Contents

[Basic Concept 4](#_Toc172141815)

[Dictionary Vs Hashtable 6](#_Toc172141816)

[C# version history 6](#_Toc172141817)

[Property 8](#_Toc172141818)

[C# Example with auto-initialize property 8](#_Toc172141819)

[Feature of C# 10](#_Toc172141820)

[Interoperability: *We can install application in any platform* 10](#_Toc172141821)

[Scalable and Updateable 10](#_Toc172141822)

[Component Oriented 10](#_Toc172141823)

[Structured Programming Language 11](#_Toc172141824)

[Rich Library 11](#_Toc172141825)

[Fast Speed 11](#_Toc172141826)

[DotNet Framework Class Library 12](#_Toc172141827)

[Garbage Collection in C# | .NET Framework 15](#_Toc172141828)

[**'using' Statement** 16](#_Toc172141829)

[**Reference Type** 20](#_Toc172141830)

[C# Implicitly-Typed Local Variable - var 20](#_Toc172141831)

[Array 20](#_Toc172141832)

[C# Single Dimensional Array 20](#_Toc172141833)

[C# Multidimensional Arrays 21](#_Toc172141834)

[Declaration of Jagged array 21](#_Toc172141835)

[C# Params 22](#_Toc172141836)

[C# Destructor 22](#_Toc172141837)

[writing single byte into file 23](#_Toc172141838)

[reading all bytes from file 23](#_Toc172141839)

[C# StreamReader 23](#_Toc172141840)

[C# TextWriter Example 24](#_Toc172141841)

[C# BinaryWriter Example 24](#_Toc172141842)

[C# BinaryReader Example 24](#_Toc172141843)

[C# FileInfo Properties 24](#_Toc172141844)

[C# FileInfo Methods 25](#_Toc172141845)

[C# FileInfo Example 26](#_Toc172141846)

[C# FileInfo Example: writing to the file 26](#_Toc172141847)

[C# FileInfo Example: Reading text from the file 26](#_Toc172141848)

[C# DirectoryInfo Example: 27](#_Toc172141849)

[C# DirectoryInfo Example: Deleting Directory 27](#_Toc172141850)

[C# Binary Serialization 27](#_Toc172141851)

[***Generic Class*** 28](#_Toc172141852)

[**Boxing and Unboxing** 31](#_Toc172141853)

[**Queues** 32](#_Toc172141854)

[Loose Coupling 33](#_Toc172141855)

[Tight Coupling 33](#_Toc172141856)

[**C# Exception Classes** 35](#_Toc172141857)

[C# Exception Filters 35](#_Toc172141858)

[C# Exception Filter Syntax 35](#_Toc172141859)

[C# User-Defined Exceptions 36](#_Toc172141860)

[**Constructor Injection** 39](#_Toc172141861)

[**Types of Collections in C#** 42](#_Toc172141862)

[**C# Anonymous Methods** 42](#_Toc172141863)

[**Singleton class vs. Static methods** 66](#_Toc172141864)

[**Advantages of Singleton Design Pattern** 66](#_Toc172141865)

[Difference between structs and classes 75](#_Toc172141866)

[**What is Serialization?** 79](#_Toc172141867)

[**What is Deserialization?** 79](#_Toc172141868)

[**Difference between Indexers and Properties** 82](#_Toc172141869)

[**Array of Delegates** 86](#_Toc172141870)

[**Anonymous Methods** 87](#_Toc172141871)

[**Events** 88](#_Toc172141872)

[C# Threading Example: Join() method 92](#_Toc172141873)

[**Create and Access Tuples** 99](#_Toc172141874)

[**Tuples in methods** 99](#_Toc172141875)

[**Return Tuples** 100](#_Toc172141876)

[**Important Methods of Static File Class** 101](#_Toc172141877)

[**Modifier Keywords** 103](#_Toc172141878)

[**Access Modifier Keywords:** 103](#_Toc172141879)

[**Const** 104](#_Toc172141880)

[**Readonly** 105](#_Toc172141881)

[Concurrent Dictionary 109](#_Toc172141882)

[**Installing the Windows Service** 114](#_Toc172141883)

[**Uninstalling a Windows Service** 114](#_Toc172141884)

[SOAP vs. REST 117](#_Toc172141885)

[**ASP.NET Web API** 117](#_Toc172141886)

[**ASP.NET Web API Characteristics** 118](#_Toc172141887)

[How to read data from XML String and insert in to table in SQL Server 120](#_Toc172141888)

