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("\*");

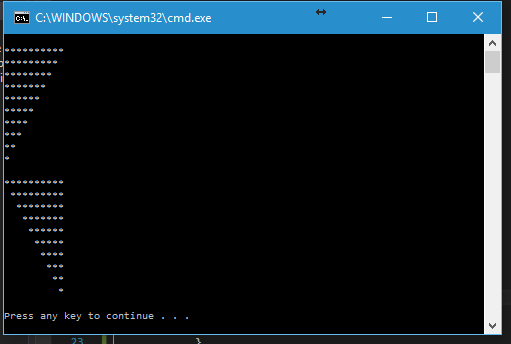
}

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;

/\*...\*/

}

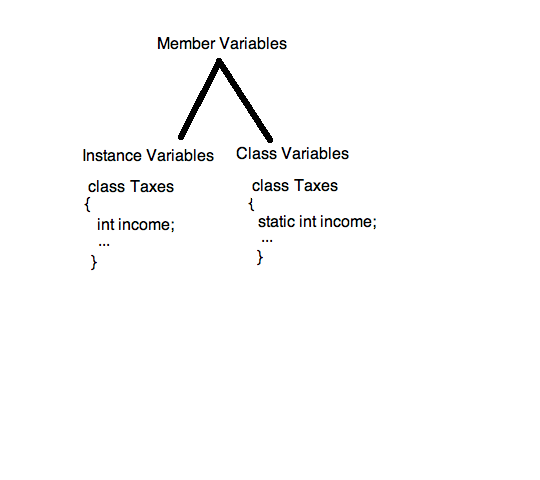
## Difference between class and instance variables

|  |
| --- |
|  |

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

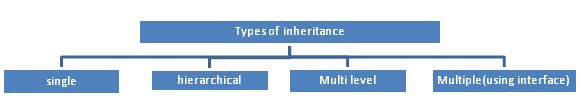
● Single Level Inheritance

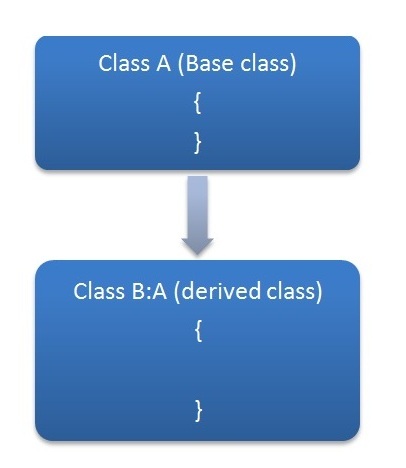
● MultiLevel

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()

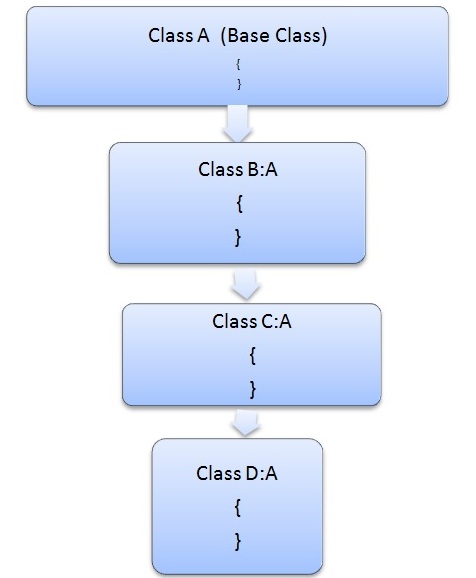
    {

        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()

        {

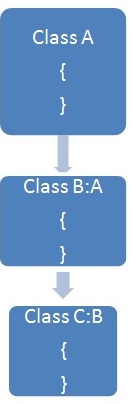
            msg();

            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()

    {

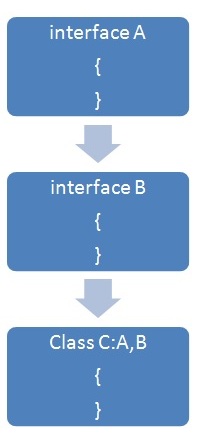
        persondet();

        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:**

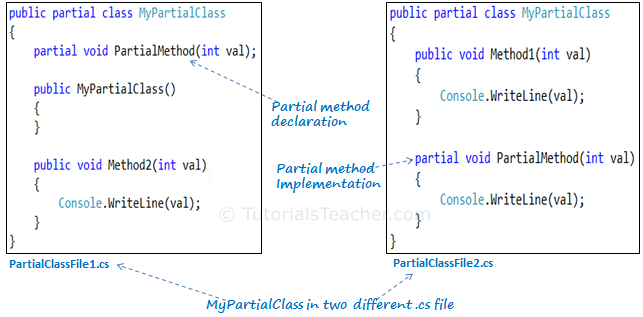
* Multiple developers can work simultaneously with a single class in separate files.
* When working with automatically generated source, code can be added to the class without having to recreate the source file. For example, Visual Studio separates HTML code for the UI and server side code into two separate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

● Need of partial class

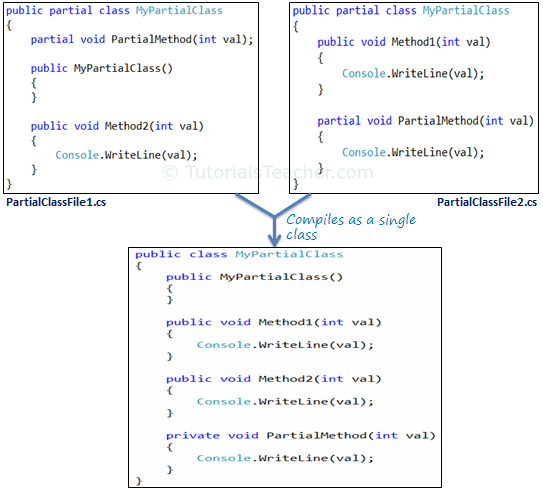
**Partial method requirements:**

* 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.

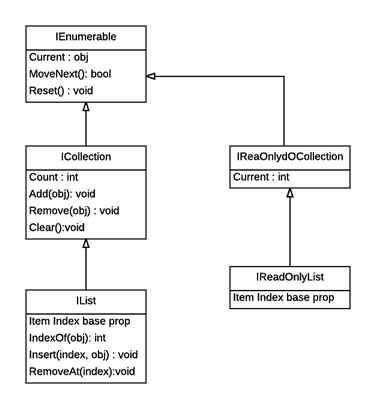
Collection classes serve various purposes, such as allocating memory dynamically to elements and accessing a list of items on the basis of an index etc. These classes create collections of objects of the Object class, which is the base class for all data types in C#.

● 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 System;

using System.Collections.Generic;

class Program

{

static Stack<int> GetStack()

{

Stack<int> stack = new **Stack**<int>();

stack.Push(100);

stack.Push(1000);

stack.Push(10000);

return stack;

}

static void Main()

{

var stack = GetStack();

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

foreach (int i in stack)

{

Console.WriteLine(i);

}

}

}

**Output**

--- Stack contents ---

10000

1000

100

QUEUE:

**First-In-First-Out:**The queue data structure implements this algorithm. Queue is a generic type with one type parameter.

**Based on:** .NET 4.6

**C# program that uses Enqueue**

using System;

using System.Collections.Generic;

class Program

{

static void Main()

{

// New Queue of integers.

Queue<int> q = new **Queue**<int>();

q.Enqueue(5); // Add 5 to the end of the Queue.

q.Enqueue(10); // Then add 10. 5 is at the start.

q.Enqueue(15); // Then add 15.

q.Enqueue(20); // Then add 20.

}

}

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

Generics

● Understanding .NET Generics

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

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

using System;

using System.Collections.Generic;

namespace GenericApplication

{

public class MyGenericArray<T>

{

private T[] array;

public MyGenericArray(int size)

{

array = new T[size + 1];

}

public T getItem(int index)

{

return array[index];

}

public void setItem(int index, T value)

{

array[index] = value;

}

}

class Tester

{

static void Main(string[] args)

{

//declaring an int array

MyGenericArray<int> intArray = new MyGenericArray<int>(5);

//setting values

for (int c = 0; c < 5; c++)

{

intArray.setItem(c, c\*5);

}

//retrieving the values

for (int c = 0; c < 5; c++)

{

Console.Write(intArray.getItem(c) + " ");

}

Console.WriteLine();

//declaring a character array

MyGenericArray<char> charArray = new MyGenericArray<char>(5);

//setting values

for (int c = 0; c < 5; c++)

{

charArray.setItem(c, (char)(c+97));

}

//retrieving the values

for (int c = 0; c< 5; c++)

{

Console.Write(charArray.getItem(c) + " ");

}

Console.WriteLine();

Console.ReadKey();

}

}

}

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

0 5 10 15 20

a b c d e

● Using Generics classes

**Generic classes** have type parameters. Separate classes, each with a different field type in them, can be replaced with a single generic class. The generic class introduces a type parameter. This becomes part of the class definition itself.[**Type**](https://www.dotnetperls.com/type)  
**Example.** One amazing feature in the C# language is the ability to specify and use generic types. These types themselves contain type parameters, which influence the compilation of the code to use any type you specify.

**Next:**In this example, the letter T denotes a type that is only known based on the calling location.

**And:**The program can act upon the instance of T like it is a real type, but it is not.

**C# program that describes generic class**

using System;

**class** Test<T>

{

T \_value;

public Test(T t)

{

// The field has the same type as the parameter.

this.\_value = t;

}

public void Write()

{

Console.WriteLine(this.\_value);

}

}

**class** Program

{

static void Main()

{

// Use the generic type Test with an int type parameter.

Test<int> test1 = new Test<int>(5);

// Call the Write method.

test1.Write();

// Use the generic type Test with a string type parameter.

Test<string> test2 = new Test<string>("cat");

test2.Write();

}

}

**Output**

5

cat

● Using Generics Interfaces

<http://www.functionx.com/csharp/interfaces/cugi.htm>

● Generics Advantages

Generics provide the solution to a limitation in earlier versions of the common language runtime and the C# language in which generalization is accomplished by casting types to and from the universal base type [Object](https://msdn.microsoft.com/en-us/library/system.object.aspx). By creating a generic class, you can create a collection that is type-safe at compile-time.

