Introduction to Microsoft .NET

● Understanding Microsoft .NET 2015

● .NET 2015 Cross platform support

● Advantages of Microsoft .NET 2015

Introduction to .NET Framework

● Understanding .NET Framework

● .NET Framework Version History

● .NET Framework 4.5 Architecture

● Common Language Runtime (CLR)

● CLR Components

● Microsoft Intermediate language (MSIL)

● Common Type System (CTS)

● Common Language Specification (CLS)

● Relationship Between CTS and CLS

● Framework Class Library (FCL)

● Just In Time Compilation (JIT)

IDE Environment: Visual Studio 2015

Introduction to Visual Studio 2015

● Understanding Visual Studio

● Visual Studio Version History

● Understanding Visual Studio Editions

● Advantage of Visual Studio

● Understanding Visual Studio Features

● Visual Studio Explorer – Solution, Team, Server

● Coding and Debugging

● Types of Projects in Visual Studio

● Creating a Project and Solution

● Toolbox

● Property Window

C# Programming Language

Introduction to C#

● History of C# Version

● C# What,

Why & When

● Structure of a C# Program

● Basic Input / Output Operations

● Compiling, Running, and Debugging

Data Type, Variable & Typecasting

● Understanding Data Type

● Types of Data Type – Value Type & Reference Type

● Naming a variable

● Boxing and Unboxing

● Data Conversions – Implicit & Explicit

● Safe Type Casting with IS and AS Operator

Assemblies

● Types of Assemblies

● Shared Assembly, Private Assembly & Satellite Assembly

● Assembly Manifest, Metadata

● Assembly Cache

Namespaces

● Creating Namespace

● Advantages of Namespace

Garbage Collection

● Understanding Garbage Collection (GC)

● Advantage of Garbage Collector

● Generations in Garbage Collector

● Garbage Collector Working Phase

● Garbage Collection Algorithm

● Finalize and Dispose Method

**Operators:**

● Different Types of 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

## Arithmetic Operators

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

[Show Examples](http://www.tutorialspoint.com/csharp/csharp_arithmetic_operators.htm)

|  |  |  |
| --- | --- | --- |
| **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:

[Show Examples](http://www.tutorialspoint.com/csharp/csharp_relational_operators.htm)

|  |  |  |
| --- | --- | --- |
| **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:

[Show Examples](http://www.tutorialspoint.com/csharp/csharp_logical_operators.htm)

|  |  |  |
| --- | --- | --- |
| **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 |

|  |  |  |
| --- | --- | --- |
| **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. |  |

Assignment Operators

There are following assignment operators supported by C#:

[Show Examples](http://www.tutorialspoint.com/csharp/csharp_assignment_operators.htm)

|  |  |  |
| --- | --- | --- |
| **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 |

Miscillaneous Operators

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

[Show Examples](http://www.tutorialspoint.com/csharp/csharp_misc_operators.htm)

|  |  |  |
| --- | --- | --- |
| **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.

[Show Examples](http://www.tutorialspoint.com/csharp/csharp_operators_precedence.htm)

|  |  |  |
| --- | --- | --- |
| **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 |

Conditional Statements

● Introduction to conditional statement

A statement that can be executed based on a condition is known as a “Conditional Statement”. The statement is often a block of code.  
  
The following are the 2 types:

1. Conditional Branching
2. Conditional Looping

● If statements

● If..else statement

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

if(boolean\_expression)

{

/\* statement(s) will execute if the boolean expression is true \*/

}

else

{

/\* statement(s) will execute if the boolean expression is false \*/

}

● If..else ladder

if (totalMarks >= 80) {

MessageBox.Show("Got Higher First Class ");

}

else if (totalMarks >= 60) {

MessageBox.Show("Got First Class ");

}

else if (totalMarks >= 40){

MessageBox.Show("Just pass only");

}

else {

MessageBox.Show("Failed");

}

● Switch statement

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

## Syntax

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

switch(expression) {

case constant-expression :

statement(s);

break; /\* optional \*/

case constant-expression :

statement(s);

break; /\* optional \*/

/\* you can have any number of case statements \*/

default : /\* Optional \*/

statement(s);

}

Loops

● Introduction to loop

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.

● do...while loop

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

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

## Syntax

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

do

{

statement(s);

}while( condition );

● while loop

A **while** loop statement in C# repeatedly executes a target statement as long as a given condition is true.

## Syntax

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

while(condition)

{

statement(s);

}

● Making star patterns

Console.WriteLine();

for (int row = 10; row >= 1; --row)

{

for (int col = 1; col <= row; ++col)

{

Console.Write("\*");

}

Console.WriteLine();

}

Console.WriteLine();

for (int row = 10; row >= 1; --row)

{

for (int spaces = 0; spaces < 10-row; ++spaces)

{

Console.Write(" ");

}

for (int col = 1; col <= row; ++col)

{

Console.Write("\*");

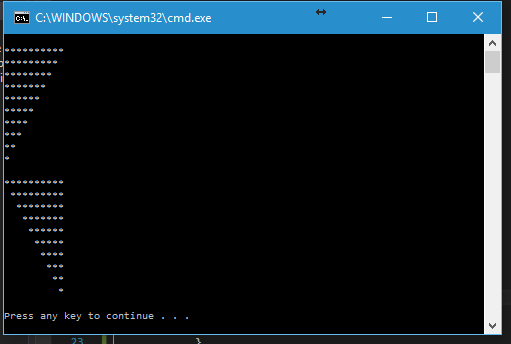
}

Console.WriteLine();

}

Console.WriteLine();

Output:



Jump Statements:

e break, goto, continue, return and throw statements are known as jump statements. These are used to transfer program control from one point in the program to another point, at any time. Let's understand how these work?

break statement

This statement terminates the execution of loop or switch in which it appears and transfers program control to the next statement which is placed immediately after the loop or switch.

1. **public class Example**
2. **{**
3. **static void Main(string[] args)**
4. **{**
5. **for (int i = 1; i <= 10; i++)**
6. **{**
7. **if (i == 5)**
8. **{**
9. **break;**
10. **}**
11. **Console.WriteLine(i);**
12. **}**
13. **Console.WriteLine("Next statement placed after loop");**
14. **}**
15. **}**
16. ***/\* Output:***
17. ***1***
18. ***2***
19. ***3***
20. ***4***
21. ***Next statement placed after loop***
22. ***\*/***

This statement is also used to terminates an inner nested loop, and return control to the outer loop.

goto statement

This statement transfers program control to a labeled statement. The label statement must exist in the scope of the goto statement. More than one goto statement can transfer control to the same label. This statement can be used to get out from a loop or an inner nested loop to outer loop.

1. **public class Example**
2. **{**
3. **static void Main(string[] args)**
4. **{**
5. **for (int i = 1; i <= 10; i++)**
6. **{**
7. **if (i == 5)**
8. **{**
9. **goto Exitlabel;**
10. **}**
11. **Console.WriteLine(i);**
12. **}**
13. **Console.WriteLine("Next statement placed after loop");**
15. **Exitlabel: *//goto label***
16. **Console.WriteLine("Labeled statement");**
17. **}**
18. **}**
19. ***/\* Output:***
20. ***1***
21. ***2***
22. ***3***
23. ***4***
24. ***Labeled statement***
25. ***\*/***

Unlike break statement, it does not transfer the program control to next statement which is placed immediately after the loop or switch.

You can also use goto statement to transfer control to a specific switch-case label or the default label in a switch statement.

It is not recommended to use goto statement since this makes the program logic complex and difficult to understand. It also becomes difficult to trace the control flow of a program execution.

continue statement

This statement skips the current iteration and passes program control to the next iteration of the enclosing loop in which it appears.

1. **public class Example**
2. **{**
3. **static void Main(string[] args)**
4. **{**
5. **for (int i = 1; i <= 10; i++)**
6. **{**
7. **if (i <= 5)**
8. **{**
9. **continue;**
10. **}**
11. **Console.WriteLine(i);**
12. **}**
13. **Console.WriteLine("Next statement placed after loop");**
14. **}**
15. **}**
16. ***/\* Output:***
17. ***6***
18. ***7***
19. ***8***
20. ***9***
21. ***10***
22. ***Next statement placed after loop***
23. ***\*/***

● break statements

The break statement is used to terminating the current flow of program and transfer controls to the next execution.

using System;  
   
namespace break\_statement  
{  
  class Program  
   {  
     static void Main(string[] args)  
      {  
        int i = 0;  
   
        while (i < 100)  
         {  
           Console.WriteLine(i);  
           if (i == 20)  
            {  
              Console.WriteLine("breaking current    segment...");  
              break;  
            }  
           i++;  
         }    
        Console.ReadLine();  
      }  
   }  
}

● continue statement

The **continue** statement in C# works somewhat like the **break** statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.

For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops,**continue** statement causes the program control passes to the conditional tests.

## Syntax

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

continue;

using System;

namespace Loops

{

class Program

{

static void Main(string[] args)

{

/\* local variable definition \*/

int a = 10;

/\* do loop execution \*/

do

{

if (a == 15)

{

/\* skip the iteration \*/

a = a + 1;

continue;

}

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

a++;

}

while (a < 20);

Console.ReadLine();

}

}

}

Loops Contd.

● for loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

## Syntax

The syntax of a **for** loop in C# is:

for ( init; condition; increment )

{

statement(s);

}

using System;

namespace Loops

{

class Program

{

static void Main(string[] args)

{

/\* for loop execution \*/

for (int a = 10; a < 20; a = a + 1)

{

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

}

Console.ReadLine();

}

}

}

● foreach loop

foreach loop is a different kind of looping constructs in C# programming that doesn’t includes initialization, termination and increment/decrement characteristics. It uses collection to take value one by one and then processes them.

**syntax:**

 foreach (string name in arr)  
 {

 }

Where, **name** is a string variable that takes value from collection as arr and then processes them in the body area.

using System;  
   
namespace foreach\_loop  
{  
  class Program  
   {  
     static void Main(string[] args)  
      {  
        string[] arr = new string[5]; // declaring array  
   
        //Storing value in array element  
        arr[0] = "Steven";  
        arr[1] = "Clark";  
        arr[2] = "Mark";  
        arr[3] = "Thompson";  
        arr[4] = "John";  
   
        //retrieving value using foreach loop  
        foreach (string name in arr)  
         {  
           Console.WriteLine("Hello " + name);  
         }  
        Console.ReadLine();  
      }  
   }  
}

● Making number patterns

class pyramid

{

public static void Main()

{

int num , space;

while(true)

{

Console.Write("Enter a number between 1 to 9 : ");

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

space=num-1;

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

{

for(space=1; space<=(num-i); space++)

{

Console.Write(" ");

}

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

{

Console.Write(j);

}

for(int k=(i-1); k>=1; k--)

{

Console.Write(k);

}

Console.WriteLine();

}

}

}

}

OUTPUT :

Enter a number between 1 to 9 : 5

1

121

12321

1234321

123454321

Arrays

● Introduction to Array

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.

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;

## 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();

}

}

}

● One Dimensional Array

● MultiDimensional Array

C# allows multidimensional arrays. Multi-dimensional arrays are also called rectangular array. You can declare a 2-dimensional array of strings as:

string [,] names;

or, a 3-dimensional array of int variables as:

int [ , , ] m;

## Two-Dimensional Arrays

The simplest form of the multidimensional array is the 2-dimensional array. A 2-dimensional array is a list of one-dimensional arrays.