[**Types of DML Triggers** 122](#_Toc172141889)

[204 No Content 125](#_Toc172141890)

[**C# Extension Method** 127](#_Toc172141891)

[**Benefits of extension methods** 129](#_Toc172141892)

[1. About SOLID 131](#_Toc172141893)

[ValidateInput Attribute to Prevent CSS Attack in MVC 142](#_Toc172141894)

[Types of Filters 147](#_Toc172141895)

[Examples of ASP.NET MVC Filters 147](#_Toc172141896)

[Authentication Filters 147](#_Toc172141897)

[Authorization Filters 147](#_Toc172141898)

[Action Filters 148](#_Toc172141899)

[Result Filters 149](#_Toc172141900)

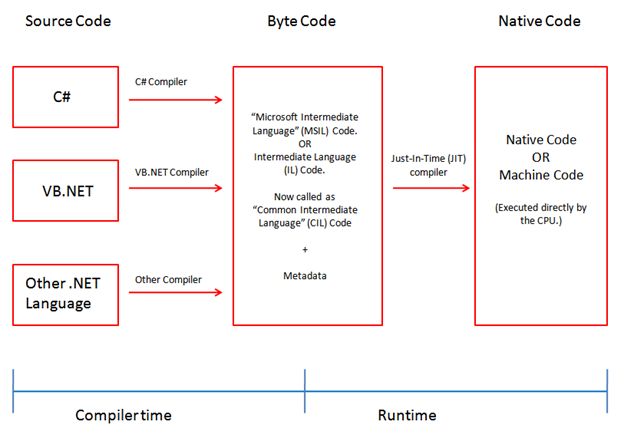
[Exception Filters 149](#_Toc172141901)

[**Isolation Levels in SQL Server** 164](#_Toc172141902)

# Basic Concept

How to convert source code to native code

* Source code convert into byte format using Intermediate Language
* With the help of JIT byte code convert into native code



Difference between CLR VS JIT

* The CLR converts the programming language code into an intermediate language
* the JIT takes that intermediate language as input and converts it to the machine language.

Primitive DataType

* Int , Double , Float , Bool ,Char

Nonprimitive DataType

* Class , Array , Structure , interface , Delegate

Difference between Java Vs C#

|  |  |  |
| --- | --- | --- |
| **No.** | **Java** | **C#** |
| 1) | Java is a **high level, robust, secured and object-oriented programming** language developed by Oracle. | C# is an **object-oriented programming** language developed by Microsoft that runs on .Net Framework. |
| 2) | Java programming language is designed to be run on a Java platform, by the help of **Java Runtime Environment (JRE).** | C# programming language is designed to be run on the **Common Language Runtime (CLR).** |
| 3) | Java type safety is safe. | C# type safety is unsafe. |
| 4) | In java, built-in data types that are passed by value are called **primitive types.** | In C#, built-in data types that are passed by value are called **simple types.** |
| 5) | Arrays in Java are direct specialization of **Object.** | Arrays in C# are specialization of **System.** |
| 6) | Java does not support **conditional compilation.** | C# supports conditional compilation using preprocessor directives. |
| 7) | Java doesn't support goto statement. | C# supports goto statement. |
| 8) | Java doesn't support **structures and unions.** | C# supports structures and unions. |
| 9) | Java supports checked exception and unchecked exception. | C# supports unchecked exception. |

# Dictionary Vs Hashtable

**Dictionary**

1. Dictionary is generic type Dictionary<TKey,TValue>
2. Dictionary class is a strong type < TKey,TValue > Hence, you must specify the data types for key and value.
3. There is no need of boxing/unboxing.
4. When you try to access non existing key dictionary, it gives runtime error.
5. Dictionary maintains an order of the stored values.
6. There is no need of boxing/unboxing, so it is faster than Hashtable.

**Hashtable**

1. Hashtable is non-generic type.
2. Hashtable is a weakly typed data structure, so you can add keys and values of any object type.
3. Values need to have boxing/unboxing.
4. When you try to access non existing key Hashtable, it gives null values.
5. Hashtable never maintains an order of the stored values.
6. Hashtable needs boxing/unboxing, so it is slower than Dictionary.