The limitations of using non-generic collection classes can be demonstrated by writing a short program that uses the [ArrayList](https://msdn.microsoft.com/en-us/library/system.collections.arraylist.aspx)collection class from the .NET Framework class library. [ArrayList](https://msdn.microsoft.com/en-us/library/system.collections.arraylist.aspx) is a highly convenient collection class that can be used without modification to store any reference or value type.

● Collections vs. Generics

**COLLECTION**   
  
Collections can hold different data type. Here all the elements are objects.   
  
**GENERICS**   
  
In generics, we can specify which datatype we want to store.   
  
  
Use the following links for examples   
  
  
For generics,   
<http://msdn.microsoft.com/en-us/library/512aeb7t(v=vs.90).aspx>   
  
For collections,   
<http://zetcode.com/language/csharptutorial/collections/>

File Handling

● File I/O and Streams

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

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

## C# I/O Classes

The System.IO namespace has various classes that are used for performing numerous operations with files, such as creating and deleting files, reading from or writing to a file, closing a file etc.

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

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

## The FileStream Class

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

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

FileStream <object\_name> = new FileStream( <file\_name>, <FileMode Enumerator>, <FileAccess Enumerator>, <FileShare Enumerator>);

For example, we create a FileStream object **F** for reading a file named**sample.txt as shown**:

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

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

## Example

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

using System;

using System.IO;

namespace FileIOApplication

{

class Program

{

static void Main(string[] args)

{

FileStream F = new FileStream("test.dat", FileMode.OpenOrCreate, FileAccess.ReadWrite);

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

{

F.WriteByte((byte)i);

}

F.Position = 0;

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

{

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

}

F.Close();

Console.ReadKey();

}

}

}

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

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

●

● Creating file

**Create a Text File in C#**

The File class in the .NET Framework class library provides static methods for creating, reading, copying, moving, and deleting files. In this article, we will see how to create a text file using different options available in .NET.

We can create a file in four different following methods

* File.Create
* File.CreateText
* FileInfo.Create
* FileInfo.CreateText

File.Create Method

The File.Create method takes a file name with the full path as its first and required parameter and creates a file at the specified location. If same file already exists at the same location, this method overwrites the file.

The following code snippet creates a file Mahesh.txt in C:\Temp folder. If file already exists, the code will delete the existing file. The code writes two arrays of bytes to the file.

The Create method creates and returns a FileStream object that is responsible for reading and writing the specified file.

|  |
| --- |
| string fileName = @"C:\Temp\Mahesh.txt";  try  {      // Check if file already exists. If yes, delete it.      if (File.Exists(fileName))      {          File.Delete(fileName);      }      // Create a new file      using (FileStream fs = File.Create(fileName))      {          // Add some text to file          Byte[] title = new UTF8Encoding(true).GetBytes("New Text File");          fs.Write(title, 0, title.Length);          byte[] author = new UTF8Encoding(true).GetBytes("Mahesh Chand");          fs.Write(author, 0, author.Length);      }      // Open the stream and read it back.      using (StreamReader sr = File.OpenText(fileName))      {          string s = "";          while ((s = sr.ReadLine()) != null)          {              Console.WriteLine(s);          }      }  }  catch (Exception Ex)  {      Console.WriteLine(Ex.ToString());  } |

The Create method has four overloaded forms provide options with a file buffer size, file options, and file security.

**Create File with Buffer Size**

File.Create(name of the file, number of bytes buffered for read and write to the file)

FileStream fs = File.Create(fileName, 1024);

**Create File with File Options**

The File.Create method takes third parameters as a FileOptions enumeration that can be used to specify advanced options for creating a FileStream object.

FileStream fs = File.Create(fileName, 1024, FileOptions.WriteThrough);

**Create File with File Security**

The Create method also has an option to specify the file security options. The fourth parameter passed within the Create method of type FileSecurity object.

|  |
| --- |
| try {      string fileName = @"C:\Temp\Mahesh2.txt";      // Create File Security      FileSecurity fSecurity = new FileSecurity();      fSecurity.AddAccessRule(new FileSystemAccessRule(@"DomainName\AccountName", FileSystemRights.ReadData, AccessControlType.Allow));      using (FileStream fs = File.Create(fileName, 1024, FileOptions.WriteThrough, fSecurity))      {          // Add some text to file          Byte[] title = new UTF8Encoding(true).GetBytes("New Text File");          fs.Write(title, 0, title.Length);          byte[] author = new UTF8Encoding(true).GetBytes("Mahesh Chand");          fs.Write(author, 0, author.Length);      }      Console.WriteLine("Adding access control entry for " + fileName);      Console.WriteLine("Done.");  }  catch (Exception e)  {      Console.WriteLine(e);  } |

File.CreateText Method

The File.CreateText method creates and opens a file for writing UTF-8 encoded text. If file already exists, this method opens the file.

The following code snippet creates a file using the CreateText method that returns a StreamWriter object. The WriteLine method of SteamLine can be used to add line text to the object and writes to the file.

|  |
| --- |
| string fileName = @"C:\Temp\MaheshTX.txt";  try  {      // Check if file already exists. If yes, delete it.      if (File.Exists(fileName))      {          File.Delete(fileName);      }      // Create a new file      using (StreamWriter sw = File.CreateText(fileName))      {          sw.WriteLine("New file created: {0}", DateTime.Now.ToString());          sw.WriteLine("Author: Mahesh Chand");          sw.WriteLine("Add one more line ");          sw.WriteLine("Add one more line ");          sw.WriteLine("Done! ");      }      // Write file contents on console.      using (StreamReader sr = File.OpenText(fileName))      {          string s = "";          while ((s = sr.ReadLine()) != null)          {              Console.WriteLine(s);          }      }  }  catch (Exception Ex)  {      Console.WriteLine(Ex.ToString());  } |

FileInfo.Create Method

The FileInfo.Create method creates a file.

The following code snippet creates a file using the Create method that returns a FileSteam object. The Write method of FileStream can be used to write text to the file.

|  |
| --- |
| string fileName = @"C:\Temp\MaheshTXFI.txt";  FileInfo fi = new FileInfo(fileName);  try  {      // Check if file already exists. If yes, delete it.      if (fi.Exists)      {          fi.Delete();      }      // Create a new file      using (FileStream fs = fi.Create())      {          Byte[] txt = new UTF8Encoding(true).GetBytes("New file.");          fs.Write(txt, 0, txt.Length);          Byte[] author = new UTF8Encoding(true).GetBytes("Author Mahesh Chand");          fs.Write(author, 0, author.Length);      }      // Write file contents on console.      using (StreamReader sr = File.OpenText(fileName))      {          string s = "";          while ((s = sr.ReadLine()) != null)          {              Console.WriteLine(s);          }      }  }  catch (Exception Ex)  {      Console.WriteLine(Ex.ToString());  } |

FileInfo.CreateText Method

The FileInfo.CreateText method creates and opens a file for writing UTF-8 encoded text. If file already exists, this method opens the file.

The following code snippet creates a file using the CreateText method that returns a StreamWriter object. The WriteLine method of SteamLine can be used to add line text to the object and writes to the file.

|  |
| --- |
| string fileName = @"C:\Temp\MaheshTXFITx.txt";  FileInfo fi = new FileInfo(fileName);  try  {      // Check if file already exists. If yes, delete it.      if (fi.Exists)      {          fi.Delete();      }      // Create a new file      using (StreamWriter sw = fi.CreateText())      {          sw.WriteLine("New file created: {0}", DateTime.Now.ToString());          sw.WriteLine("Author: Mahesh Chand");          sw.WriteLine("Add one more line ");          sw.WriteLine("Add one more line ");          sw.WriteLine("Done! ");      }      // Write file contents on console.      using (StreamReader sr = File.OpenText(fileName))      {          string s = "";          while ((s = sr.ReadLine()) != null)          {              Console.WriteLine(s);          }      }  }  catch (Exception Ex)  {      Console.WriteLine(Ex.ToString());  } |

Stream readers and writers

● Writing data to file

The **StreamReader** and **StreamWriter** classes are used for reading from and writing data to text files. These classes inherit from the abstract base class Stream, which supports reading and writing bytes into a file stream.

The StreamReader Class

The **StreamReader** class also inherits from the abstract base class TextReader that represents a reader for reading series of characters. The following table describes some of the commonly used **methods** of the StreamReader class:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods** |
| 1 | **public override void Close()**  It closes the StreamReader object and the underlying stream, and releases any system resources associated with the reader. |
| 2 | **public override int Peek()**  Returns the next available character but does not consume it. |
| 3 | **public override int Read()**  Reads the next character from the input stream and advances the character position by one. |

Example

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

Down the way where the nights are gay

And the sun shines daily on the mountain top

I took a trip on a sailing ship

And when I reached Jamaica

I made a stop

using System;

using System.IO;

namespace FileApplication

{

class Program

{

static void Main(string[] args)

{

try

{

// Create an instance of StreamReader to read from a file.

// The using statement also closes the StreamReader.

using (StreamReader sr = new StreamReader("c:/jamaica.txt"))

{

string line;

// Read and display lines from the file until

// the end of the file is reached.

while ((line = sr.ReadLine()) != null)

{

Console.WriteLine(line);

}

}

}

catch (Exception e)

{

// Let the user know what went wrong.

Console.WriteLine("The file could not be read:");

Console.WriteLine(e.Message);

}

Console.ReadKey();

}

}

}

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

The StreamWriter Class

The **StreamWriter** class inherits from the abstract class TextWriter that represents a writer, which can write a series of character.