A 2-dimensional array can be thought of as a table, which has x number of rows and y number of columns. Following is a 2-dimensional array, which contains 3 rows and 4 columns:

## Initializing Two-Dimensional Arrays

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

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

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

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

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

};

## Accessing Two-Dimensional Array Elements

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

int val = a[2,3];

The above statement takes 4th element from the 3rd row of the array. You can verify it in the above diagram. Let us check the program to handle a two dimensional array:

using System;

namespace ArrayApplication

{

class MyArray

{

static void Main(string[] args)

{

/\* an array with 5 rows and 2 columns\*/

int[,] a = new int[5, 2] {{0,0}, {1,2}, {2,4}, {3,6}, {4,8} };

int i, j;

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

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

{

for (j = 0; j < 2; j++)

{

Console.WriteLine("a[{0},{1}] = {2}", i, j, a[i,j]);

}

}

Console.ReadKey();

}

}

}

● Jagged Array no

need

Strings

● Introduction to 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

The following example demonstrates this:

using System;

namespace StringApplication

{

class Program

{

static void Main(string[] args)

{

//from string literal and string concatenation

string fname, lname;

fname = "Rowan";

lname = "Atkinson";

string fullname = fname + lname;

Console.WriteLine("Full Name: {0}", fullname);

//by using string constructor

char[] letters = { 'H', 'e', 'l', 'l','o' };

string greetings = new string(letters);

Console.WriteLine("Greetings: {0}", greetings);

//methods returning string

string[] sarray = { "Hello", "From", "Tutorials", "Point" };

string message = String.Join(" ", sarray);

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

//formatting method to convert a value

DateTime waiting = new DateTime(2012, 10, 10, 17, 58, 1);

string chat = String.Format("Message sent at {0:t} on {0:D}", waiting);

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

}

}

}

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

Full Name: Rowan Atkinson

Greetings: Hello

Message: Hello From Tutorials Point

Message: Message sent at 5:58 PM on Wednesday, October 10, 2012

● Mutable strings

We can assign multiple values to mutable string and object state can be altered.  
Best example of Mutable is stringbuilder object can be used. We can multiple concatenation to mutable string. Mutable have very good flexibility to change object state, mutable string can generate long string.  
We can use given below namespace to use mutable string in c#

|  |  |
| --- | --- |
|  | using System.Text;   //\*  namespace used to import string builder  StringBuilder strLong = null;   //\*  [Mutable String] example   strLong.Append("Technology");   strLong.Append("Crowds"); |

● Immutable strings

Immutable string can’t be alter, once we have assign a value to immutable object state can’t be changed. We can use System.String to use immutable string.

|  |  |
| --- | --- |
| 1  2 | String str = “Technology Crowds”;  //\*  Immutable String Example  string str = “Technology Crowds” + “Technology”; |

● Strings methods

## 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:

|  |  |
| --- | --- |
| **Sr.No** | **Methods** |
| 1 | **public static int Compare(string strA, string strB)**  Compares two specified string objects and returns an integer that indicates their relative position in the sort order. |
| 2 | **public static int Compare(string strA, string strB, bool ignoreCase )**  Compares two specified string objects and returns an integer that indicates their relative position in the sort order. However, it ignores case if the Boolean parameter is true. |
| 3 | **public static string Concat(string str0, string str1)**  Concatenates two string objects. |
| 4 | **public static string Concat(string str0, string str1, string str2)**  Concatenates three string objects. |
| 5 | **public static string Concat(string str0, string str1, string str2, string str3)**  Concatenates four string objects. |
| 6 | **public bool Contains(string value)**  Returns a value indicating whether the specified String object occurs within this string. |
| 7 | **public static string Copy(string str)**  Creates a new String object with the same value as the specified string. |
| 8 | **public void CopyTo(int sourceIndex, char[] destination, int destinationIndex, int count)**  Copies a specified number of characters from a specified position of the String object to a specified position in an array of Unicode characters. |
| 9 | **public bool EndsWith(string value)**  Determines whether the end of the string object matches the specified string. |
| 10 | **public bool Equals(string value)**  Determines whether the current String object and the specified String object have the same value. |
| 11 | **public static bool Equals(string a, string b)**  Determines whether two specified String objects have the same value. |
| 12 | **public static string Format(string format, Object arg0)**  Replaces one or more format items in a specified string with the string representation of a specified object. |
| 13 | **public int IndexOf(char value)**  Returns the zero-based index of the first occurrence of the specified Unicode character in the current string. |
| 14 | **public int IndexOf(string value)**  Returns the zero-based index of the first occurrence of the specified string in this instance. |
| 15 | **public int IndexOf(char value, int startIndex)**  Returns the zero-based index of the first occurrence of the specified Unicode character in this string, starting search at the specified character position. |
| 16 | **public int IndexOf(string value, int startIndex)**  Returns the zero-based index of the first occurrence of the specified string in this instance, starting search at the specified character position. |
| 17 | **public int IndexOfAny(char[] anyOf)**  Returns the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters. |
| 18 | **public int IndexOfAny(char[] anyOf, int startIndex)**  Returns the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters, starting search at the specified character position. |
| 19 | **public string Insert(int startIndex, string value)**  Returns a new string in which a specified string is inserted at a specified index position in the current string object. |
| 20 | **public static bool IsNullOrEmpty(string value)**  Indicates whether the specified string is null or an Empty string. |
| 21 | **public static string Join(string separator, params string[] value)**  Concatenates all the elements of a string array, using the specified separator between each element. |
| 22 | **public static string Join(string separator, string[] value, int startIndex, int count)**  Concatenates the specified elements of a string array, using the specified separator between each element. |
| 23 | **public int LastIndexOf(char value)**  Returns the zero-based index position of the last occurrence of the specified Unicode character within the current string object. |
| 24 | **public int LastIndexOf(string value)**  Returns the zero-based index position of the last occurrence of a specified string within the current string object. |
| 25 | **public string Remove(int startIndex)**  Removes all the characters in the current instance, beginning at a specified position and continuing through the last position, and returns the string. |
| 26 | **public string Remove(int startIndex, int count)**  Removes the specified number of characters in the current string beginning at a specified position and returns the string. |
| 27 | **public string Replace(char oldChar, char newChar)**  Replaces all occurrences of a specified Unicode character in the current string object with the specified Unicode character and returns the new string. |
| 28 | **public string Replace(string oldValue, string newValue)**  Replaces all occurrences of a specified string in the current string object with the specified string and returns the new string. |
| 29 | **public string[] Split(params char[] separator)**  Returns a string array that contains the substrings in the current string object, delimited by elements of a specified Unicode character array. |
| 30 | **public string[] Split(char[] separator, int count)**  Returns a string array that contains the substrings in the current string object, delimited by elements of a specified Unicode character array. The int parameter specifies the maximum number of substrings to return. |
| 31 | **public bool StartsWith(string value)**  Determines whether the beginning of this string instance matches the specified string. |
| 32 | **public char[] ToCharArray()**  Returns a Unicode character array with all the characters in the current string object. |
| 33 | **public char[] ToCharArray(int startIndex, int length)**  Returns a Unicode character array with all the characters in the current string object, starting from the specified index and up to the specified length. |
| 34 | **public string ToLower()**  Returns a copy of this string converted to lowercase. |
| 35 | **public string ToUpper()**  Returns a copy of this string converted to uppercase. |
| 36 | **public string Trim()**  Removes all leading and trailing white-space characters from the current String object. |

Object Oriented Programming Concepts

● Introduction to object oriented programming

OOP is a design philosophy. It stands for Object Oriented Programming. **O**bject-**O**riented **P**rogramming (OOP) uses a different set of programming languages than old procedural programming languages (C, Pascal, etc.). Everything in OOP is grouped as self sustainable "objects". Hence, you gain reusability by means of four main object-oriented programming concepts.

● Classes and Objects

A class is simply a representation of a type of object. It is the blueprint, or plan, or template, that describes the details of an object. A class is the blueprint from which the individual objects are created. Class is composed of three things: a name, attributes, and operations.

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

}

}

● 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.

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;

}

// Derived class

class Rectangle: Shape

{

public int getArea()

{

return (width \* height);

}

}

class RectangleTester

{

static void Main(string[] args)

{

Rectangle Rect = new Rectangle();

Rect.setWidth(5);

Rect.setHeight(7);

// Print the area of the object.

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

Console.ReadKey();

}

}

}

● 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

● Abstraction

**Abstraction**  
  
The word abstract means a concept or an idea not associated with any specific instance.  
  
In programming we apply the same meaning of abstraction by making classes not associated with any specific instance.  
  
The abstraction is done when we need to only inherit from a certain class, but not need to instantiate objects of that class. In such case the base  
class can be regarded as "Incomplete". Such classes are known as an "Abstract Base Class".  
  
**Abstract Base Class**  
  
There are some important points about Abstract Base Class :

1. An Abstract Base class can not be instantiated; it means the object of that class can not be created.
2. Class having abstract keyword and having, abstract keyword with some of its methods (not all) is known as an Abstract Base Class.
3. Class having Abstract keyword and having abstract keyword with all of its methods is known as pure Abstract Base Class.
4. The method of abstract class that has no implementation is known as "operation". It can be defined as abstract void method ();
5. An abstract class holds the methods but the actual implementation of those methods is made in derived class.

Lets have a look of this code!  
  
    abstract class animal  
    {  
        public abstract void eat();  
        public void sound()  
        {  
            Console.WriteLine("dog can sound");  
        }  
    }   
This is the Abstract Base Class, if I make both of its methods abstract then this class would become a pure Abstract Base Class.  
  
Now we derive a class of 'dog' from the class animal.  
  
    abstract class animal  
    {  
        public abstract void eat();  
        public void sound()  
        {  
            Console.WriteLine("dog can sound");  
        }  
    }  
    class dog : animal  
    {  
        public override void eat() { Console.WriteLine("dog can eat"); }  
    }  
  
Here you can see we have 2 methods in the Abstract Base Class, the method eat() has no implementation; that is why it is being declared as 'abstract' while the method sound() has its own body so it is not declared as 'abstract'.  
  
In the derived class we have the same name method but this method has it's body.  
  
We are doing abstraction here so that we can access the method of derived class without any trouble.  
  
Let's have a look!  
  
    class program  
    {  
        abstract class animal  
        {  
            public abstract void eat();  
            public void sound()  
            {  
                Console.WriteLine("dog can sound");  
            }  
        }  
        class dog : animal  
        {  
            public override void eat() { Console.WriteLine("dog can eat"); }  
        }  
        static void Main(string[] args)  
        {  
            dog mydog = new dog();  
            animal thePet = mydog;  
            thePet.eat();  
            mydog.sound();  
        }  
    }

Finally we created an Object 'mydog' of class dog, but we didn't instantiate any object of Abstract Base Class 'animal'.

● 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.

The following example illustrates this:

using System;

namespace RectangleApplication

{

class Rectangle

{

//member variables

public double length;

public double width;

public double GetArea()

{

return length \* width;

}

public void Display()

{

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

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

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

}

}//end class Rectangle

class ExecuteRectangle

{

static void Main(string[] args)

{

Rectangle r = new Rectangle();

r.length = 4.5;

r.width = 3.5;

r.Display();

Console.ReadLine();

}

}

}

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

Length: 4.5

Width: 3.5

Area: 15.75

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

The member function *Display()* and *GetArea()* can also access these variables directly without using any instance of the class.

The member functions *Display()* is also declared **public**, so it can also be accessed from *Main()* using an instance of the Rectangle class, named **r**.

Private Access Specifier

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

The following example illustrates this:

using System;

namespace RectangleApplication

{

class Rectangle

{

//member variables

private double length;

private double width;

public void Acceptdetails()

{

Console.WriteLine("Enter Length: ");

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

Console.WriteLine("Enter Width: ");

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

}

public double GetArea()

{

return length \* width;

}

public void Display()

{

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

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

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

}

}//end class Rectangle

class ExecuteRectangle

{

static void Main(string[] args)

{

Rectangle r = new Rectangle();

r.Acceptdetails();

r.Display();

Console.ReadLine();

}

}

}

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

Enter Length:

4.4

Enter Width:

3.3

Length: 4.4

Width: 3.3

Area: 14.52

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

Protected Access Specifier

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

Internal Access Specifier

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

The following program illustrates this:

using System;

namespace RectangleApplication

{

class Rectangle

{

//member variables

internal double length;

internal double width;

double GetArea()

{

return length \* width;

}

public void Display()

{

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

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

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

}

}//end class Rectangle

class ExecuteRectangle

{

static void Main(string[] args)

{

Rectangle r = new Rectangle();

r.length = 4.5;

r.width = 3.5;

r.Display();

Console.ReadLine();

}

}

}

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

Length: 4.5

Width: 3.5

Area: 15.75

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

Protected Internal Access Specifier

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

Class and Objects

● Creating a class

● Access Modifiers

● Instance members

# What’s the difference between a class variable and an instance variable?

|  |
| --- |
|  |

Knowing the terminology is important. Instance variables and class variables are both member variables. They are both member variables because they are both associated with a **specific** class. But, there are differences between instance variables and class variables.

