and their Description** |
| **(1) Common Language Runtime or CLR**  It performs memory management, exception handling, debugging, security checking, thread execution, code execution, code safety, verification, and compilation. The code that is directly managed by the CLR is called the managed code. When the managed code is compiled, the compiler converts the source code into a CPU independent intermediate language (IL) code. A Just In Time(JIT) compiler compiles the IL code into native code, which is CPU specific. |
| **(2) .Net Framework Class Library**  It contains a huge library of reusable types. classes, interfaces, structures, and enumerated values, which are collectively called types. |
| **(3) Common Language Specification**  It contains the specifications for the .Net supported languages and implementation of language integration. |
| **(4) Common Type System**  It provides guidelines for declaring, using, and managing types at runtime, and cross-language communication. |
| **(5) Metadata and Assemblies**  Metadata is the binary information describing the program, which is either stored in a portable executable file (PE) or in the memory. Assembly is a logical unit consisting of the assembly manifest, type metadata, IL code, and a set of resources like image files. |
| **(6) Windows Forms**  Windows Forms contain the graphical representation of any window displayed in the application. |
| **(7) ASP.NET and ASP.NET AJAX**  ASP.NET is the web development model and AJAX is an extension of ASP.NET for developing and implementing AJAX functionality. ASP.NET AJAX contains the components that allow the developer to update data on a website without a complete reload of the page. |
| **(8) ADO.NET**  It is the technology used for working with data and databases. It provides access to data sources like SQL server, OLE DB, XML etc. The ADO.NET allows connection to data sources for retrieving, manipulating, and updating data. |
| **(9) Windows Workflow Foundation (WF)**  It helps in building workflow-based applications in Windows. It contains activities, workflow runtime, workflow designer, and a rules engine. |
| **(10) Windows Presentation Foundation**  It provides a separation between the user interface and the business logic. It helps in developing visually stunning interfaces using documents, media, two and three dimensional graphics, animations, and more. |
| **(11) Windows Communication Foundation (WCF)**  It is the technology used for building and executing connected systems. |
| **(12) Windows CardSpace**  It provides safety for accessing resources and sharing personal information on the internet. |
| **(13) LINQ**  It imparts data querying capabilities to .Net languages using a syntax which is similar to the tradition query language SQL. |

## ASP.NET Page Life Cycle

The page life cycle phases are:

* Initialization
* Instantiation of the controls on the page
* Restoration and maintenance of the state
* Execution of the event handler codes
* Page rendering
* The following code snippet provides a sample ASP.NET page explaining Page directives, code section and page layout written in C#:
* <!-- directives -->
* <% @Page Language="C#" %>
* <!-- code section -->
* <script runat="server">
* private void convertoupper(object sender, EventArgs e)
* {
* string str = mytext.Value;
* changed\_text.InnerHtml = str.ToUpper();
* }
* </script>
* <!-- Layout -->
* <html>
* <head>
* <title> Change to Upper Case </title>
* </head>
* <body>
* <h3> Conversion to Upper Case </h3>
* <form runat="server">
* <input runat="server" id="mytext" type="text" />
* <input runat="server" id="button1" type="submit" value="Enter..." OnServerClick="convertoupper"/>
* <hr />
* <h3> Results: </h3>
* <span runat="server" id="changed\_text" />
* </form>
* </body>
* </html>

# ASP.NET - Web Services

A web service is a web-based functionality accessed using the protocols of the web to be used by the web applications. There are three aspects of web service development:

* Creating the web service
* Creating a proxy
* Consuming the web service

**Creating a Web Service**

A web service is a web application which is basically a class consisting of methods that could be used by other applications. It also follows a code-behind architecture such as the ASP.NET web pages, although it does not have a user interface.

To understand the concept let us create a web service to provide stock price information. The clients can query about the name and price of a stock based on the stock symbol. To keep this example simple, the values are hardcoded in a two-dimensional array. This web service has three methods:

* A default HelloWorld method
* A GetName Method
* A GetPrice Method

Take the following steps to create the web service:

**Step (1)** : Select File -> New -> Web Site in Visual Studio, and then select ASP.NET Web Service.

**Step (2)** : A web service file called Service.asmx and its code behind file, Service.cs is created in the App\_Code directory of the project.

**Step (3)** : Change the names of the files to StockService.asmx and StockService.cs.

**Step (4)** : The .asmx file has simply a WebService directive on it:

<%@ WebService Language="C#" CodeBehind="~/App\_Code/StockService.cs" Class="StockService" %>

**Step (5)** : Open the StockService.cs file, the code generated in it is the basic Hello World service. The default web service code behind file looks like the following:

using System;

using System.Collections;

using System.ComponentModel;

using System.Data;

using System.Linq;

using System.Web;

using System.Web.Services;

using System.Web.Services.Protocols;

using System.Xml.Linq;

namespace StockService

{

// <summary>

// Summary description for Service1

// <summary>

[WebService(Namespace = "http://tempuri.org/")]

[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1\_1)]

[ToolboxItem(false)]

// To allow this Web Service to be called from script,

// using ASP.NET AJAX, uncomment the following line.

// [System.Web.Script.Services.ScriptService]

public class Service1 : System.Web.Services.WebService

{

[WebMethod]

public string HelloWorld()

{

return "Hello World";

}

}

}

**Step (6)** : Change the code behind file to add the two dimensional array of strings for stock symbol, name and price and two web methods for getting the stock information.

using System;

using System.Linq;

using System.Web;

using System.Web.Services;

using System.Web.Services.Protocols;

using System.Xml.Linq;

[WebService(Namespace = "http://tempuri.org/")]

[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1\_1)]

// To allow this Web Service to be called from script,

// using ASP.NET AJAX, uncomment the following line.

// [System.Web.Script.Services.ScriptService]

public class StockService : System.Web.Services.WebService

{

public StockService () {

//Uncomment the following if using designed components

//InitializeComponent();

}

string[,] stocks =

{

{"RELIND", "Reliance Industries", "1060.15"},

{"ICICI", "ICICI Bank", "911.55"},

{"JSW", "JSW Steel", "1201.25"},

{"WIPRO", "Wipro Limited", "1194.65"},

{"SATYAM", "Satyam Computers", "91.10"}

};

[WebMethod]

public string HelloWorld() {

return "Hello World";

}

[WebMethod]

public double GetPrice(string symbol)

{

//it takes the symbol as parameter and returns price

for (int i = 0; i < stocks.GetLength(0); i++)

{

if (String.Compare(symbol, stocks[i, 0], true) == 0)

return Convert.ToDouble(stocks[i, 2]);

}

return 0;

}

[WebMethod]

public string GetName(string symbol)

{

// It takes the symbol as parameter and

// returns name of the stock

for (int i = 0; i < stocks.GetLength(0); i++)

{

if (String.Compare(symbol, stocks[i, 0], true) == 0)

return stocks[i, 1];

}

return "Stock Not Found";

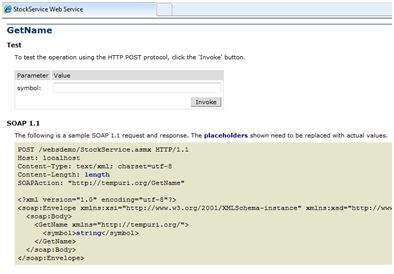
}

}

**Step (7)** : Running the web service application gives a web service test page, which allows testing the service methods.



**Step (8)** : Click on a method name, and check whether it runs properly.



**Step (9)** : For testing the GetName method, provide one of the stock symbols, which are hard coded, it returns the name of the stock



.