The following table describes the most commonly used methods of this class:

|  |  |
| --- | --- |
| **Sr.No.** | **Methods** |
| 1 | **public override void Close()**  Closes the current StreamWriter object and the underlying stream. |
| 2 | **public override void Flush()**  Clears all buffers for the current writer and causes any buffered data to be written to the underlying stream. |
| 3 | **public virtual void Write(bool value)**  Writes the text representation of a Boolean value to the text string or stream. (Inherited from TextWriter.) |
| 4 | **public override void Write(char value)**  Writes a character to the stream. |
| 5 | **public virtual void Write(decimal value)**  Writes the text representation of a decimal value to the text string or stream. |
| 6 | **public virtual void Write(double value)**  Writes the text representation of an 8-byte floating-point value to the text string or stream. |
| 7 | **public virtual void Write(int value)**  Writes the text representation of a 4-byte signed integer to the text string or stream. |
| 8 | **public override void Write(string value)**  Writes a string to the stream. |
| 9 | **public virtual void WriteLine()**  Writes a line terminator to the text string or stream. |

For a complete list of methods, please visit Microsoft's C# documentation.

Example

The following example demonstrates writing text data into a file using the StreamWriter class:

using System;

using System.IO;

namespace FileApplication

{

class Program

{

static void Main(string[] args)

{

string[] names = new string[] {"Zara Ali", "Nuha Ali"};

using (StreamWriter sw = new StreamWriter("names.txt"))

{

foreach (string s in names)

{

sw.WriteLine(s);

}

}

// Read and show each line from the file.

string line = "";

using (StreamReader sr = new StreamReader("names.txt"))

{

while ((line = sr.ReadLine()) != null)

{

Console.WriteLine(line);

}

}

Console.ReadKey();

}

}

}

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

Zara Ali

Nuha Ali

● File opening mode

## Open existing file for read and write

The following examples require to add namespace using System.IO;

[C#]

using (var fileStream = new FileStream(@"c:\file.txt", FileMode.Open))

{

// read from file or write to file

}

## Open existing file for reading

[C#]

using (var fileStream = new FileStream(@"c:\file.txt", FileMode.Open, FileAccess.Read))

{

// read from file

}

## Open existing file for writing

[C#]

using (var fileStream = new FileStream(@"c:\file.txt", FileMode.Open, FileAccess.Write))

{

// write to file

}

## Open file for writing (with seek to end), if the file doesn't exist create it

[C#]

using (var fileStream = new FileStream(@"c:\file.txt", FileMode.Append))

{

// append to file

}

## Create new file and open it for read and write, if the file exists overwrite it

[C#]

using (var fileStream = new FileStream(@"c:\file.txt", FileMode.Create))

{

// write to just created file

}

## Create new file and open it for read and write, if the file exists throw exception

[C#]

using (var fileStream = new FileStream(@"c:\file.txt", FileMode.CreateNew))

{

// write to just created file

}

## Files and Folders Examples

* [[C#] FileStream Open File](http://www.csharp-examples.net/filestream-open-file/) – how to open file using file stream
* [[C#] FileStream Read File](http://www.csharp-examples.net/filestream-read-file/) – how to safely read file stream
* [[C#] Read Text File](http://www.csharp-examples.net/read-text-file/) – how to read lines from text file
* [[C#] Load Text File to String](http://www.csharp-examples.net/load-text-file-to-string/) – how to load text from file to string
* [[C#] Get Files from Directory](http://www.csharp-examples.net/get-files-from-directory/) – how to get files from directory
* [[C#] Delete All Files](http://www.csharp-examples.net/delete-all-files/) – how to delete files from the specified directory
* [[C#] Get Application Directory](http://www.csharp-examples.net/get-application-directory/) – how to get application or assembly folder
* [[C#] File Attributes](http://www.csharp-examples.net/file-attributes/) – how to get or set file attributes
* [[C#] Get File Time](http://www.csharp-examples.net/file-creation-modification-time/) – how to get last modification time of a file
* [[C#] Open File With Associated Application](http://www.csharp-examples.net/open-file-with-default-application/) – how to launch the default application

File Class and Operations

● File class

Provides static methods for the creation, copying, deletion, moving, and opening of a single file, and aids in the creation of[FileStream](https://msdn.microsoft.com/en-us/library/system.io.filestream(v=vs.110).aspx) objects.

<https://www.dotnetperls.com/file>

● Directory class

<https://www.dotnetperls.com/directory>

Exposes static methods for creating, moving, and enumerating through directories and subdirectories. This class cannot be inherited.

**Directory.** The Directory type is found in System.IO. It provides methods that interact with the file system. With it we create directories, get file lists from directories, test directories for existence, and even delete directories.  
**GetFiles.** The Directory.GetFiles method returns a string array of file names. It receives the directory name and an optional pattern parameter. This is a useful and popular Directory method.

● Path class

<https://www.dotnetperls.com/path>

**With Path,** a class in the .NET Framework, we have built-in methods. This class helps when handling file paths. It is part of System.IO.

● Performing operations on a file

<http://csharp.net-informations.com/file/csharp-file-tutorial.htm>

ASP.NET

Introduction to HTML

● Introduction to ASP.NET

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

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

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

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

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

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

## ASP.NET Web Forms Model

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

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

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

* Page state
* Session state

The page state is the state of the client, i.e., the content of various input fields in the web form. The session state is the collective information obtained from various pages the user visited and worked with, i.e., the overall session state. To clear the concept, let us take an example of a shopping cart.

User adds items to a shopping cart. Items are selected from a page, say the items page, and the total collected items and price are shown on a different page, say the cart page. Only HTTP cannot keep track of all the information coming from various pages. ASP.NET session state and server side infrastructure keeps track of the information collected globally over a session.

The ASP.NET runtime carries the page state to and from the server across page requests while generating ASP.NET runtime codes, and incorporates the state of the server side components in hidden fields.

This way, the server becomes aware of the overall application state and operates in a two-tiered connected way.

## The ASP.NET Component Model

The ASP.NET component model provides various building blocks of ASP.NET pages. Basically it is an object model, which describes:

* Server side counterparts of almost all HTML elements or tags, such as <form> and <input>.
* Server controls, which help in developing complex user-interface. For example, the Calendar control or the Gridview control.

ASP.NET is a technology, which works on the .Net framework that contains all web-related functionalities. The .Net framework is made of an object-oriented hierarchy. An ASP.NET web application is made of pages. When a user requests an ASP.NET page, the IIS delegates the processing of the page to the ASP.NET runtime system.

The ASP.NET runtime transforms the .aspx page into an instance of a class, which inherits from the base class page of the .Net framework. Therefore, each ASP.NET page is an object and all its components i.e., the server-side controls are also objects.

## Components of .Net Framework 3.5

Before going to the next session on Visual Studio.Net, let us go through at the various components of the .Net framework 3.5. The following table describes the components of the .Net framework 3.5 and the job they perform:

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

● ASP.NET Version History

● Difference Between ASP and ASP.NET

**ASP** and **ASP.Net** are very different [programming languages](http://www.diffen.com/difference/Category:Programming_Languages). **ASP** is a scripting language, where as **ASP.NET** is the web formulation of a compiled language (Visual Basic, C#, J#, [C++](http://www.diffen.com/difference/C_vs_C%2B%2B), .Net). Moreover, unlike ASP, ASP.NET is an [object](http://www.diffen.com/difference/Class_vs_Object)-oriented language.

## Contents: ASP vs ASP.Net

* [1 Process Isolation](http://www.diffen.com/difference/ASP_vs_ASP.Net#Process_Isolation)
* [2 Interpretation vs. Compilation](http://www.diffen.com/difference/ASP_vs_ASP.Net#Interpretation_vs._Compilation)
  + [2.1 Performance Implications](http://www.diffen.com/difference/ASP_vs_ASP.Net#Performance_Implications)
* [3 Debugging](http://www.diffen.com/difference/ASP_vs_ASP.Net#Debugging)
* [4 References](http://www.diffen.com/difference/ASP_vs_ASP.Net#References)

## Process Isolation

ASP is run under the inetinfo.exe(IIS) process space and is therefore susceptible to application crashes due to IIS being stopped or restarted.

On the other hand, the ASP.NET worker process is a distinct process (aspnet\_wp.exe) separate from the IIS process inetinfo.exe. The process model in ASP.NET is unrelated to process isolation settings in IIS.

## Interpretation vs. Compilation

When a traditional ASP page is requested, the text of that page is parsed linearly. All content that is not server-side script is rendered as-is back to the response. All server-side script in the page is first run through the appropriate interpreter (JScript or VBScript), the output of which is then rendered back to the response.

In contrast, ASP.NET pages are always compiled into .NET [classes](http://www.diffen.com/difference/Class_vs_Object) housed within assemblies. This class includes all of the server-side code and the static HTML, so once a page is accessed for the first time (or any page within a particular directory is accessed), subsequent rendering of that page is serviced by executing compiled code. This eliminates all the inefficiencies of the scripting model of traditional ASP.

### Performance Implications

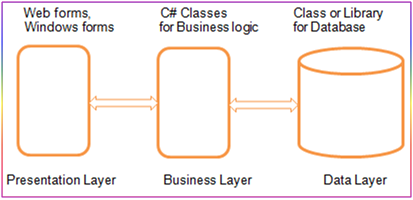
* Since ASP scripts are interpreted on the fly, there is a performance impact. A common optimization for ASP applications, therefore, is to move a lot of server-side script into precompiled COM components to improve response times. Since all components in ASP.NET are assemblies, there is no performance degradation by using server-side code.
* With ASP, intermingling server-side evaluation blocks with static HTML is less efficient than a single server-side script block, because the interpreter has to be invoked multiple times. To avoid this, many ASP developers resort to large blocks of server-side script, replacing static HTML elements with Response.Write() invocations instead. For ASP.NET, such steps are not required for performance improvement.
* ASP allows different blocks of script within a page to be written in different scripting languages. While this may be appealing in some ways, it also degrades performance by requiring that a particular page load both scripting engines (JScript, VBScript) to process a request, which takes more time and memory than using just one language. ASP.NET has "code-behind" in .aspx files that are parsed and compiled. Multiple server-side languages cannot be used within a single .aspx file.