## Instance variables

Instance variables belong to an instance of a class. Another way of saying that is instance variables belong to an object, since an object is an instance of a class. Every object has it’s own copy of the instance variables. Here is what a declaration of an instance variable would look like:

## Example of an instance variable:

class Taxes

{

int count;

/\*...\*/

}

## Class variables – also known as static member variables

Class variables, however, only have **one** copy of the variable(s) shared with all instances of the class. It’s important to remember that **class variables are also known as static member variables** in C++, Java, and C#. Each object of the class does not have its own copy of a class variable. Instead, every object shares the **one and only** copy of that class variable – and any changes made to that copy are seen by all of the objects of that class. Here is what a class variable – or a static member variable – would look like in C++:

## Example of a class variable:

class Taxes

{

static int count;

/\*...\*/

}

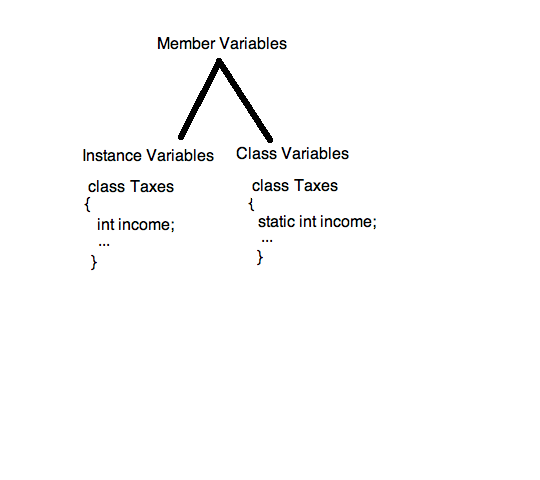
## Difference between class and instance variables

|  |
| --- |
|  |

Now, it should be clear what the difference between instance and class variables is. Class variables only have one copy that is shared by all the different objects of a class, whereas every object has it’s own personal copy of an instance variable. So, instance variables across different objects can have different values whereas class variables across different objects can have only one value.

## Class and Instance variables are both Member variables

Here’s a little diagram to help you remember the differences between instance and class variables:



● Creating an object

Constructor and Destructor

● Introduction to Constructor

Broadly speaking, a constructor is a method in the class which gets executed when its object is created. Usually, we put the initialization code in the constructor. Writing a constructor in the class is damn simple, have a look at the following sample:

Hide   Copy Code

public class mySampleClass

{

public mySampleClass()

{

*// This is the constructor method.*

}

*// rest of the class members goes here.*

}

When the object of this class is instantiated, this constructor will be executed. Something like this:

Hide   Copy Code

mySampleClass obj = new mySampleClass()

*// At this time the code in the constructor will // be executed*

● Types of Constructor

**onstructors can be divided into 5 types:**

1. Default Constructor
2. Parametrized Constructor
3. Copy Constructor
4. Static Constructor
5. Private Constructor

Now let us see  each constructor type with example as below

**Default Constructor**  
A constructor without any parameters is called a default constructor; in other words this type of constructor does not take parameters. The drawback of a default constructor is that every instance of the class will be initialized to the same values and it is not possible to initialize each instance of the class to different values. The default constructor initializes:

1. All numeric fields in the class to zero.
2. All string and object fields to null.

**Example**

using System;  
namespace DefaultConstractor  
 {  
    class addition  
    {  
        int a, b;

        public addition()   //default contructor

        {

            a = 100;

            b = 175;

        }

        public static void Main()

        {

            addition obj = new addition(); //an object is created , constructor is called

            Console.WriteLine(obj.a);

            Console.WriteLine(obj.b);

            Console.Read();

        }

      }

    }

**Parameterized Constructor**

A constructor with at least one parameter is called a parametrized constructor. The advantage of a parametrizedconstructor is that you can initialize each instance of the class to different values.

using System;

namespace Constructor

{  
    class paraconstrctor  
    {  
      public  int a, b;  
      public paraconstrctor(int x, int y)  // decalaring Paremetrized Constructor with ing x,y parameter

        {

            a = x;

            b = y;

        }

   }

    class MainClass

    {

        static void Main()

        {

            paraconstrctor v = new paraconstrctor(100, 175);   // Creating object of Parameterized Constructor and ing values

            Console.WriteLine("-----------parameterized constructor example by vithal wadje---------------");

            Console.WriteLine("\t");

            Console.WriteLine("value of a=" + v.a );

            Console.WriteLine("value of b=" + v.b);

            Console.Read();

        }

    }

}

**Copy Constructor**  
The constructor which creates an object by copying variables from another object is called a copy constructor. The purpose of a copy constructor is to initialize a new instance to the values of an existing instance.  
  
**Syntax**  
public employee(employee emp)  
{  
name=emp.name;  
age=emp.age;  
}  
  
The copy constructor is invoked by instantiating an object of type employee and ing it the object to be copied.  
  
**Example**

employee emp1=new  employee (emp2);

Now, emp1 is a copy of emp2.   
So let us see its practical implementation.

using System;

namespace copyConstractor

{

    class employee

    {

        private string name;

        private int age;

        public employee(employee emp)   // declaring Copy constructor.

        {

            name = emp.name;

            age = emp.age;

        }

        public employee(string name, int age)  // Instance constructor.

        {

            this.name = name;

            this.age = age;

        }

        public string Details     // Get deatils of employee

        {

            get

            {

                return  " The age of " + name +" is "+ age.ToString();

            }

        }

    }

    class empdetail

    {

        static void Main()

        {

            employee emp1 = new employee("Vithal", 23);  // Create a new employee object.

            employee emp2 = new employee(emp1);         **// here is emp1 details is copied to emp2.**

            Console.WriteLine(emp2.Details);

            Console.ReadLine();

        }

    }

}

**Static Constructor**  
When a constructor is created as static, it will be invoked only once for all of instances of the class and it is invoked during the creation of the first instance of the class or the first reference to a static member in the class. A static constructor is used to initialize static fields of the class and to write the code that needs to be executed only once.  
  
**Some key points of a static constructor is:**

1. A static constructor does not take access modifiers or have parameters.
2. A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.
3. A static constructor cannot be called directly.
4. The user has no control on when the static constructor is executed in the program.
5. A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.

**Syntax**  
class employee  
 {// Static constructor  
  static employee(){}  
 }

Now let us see it with practically

using System;

namespace staticConstractor

{

public class employee

{

    static employee() // Static constructor declaration{Console.WriteLine("The static constructor ");

}

public static void Salary()

 {

    Console.WriteLine();

    Console.WriteLine("The Salary method");

 }

}

class details

{

    static void Main()

    {

        Console.WriteLine("----------Static constrctor example by vithal wadje------------------");

        Console.WriteLine();

        employee.Salary();

        Console.ReadLine();

    }

  }

}

**Private Constructor**

When a constructor is created with a private specifier, it is not possible for other classes to derive from this class,

neither is it possible to create an instance of this class. They are usually used in classes that contain static members

only. Some key points of a private constructor are:

1. One use of a private constructor is when we have only static members.
2. It provides an implementation of a singleton class pattern
3. Once we provide a constructor that is either private or public or any, the compiler will not add the parameter-less public constructor to the class.

Now let us see it practically.

using System;

namespace defaultConstractor

{

    public class Counter

    {

        private Counter()   //private constrctor declaration

        {

        }

        public static int currentview;

        public static int visitedCount()

        {

            return ++ currentview;

        }

    }

    class viewCountedetails

    {

        static void Main()

        {

            // Counter aCounter = new Counter();   // Error

            Console.WriteLine("-------Private constructor example by vithal wadje----------");

            Console.WriteLine();

            Counter.currentview = 500;

            Counter.visitedCount();

            Console.WriteLine("Now the view count is: {0}", Counter.currentview);

            Console.ReadLine();

        }

    }

}

● Introduction to Destructor

**Destructor.** A destructor runs after a class becomes unreachable. It has the special "~" character in its name. The exact time it is executed is not specified. But it always runs when the class is not reachable in memory by any references.  
**Example.** Let's begin by looking at this Example class. It contains a constructor "Example()" and a destructor "~Example()". The destructor in a class must be prefixed with the tilde "~" character.  
**The class Example** is instantiated in the Main method. We write the method type to the console. The output, below, shows that the constructor is run and then the destructor is run before the program exits.[**Console.WriteLine**](http://www.dotnetperls.com/console)

**C# program that uses destructor**

using System;

class Example

{

public Example()

{

Console.WriteLine("Constructor");

}

**~Example()**

{

Console.WriteLine("Destructor");

}

}

class Program

{

static void Main()

{

Example x = new Example();

}

}

**Output**

Constructor

Destructor

Methods

● Introduction to 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

}

● Methods Categories

● Methods Parameters

There are 3 types:

* [params](https://msdn.microsoft.com/en-us/library/w5zay9db.aspx)

By using the **params** keyword, you can specify a [method parameter](https://msdn.microsoft.com/en-us/library/8f1hz171.aspx) that takes a variable number of arguments.

You can send a comma-separated list of arguments of the type specified in the parameter declaration or an array of arguments of the specified type. You also can send no arguments. If you send no arguments, the length of the **params** list is zero.

No additional parameters are permitted after the **params** keyword in a method declaration, and only one **params** keyword is permitted in a method declaration.

public class MyClass

{

public static void UseParams(params int[] list)

{

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

{

Console.Write(list[i] + " ");

}

Console.WriteLine();

}

public static void UseParams2(params object[] list)

{

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

{

Console.Write(list[i] + " ");

}

Console.WriteLine();

}

static void Main()

{

// You can send a comma-separated list of arguments of the

// specified type.

UseParams(1, 2, 3, 4);

UseParams2(1, 'a', "test");

// A params parameter accepts zero or more arguments.

// The following calling statement displays only a blank line.

UseParams2();

// An array argument can be passed, as long as the array

// type matches the parameter type of the method being called.

int[] myIntArray = { 5, 6, 7, 8, 9 };

UseParams(myIntArray);

object[] myObjArray = { 2, 'b', "test", "again" };

UseParams2(myObjArray);

// The following call causes a compiler error because the object

// array cannot be converted into an integer array.

//UseParams(myObjArray);

// The following call does not cause an error, but the entire

// integer array becomes the first element of the params array.

UseParams2(myIntArray);

}

}

/\*

Output:

1 2 3 4

1 a test

5 6 7 8 9

2 b test again

System.Int32[]

\*/

* [ref](https://msdn.microsoft.com/en-us/library/14akc2c7.aspx)

The **ref** keyword causes an argument to be passed by reference, not by value. The effect of passing by reference is that any change to the parameter in the called method is reflected in the calling method.

class RefExample

{

static void Method(ref int i)

{

// Rest the mouse pointer over i to verify that it is an int.

// The following statement would cause a compiler error if i

// were boxed as an object.

i = i + 44;

}

static void Main()

{

int val = 1;

Method(ref val);

Console.WriteLine(val);

// Output: 45

}

}

* [out](https://msdn.microsoft.com/en-us/library/t3c3bfhx.aspx)

class OutExample

{

static void Method(out int i)

{

i = 44;

}

static void Main()

{

int value;

Method(out value);

// value is now 44

}

}

● Optional Parameters

**Optional Parameters.** An optional parameter has a default value. A method with an optional parameter can be called with only some of its parameters specified. Using this feature in new versions of the C# language, we add default values for formal parameters.  
**Example.** In this example, we introduce a method named "Method" that has two parameters. Each of the parameters is optional. To specify an optional parameter, assign the formal parameter in the method parameter list to an appropriate value.  
**Here,** we set the formal parameter 'value' to 1, and the formal parameter 'name' to "Perl". Whenever Method is called without a parameter specified, its default value is used instead in the method body.

**C# program that uses optional parameters**

using System;

class Program

{

static void Main()

{

// Omit the optional parameters.

Method();

// Omit second optional parameter.

Method(4);

// You can't omit the first but keep the second.

// Method("Dot");

// Classic calling syntax.

Method(4, "Dot");

// Specify one named parameter.

Method(name: "Sam");

// Specify both named parameters.

Method(value: 5, name: "Allen");

}

static void Method(**int value = 1**, **string name = "Perl"**)

{

Console.WriteLine("value = {0}, name = {1}", value, name);

}

}

**Output**

value = 1, name = Perl

value = 4, name = Perl

value = 4, name = Dot

value = 1, name = Sam

value = 5, name = Allen

● Call By Value

* **Input parameter-**This kind of parameter is specify to give the same input to method   for calling it.

There  are two type of parameter

1. Call by value
2. Call by reference

* **Call by value  ->**In this case when we call the method of any **class** **(**which takes some **parameter)** from**main method**using object**.T**hen value of parameter in **main** method will directly copy to the **class**method to parameter values respectively. In this   case if some changes occurs in values within the method that **change not occurs** in actual variable .I have full describe this concept through programming which is given below.
* **Call by reference ->** In this case when we call the method,the**reference address** of variable is passed to the method.If some changes occurs in values within the method that changes occurs in **actual variable.**To specify this parameter we use**'ref'**Keyword at the time of parameter declaration as well as the calling method.

|  |
| --- |
| * namespace callbyvalue |

|  |  |
| --- | --- |
| 03 | { |

|  |  |
| --- | --- |
| 04 | class Program |

|  |  |
| --- | --- |
| 05 | { |

|  |  |
| --- | --- |
| 06 | public class employee |

|  |  |
| --- | --- |
| 07 | { |

|  |  |
| --- | --- |
| 08 | public void display(int a, String b) |

|  |  |
| --- | --- |
| 09 | { |

|  |  |
| --- | --- |
| 10 | Console.WriteLine("Integer value is"+" " +a); |

|  |  |
| --- | --- |
| 11 | Console.WriteLine(" String value is" + " " + b); |

|  |  |
| --- | --- |
| 12 | Console.ReadLine(); |

|  |  |
| --- | --- |
| 13 | } |

|  |  |
| --- | --- |
| 14 |  |

|  |  |
| --- | --- |
| 15 | } |

|  |  |
| --- | --- |
| 16 | public class student |

|  |  |
| --- | --- |
| 17 | { |

|  |  |
| --- | --- |
| 18 | public void show(ref String str) |

|  |  |
| --- | --- |
| 19 | { |

|  |  |
| --- | --- |
| 20 | Console.WriteLine("Enter the value"); |

|  |  |
| --- | --- |
| 21 | string s = Console.ReadLine(); |

|  |  |
| --- | --- |
| 22 | str = str + s; |

|  |  |
| --- | --- |
| 23 | Console.WriteLine("value in str variable is"+" "+str); |

|  |  |
| --- | --- |
| 24 | Console.ReadLine(); |

|  |  |
| --- | --- |
| 25 | } |

|  |  |
| --- | --- |
| 26 |  |

|  |  |
| --- | --- |
| 27 | } |

|  |  |
| --- | --- |
| 28 | //all class member is called through main method. |

|  |  |
| --- | --- |
| 29 |  |

|  |  |
| --- | --- |
| 30 | static void Main(string[] args) |

|  |  |
| --- | --- |
| 31 | { |

|  |  |
| --- | --- |
| 32 | //creating the object of employee class first and implementing the call by value concept. |

|  |  |
| --- | --- |
| 33 | String m = "sunil"; |

|  |  |
| --- | --- |
| 34 | employee emp = new employee(); |

|  |  |
| --- | --- |
| 35 | emp.display(200,m); |

|  |  |
| --- | --- |
| 36 | Console.WriteLine("value in variable m is" +" "+m); |

|  |  |
| --- | --- |
| 37 | Console.ReadLine(); |

|  |  |
| --- | --- |
| 38 |  |

|  |  |
| --- | --- |
| 39 | //creating the object of employee class first and implementing the call by Reference concept |

|  |  |
| --- | --- |
| 40 | string msg="Hello"; |

|  |  |
| --- | --- |
| 41 | student st = new student(); |

|  |  |
| --- | --- |
| 42 | st.show(ref msg); |

|  |  |
| --- | --- |
| 43 | Console.WriteLine("value in msg is" +" "+msg);//value at address msg will  be print,because here address is copy not value thatswhy at same address value will be print |

|  |  |
| --- | --- |
| 44 | Console.ReadLine(); |

|  |  |
| --- | --- |
| 45 | } |

|  |  |
| --- | --- |
| 46 | } |

|  |  |
| --- | --- |
| 47 | } |

Polymorphism

● Introduction to Polymorphism

● Types of Polymorphism

● Methods Overloading

### **Static or Compile Time Polymorphism**

In static polymorphism, the decision is made at compile time.

* Which method is to be called is decided at compile-time only.
* Method overloading is an example of this.
* Compile time polymorphism is method overloading, where the compiler knows which overloaded method it is going to call.

Method overloading is a concept where a class can have more than one method with the same name and different parameters.

Compiler checks the type and number of parameters passed on to the method and decides which method to call at compile time and it will give an error if there are no methods that match the method signature of the method that is called at compile time.

#### Example

Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png   Copy Code

namespace MethodOverloadingByManishAgrahari

{

class Program

{

public class TestOverloading

{

public void Add(string a1, string a2)

{

Console.WriteLine("Adding Two String :" + a1 + a2);

}

public void Add(int a1, int a2)

{

Console.WriteLine("Adding Two Integer :" + a1 + a2);

}

}

static void Main(string[] args)

{

TestOverloading obj = new TestOverloading();

obj.Add("Manish " , "Agrahari");

obj.Add(5, 10);

Console.ReadLine();

}

}

}

### **Dynamic or Runtime Polymorphism**

Run-time polymorphism is achieved by method overriding.

Method overriding allows us to have methods in the base and derived classes with the same name and the same parameters.

By runtime polymorphism, we can point to any derived class from the object of the base class at runtime that shows the ability of runtime binding.

Through the reference variable of a base class, the determination of the method to be called is based on the object being referred to by reference variable.

Compiler would not be aware whether the method is available for overriding the functionality or not. So compiler would not give any error at compile time. At runtime, it will be decided which method to call and if there is no method at runtime, it will give an error.

See the following example:

Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png   Copy Code

namespace PolymorphismByManishAgrahari

{

class Program

{

public class Base

{

public virtual void Show()

{

Console.WriteLine("Show From Base Class.");

}

}

public class Derived : Base

{

public override void Show()

{

Console.WriteLine("Show From Derived Class.");

}

}

static void Main(string[] args)

{

Base objBase;

objBase = new Base();

objBase.Show();*// Output ----> Show From Base Class.*

objBase = new Derived();

objBase.Show();*//Output--> Show From Derived Class.*

Console.ReadLine();

}

}

}

Compiler demands virtual Show() method and it compiles successfully. The right version of Show() method cannot be determined until run-time since only at that time Base objBase is initialized as Derived.

● Abstract, override and new keywords

**Abstract.** A method can be abstract. A class can be abstract. An abstract method has no implementation. Its implementation logic is provided instead by classes that derive from it.

**Methods:**Abstract methods cannot have bodies. This makes sense: these bodies would never be used.

**Classes:**Abstract classes have certain restrictions. They cannot be constructed directly.

**C# program that uses abstract class**

using System;

**abstract** class Test

{

public int \_a;

public **abstract** void A();

}

class Example1 : Test

{

public override void A()

{

Console.WriteLine("Example1.A");

base.\_a++;

}

}

class Example2 : Test

{

public override void A()

{

Console.WriteLine("Example2.A");

base.\_a--;

}

}

class Program

{

static void Main()

{

// Reference Example1 through Test type.

Test test1 = new Example1();

test1.A();

// Reference Example2 through Test type.

Test test2 = new Example2();

test2.A();

}

}

**Output**

Example1.A

Example2.A

**Override** affects virtual method usage. Virtual methods are meant to be re-implemented in derived classes. The override keyword specifies that a method replaces its virtual base method.

In the example, the class A is the base class. It has the virtual method Y.

[**Virtual**](https://www.dotnetperls.com/virtual)

**And:**In class B, we override Y. In class C, we implement Y but do not specify that it overrides the base method.

[**Class**](https://www.dotnetperls.com/class)

**C# program that uses override modifier**

using System;

class A

{

public virtual void Y()

{

// Used when C is referenced through A.

Console.WriteLine("A.Y");

}

}

class B : A

{

public **override** void Y()

{

// Used when B is referenced through A.

Console.WriteLine("B.Y");

}

}

class C : A

{

public void Y() // Can be "new public void Y()"

{

// Not used when C is referenced through A.

Console.WriteLine("C.Y");

}

}

class Program

{

static void Main()

{

// Reference B through A.

A ab = new B();

ab.Y();

// Reference C through A.

A ac = new C();

ac.Y();

}

}

**Output**

B.Y

A.Y

### **Virtual Keyword**

Virtual keyword is used for generating a virtual path for its derived classes on implementing method overriding. Virtual keyword is used within a set with override keyword. It is used as:

Hide   Copy Code

*// Base Class*

class A

{

public virtual void show()

{

Console.WriteLine("Hello: Base Class!");

Console.ReadLine();

}

}

### **Override Keyword**

Override keyword is used in the derived class of the base class in order to override the base class method.Override keyword is used with virtual keyword, as:

Hide   Copy Code

*// Base Class*

class A

{

public virtual void show()

{

Console.WriteLine("Hello: Base Class!");

Console.ReadLine();

}

}

*// Derived Class*

class B : A

{

public override void show()

{

Console.WriteLine("Hello: Derived Class!");

Console.ReadLine();

}

}

### **New Keyword**

New keyword is also used in polymorphism concept, but in the case of method overloading So what does overloading means, in simple words we can say procedure of hiding your base class through your derived class.

It is implemented as:

Hide   Copy Code

class A

{

public void show()

{

Console.WriteLine("Hello: Base Class!");

Console.ReadLine();

}

}

class B : A

{

public new void show()

{

Console.WriteLine("Hello: Derived Class!");

Console.ReadLine();

}

}

Inheritance

● Introduction to Inheritance

the ability to create classes which inherits certain aspects from parent classes. The entire .NET framework is built on this concept, with the "everything is an object" as a result of it. Even a simple number is an instance of a class, which inherits from the System.Object class, although .NET helps you out a bit, so you can assign a number directly, instead of having to create a new instance of e.g. the integer class.

public class Animal

{

public virtual void Greet()

{

Console.WriteLine("Hello, I'm some sort of animal!");

}

}

public class Dog : Animal

{

public override void Greet()

{

Console.WriteLine("Hello, I'm a dog!");

}

● Types of inheritance

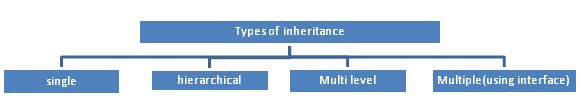
● Single Level Inheritance

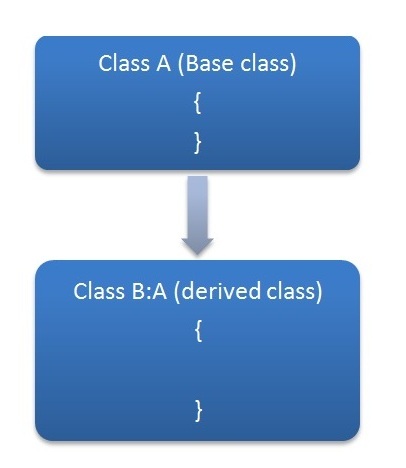
● MultiLevel

Inheritance

● Multiple Inheritance

**What Inheritance is**

Acquiring (taking) the properties of one class into another class is called inheritance. Inheritance provides reusability by allowing us to extend an existing class.    
The reason behind OOP programming is to promote the reusability of code and to reduce complexity in code and it is possible by using inheritance.  
   
The following are the types of inheritance in C#.  
  
  
   
The inheritance concept is based on a base class and derived class. Let us see the definition of a base and derived class.

**Base class:** is the class from which features are to be inherited into another class.  
**Derived class:** it is the class in which the base class features are inherited.  
  
**Single inheritance**   
It is the type of inheritance in which there is one base class and one derived class.   
   
  
For example:

public class Accountcreditinfo //base class

{

    public string Credit()

    {

        return "balance is credited";

    }

}

public class debitinfo : Accountcreditinfo //derived class

{

    public string debit()

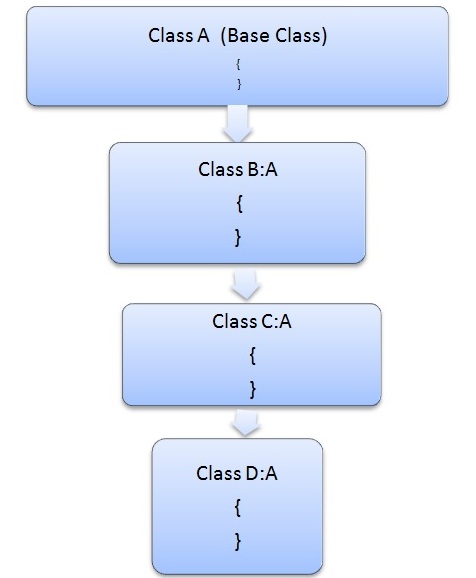
    {

        Credit();                       ////derived class method

        return "balance is debited";

    }

}

In the preceding sample program Accountcreditinfo is the base class and debitinfo is the derived class.  
  