## Debugging

Since ASP involves scripts being interpreted, debugging is difficult. But with ASP.NET, all the tools available to the .NET developer are applicable to the .aspx developer. Errors with pages are generated as compiler errors, and there is a good chance that most errors will be found at compilation time instead of runtime, because VB.NET and C# are both strongly typed languages.

● ASP.NET Architecture

[**http://www.codemag.com/article/0511061**](http://www.codemag.com/article/0511061)

**) Explain Asp.Net architecture in detail.**  
  
**Ans.**  
  
ASP.NET works on three tier architecture. This architecture is extremely popular because they increase application performance, scalability, flexibility, and code reuse. In three tier architecture, applications are divided into three major areas of functionality:  
  
- The data layer manages the data storage and retrieval.  
- The business layer maintains business rules and logic  
- The presentation layer manages the user interface and related presentation code.   
  
**The presentation Layer**  
  
Presentation layer provides the interface to the users of website or application. It contains pages like .aspx or Windows forms where data is presented to the user or user enters the information. This layer communicates with business layer. In ASP.NET the code behind file and HTML file are separate file. The HTML file defines the look and layout of the web form and the code behind file contains the presentation logic. This separation helps designers and developers. Designers don't have to worry about business logic to make user interface changes, and developers don't have to worry about to update code.  
  
**The business Layer**  
  
Business layer or Application layer acts between Application layer and Data Access Layer. This layer contains our logic, validating the data and other functions. In this layer programmer access the data access layer (also known as the DAL), functionality. As example suppose any user wants to register and he fills all the detail and click the submit button on presentation layer then call goes to business layer. In business layer programmers only access the function and pass the parameter that is provided by the end user as example.  
**InserUserDetails**(string Username, string Password, string Email, string Firstname, string Lastname, string Address, float Salary);  
  
**The data Layer**  
  
All the code related to database is written in this layer. It contains methods to connect with database. It also contains the methods to perform insert, update, delete, select data according to the SQL query.  
  
  
  
**The main advantages of the 3 Tier Architecture is as follows:**  
  
**Flexibility** – All the layers have separate code so it makes the application much more flexible to changes.  
  
**Maintainability** – You can change to the components in one layer and it will no effect on any others layers. So you can maintain the application very easily.  
  
**Reusability** – A single library/component may be accessed by multiple components in the presentation layer. It easier to implement re-usable components by separating the application into multiple layers makes.   
  
**Scalability** – You can deploy application components across multiple servers, so it makes the system much more scalable.

● Inline Technique & CodeBehind Technique

Although Microsoft® Visual Studio® .NET makes it easy to create and work with Web Forms pages using the ASP.NET code-behind model, you might find yourself working with single-file Web Forms pages by circumstance or by preference. This article gives an overview of the differences between the two models, describes how to work with single-file Web Forms pages in Visual Studio, and shows you how to convert single-file *.aspx* pages to code-behind Web Forms pages.

There are a few differences in the processing of code-behind and single-file pages.

|  |  |
| --- | --- |
| Code Behind | Single File |
| The HTML and controls are in the *.aspx* file, and the code is in a separate *.aspx.vb* or *.aspx.cs* file. | The code is in <script> blocks in the same*.aspx* file that contains the HTML and controls. |
| The code for the page is compiled into a separate class from which the *.aspx* file derives. | The *.aspx* file derives from the Page class. |
| All project class files (without the *.aspx* file itself) are compiled into a *.dll* file, which is deployed to the server without any source code. When a request for the page is received, then an instance of the project *.dll* file is created and executed. | When the page is deployed, the source code is deployed along with the Web Forms page, because it is physically in the *.aspx* file. However, you do not see the code, only the results are rendered when the page runs. |

[Quote MSDN: Working with Single-File Web Forms Pages in Visual Studio .NET]

## Using the code

My personal preference is Code Behind. Most free ASP.NET hosting servers don’t allow Code Behind, not sure why, yet.

What I sometimes do is I write a base class, which derives from Page class, and all my pages derive from my base class. But on a Single File Web Form, it derives from the Page class. So, this limits you to create your own base derived class.

Let's start off with a simple app that uses Code Behind and convert that into Single File.

1. Open an existing project, or create a new ASP.NET Web application.
2. On the Project menu, click Add HTML Page.
3. Name the new page with the *.aspx* extension, for example, *SingleForm1.aspx*.
4. Design the form.
5. When your form works in code behind now, we can move it to a Single File.
6. Change your design view to HTML.
7. Replace your Page directive with:

Hide   Copy Code

<%@ Page language="c#" %>

1. Between the <Head></Head> tags, add the following code:

Hide   Copy Code

<Head>

<script language="CS" runat="server"></script>

</Head>

1. Copy and paste your Code Behind code in between the <script> tags.

Note that I don't have any private / public / ... modifier.

Hide   Copy Code

<Head>

<script language="CS" runat="server">

void Page\_Load(object sender, System.EventArgs e)

{

*// Put user code to initialize the page here*

}

void btnLogon\_Click(object sender, System.EventArgs e)

{

this.txtUserID.Text = "Logon";

this.txtPassword.Text = "";

}

</script>

1. Now. we need to “Register” the event. Due to the fact the I could find the InitializeComponentmethod, I registered my events in the Page\_Load.

Hide   Copy Code

<Head>

<script language="CS" runat="server">

void Page\_Load(object sender, System.EventArgs e)

{

*// Put user code to initialize the page here*

**this.btnLogon.Click +=**

**new System.EventHandler(this.btnLogon\_Click);**

}

void btnLogon\_Click(object sender, System.EventArgs e)

{

this.txtUserID.Text = "Logon";

this.txtPassword.Text = "";

}

</script>

1. There we go.

#### Now how do I use classes in inline code???

1. Well to start off, you can create a *.cs* file where your class code is declared.
2. Between the <Head></Head> tags, add the following code:

Hide   Copy Code

<Head>

<script language="cs" runat="server" src= "MySource.cs"/>

**<script language="CS" runat="server"></script>**

</Head>

1. Note that in your source file, you don’t include the namespace. You only declare the class:

Hide   Copy Code

public class MyClass1 { }

public class MyClass2

{

protected int Index = 0;

public MyClass2()

{

}

}

public class MyClass3 : System.Collections.CollectionBase

{

public int this[int Index]

{

get

{

return (int)List[Index];

}

set

{

List[Index] = value;

}

}

}

1. And you use it the same as what you would have in code behind.

Hide   Copy Code

<Head>

<script language="CS" runat="server">

void Page\_Load(object sender, System.EventArgs e)

{

*// Put user code to initialize the page here*

this.btnLogon.Click +=

new System.EventHandler(this.btnLogon\_Click);

MyClass3 class3 = new MyClass3();

class[0] = "Test";

}

void btnLogon\_Click(object sender, System.EventArgs e)

{

this.txtUserID.Text = "Logon";

this.txtPassword.Text = "";

}

</script>

● Server Controls

<https://www.tutorialspoint.com/asp.net/asp.net_server_controls.htm>

Controls are small building blocks of the graphical user interface, which include text boxes, buttons, check boxes, list boxes, labels, and numerous other tools. Using these tools, the users can enter data, make selections and indicate their preferences.

Controls are also used for structural jobs, like validation, data access, security, creating master pages, and data manipulation.

ASP.NET uses five types of web controls, which are:

* HTML controls
* HTML Server controls
* ASP.NET Server controls
* ASP.NET Ajax Server controls
* User controls and custom controls

ASP.NET server controls are the primary controls used in ASP.NET. These controls can be grouped into the following categories:

* **Validation controls** - These are used to validate user input and they work by running client-side script.
* **Data source controls** - These controls provides data binding to different data sources.
* **Data view controls** - These are various lists and tables, which can bind to data from data sources for displaying.
* **Personalization controls** - These are used for personalization of a page according to the user preferences, based on user information.
* **Login and security controls** - These controls provide user authentication.
* **Master pages** - These controls provide consistent layout and interface throughout the application.
* **Navigation controls** - These controls help in navigation. For example, menus, tree view etc.
* **Rich controls** - These controls implement special features. For example, AdRotator, FileUpload, and Calendar control.

The syntax for using server controls is:

<asp:controlType ID ="ControlID" runat="server" Property1=value1 [Property2=value2] />

In addition, visual studio has the following features, to help produce in error-free coding:

* Dragging and dropping of controls in design view
* IntelliSense feature that displays and auto-completes the properties
* The properties window to set the property values directly

## Properties of the Server Controls

ASP.NET server controls with a visual aspect are derived from the WebControl class and inherit all the properties, events, and methods of this class.

The WebControl class itself and some other server controls that are not visually rendered are derived from the System.Web.UI.Control class. For example, PlaceHolder control or XML control.

ASP.Net server controls inherit all properties, events, and methods of the WebControl and System.Web.UI.Control class.