**Hierarchical inheritance**   
This is the type of inheritance in which there are multiple classes derived from one base class. This type of inheritance is used when there is a requirement of one class feature that is needed in multiple classes.  
   
  
  
For example:

class A  //base class

{

    public string msg()

    {

        return "this is A class Method";

    }

}

class B : A

{

    public string info()

    {

        msg();

        return "this is B class Method";

    }

    class C : A

    {

        public string getinfo()

        {

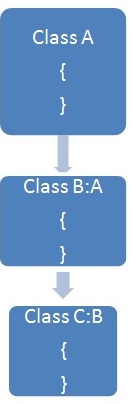
            msg();

            return "this is B class Method";

        }

    }

}

In the preceding program one base class is derived in many classes hence it is a called a Hierarchical Inheritance.  
   
**Multilevel inheritance**  
When one class is derived from another derived class then this type of inheritance is called multilevel inheritance.  
   
  
  
For example:

public class Person

{

    public string persondet()

    {

        return "this is the person class";

    }

}

public class Bird : Person

{

    public string birddet()

    {

        persondet();

        return "this is the birddet Class";

    }

}

public class Animal : Bird

{

    public string animaldet()

    {

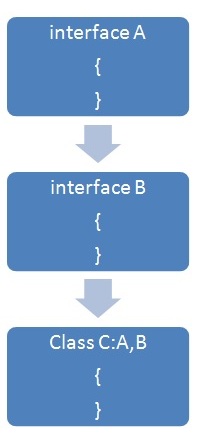
        persondet();

        birddet();

        return "this is the Animal Class";

    }

}

In the preceding program, each class is derived from one class that is derived from another class hence this type of inheritance is called Multilevel Inheritance.  
   
**Multiple inheritance using Interfaces**   
C# does not support multiple inheritances of classes. To overcome this problem we can use interfaces, we will see more about interfaces in my next article in detail.  
   
  
  
For example:

public interface IA //ineterface  1

{

    string setImgs(string a);

}

public interface IB  //Interface 2

{

    int getAmount(int Amt);

}

public class ICar : IA, IB //implementatin

{

    public int getAmount(int Amt)

    {

        return 100;

    }

    public string setImgs(string a)

    {

        return "this is the car";

    }

}

In the preceding program the ICar class inherits the features of the two interfaces hence this type of inheritance is called Multiple Inheritance.  
   
The following are some key points about inheritance:

1. C# does not support multiple inheritances of classes, the same thing can be done using interfaces.
2. Private members are not accessed in a derived class when one class is derived from another.

● Hybrid Inheritance: Multilevel, multiple inheritance:

● Advantage of Inheritance

Once a behavior (method) or property is defined in a super class(base class),that behavior or property is automatically inherited by all subclasses (derived class).

         Code reusability increased through inheritance.

         Inheritance provide a clear model structure which is easy to understand without much complexity Using inheritance, classes become grouped together in a hierarchical tree structure Code are easy to manage and divided into parent and child classes.

● Sealed class and private constructor

-------------------------+---+--------+--------+--------+----------+

| Class Type | | normal | static | sealed | abstract |

+-------------------------+---+--------+--------+--------+----------+

| Can be instantiated | : | YES | NO | YES | NO |

| Can be inherited | : | YES | NO | NO | YES |

| Can inherit from others | : | YES | NO | YES | YES |

+-------------------------+---+--------+--------+--------+----------+

Sealed classes are used to restrict the inheritance feature of object oriented programming. Once a class is defined as a **sealed class,** the class cannot be inherited.

 The main purpose of a sealed class is to take away the inheritance feature from the user so they cannot derive a class from a sealed class. One of the best usage of sealed classes is when you have a class with static members. For example, the "Pens" and "Brushes" classes of the "System.Drawing" namespace.

Private constructor:

**Private Constructor.** A private constructor cannot be externally called. It is used to ensure higher-quality code bases on complex projects. A private constructor forces the class to provide a controlled and unified access pattern.

Private constructor is a special instance constructor used in a class that contains static member only. If a class has one or more private constructor and no public constructor then other classes is not allowed to create instance of this class this mean we can neither create the object of the class nor it can be inherit by other class. The main purpose of creating private constructor is used to restrict the class from being instantiated when it contains every member as static.

|  |
| --- |
| using System;  namespace ConsoleApplication3  {  public class Sample  {  public string param1, param2;  public Sample(string a,string b)  {  param1 = a;  param2 = b;  }  private Sample()  // Private Constructor Declaration  {  Console.WriteLine("Private Constructor with no prameters");  }  }  class Program  {  static void Main(string[] args)  {  // Here we don't have chance to create instace for private constructor  Sample obj = new Sample("Welcome","to Aspdotnet-Suresh");  Console.WriteLine(obj.param1 +" " + obj.param2);  Console.ReadLine();  }  }  } |

Property

● Introduction to Property

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called accessors. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.

class TimePeriod

{

private double seconds;

public double Hours

{

get { return seconds / 3600; }

set { seconds = value \* 3600; }

}

}

class Program

{

static void Main()

{

TimePeriod t = new TimePeriod();

// Assigning the Hours property causes the 'set' accessor to be called.

t.Hours = 24;

// Evaluating the Hours property causes the 'get' accessor to be called.

System.Console.WriteLine("Time in hours: " + t.Hours);

}

}

● Types of Property

Properties provide the convenience of public data members without the risks that come with unprotected, uncontrolled, and unverified access to an object's data. This is accomplished through accessors: special methods that assign and retrieve values from the underlying data member. The [set](https://msdn.microsoft.com/en-us/library/ms228368.aspx)accessor enables data members to be assigned, and the [get](https://msdn.microsoft.com/en-us/library/ms228503.aspx) accessor retrieves data member values.

● ReadWrite Property

class Person

{

private string name = "N/A";

private int age = 0;

// Declare a Name property of type string:

public string Name

{

get

{

return name;

}

set

{

name = value;

}

}

● Read Only Property

Omitting the **set** accessor, for example, makes the property read-only:

public string Name

{

get

{

return name;

}

}

● Write Only Property

class User

{

public string Password

{

set { /\* password encryption here \*/ }

}

}

● Advantages of Property

they are supposed to provide controlled access to the fields of our class. As the state of the class depends upon the values of its fields, using properties we can assure that no invalid (or unacceptable) value is assigned to the fields.

one more big big advantage of having the Properties are to have a control over the freedom to allow either write or read alone right to the class using this property with the object instantiated,

Indexer

● Introduction to Indexer

Indexers allow instances of a class or struct to be indexed just like arrays. Indexers resemble [properties](https://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx) except that their accessors take parameters.

● Types of Indexer

● Advantages of Indexer

Abstract Class

● Introduction to Abstract Class

 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

● Creating Abstract class

● Need of Abstract class

<http://www.codeproject.com/Articles/6118/All-about-abstract-classes>

● Advantages of abstract class

Interface

● Introduction to Interface

**Interface**s in C # provide a way to achieve runtime polymorphism. Using **interface**s we can invoke functions from different classes through the same **Interface** reference, whereas using virtual functions we can invoke functions from different classes in the same inheritance hierarchy through the same reference. Before things start getting difficult let me start using simple and short examples to explain the concept of **interface**s. Here's a short example that shows you what an **interface** looks like.

● Creating Interface

<http://www.codeproject.com/Articles/18743/Interfaces-in-C-For-Beginners>

● Need of Interface

● Abstract class vs. Interface

### **What is an Abstract Class?**

An abstract class is a special kind of class that cannot be instantiated. So the question is why we need a class that cannot be instantiated? An abstract class is only to be sub-classed (inherited from). In other words, it only allows other classes to inherit from it but cannot be instantiated. The advantage is that it enforces certain hierarchies for all the subclasses. In simple words, it is a kind of contract that forces all the subclasses to carry on the same hierarchies or standards.

### **What is an Interface?**

An interface is not a class. It is an entity that is defined by the word Interface. An interface has no implementation; it only has the signature or in other words, just the definition of the methods without the body. As one of the similarities to Abstract class, it is a contract that is used to define hierarchies for all subclasses or it defines specific set of methods and their arguments. The main difference between them is that a class can implement more than one interface but can only inherit from one abstract class. Since C# doesn’t support multiple inheritance, interfaces are used to implement multiple inheritance.

### **Both Together**

When we create an interface, we are basically creating a set of methods without any implementation that must be overridden by the implemented classes. The advantage is that it provides a way for a class to be a part of two classes: one from inheritance hierarchy and one from the interface.

When we create an abstract class, we are creating a base class that might have one or more completed methods but at least one or more methods are left uncompleted and declared abstract. If all the methods of an abstract class are uncompleted then it is same as an interface. The purpose of an abstract class is to provide a base class definition for how a set of derived classes will work and then allow the programmers to fill the implementation in the derived classes.

There are some similarities and differences between an interface and an abstract class that I have arranged in a table for easier comparison:

| **Feature** | **Interface** | **Abstract class** |
| --- | --- | --- |
| Multiple inheritance | A class may inherit several interfaces. | A class may inherit only one abstract class. |
| Default implementation | An interface cannot provide any code, just the signature. | An abstract class can provide complete, default code and/or just the details that have to be overridden. |
| Access Modfiers | An interface cannot have access modifiers for the subs, functions, properties etc everything is assumed as public | An abstract class can contain access modifiers for the subs, functions, properties |
| Core VS Peripheral | Interfaces are used to define the peripheral abilities of a class. In other words both Human and Vehicle can inherit from a IMovable interface. | An abstract class defines the core identity of a class and there it is used for objects of the same type. |
| Homogeneity | If various implementations only share method signatures then it is better to use Interfaces. | If various implementations are of the same kind and use common behaviour or status then abstract class is better to use. |
| Speed | Requires more time to find the actual method in the corresponding classes. | Fast |
| Adding functionality (Versioning) | If we add a new method to an Interface then we have to track down all the implementations of the interface and define implementation for the new method. | If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly. |
| Fields and Constants | No fields can be defined in interfaces | An abstract class can have fields and constrants defined |

You can implement multiple interfaces, but only inherit from one abstract class.

An **interface** is an empty shell, there are only the signatures (name / params / return type) of the methods. The methods do not contain anything. The interface can't do anything. It's just a pattern

**Abstract classes**, unlike interfaces, are classes. There are more expensive to use because there is a lookup to do when you inherit from them.

Abstract classes look a lot like interfaces, but they have something more : you can define a behavior for them. It's more about a guy saying "these classes should look like that, and they got that in common, so fill in the blanks!".

Static Class

● Introduction to static class

**Static class.** A static class is never instantiated. The static keyword on a class enforces that a type not be created with a constructor. This eliminates misuse of the class.

**Note:**A static class cannot have non-static members. All methods, fields and properties in it must also be static.

● Creating static class and static methods

class Program

{

static void Main()

{

// Cannot declare a variable of type Perl.

// This won't blend.

// Perl perl = new Perl();

// Program is a regular class so you can create it.

Program program = new Program();

// You can call static methods inside a static class.

Perl.\_ok = true;

Perl.Blend();

}

}

**static** class Perl

{

// Cannot declare instance members in a static class!

// int \_test;

// This is ok.

public **static** bool \_ok;

// Can only have static methods in static classes.

public **static** void Blend()

{

Console.WriteLine("Blended");

}

}

**Output**

Blended

● Need of static class

A [static](https://msdn.microsoft.com/en-us/library/98f28cdx.aspx) class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated.

The following list provides the main features of a static class:

* Contains only static members.
* Cannot be instantiated.
* Is sealed.
* Cannot contain [Instance Constructors](https://msdn.microsoft.com/en-us/library/k6sa6h87.aspx).

Creating a static class is therefore basically the same as creating a class that contains only static members and a private constructor. A private constructor prevents the class from being instantiated. The advantage of using a static class is that the compiler can check to make sure that no instance members are accidentally added.

● Normal class vs. Static class

Static classes are sealed and therefore cannot be inherited. They cannot inherit from any class except [Object](https://msdn.microsoft.com/en-us/library/system.object.aspx). Static classes cannot contain an instance constructor; however, they can contain a static constructor. Non-static classes should also define a static constructor if the class contains static members that require non-trivial initialization.

* static classes cannot be instantiated or inherited.
* static classes are marked as sealed and abstract by compiler in the output MSIL.
* all members of static classes must be static as well.
* only static classes can host extension methods.
* static classes cannot be used as generic type arguments.

Partial Class

● Introduction to partial class

Each class in C# resides in a separate physical file with a .cs extension. C# provides the ability to have a single class implementation in multiple .cs files using the ***partial*** modifier [keyword](http://www.tutorialsteacher.com/csharp/csharp-keywords). The*partial* modifier can be applied to a class, method, interface or structure.

● Creating partial class

For example, the following MyPartialClass splits into two files, PartialClassFile1.cs and PartialClassFile2.cs:

PartialClassFile1.cs:

public partial class MyPartialClass

{

public MyPartialClass()

{

}

public void Method1(int val)

{

Console.WriteLine(val);

}

}

PartialClassFile2.cs:

public partial class MyPartialClass

{

public void Method2(int val)

{

Console.WriteLine(val);

}

}

MyPartialClass in PartialClassFile1.cs defines the constructor and one public method, Method1, whereas PartialClassFile2 has only one public method, Method2. The compiler combines these two partial classes into one class as below:

Partial class:

public class MyGenericClass

{

public MyPartialClass()

{

}

public void Method1(int val)

{

Console.WriteLine(val);

}

public void Method2(int val)

{

Console.WriteLine(val);

}

}

### **Partial Class Requirements:**

* All the partial class definitions must be in the same assembly and namespace.
* All the parts must have the same accessibility like public or private, etc.
* If any part is declared abstract, sealed or base type then the whole class is declared of the same type.
* Different parts can have different base types and so the final class will inherit all the base types.
* The PartialÂ modifier can only appear immediately before the keywordsÂ class,Â struct, orÂ interface.
* Nested partial types are allowed.

### **Advantages of Partial class:**

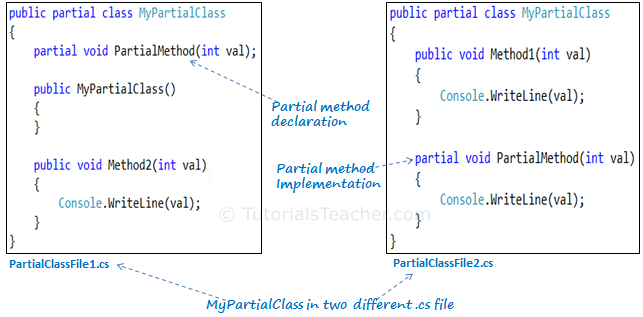
* Multiple developers can work simultaneously with a single class in separate files.
* When working with automatically generated source, code can be added to the class without having to recreate the source file. For example, Visual Studio separates HTML code for the UI and server side code into two separate files: .aspx and .cs files.

● Need of partial class

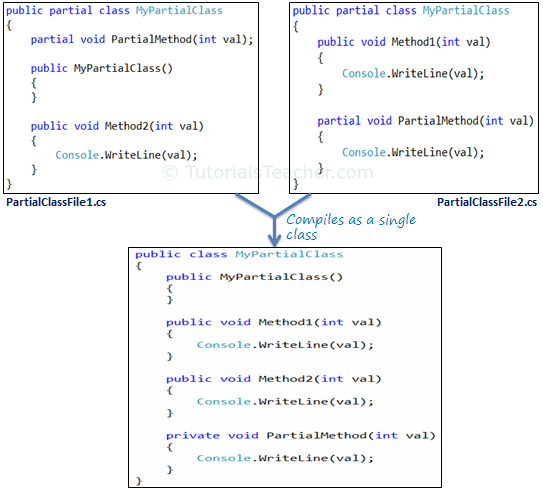
**Partial method requirements:**

* The partial method declaration must began with the partial modifier.
* The partial method can have a ref but not an out parameter.
* Partial methods are implicitly private methods.
* Partial methods can be static methods.
* Partial methods can be generic.

The following image illustrates partial class and partial method:

[](http://www.tutorialsteacher.com/Content/images/csharp/partial-method.png)Partial Method

The compiler combines the two partial classes into a single final class:

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

Structure

● Introduction to structure

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.

● Creating structure

## 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

Enum

● Introduction to enum

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

● Need of enum

**Enum advantages.** With an enum, magic constants are separate. This modular design makes things easier to understand. Fewer bugs will be introduced.

Exception Handling

● Understanding Exceptions

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.

● Throw exception

● Handling Exception

● Custom Exception

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:

|  |  |
| --- | --- |
| **Exception Class** | **Description** |
| System.IO.IOException | Handles I/O errors. |
| System.IndexOutOfRangeException | Handles errors generated when a method refers to an array index out of range. |
| System.ArrayTypeMismatchException | Handles errors generated when type is mismatched with the array type. |
| System.NullReferenceException | Handles errors generated from deferencing a null object. |
| System.DivideByZeroException | Handles errors generated from dividing a dividend with zero. |
| System.InvalidCastException | Handles errors generated during typecasting. |
| System.OutOfMemoryException | Handles errors generated from insufficient free memory. |
| 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

}

Delegates

● Introduction to delegates

Delegate is a type which  holds the method(s) reference in an object. It is also referred to as a type safe function pointer.

### **Advantages**

* Encapsulating the method's call from caller
* Effective use of delegate improves the performance of application
* Used to call a method asynchronously

### **Declaration**

Hide   Copy Code

public delegate type\_of\_delegate delegate\_name()

Example:

Hide   Copy Code

public delegate int mydelegate(int delvar1,int delvar2)

### **Note**

* You can use delegates without parameters or with parameter list
* You should follow the same syntax as in the method   
  (If you are referring to the method with two int parameters and int return type, the delegate which you are declaring should be in the same format. This is why it is referred to as type safe function pointer.)

### **Sample Program using Delegate**

Hide   Copy Code

public delegate double Delegate\_Prod(int a,int b);

class Class1

{

static double fn\_Prodvalues(int val1,int val2)

{

return val1\*val2;

}

static void Main(string[] args)

{

*//Creating the Delegate Instance*

Delegate\_Prod delObj = new Delegate\_Prod(fn\_Prodvalues);

Console.Write("Please Enter Values");

int v1 = Int32.Parse(Console.ReadLine());

int v2 = Int32.Parse(Console.ReadLine());

*//use a delegate for processing*

double res = delObj(v1,v2);

Console.WriteLine ("Result :"+res);

Console.ReadLine();

}

}

### **Explanation**

Here I have used a small program which demonstrates the use of delegate.

The delegate "Delegate\_Prod" is declared with double return type and accepts only two integer parameters.

Inside the class, the method named fn\_Prodvalues is defined with double return type and two integer parameters. (The delegate and method have the same signature and parameter type.)

Inside the Main method, the delegate instance is created and the function name is passed to the delegate instance as follows:

Hide   Copy Code

Delegate\_Prod delObj = new Delegate\_Prod(fn\_Prodvalues);

● Types of delegates

● Single Delegate

● Multicast Delegate

delegate void Del(string s);

class TestClass

{

static void Hello(string s)

{

System.Console.WriteLine(" Hello, {0}!", s);

}

static void Goodbye(string s)

{

System.Console.WriteLine(" Goodbye, {0}!", s);

}

static void Main()

{

Del a, b, c, d;

// Create the delegate object a that references

// the method Hello:

a = Hello;

// Create the delegate object b that references

// the method Goodbye:

b = Goodbye;

// The two delegates, a and b, are composed to form c:

c = a + b;

// Remove a from the composed delegate, leaving d,

// which calls only the method Goodbye:

d = c - a;

System.Console.WriteLine("Invoking delegate a:");

a("A");

System.Console.WriteLine("Invoking delegate b:");

b("B");

System.Console.WriteLine("Invoking delegate c:");

c("C");

System.Console.WriteLine("Invoking delegate d:");

d("D");

}

}

/\* Output:

Invoking delegate a:

Hello, A!

Invoking delegate b:

Goodbye, B!

Invoking delegate c:

Hello, C!

Goodbye, C!

Invoking delegate d:

Goodbye, D!

\*/

Generic Delegates

● Introduction to generic delegates

In the preceding article and code sample, we have a delegate called DelegateInt that takes two integer parameters and returns an int type.

1. **public** **delegate** **int** DelegateInt(**int** a, **int** b);

The DelegateInt works only with methods that have two integer parameters. What if we want to create a delegate that will work with any type of two parameters and return any type? In that case, it will not work. This is where generics are useful and generics play a major role in LINQ.

The following code snippet declares a generic delegate.

1. **public** **delegate** **string** GenericDelegateNumber<T1, T2>(T1 a, T2 b);

The following code snippet defines two methods for creating instances of generic delegates.

1. **public** **static** **string** AddDoubles(**double** a, **double** b)
2. {
3. **return** (a + b).ToString();
4. }
6. **public** **static** **string** AddInt(**int** a, **int** b)
7. {
8. **return** (a + b).ToString();
9. }

The following code snippet creates two delegate instances where the first one uses integers and the second delegate uses double parameter values.

1. GenericDelegateNumber<**int**, **int**> gdInt = **new** GenericDelegateNumber<**int**, **int**>(AddInt);
2. Console.WriteLine(gdInt(3, 6));
3. GenericDelegateNumber<**double**, **double**> gdDouble = **new** GenericDelegateNumber<**double**, **double**>(AddDoubles);
4. Console.WriteLine(gdDouble(3.2, 6.9));

The following code lists the complete sample.

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Threading.Tasks;
7. **namespace** GenericDelegateSample
8. {
9. **class** Program
10. {
11. **static** **void** Main(**string**[] args)
12. {
13. GenericDelegateNumber<**int**, **int**> gdInt = **new** GenericDelegateNumber<**int**, **int**>(AddInt);
14. Console.WriteLine(gdInt(3, 6));
15. GenericDelegateNumber<**double**, **double**> gdDouble = **new** GenericDelegateNumber<**double**, **double**>(AddDoubles);
16. Console.WriteLine(gdDouble(3.2, 6.9));
17. Console.ReadKey();
18. }
20. // Generic Delegate takes generic types and returns a string
21. **public** **delegate** **string** GenericDelegateNumber<T1, T2>(T1 a, T2 b);
23. **public** **static** **string** AddDoubles(**double** a, **double** b)
24. {
25. **return** (a + b).ToString();
26. }
28. **public** **static** **string** AddInt(**int** a, **int** b)
29. {
30. **return** (a + b).ToString();
31. }
32. }
34. }

● Func

● Action

● Predicate

Generic Delegate was introduced in **.NET 3.5** that don't require to define the delegate instance in order to invoke the methods.

There are three types of generic delegates:

* Func
* Action
* Predicate

#### Generic Delegate: Func

The Func delegate defines a method that can be called on arguments and returns a result. In the given code example, delegate Func<interest,double> is defined with Interest type as argument and double as return type.

Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png   Copy Code

/// *<summary>*

/// *Sample example of generic delegate*

///

class Program

{

*// Declare a delegate*

delegate double CalculateSimpleInterest(double para1, double para2, double para3);

static CalculateSimpleInterest dObjSI = getTotalInterest;

static void Main(string[] args)

{

double SI;

*//Declare a generic Func delegate*

Func<interest,double> calcSI = SIObj =>(SIObj.P\*SIObj.T\*SIObj.R)/100;

Interest obj = new Interest();

obj.P = 120; obj.T = 1; obj.R = 3.25;

*// Consuming delegate*

SI = calcSI(obj);

Console.WriteLine("Total Interest of $120 in a year at rate of 3.25% APR is {0}", SI);

Console.ReadKey();

}

}

class Interest

{

public double P { get; set; }

public double T { get; set; }

public double R { get; set; }

}

</interest,double>

#### Generic Delegate: Action

The Action delegate defines a method that can be called on arguments but does not return a result. In the given code example, delegate Action<string> is defined with string as argument.

Hide   Copy Code

Action<string> MyAction = y => Console.Write(y);

MyAction("Hello");

Console.ReadKey();

#### Generic Delegate: Predicate

The **Predicate** delegate defines a method that can be called on arguments and always returns **Boolean** type result. In the given code example, delegate Predicate<string> checkValidDate is defined with stringtype as argument and returns bool type.