The following table shows the inherited properties, common to all server controls:

|  |  |
| --- | --- |
| **Property** | **Description** |
| AccessKey | Pressing this key with the Alt key moves focus to the control. |
| Attributes | It is the collection of arbitrary attributes (for rendering only) that do not correspond to properties on the control. |
| BackColor | Background color. |
| BindingContainer | The control that contains this control's data binding. |
| BorderColor | Border color. |
| BorderStyle | Border style. |
| BorderWidth | Border width. |
| CausesValidation | Indicates if it causes validation. |
| ChildControlCreated | It indicates whether the server control's child controls have been created. |
| ClientID | Control ID for HTML markup. |
| Context | The HttpContext object associated with the server control. |
| Controls | Collection of all controls contained within the control. |
| ControlStyle | The style of the Web server control. |
| CssClass | CSS class |
| DataItemContainer | Gets a reference to the naming container if the naming container implements IDataItemContainer. |
| DataKeysContainer | Gets a reference to the naming container if the naming container implements IDataKeysControl. |
| DesignMode | It indicates whether the control is being used on a design surface. |
| DisabledCssClass | Gets or sets the CSS class to apply to the rendered HTML element when the control is disabled. |
| Enabled | Indicates whether the control is grayed out. |
| EnableTheming | Indicates whether theming applies to the control. |
| EnableViewState | Indicates whether the view state of the control is maintained. |
| Events | Gets a list of event handler delegates for the control. |
| Font | Font. |
| Forecolor | Foreground color. |
| HasAttributes | Indicates whether the control has attributes set. |
| HasChildViewState | Indicates whether the current server control's child controls have any saved view-state settings. |
| Height | Height in pixels or %. |
| ID | Identifier for the control. |
| IsChildControlStateCleared | Indicates whether controls contained within this control have control state. |
| IsEnabled | Gets a value indicating whether the control is enabled. |
| IsTrackingViewState | It indicates whether the server control is saving changes to its view state. |
| IsViewStateEnabled | It indicates whether view state is enabled for this control. |
| LoadViewStateById | It indicates whether the control participates in loading its view state by ID instead of index. |
| Page | Page containing the control. |
| Parent | Parent control. |
| RenderingCompatibility | It specifies the ASP.NET version that the rendered HTML will be compatible with. |
| Site | The container that hosts the current control when rendered on a design surface. |
| SkinID | Gets or sets the skin to apply to the control. |
| Style | Gets a collection of text attributes that will be rendered as a style attribute on the outer tag of the Web server control. |
| TabIndex | Gets or sets the tab index of the Web server control. |
| TagKey | Gets the HtmlTextWriterTag value that corresponds to this Web server control. |
| TagName | Gets the name of the control tag. |
| TemplateControl | The template that contains this control. |
| TemplateSourceDirectory | Gets the virtual directory of the page or control containing this control. |
| ToolTip | Gets or sets the text displayed when the mouse pointer hovers over the web server control. |
| UniqueID | Unique identifier. |
| ViewState | Gets a dictionary of state information that saves and restores the view state of a server control across multiple requests for the same page. |
| ViewStateIgnoreCase | It indicates whether the StateBag object is case-insensitive. |
| ViewStateMode | Gets or sets the view-state mode of this control. |
| Visible | It indicates whether a server control is visible. |
| Width | Gets or sets the width of the Web server control. |

## Methods of the Server Controls

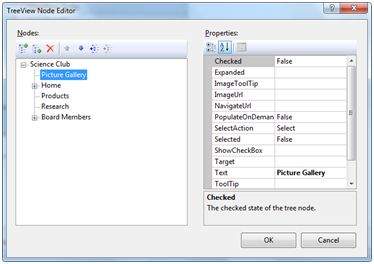
The following table provides the methods of the server controls:

|  |  |
| --- | --- |
| **Method** | **Description** |
| AddAttributesToRender | Adds HTML attributes and styles that need to be rendered to the specified HtmlTextWriterTag. |
| AddedControl | Called after a child control is added to the Controls collection of the control object. |
| AddParsedSubObject | Notifies the server control that an element, either XML or HTML, was parsed, and adds the element to the server control's control collection. |
| ApplyStyleSheetSkin | Applies the style properties defined in the page style sheet to the control. |
| ClearCachedClientID | Infrastructure. Sets the cached ClientID value to null. |
| ClearChildControlState | Deletes the control-state information for the server control's child controls. |
| ClearChildState | Deletes the view-state and control-state information for all the server control's child controls. |
| ClearChildViewState | Deletes the view-state information for all the server control's child controls. |
| CreateChildControls | Used in creating child controls. |
| CreateControlCollection | Creates a new ControlCollection object to hold the child controls. |
| CreateControlStyle | Creates the style object that is used to implement all style related properties. |
| DataBind | Binds a data source to the server control and all its child controls. |
| DataBind(Boolean) | Binds a data source to the server control and all its child controls with an option to raise the DataBinding event. |
| DataBindChildren | Binds a data source to the server control's child controls. |
| Dispose | Enables a server control to perform final clean up before it is released from memory. |
| EnsureChildControls | Determines whether the server control contains child controls. If it does not, it creates child controls. |
| EnsureID | Creates an identifier for controls that do not have an identifier. |
| Equals(Object) | Determines whether the specified object is equal to the current object. |
| Finalize | Allows an object to attempt to free resources and perform other cleanup operations before the object is reclaimed by garbage collection. |
| FindControl(String) | Searches the current naming container for a server control with the specified id parameter. |
| FindControl(String, Int32) | Searches the current naming container for a server control with the specified id and an integer. |
| Focus | Sets input focus to a control. |
| GetDesignModeState | Gets design-time data for a control. |
| GetType | Gets the type of the current instance. |
| GetUniqueIDRelativeTo | Returns the prefixed portion of the UniqueID property of the specified control. |
| HasControls | Determines if the server control contains any child controls. |
| HasEvents | Indicates whether events are registered for the control or any child controls. |
| IsLiteralContent | Determines if the server control holds only literal content. |
| LoadControlState | Restores control-state information. |
| LoadViewState | Restores view-state information. |
| MapPathSecure | Retrieves the physical path that a virtual path, either absolute or relative, maps to. |
| MemberwiseClone | Creates a shallow copy of the current object. |
| MergeStyle | Copies any nonblank elements of the specified style to the web control, but does not overwrite any existing style elements of the control. |
| OnBubbleEvent | Determines whether the event for the server control is passed up the page's UI server control hierarchy. |
| OnDataBinding | Raises the data binding event. |
| OnInit | Raises the Init event. |
| OnLoad | Raises the Load event. |
| OnPreRender | Raises the PreRender event. |
| OnUnload | Raises the Unload event. |
| OpenFile | Gets a Stream used to read a file. |
| RemovedControl | Called after a child control is removed from the controls collection of the control object. |
| Render | Renders the control to the specified HTML writer. |
| RenderBeginTag | Renders the HTML opening tag of the control to the specified writer. |
| RenderChildren | Outputs the contents of a server control's children to a provided HtmlTextWriter object, which writes the contents to be rendered on the client. |
| RenderContents | Renders the contents of the control to the specified writer. |
| RenderControl(HtmlTextWriter) | Outputs server control content to a provided HtmlTextWriter object and stores tracing information about the control if tracing is enabled. |
| RenderEndTag | Renders the HTML closing tag of the control into the specified writer. |
| ResolveAdapter | Gets the control adapter responsible for rendering the specified control. |
| SaveControlState | Saves any server control state changes that have occurred since the time the page was posted back to the server. |
| SaveViewState | Saves any state that was modified after the TrackViewState method was invoked. |
| SetDesignModeState | Sets design-time data for a control. |
| ToString | Returns a string that represents the current object. |
| TrackViewState | Causes the control to track changes to its view state so that they can be stored in the object's view state property. |

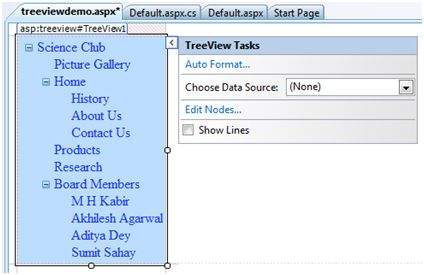
## Example

Let us look at a particular server control - a tree view control. A Tree view control comes under navigation controls. Other Navigation controls are: Menu control and SiteMapPath control.

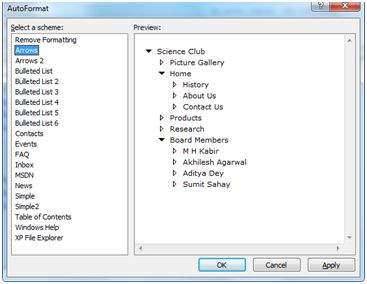
Add a tree view control on the page. Select Edit Nodes... from the tasks. Edit each of the nodes using the Tree view node editor as shown:



Once you have created the nodes, it looks like the following in design view:



The AutoFormat... task allows you to format the tree view as shown:



Add a label control and a text box control on the page and name them lblmessage and txtmessage respectively.

Write a few lines of code to ensure that when a particular node is selected, the label control displays the node text and the text box displays all child nodes under it, if any. The code behind the file should look like this:

using System;

using System.Collections;

using System.Configuration;

using System.Data;

using System.Linq;

using System.Web;

using System.Web.Security;

using System.Web.UI;

using System.Web.UI.HtmlControls;

using System.Web.UI.WebControls;

using System.Web.UI.WebControls.WebParts;

using System.Xml.Linq;

namespace eventdemo {

public partial class treeviewdemo : System.Web.UI.Page {

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

txtmessage.Text = " ";

}

protected void TreeView1\_SelectedNodeChanged(object sender, EventArgs e) {

txtmessage.Text = " ";

lblmessage.Text = "Selected node changed to: " + TreeView1.SelectedNode.Text;

TreeNodeCollection childnodes = TreeView1.SelectedNode.ChildNodes;

if(childnodes != null) {

txtmessage.Text = " ";

foreach (TreeNode t in childnodes) {

txtmessage.Text += t.Value;

}

}

}

}

}

● Page Lifecycle

<https://www.tutorialspoint.com/asp.net/asp.net_life_cycle.htm>

## ASP.NET Page Life Cycle

When a page is requested, it is loaded into the server memory, processed, and sent to the browser. Then it is unloaded from the memory. At each of these steps, methods and events are available, which could be overridden according to the need of the application. In other words, you can write your own code to override the default code.

The Page class creates a hierarchical tree of all the controls on the page. All the components on the page, except the directives, are part of this control tree. You can see the control tree by adding trace= "true" to the page directive. We will cover page directives and tracing under 'directives' and 'event handling'.

The page life cycle phases are:

* Initialization
* Instantiation of the controls on the page
* Restoration and maintenance of the state
* Execution of the event handler codes
* Page rendering

Understanding the page cycle helps in writing codes for making some specific thing happen at any stage of the page life cycle. It also helps in writing custom controls and initializing them at right time, populate their properties with view-state data and run control behavior code.