Hide   Copy Code

/// *<summary>*

/// *Sample example of generic delegate: Predicate*

///

class Program

{

static void Main(string[] args)

{

string date="05/12/20143";

Predicate<string> checkValidDate = d => IsDate(d) ;

if (checkValidDate(date))

{

Console.WriteLine("Valid Date");

}

else

{

Console.WriteLine("Invalid Date");

}

Console.ReadKey();

}

private static bool IsDate(string date)

{

DateTime dt;

return DateTime.TryParse(date,out dt);

}

}

</string>

## Expression Tree

Expression trees allow you to build code dynamically at runtime instead of statically typing it in the IDE and using a compiler. Expression Trees use generic delegates to create and parse the expressions.

Expression trees are used in the following cases:

* Expression trees can be used to create LINQ to SQL and EF to SQL.
* Expression trees can be used for ASP.NET MVC's HTML extensions.
* Expression trees can be used to determine the selected property or field in MVC.

In the given code example, a expression **(3+5)-(4-2)** is divided into three expressions as **Exp1 for (3+5), Exp2 for (4-2) and Exp3** for adding **Exp1 and Exp2**. The expression Expression.Lambda<func<int>>(resultexp).compile()() uses Func generic delegate to parse the expressions.

Hide   Copy Code

/// *<summary>*

/// *Sample example of Expression Tree*

///

class Program

{

static void Main(string[] args)

{

*//Express tree (3+5)-(4-2)*

*//3+5*

BinaryExpression Exp1 = Expression.MakeBinary(ExpressionType.Add, Expression.Constant(3),

Expression.Constant(5));

*//4-2*

BinaryExpression Exp2 = Expression.MakeBinary(ExpressionType.Subtract, Expression.Constant(4),

Expression.Constant(2));

*// (3+5)-(4-2)*

BinaryExpression resultExp = Expression.MakeBinary(ExpressionType.Subtract, Exp1, Exp2);

*//this stmt will create a delegates by parsing the expression three*

int result = Expression.Lambda<func<int>>

(resultexp).compile()(); console.writeline("result="{0}",">

## Difference Between Each Type of Generic Delegate

|  | **Func** | **Action** | **Predicate** |
| --- | --- | --- | --- |
| Arguments | Yes | Yes | Yes |
| Returns | Yes | No | Boolean Type Only |

Events

● Introduction to 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.

● Creating Events

## 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

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

● Event Handler

### **Step 1 - Create an EventArgs class**

If you need to pass arguments to the event handler, a specific EventArgs class has to be made. Eventually, a suitable EventArgs class might already be available, but in most cases, you will have to create one to tailor your specific arguments.

If you do not need to pass arguments to the event handler (except the 'sender'), then no subclass of EventArgsis needed. However, note that you should still define the event method with an argument of typeSystem.EventArgs, and then pass System.EventArgs.Empty.

Create the EventArgs subclass as follows:

Hide   Copy Code

public class StartEventArgs : System.EventArgs {

*// Provide one or more constructors, as well as fields and*

*// accessors for the arguments.*

}

**Note**: all EventArgs classes should have a name ending on 'EventArgs'.

### **Step 2 - Create a delegate**

For each EventArgs subclass you have created, a matching delegate should be created. Best practice is to declare the EventArgs subclass and the delegate in the same scope (inside the same namespace or inside the same class). A delegate declaration looks pretty much like a method declaration, but in fact, when declaring a delegate, you create a new class, so placing a delegate declaration directly in a namespace is no problem.

When you use the default System.EventArgs class instead of a subclass of it, there is no need to declare a delegate, you can use the System.EventHandler delegate.

You might also decide to create a specific EventArgs subclass, but no delegate for it. In that case, the event will be delcared of type System.EventHandler (or any superclass of your specific EventArgs subclass for which a delegate is available) and casting of the EventArgs argument in the eventhandler method will be needed.

Declare a delegate as follows:

Hide   Copy Code

public delegate void StartEventHandler(object sender, StartEventArgs e);

**Note**: all delegate classes used for event handling should have a name ending on 'EventHandler' and should be of type void. (Delegates that are not of type void are not suitable for multicasting and therefore not suitable for event handling.)

### **Step 3 - Create Events**

For each kind of event, an Event is created in the sending class. The event is declared as of type it's delegate. Multiple events can use the same delegate (think of the delegate as a class, and an event as an instance of it):

Hide   Copy Code

public class Sender {

public event StartEventHandler BeforeStart;

public event StartEventHandler AfterStart;

*//...*

}

### **Step 4 - Create OnEvent methods**

Although not required, it is a good convention to create protected virtual OnEvent methods for your events:

Hide   Copy Code

protected virtual void OnBeforeStart(StartEventArgs e) {

if (BeforeStart != null) BeforeStart(this, e);

}

protected virtual void OnAfterStart(StartEventArgs e) {

if (AfterStart != null) AfterStart(this, e);

}

You can now call this method whenever you want to send the Start event. I.e:

Hide   Copy Code

OnBeforeStart(new StartEventArgs());

### **Step 5 - Make the event default**

When you class represents a component, it is good practice to make the most commonly used event the default event. Add a DefaultEvent attribute in front of your sender class:

Hide   Copy Code

[System.ComponentModel.DefaultEvent("BeforeStart")]

public class Sender : System.ComponentModel.Component {

*//...*

## Handling events

A add a handler to an event, use the (overloaded) += assignment operator, as in:

Hide   Copy Code

sender.BeforeStart += new StartEventHandler(this.sender\_BeforeStart);

To remove the handler, use the (overloaded) -= assignement operator:

Hide   Copy Code

sender.BeforeStart -= new StartEventHandler(this.sender\_BeforeStart);

It might look strange to remove a newly created object. In reality, when you remove a handler, the .NET framework will look for a handler using the same handler method, and remove that handler. The newly created object is then garbage collected.

Here, sender\_BeforeStart is the method that will handle the event. It looks like:

Hide   Copy Code

private void sender\_BeforeStart(object sender, StartEventArgs e) {

*//...*

}

Example program:

public delegate void StartEventHandler(object sender, StartEventArgs e);

public class StartEventArgs : System.EventArgs {

*// Provide one or more constructors, as well as fields and*

*// accessors for the arguments.*

}

public class Sender {

public event StartEventHandler BeforeStart;

public event StartEventHandler AfterStart;

protected virtual void OnBeforeStart(StartEventArgs e) {

if (BeforeStart != null) BeforeStart(this, e);

}

protected virtual void OnAfterStart(StartEventArgs e) {

if (AfterStart != null) AfterStart(this, e);

}

void DoStart() {

OnBeforeStart(new StartEventArgs());

*//...*

OnAfterStart(new StartEventArgs());

}

*//...*

}

public class Receiver {

Sender sender;

public Receiver() {

sender = new Sender();

}

void Connect() {

sender.BeforeStart += new StartEventHandler(this.sender\_BeforeStart);

}

private void sender\_BeforeStart(object sender, StartEventArgs e) {

*//...*

}

*//...*

}

Anonymous Method and Lambda Expression

● Anonymous Method

**Anonymous methods** provide a technique to pass a code block as a delegate parameter. Anonymous methods are the methods without a name, just the body.

You need not specify the return type in an anonymous method; it is inferred from the return statement inside the method body.

## Writing an Anonymous Method

Anonymous methods are declared with the creation of the delegate instance, with a **delegate** keyword. For example,

delegate void NumberChanger(int n);

...

NumberChanger nc = delegate(int x)

{

Console.WriteLine("Anonymous Method: {0}", x);

};

The code block *Console.WriteLine("Anonymous Method: {0}", x);* is the body of the anonymous method.

The delegate could be called both with anonymous methods as well as named methods in the same way, i.e., by passing the method parameters to the delegate object.

For example,

nc(10);

## Example

The following example demonstrates the concept:

using System;

delegate void NumberChanger(int n);

namespace DelegateAppl

{

class TestDelegate

{

static int num = 10;

public static void AddNum(int p)

{

num += p;

Console.WriteLine("Named Method: {0}", num);

}

public static void MultNum(int q)

{

num \*= q;

Console.WriteLine("Named Method: {0}", num);

}

public static int getNum()

{

return num;

}

static void Main(string[] args)

{

//create delegate instances using anonymous method

NumberChanger nc = delegate(int x)

{

Console.WriteLine("Anonymous Method: {0}", x);

};

//calling the delegate using the anonymous method

nc(10);

//instantiating the delegate using the named methods

nc = new NumberChanger(AddNum);

//calling the delegate using the named methods

nc(5);

//instantiating the delegate using another named methods

nc = new NumberChanger(MultNum);

//calling the delegate using the named methods

nc(2);

Console.ReadKey();

}

}

}

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

Anonymous Method: 10

Named Method: 15

Named Method: 30

● Lambda Expression

A lambda expression is an anonymous function and it is mostly used to create delegates in LINQ. Simply put, it's a method without a declaration, i.e., access modifier, return value declaration, and name.

Convenience. It's a shorthand that allows you to write a method in the same place you are going to use it. Especially useful in places where a method is being used only once, and the method definition is short. It saves you the effort of declaring and writing a separate method to the containing class.

### **Benefits**

Lambda expressions should be short. A complex definition makes the calling code difficult to read.

Lambda basic definition: Parameters => Executed code

1. **What** is a Lambda Expression?
2. **Why** do we need lambda expressions? (Why would we need to write a method without a name?)
   1. Reduced typing. No need to specify the name of the function, its return type, and its access modifier.
   2. When reading the code, you don't need to look elsewhere for the method's definition.
3. **How** do we define a lambda expression?

#### Simple Example

Hide   Copy Code

n => n % 2 == 1

* n is the input parameter
* n % 2 == 1 is the expression

You can read n => n % 2 == 1 like: "input parameter named n goes to anonymous function which returnstrue if the input is odd".

Same example (now execute the lambda):

Hide   Copy Code

List<int> numbers = new List<int>{11,37,52};

List<int> oddNumbers = numbers.where(n => n % 2 == 1).ToList();

*//Now oddNumbers is equal to 11 and 37*

That's all, now you know the basics of Lambda Expressions.

* I didn't mention expression trees/run time advantages of lambda expression due to limited scope.

Attributes

● Introduction to attributes

<https://www.tutorialspoint.com/csharp/csharp_attributes.htm>

**Attributes** extend classes and types. This C# feature allows you to attach declarative information to any type. Attributes are accessed at compile-time or runtime through the metadata. We then can handle types based on their attributes.

● Creating attribute class

The Obsolete attribute is a way to declare that a method is deprecated and should be avoided. When you look at the program in Visual Studio, this will result in a warning.[**Visual Studio**](https://www.dotnetperls.com/vs)

**Tip:**The actual type referenced by [Obsolete] is ObsoleteAttribute, but you can omit the word Attribute.

**Tip 2:**The attribute modifies the compiler's view of the Program.Text method. It doesn't affect runtime.

[**Compiler**](https://www.dotnetperls.com/compiler)

**Based on:** .NET 4.6

**C# program that uses attribute**

using System;

class Program

{

static void Main()

{

// Warning: 'Program.Test()' is obsolete

Test();

}

**[Obsolete]**

static void Test()

{

}

}

**Warning generated by program:**

'Program.Test()' is obsolete

● Need of attributes

In the .NET Framework, attributes can be used for many reasons -- like

* Defining which classes are serializable
* Choosing which methods are exposed in a Web service

Attributes allow us to add descriptions to classes, properties, and methods at design time that can then be examined at runtime via reflection.

Reflection

● Introduction to reflection

**Reflection** objects are used for obtaining type information at runtime. The classes that give access to the metadata of a running program are in the**System.Reflection** namespace.

The **System.Reflection** namespace contains classes that allow you to obtain information about the application and to dynamically add types, values, and objects to the application.

## Applications of Reflection

Reflection has the following applications:

* It allows view attribute information at runtime.
* It allows examining various types in an assembly and instantiate these types.
* It allows late binding to methods and properties
* It allows creating new types at runtime and then performs some tasks using those types.

● Viewing Metadata

We have mentioned in the preceding chapter that using reflection you can view the attribute information.

The **MemberInfo** object of the **System.Reflection** class needs to be initialized for discovering the attributes associated with a class. To do this, you define an object of the target class, as:

System.Reflection.MemberInfo info = typeof(MyClass);

The following program demonstrates this:

using System;

[AttributeUsage(AttributeTargets.All)]

public class HelpAttribute : System.Attribute

{

public readonly string Url;

public string Topic // Topic is a named parameter

{

get

{

return topic;

}

set

{

topic = value;

}

}

public HelpAttribute(string url) // url is a positional parameter

{

this.Url = url;

}

private string topic;

}

[HelpAttribute("Information on the class MyClass")]

class MyClass

{

}

namespace AttributeAppl

{

class Program

{

static void Main(string[] args)

{

System.Reflection.MemberInfo info = typeof(MyClass);

object[] attributes = info.GetCustomAttributes(true);

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

{

System.Console.WriteLine(attributes[i]);

}

Console.ReadKey();

}

}

}

When it is compiled and run, it displays the name of the custom attributes attached to the class *MyClass*:

HelpAttribute

Collections

● Introduction to .NET Collections

Collection classes are specialized classes for data storage and retrieval. These classes provide support for stacks, queues, lists, and hash tables. Most collection classes implement the same interfaces.

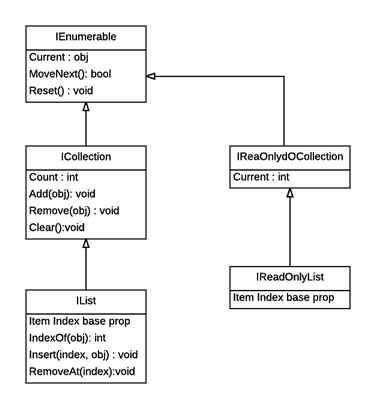
Collection classes serve various purposes, such as allocating memory dynamically to elements and accessing a list of items on the basis of an index etc. These classes create collections of objects of the Object class, which is the base class for all data types in C#.

● Introduction to Collection Classes – Array List, Hash Table, Dictionary, Stack, Queue

The following are the various commonly used classes of the**System.Collection** namespace. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Class** | **Description and Useage** |
| [**ArrayList**](https://www.tutorialspoint.com/csharp/csharp_arraylist.htm) | It represents ordered collection of an object that can be**indexed** individually.  It is basically an alternative to an array. However, unlike array you can add and remove items from a list at a specified position using an **index** and the array resizes itself automatically. It also allows dynamic memory allocation, adding, searching and sorting items in the list. |
| [**Hashtable**](https://www.tutorialspoint.com/csharp/csharp_hashtable.htm) | It uses a **key** to access the elements in the collection.  A hash table is used when you need to access elements by using key, and you can identify a useful key value. Each item in the hash table has a **key/value** pair. The key is used to access the items in the collection. |
| [**SortedList**](https://www.tutorialspoint.com/csharp/csharp_sortedlist.htm) | It uses a **key** as well as an **index** to access the items in a list.  A sorted list is a combination of an array and a hash table. It contains a list of items that can be accessed using a key or an index. If you access items using an index, it is an ArrayList, and if you access items using a key , it is a Hashtable. The collection of items is always sorted by the key value. |
| [**Stack**](https://www.tutorialspoint.com/csharp/csharp_stack.htm) | It represents a **last-in, first out** collection of object.  It is used when you need a last-in, first-out access of items. When you add an item in the list, it is called **pushing** the item and when you remove it, it is called **popping** the item. |
| [**Queue**](https://www.tutorialspoint.com/csharp/csharp_queue.htm) | It represents a **first-in, first out** collection of object.  It is used when you need a first-in, first-out access of items. When you add an item in the list, it is called**enqueue** and when you remove an item, it is called**deque**. |
| [**BitArray**](https://www.tutorialspoint.com/csharp/csharp_bitarray.htm) | It represents an array of the **binary representation** using the values 1 and 0.  It is used when you need to store the bits but do not know the number of bits in advance. You can access items from the BitArray collection by using an **integer index**, which starts from zero. |

● Introduction to Collection Interfaces – IEnumerable, IQuerable, IList, ICollection

NET framework provides interfaces that implements by collections in language to provide functionality of iterating over objects in collection, adding and removing object from collection to randomly access object from collection.   
  
As different interfaces provide different set of functionality most of the developers has problem when to use which interface to achieve functionality. The following post provides information about interfaces implemented by collection.  
  
**Interfaces**  
The following diagram is for relation between the interfaces.  
  
  
  
**Note:**

1. Class diagram are not having all the methods but contains important method that belongs to each collection interface.
2. Collection interface is available in both generic and non-generic form, so in diagram obj type is object in nongeneric form and obj type is T(template type) in generic form.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Functionality Provided** | **Read** | **Count** | **Add & Remove** | **Index Based Read** | **Index Based Add & Remove** |
| IEnumerable | 1. Provide Read only Collection. 2. Allow to read each object of collection in forward only mode. | Y | N | N | N | N |
| ICollection | 1. Allow to modify collection.  2. Allow to get size of collection. 3. Allow to Add and Remove object in/from collection. | Y (Inherited) | Y | Y | N | N |
| IReadOnlyCollection | 1. Allow to read collection. 2. Allow to get size of collection. | Y (Inherited) | Y | N | N | N |
| IList | 1. Allows to access collection by Index. 2. Allow to Add and Remove object in/from collection by index. | Y (Inherited) | Y  (Inherited) | Y  (Inherited) | Y | Y |
| IReadOnlyList | 1. Allow to read collection by Index. | Y (Inherited) | Y  (Inherited) | N | N | Y |

**Note:**  
In above diagram (Inherited), the columns indicate that the features are inherited from parent and to find out from which parent one must look in the interface collection diagram.

So from above table three main interfaces functionality concluded in following way:

**IEnumerable** – interface provide minimum functionality which is Enumration.

**ICollection** – interface provide medium functionality which is getting size, adding, removing and clearing collection i.e. modification of collection. As it inherited from IEnumerable so includes functionality of IEnumerable.

**IList** – interface provide full functionality which is index base accessing of collection element, index base adding, index base removing from collection. As it inherited from ICollection it includes functionality of Enumerable and ICollection.

The following are some important things to know:

1. IEnumerable interface under the hood make use of IEnumerator for providing reaonly and forward mode read.
2. IReadOnly\*\*\* and IEnumerable are used for providing readonly collection. But difference is that IEnumerable allows collection to read in forward only mode where IReadOnly\*\*\* provide feature of Collection /List but only in readonly mode i.e. without modification feature like add & remove.
3. IReadOnly is part of collection interface from framework 4.5.

Above table list down the features provided by each interface when collection gets converted to interface type or class implement interface to provide feature of collection.

● Relations between Collection Interfaces and Classes

● Using Collection classes – Array List, Hash Table & Dictionary

Collections Contd.

● Using Collection classes – Stack, Queue

● Using Collection Interfaces – IEnumerable, IQuerable, IList, ICollection

Generics

● Understanding .NET Generics

● Using Generics classes

● Using Generics Interfaces

● Generics Advantages

● Collection s vs. Generics

File Handling

● File I/O and Streams

● Stream readers and writers

● Creating file

● Writing data to file

● File opening mode

File Class and Operations

● File class

● Directory class

● Path class

● Performing operations on a file

ASP.NET

Introduction to HTML

● Introduction to ASP.NET

● ASP.NET Version History

● Difference Between ASP and ASP.NET

● ASP.NET Architecture

● Inline Technique & CodeBehind

Technique

● Server Controls

● Page Lifecycle

● Postback

● Page Directives

Web Form Control

● Text Box

● Check Box

● List Box

● Drop DownList

● Image Control

● Radio Button

HTML

Introduction to HTML

● History of Html

● Getting Started With HTML

● HTML Page Structure

● Html Element

● Inline and Block Elements

● Images

● Ordered list and Unordered list

● Tables and Nesting tables

CSS

Introduction to CSS

● Introduction to CSS

● CSS Id and Class

● Colors and Backgrounds

● Text and Fonts

● Lists Styles

● Tables Styles

● Introduction to CSS3

● CSS Box Model

● CSS Grouping/Nesting, positions and advance

Master Page

● Introduction to Master Page

● Creating Master Page

● Advantages of Master Pages

Forms & Validations

● Creating Forms

● Creating User SignUp Page

● Using Data Validations Controls – Required Field, Regular Expression, Compare and

Range Validator

State Management

● Client Side State Management Techniques – Cookies, ViewState, Hidden Field, Query

String

● Server Side Statement Management – Session, Application State

Bootstrap

Introduction to Bootstrap

● Introduction to Responsive Design

● Need of Bootstrap

● Bootstrap Fundamentals

● Bootstrap Grid System

● Bootstrap Components

● Header, Footer, Navigation Bar

● Bootstrap Modals

● Bootstrap Form Elements

● Bootstrap Icons

● Typography

● Customizing Bootstrap

● Panels, Sliders, Tooltips

SQL Server

Introduction to SQL Server

● Understanding SQL Server

● SQL Server Version History

● RDBMS VS DBMS VS NoSQL

● Advantage of SQL Server

T SQL

● Types of SQL Commands

● Data Definition Language (DDL)

● Data Manipulation Language (DML)

● Data Query Language (DQL)

● Data Control Language (DCL)

● Transaction Control Language (TCL)

● Creating Database

● Altering Database

● Deleting Database

● Creating Table

● Altering Table

● Deleting Table

SQL Keys

● Types of SQL Keys

● Super Key

● Primary Key

● Unique Key

● Foreign Key

● Candidate Key

● Alternate Key

● Composite Key

SQL Commands & Predicates

● Insert

● Update

● Delete

● Truncate

● Delete VS Truncate

● Distinct

● Between…And

● In

● Like

● Is Null

SQL Clauses

● Select

● Where

● Order By

● Group By

● Having

Query & SubQuery

● Understanding Query and SubQuery

● Writing and Executing Query

SQL Joins

● Types of Joins

● Inner Join

● Outer Join

● Cross Join

● SelfJoin

Views

● Understanding Views

● Need of Views

● Creating , Altering and Dropping Views

● Simple and Complex Views

Functions

● BuiltIn

Functions

● Scalar Functions

● Aggregate Functions

● Creating, Altering and Dropping Functions

Stored Procedures

● Creating , Altering and Dropping Procedure

● Optional Parameters

● Input and Output Parameters

Triggers

● Understanding Triggers

● Stored Procedures VS Functions VS Triggers

● Types of Triggers

● Creating, Altering and Dropping Triggers

ADO.NET

Introduction to ADO.NET

● Understanding ADO.NET

● ADO VS ADO.NET

● Advantage of ADO.NET

● ADO.NET Version History

● ADO.NET Architecture

● Connected and Disconnected Architecture

● ADO.NET Objects

● .NET Data Providers

Connection & Command

● Connection Classes

● Command Objects

● Creating Commands

● Executing Commands

● Parameterized Queries

● Command Types

Data Readers & Connected Access

● Understanding DataReader

● Reading Data using DataReader

● ExecuteReader Options

Data Sets & Disconnected Access

● Understanding DataSet

● Understanding DataAdapter

● Understanding DataTable, DataColumn, DataRow

Transaction

● Understanding Transaction

● Transaction ACID properties

● Implementing Transaction

● Commit and Rollback

Data Controls

● Data Source Controls

● Choosing Data Binding Controls

● GridView

● DataList

● Paging, Sorting

AJAX

Introduction to AJAX

● Understanding AJAX

● Script Manager

● Update Panel

● Remote Method Calls

● AJAX Control Toolkit

WCF

Introduction to WCF

● Introduction to WCF

● Understanding WCF ABC

● WCF Addresses

● WCF Bindings

● WCF Contracts

● WCF Endpoints

Building & Consuming WCF Service

● Creating WCF Service

● Configuring Endpoints

● Creating Service Client

● Consuming WCF Service

● WCF Hosting

ASP.NET MVC

Introduction to ASP.NET MVC

● Understanding ASP.NET MVC

● ASP.NET VS MVC VS ASP.NET Web Form VS ASP.NET MVC

● ASP.NET MVC Version History

● Understanding Model, View and Controller

● Advantages of ASP.NET MVC

● Design Pattern VS Architectural Pattern

● Layer VS Tier

● MVC pattern VS 3Tier

Architecture

Visual Studio Project Templates

● Understanding Visual Studio ASP.NET MVC Templates

● Creating an ASP.NET MVC project

● Understanding ASP.NET MVC project folder structure

● Understanding configuration files

Controller & Actions

● Understanding Controller

● Creating Controller

● Understanding Actions

● Actions and Non Actions Methods

● Understanding Action Results

View

● Types of Views

● Creating Standard View

● Creating Layout Page

● Communication between Controller and View

Model & Validation

● Creating Model

● Understanding ASP.NET MVC Validation

● Need of Server Side and Client Side Validation

● Validation with Data Annotation