Following are the different stages of an ASP.NET page:

* **Page request** - When ASP.NET gets a page request, it decides whether to parse and compile the page, or there would be a cached version of the page; accordingly the response is sent.
* **Starting of page life cycle** - At this stage, the Request and Response objects are set. If the request is an old request or post back, the IsPostBack property of the page is set to true. The UICulture property of the page is also set.
* **Page initialization** - At this stage, the controls on the page are assigned unique ID by setting the UniqueID property and the themes are applied. For a new request, postback data is loaded and the control properties are restored to the view-state values.
* **Page load** - At this stage, control properties are set using the view state and control state values.
* **Validation** - Validate method of the validation control is called and on its successful execution, the IsValid property of the page is set to true.
* **Postback event handling** - If the request is a postback (old request), the related event handler is invoked.
* **Page rendering** - At this stage, view state for the page and all controls are saved. The page calls the Render method for each control and the output of rendering is written to the OutputStream class of the Response property of page.
* **Unload** - The rendered page is sent to the client and page properties, such as Response and Request, are unloaded and all cleanup done.

● Postback

## Introduction

In the old HTML, the only way to make something updated on the webpage is to resend a new webpage to the client browser. That's what ASP used to do, you have to do this thing call a "PostBack" to send an updated page to the client.

In ASP .NET, you don't have to resend the entire webpage. You can now use AJAX, or other ASP.NET controls such that you don't have to resend the entire webpage.

If you visit some old website, you would notice that once you click something, the entire page has to be refresh, this is the old ASP. In most of the modern website, you will notice your browser doesn't have to refresh the entire page, it only updates the part of the content that needs to be updated. For example, in Stackoverflow, you see the page update only the content, not the entire webpage.

Programming model in old ASP for using POST method in form is to post the values of a Form to a second page. The second asp page will receive the data and process it for doing any validation or processing on the server side.

With ASP .Net, the whole model has changed. Each of the asp .net pages will be a separate entity with ability to process its own posted data. That is, the values of the Form are posted to the same page and the very same page can process the data. This model is called post back.

Each Asp .net page when loaded goes through a regular creation and destruction cycle like Initialization, Page load etc., in the beginning and unload while closing it. This Postback is a read only property with each Asp .Net Page (System.Web.UI.Page) class. This is false when the first time the page is loaded and is true when the page is submitted and processed. This enables users to write the code depending on if the PostBack is true or false (with the use of the function Page.IsPostBack()).

## Understanding the PostBack

PostBack is the name given to the process of submitting an ASP.NET page to the server for processing. PostBack is done if certain credentials of the page are to be checked against some sources (such as verification of username and password using database). This is something that a client machine is not able to accomplish and thus these details have to be 'posted back' to the server.

A post back is round trip from the client (Browser) to the server and then back to the client. This enables you page to go through the asp engine on the server and any dynamic content to be updated.

### **Using the Code**

Post back is implemented with the use javascript in the client side. The HTML page generated for each .aspx page will have the action property of the form tag set to the same page. This makes the page to be posted on to itself. If we check the entry on the HTML file, it will look something like this.

Hide   Copy Code

<form name=&rdquo;\_ctl1&Prime; method=&rdquo;post&rdquo; action=&rdquo;pagename.aspx?getparameter1=134&Prime; language="&rdquo;javascript&rdquo;" onsubmit=&rdquo;if (!ValidatorOnSubmit()) return false;&rdquo; id=&rdquo;\_ctl1&Prime; >

Also, all the validation code that is written (Required Field Validation, Regular Expression validation etc.,) will all be processed at the client side using the .js(javascript) file present in the webserver\_wwwroot/aspnet\_client folder.

With this new ASP .Net model, even if the user wants to post the data to a different .aspx page, the web server will check for the runat=’server’ tag in the form tag and post the web form to the same .aspx page. A simple declaration as in the following code snippet will be enough to create such a web form.

Hide   Copy Code

<form id=&rdquo;form1&Prime; runat="&rdquo;server&rdquo;" >

<!&ndash; place the controls inside &ndash;>

</form>

## Understanding the AutoPostBack

AutopostBack is a property which you assign to web controls if you want to post back the page when any event occurs at them.

### **Using the Code**

Hide   Copy Code

<asp:DropDownList id="id" runat="server" AutoPostBack="true" OnSelectIndexChanged="..."/>

This ddl not need asp:button for example in order to post, when you change ddl is autoposted.

Defaut value on control is false.

ASP.NET also adds two additional hidden input fields that are used to pass information back to the server. This information consists of ID of the Control that raised the event and any additional information if needed. These fields will empty initially as shown below,

The \_doPostBack() function has the responsibility for setting these values with the appropriate information about the event and the submitting the form. The \_doPostBack() function is shown below:

Hide   Copy Code

<script language="text/javascript">

function \_\_doPostBack(eventTarget, eventArgument) {

if (!theForm.onsubmit || (theForm.onsubmit() != false)) {

theForm.\_\_EVENTTARGET.value = eventTarget;

theForm.\_\_EVENTARGUMENT.value = eventArgument;

theForm.submit();

}

</script>

ASP.NET generates the \_doPostBack() function automatically, provided at least one control on the page uses automatic postbacks.

## Understanding the Page.IsPostBack

* Gets a value that indicates whether the page is being rendered for the first time or is being loaded in response to a postback.
* The IsPostBack property can be used to determine if the page is submitted to itself. When a form is submitted back to the same page that contains it, it's called a post back. ASP.NET provides a property called IsPostBack that is TRUE when the page is being loaded as a result of a post back, and is FALSE otherwise.

### **Using the code**

Hide   Copy Code

private void Page\_Load()

{

if (!IsPostBack)

{

*//You can write here the code, which you want to execute in the first time when the page is loaded.*

FunctionToBindSomething();

}

}

## Usage

1. The One of the Most Common Use of AutoPostBack is for [Cascading Dropdown](http://www.codeproject.com/Articles/32725/Using-CascadingDropDown-with-a-Database) list.

● Page Directives

<https://www.tutorialspoint.com/asp.net/asp.net_directives.htm>

The Page Directive

The Page directive defines the attributes specific to the page file for the page parser and the compiler.

The basic syntax of Page directive is:

<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="\_Default" Trace="true" %>

The attributes of the Page directive are:

|  |  |
| --- | --- |
| **Attributes** | **Description** |
| AutoEventWireup | The Boolean value that enables or disables page events that are being automatically bound to methods; for example, Page\_Load. |
| Buffer | The Boolean value that enables or disables HTTP response buffering. |
| ClassName | The class name for the page. |
| ClientTarget | The browser for which the server controls should render content. |
| CodeFile | The name of the code behind file. |
| Debug | The Boolean value that enables or disables compilation with debug symbols. |
| Description | The text description of the page, ignored by the parser. |
| EnableSessionState | It enables, disables, or makes session state read-only. |
| EnableViewState | The Boolean value that enables or disables view state across page requests. |
| ErrorPage | URL for redirection if an unhandled page exception occurs. |
| Inherits | The name of the code behind or other class. |
| Language | The programming language for code. |
| Src | The file name of the code behind class. |
| Trace | It enables or disables tracing. |
| TraceMode | It indicates how trace messages are displayed, and sorted by time or category. |
| Transaction | It indicates if transactions are supported. |
| ValidateRequest | The Boolean value that indicates whether all input data is validated against a hardcoded list of values. |

Web Form Control

<https://www.tutorialspoint.com/asp.net/asp.net_basic_controls.htm>

● Text Box

Text box controls are typically used to accept input from the user. A text box control can accept one or more lines of text depending upon the settings of the TextMode attribute.

Label controls provide an easy way to display text which can be changed from one execution of a page to the next. If you want to display text that does not change, you use the literal text.

Basic syntax of text control:

<asp:TextBox ID="txtstate" runat="server" ></asp:TextBox>

Common Properties of the Text Box and Labels:

|  |  |
| --- | --- |
| **Property** | **Description** |
| TextMode | Specifies the type of text box. SingleLine creates a standard text box, MultiLIne creates a text box that accepts more than one line of text and the Password causes the characters that are entered to be masked. The default is SingleLine. |
| Text | The text content of the text box. |
| MaxLength | The maximum number of characters that can be entered into the text box. |
| Wrap | It determines whether or not text wraps automatically for multi-line text box; default is true. |
| ReadOnly | Determines whether the user can change the text in the box; default is false, i.e., the user can change the text. |
| Columns | The width of the text box in characters. The actual width is determined based on the font that is used for the text entry. |
| Rows | The height of a multi-line text box in lines. The default value is 0, means a single line text box. |

The mostly used attribute for a label control is 'Text', which implies the text displayed on the label.

● Check Box

● Radio Button

A check box displays a single option that the user can either check or uncheck and radio buttons present a group of options from which the user can select just one option.

To create a group of radio buttons, you specify the same name for the GroupName attribute of each radio button in the group. If more than one group is required in a single form, then specify a different group name for each group.

If you want check box or radio button to be selected when the form is initially displayed, set its Checked attribute to true. If the Checked attribute is set to true for multiple radio buttons in a group, then only the last one is considered as true.

Basic syntax of check box:

<asp:CheckBox ID= "chkoption" runat= "Server">

</asp:CheckBox>

Basic syntax of radio button:

<asp:RadioButton ID= "rdboption" runat= "Server">

</asp: RadioButton>

Common properties of check boxes and radio buttons:

|  |  |
| --- | --- |
| **Property** | **Description** |
| Text | The text displayed next to the check box or radio button. |
| Checked | Specifies whether it is selected or not, default is false. |
| GroupName | Name of the group the control belongs to. |

● List Box

● Drop Down List

## List Controls

ASP.NET provides the following controls

* Drop-down list,
* List box,
* Radio button list,
* Check box list,
* Bulleted list.

These control let a user choose from one or more items from the list. List boxes and drop-down lists contain one or more list items. These lists can be loaded either by code or by the ListItemCollection editor.

Basic syntax of list box control:

<asp:ListBox ID="ListBox1" runat="server" AutoPostBack="True" OnSelectedIndexChanged="ListBox1\_SelectedIndexChanged">

</asp:ListBox>

Basic syntax of drop-down list control:

<asp:DropDownList ID="DropDownList1" runat="server" AutoPostBack="True" OnSelectedIndexChanged="DropDownList1\_SelectedIndexChanged">

</asp:DropDownList>

Common properties of list box and drop-down Lists:

|  |  |
| --- | --- |
| **Property** | **Description** |
| Items | The collection of ListItem objects that represents the items in the control. This property returns an object of type ListItemCollection. |
| Rows | Specifies the number of items displayed in the box. If actual list contains more rows than displayed then a scroll bar is added. |
| SelectedIndex | The index of the currently selected item. If more than one item is selected, then the index of the first selected item. If no item is selected, the value of this property is -1. |
| SelectedValue | The value of the currently selected item. If more than one item is selected, then the value of the first selected item. If no item is selected, the value of this property is an empty string (""). |
| SelectionMode | Indicates whether a list box allows single selections or multiple selections. |

Common properties of each list item objects:

|  |  |
| --- | --- |
| **Property** | **Description** |
| Text | The text displayed for the item. |
| Selected | Indicates whether the item is selected. |
| Value | A string value associated with the item. |

It is important to notes that:

* To work with the items in a drop-down list or list box, you use the Items property of the control. This property returns a ListItemCollection object which contains all the items of the list.
* The SelectedIndexChanged event is raised when the user selects a different item from a drop-down list or list box.

● Image Control

Image Control

The image control is used for displaying images on the web page, or some alternative text, if the image is not available.

Basic syntax for an image control:

<asp:Image ID="Image1" runat="server">

It has the following important properties:

|  |  |
| --- | --- |
| **Property** | **Description** |
| AlternateText | Alternate text to be displayed in absence of the image. |
| ImageAlign | Alignment options for the control. |
| ImageUrl | Path of the image to be displayed by the control. |

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

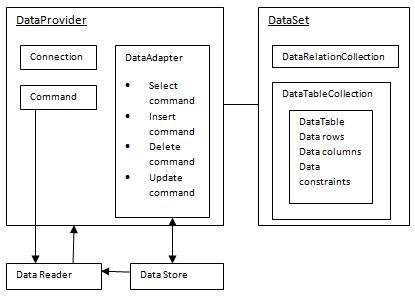
ADO.NET

Introduction to ADO.NET

● Understanding ADO.NET

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

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



## The DataSet Class

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

Following table shows some important properties of the DataSet class:

|  |  |
| --- | --- |
| **Properties** | **Description** |
| CaseSensitive | Indicates whether string comparisons within the data tables are case-sensitive. |
| Container | Gets the container for the component. |
| DataSetName | Gets or sets the name of the current data set. |
| DefaultViewManager | Returns a view of data in the data set. |
| DesignMode | Indicates whether the component is currently in design mode. |
| EnforceConstraints | Indicates whether constraint rules are followed when attempting any update operation. |
| Events | Gets the list of event handlers that are attached to this component. |
| ExtendedProperties | Gets the collection of customized user information associated with the DataSet. |
| HasErrors | Indicates if there are any errors. |
| IsInitialized | Indicates whether the DataSet is initialized. |
| Locale | Gets or sets the locale information used to compare strings within the table. |
| Namespace | Gets or sets the namespace of the DataSet. |
| Prefix | Gets or sets an XML prefix that aliases the namespace of the DataSet. |
| Relations | Returns the collection of DataRelation objects. |
| Tables | Returns the collection of DataTable objects. |

The following table shows some important methods of the DataSet class:

|  |  |
| --- | --- |
| **Methods** | **Description** |
| AcceptChanges | Accepts all changes made since the DataSet was loaded or this method was called. |
| BeginInit | Begins the initialization of the DataSet. The initialization occurs at run time. |
| Clear | Clears data. |
| Clone | Copies the structure of the DataSet, including all DataTable schemas, relations, and constraints. Does not copy any data. |
| Copy | Copies both structure and data. |
| CreateDataReader() | Returns a DataTableReader with one result set per DataTable, in the same sequence as the tables appear in the Tables collection. |
| CreateDataReader(DataTable[]) | Returns a DataTableReader with one result set per DataTable. |
| EndInit | Ends the initialization of the data set. |
| Equals(Object) | Determines whether the specified Object is equal to the current Object. |
| Finalize | Free resources and perform other cleanups. |
| GetChanges | Returns a copy of the DataSet with all changes made since it was loaded or the AcceptChanges method was called. |
| GetChanges(DataRowState) | Gets a copy of DataSet with all changes made since it was loaded or the AcceptChanges method was called, filtered by DataRowState. |
| GetDataSetSchema | Gets a copy of XmlSchemaSet for the DataSet. |
| GetObjectData | Populates a serialization information object with the data needed to serialize the DataSet. |
| GetType | Gets the type of the current instance. |
| GetXML | Returns the XML representation of the data. |
| GetXMLSchema | Returns the XSD schema for the XML representation of the data. |
| HasChanges() | Gets a value indicating whether the DataSet has changes, including new, deleted, or modified rows. |
| HasChanges(DataRowState) | Gets a value indicating whether the DataSet has changes, including new, deleted, or modified rows, filtered by DataRowState. |
| IsBinarySerialized | Inspects the format of the serialized representation of the DataSet. |
| Load(IDataReader, LoadOption, DataTable[]) | Fills a DataSet with values from a data source using the supplied IDataReader, using an array of DataTable instances to supply the schema and namespace information. |
| Load(IDataReader, LoadOption, String[]) | Fills a DataSet with values from a data source using the supplied IDataReader, using an array of strings to supply the names for the tables within the DataSet. |
| Merge() | Merges the data with data from another DataSet. This method has different overloaded forms. |
| ReadXML() | Reads an XML schema and data into the DataSet. This method has different overloaded forms. |
| ReadXMLSchema(0) | Reads an XML schema into the DataSet. This method has different overloaded forms. |
| RejectChanges | Rolls back all changes made since the last call to AcceptChanges. |
| WriteXML() | Writes an XML schema and data from the DataSet. This method has different overloaded forms. |
| WriteXMLSchema() | Writes the structure of the DataSet as an XML schema. This method has different overloaded forms. |

## The DataTable Class

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

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

The following table shows some important methods of the DataTable class:

|  |  |
| --- | --- |
| **Methods** | **Description** |
| AcceptChanges | Commits all changes since the last AcceptChanges. |
| Clear | Clears all data from the table. |
| GetChanges | Returns a copy of the DataTable with all changes made since the AcceptChanges method was called. |
| GetErrors | Returns an array of rows with errors. |
| ImportRows | Copies a new row into the table. |
| LoadDataRow | Finds and updates a specific row, or creates a new one, if not found any. |
| Merge | Merges the table with another DataTable. |
| NewRow | Creates a new DataRow. |
| RejectChanges | Rolls back all changes made since the last call to AcceptChanges. |
| Reset | Resets the table to its original state. |
| Select | Returns an array of DataRow objects. |

## The DataRow Class

The DataRow object represents a row in a table. It has the following important properties:

|  |  |
| --- | --- |
| **Properties** | **Description** |
| HasErrors | Indicates if there are any errors. |
| Items | Gets or sets the data stored in a specific column. |
| ItemArrays | Gets or sets all the values for the row. |
| Table | Returns the parent table. |

The following table shows some important methods of the DataRow class:

|  |  |
| --- | --- |
| **Methods** | **Description** |
| AcceptChanges | Accepts all changes made since this method was called. |
| BeginEdit | Begins edit operation. |
| CancelEdit | Cancels edit operation. |
| Delete | Deletes the DataRow. |
| EndEdit | Ends the edit operation. |
| GetChildRows | Gets the child rows of this row. |
| GetParentRow | Gets the parent row. |
| GetParentRows | Gets parent rows of DataRow object. |
| RejectChanges | Rolls back all changes made since the last call to AcceptChanges. |

## The DataAdapter Object

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

## The DataReader Object

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

## DbCommand and DbConnection Objects

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

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

## Example

So far, we have used tables and databases already existing in our computer. In this example, we will create a table, add column, rows and data into it and display the table using a GridView object.

The source file code is as given:

<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="createdatabase.\_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

<head runat="server">

<title>

Untitled Page

</title>

</head>

<body>

<form id="form1" runat="server">

<div>

<asp:GridView ID="GridView1" runat="server">

</asp:GridView>

</div>

</form>

</body>

</html>

The code behind file is as given:

namespace createdatabase

{

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

{

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

{

if (!IsPostBack)

{

DataSet ds = CreateDataSet();

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

GridView1.DataBind();

}

}

private DataSet CreateDataSet()

{

//creating a DataSet object for tables

DataSet dataset = new DataSet();

// creating the student table

DataTable Students = CreateStudentTable();

dataset.Tables.Add(Students);

return dataset;

}

private DataTable CreateStudentTable()

{

DataTable Students = new DataTable("Student");

// adding columns

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

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

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

// adding rows

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

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

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

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

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

return Students;

}

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

{

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

}

//adding data into the table

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

{

DataRow newrow = table.NewRow();

newrow["StudentID"] = id;

newrow["StudentName"] = name;

newrow["StudentCity"] = city;

table.Rows.Add(newrow);

}

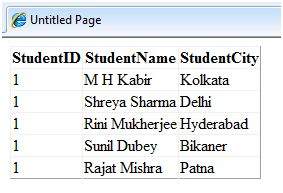
}

}

When you execute the program, observe the following:

* The application first creates a data set and binds it with the grid view control using the DataBind() method of the GridView control.
* The Createdataset() method is a user defined function, which creates a new DataSet object and then calls another user defined method CreateStudentTable() to create the table and add it to the Tables collection of the data set.
* The CreateStudentTable() method calls the user defined methods AddNewColumn() and AddNewRow() to create the columns and rows of the table as well as to add data to the rows.

When the page is executed, it returns the rows of the table as shown:



● ADO VS ADO.NET

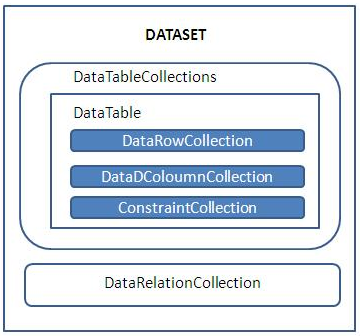
<http://net-informations.com/faq/ado/ado-difference.htm>

**Classic ADO and ADO.NET**

ADO is based on COM Technology and it used OLEDB data provider for accessing data . It has a limited number of data types which are defined by the COM standard. ADO.NET basically designed by .NET Framework for smooth interaction of application and database, it support large and rich datatypes.

Classic ADO used OLE DB data provider to access data and is COM based, while ADO.net uses XML as the format for transmitting data to and from your database and applications. It is compatible with any component on any platform that understands XML.

ADO works with connected data architecture. That means, when you access the data from data source, such as viewing or updating data, ADO recordset is keeping connection with the data source. This is barring, of course, you have to develop special routines to pull all your data into temporary tables.



ADO.NET uses data in a disconnected manner. When you access data, ADO.NET makes a copy of the data using XML. ADO.NET only holds the connection open long enough to either pull down the data or to make any requested updates and immediatly close the connection after operation. This makes ADO.NET efficient to use in networking environment.

Classic ADO has one main object that is used to reference data, called the Recordset object, it acts like a single table or query result. If an ADO recordset is to contain data from multiple database tables, it should use a JOIN query, which assembles the data from the different database tables into a single result table.

In ADO.NET, you have various objects that allow you to access data in various ways. ADO.net Dataset can contain multiple tables from various data sources. The tables within a dataset are called data tables. The DataSet object will actually allow you to store the relational model of your database. If a dataset contains data from multiple database tables, it will typically contain multiple DataTable objects. That is, each DataTable object typically corresponds to a single database table or view.

ADO allows you to create client side cursors only, whereas ADO.NET gives you the choice of either using client side or server side cursors. In ADO.NET, classes actually handle the work of cursors. This allows a choice for programmers. In networking development environment , this choice is crucial in creating efficient applications.

In ADO.Net we can send multiple transactions using a single connection instance, whereas in ADO, we cannot send multiple transactions using a single connection instance.

In ADO, it is sometime create problems because firewall prohibits many types of request, while in ADO.net there is no such problem because XML is completely firewall-proof.

● Advantage of ADO.NET

ADO.NET is the common name for the classes and interfaces of the namespace System.Data. ADO.NET data components in Visual Studio environment encapsulate data access functionality in various ways that help you to develop applications more easily and with fewer errors.

ADO.Net offers performance advantages by its Disconnected Architecture, it is a remarkably efficient and scalable architecture. The DataSet class in ADO.Net operates in an entirely disconnected nature. This model allows for the DataSet class to be unaware of the origin of its data source, an unlimited number of supported data sources can be plugged into code without any hassle in the future

ADO.NET programs can take advantage of the flexibility and broad acceptance of Extensible Markup Language (XML). XML is the format for transmitting datasets across the network, so any component that can read the XML format can process data. In fact, many of the classes in ADO.NET, like the DataSet, are so intertwined with XML that they simply cannot exist or function without utilizing the technology.

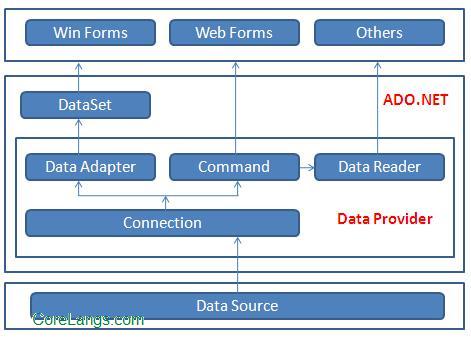
The role of ADO.NET data providers is to permit direct manipulation of data through SQL. ADO.NET includes a SQL Server Data Provider that is highly optimized for interaction with SQL Server. It uses SQL Server's own Tabular Data Stream (TDS) format for exchanging information. This is deftly handled by the Advantage ADO.NET Data Provider.

Another advantage of ADO.NET is its rich object model. The entire ADO.NET architecture is built on a hierarchy of class inheritance and interface implementation. Once you trace for things you need within this namespace, you can find that the logical inheritance of features and base class support makes the entire system extremely easy to use.

● ADO.NET Version History

● ADO.NET Architecture

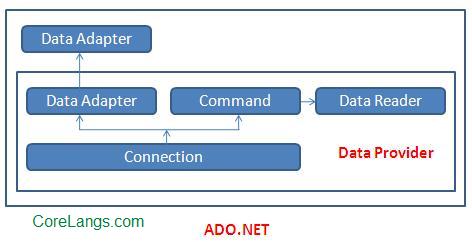
<http://asp.net-informations.com/ado.net/ado-architecture.htm>



ADO.NET consist of a set of Objects that expose data access services to the .NET environment. It is a data access technology from Microsoft .Net Framework , which provides communication between relational and non relational systems through a common set of components .

System.Data namespace is the core of ADO.NET and it contains classes used by all data providers. ADO.NET is designed to be easy to use, and Visual Studio provides several wizards and other features that you can use to generate ADO.NET data access code.

**Data Providers and DataSet**



The two key components of ADO.NET are Data Providers and DataSet . The Data Provider classes are meant to work with different kinds of data sources. They are used to perform all data-management operations on specific databases. DataSet class provides mechanisms for managing data when it is disconnected from the data source.

● Connected and Disconnected Architecture

The ADO.NET Framework supports two models of Data Access Architecture, Connection Oriented Data Access Architecture and Disconnected Data Access Architecture.

In Connection Oriented Data Access Architecture the application makes a connection to the Data Source and then interact with it through SQL requests using the same connection. In these cases the application stays connected to the database system even when it is not using any Database Operations.

ADO.Net solves this problem by introduces a new component called Dataset. The DataSet is the central component in the ADO.NET Disconnected Data Access Architecture. A DataSet is an in-memory data store that can hold multiple tables at the same time. DataSets only hold data and do not interact with a Data Source. One of the key characteristics of the DataSet is that it has no knowledge of the underlying Data Source that might have been used to populate it.

**DataSet ds = new DataSet();**

In Connection Oriented Data Access, when you read data from a database by using a DataReader object, an open connection must be maintained between your application and the Data Source. Unlike the DataReader, the DataSet is not connected directly to a Data Source through a Connection object when you populate it. It is the DataAdapter that manages connections between Data Source and Dataset by fill the data from Data Source to the Dataset and giving a disconnected behavior to the Dataset. The DataAdapter acts as a bridge between the Connected and Disconnected Objects.

**SqlDataAdapter adapter = new SqlDataAdapter("sql", "connection");**

**DataSet ds = new DataSet();**

**adapter.Fill(ds, "Src Table");**

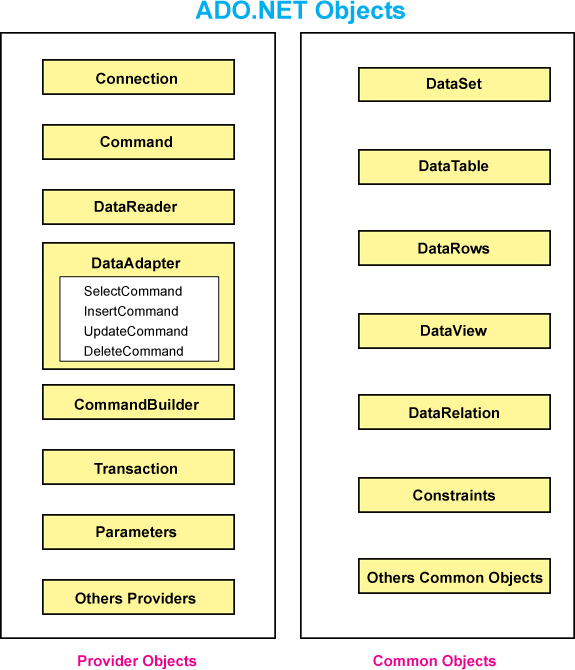
By keeping connections open for only a minimum period of time, ADO .NET conserves system resources and provides maximum security for databases and also has less impact on system performance.

● ADO.NET Objects

<http://www.c-sharpcorner.com/UploadFile/puranindia/ado-net-objects-part-i/>

The ADO.NET object model consists of two key components as follows:

* Connected model (.NET Data Provider - a set of components including the Connection, Command, DataReader, and DataAdapter objects)
* Disconnected model (DataSet).

****

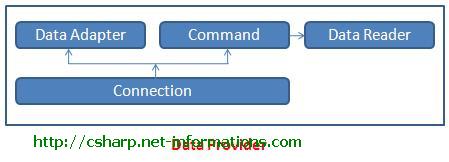
● .NET Data Providers

The [.Net Framework](http://vb.net-informations.com/framework/what_is_net_framework.htm) includes mainly three Data Providers for ADO.NET. They are the Microsoft **SQL Server** Data Provider ,**OLEDB** Data Provider and **ODBC** Data provider. You can see from the following links how these Data Providers making connection to the specified data Sources from your C# applications.

[C# SQL Server Connection](http://csharp.net-informations.com/data-providers/csharp-sql-server-connection.htm)

[C# OLEDB Connection](http://csharp.net-informations.com/data-providers/csharp-oledb-connection.htm)

[C# ODBC Connection](http://csharp.net-informations.com/data-providers/csharp-odbc-connection.htm)



The four Objects from the .Net Framework provide the functionality of Data Providers in ADO.NET. They are **Connection** Object,**Command** Object , **DataReader** Object and **DataAdapter** Object. The following link shows how to use these Objects in C# applications .

[C# Connection](http://csharp.net-informations.com/data-providers/csharp-ado.net-connection.htm)

[C# Command](http://csharp.net-informations.com/data-providers/csharp-ado.net-Command.htm)

[C# DataReader](http://csharp.net-informations.com/data-providers/csharp-datareader.htm)

[C# DataAdapter](http://csharp.net-informations.com/data-providers/csharp-dataadapter.htm)

Connection & Command

● Connection Classes

<https://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection(v=vs.110).aspx>

● Command Objects

● Creating Commands

● Executing Commands

● Parameterized Queries

● Command Types

Data Readers & Connected Access

● Understanding DataReader

<http://csharp-station.com/Tutorial/AdoDotNet/Lesson04>

● 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