# C# version history

| Version | .NET Framework | Visual Studio | Important Features |
| --- | --- | --- | --- |
| C# 1.0 | .NET Framework 1.0/1.1 | Visual Studio .NET 2002 | * Basic features |
| C# 2.0 | .NET Framework 2.0 | Visual Studio 2005 | * Generics * Partial types * Anonymous methods * Iterators * Nullable types * Private setters (properties) * Method group conversions (delegates) * Covariance and Contra-variance * Static classes |
| C# 3.0 | .NET Framework 3.0\3.5 | Visual Studio 2008 | * Implicitly typed local variables * Object and collection initializers * Auto-Implemented properties * Anonymous types * Extension methods * Query expressions * Lambda expressions * Expression trees * Partial Methods |
| C# 4.0 | .NET Framework 4.0 | Visual Studio 2010 | * Dynamic binding (late binding) * Named and optional arguments * Generic co- and contravariance * Embedded interop types |
| C# 5.0 | .NET Framework 4.5 | Visual Studio 2012/2013 | * Async features * Caller information |
| C# 6.0 | .NET Framework 4.6 | Visual Studio 2013/2015 | * Expression Bodied Methods * Auto-property initializer * nameof Expression * Primary constructor * Await in catch block * Exception Filter * String Interpolation |
| C# 7.0 | .NET Core 2.0 | Visual Studio 2017 | * out variables * Tuples * Discards * Pattern Matching * Local functions * Generalized async return types * throw Expressions |
| C# 8.0 | .NET Core 3.0 | Visual Studio 2019 | Readonly members  Default interface methods  Using declarations  Static local functions  Disposable ref structs  Nullable reference types |
| C# 9.0 | .NET 5.0 | Visual Studio 2019 | Records  Init-only properties  Top-level statements  Init accessors and readonly fields  With-expressions  Value-based equality |
| C# 10.0 | .NET 6.0 | Visual Studio 2022 | Record structs  Global using directives  File-scoped namespace declaration  Extended Proptery Patterns  Null Parameter Checking  Constant interpolated strings |

# Property

**String interpolation in C#**

// Simple String Interpolation

**string** author = "Mahesh Chand";

**string** book = "Programming C#";

**int** year = 2018;

**decimal** price = 45.95m;

**string** hello = $"{author} is an author of {book} . \n" +

$"The book price is ${price} and was published in year {year}. ";

Console.WriteLine(hello);

### C# Example with auto-initialize property

**namespace** CSharpFeatures

{

**public** **class** PropertyInitializer

    {

**public** **string** Name { **get**; **set**; } = "Rahul Kumar";

**public** **static** **void** Main(**string**[] args)

        {

            PropertyInitializer pin = **new** PropertyInitializer();

            Console.WriteLine(pin.Name);

        }

    }

}

**namespace** CSharpFeatures

{

**Public class** Student

    {

        // Auto-property initializer

**public** **string** Name { **get**; **private** **set**; } = "Rahul Kumar";

    }

**public** **class** PropertyInitializer

    {

**public** **static** **void** Main(**string**[] args)

        {

            Student student = **new** Student();

            Console.WriteLine(student.Name);

        }

    }

}

public class PropertyInitialization

{

public virtual string First { get; set; } = "Adam";

}

public class ZopertyInitalization : PropertyInitialization

{

public override string First

{

get { return base.First; }

set

{

Console.WriteLine($"Child property hit with the value: '{0}'");

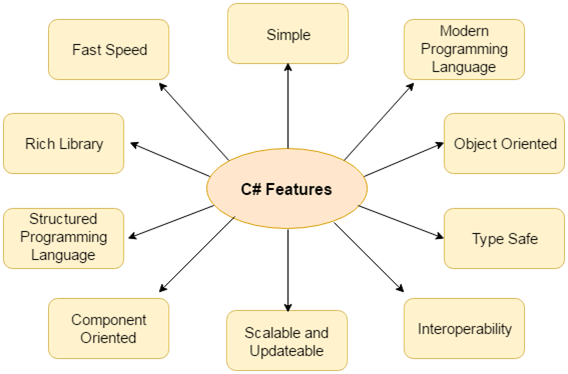
base.First = value;

}

}

}

# Feature of C#



C#:- It is known as object oriented programing language.OPPS mainly works on class and object

OPPS: [CLASS, OBJECT, INHERITANCE, POLYMORPHISM, ABSTRACTION, ENCAPSILATION]

**Type Safe:** - *Value type initialized to zero and reference type initialized to null by complier automatically*

C# type safe code can only access the memory location that it has permission to execute. Therefore it improves a security of the program.

### Interoperability: *We can install application in any platform*

Interoperability process enables the C# programs to do almost anything that a native C++ application can do.

### Scalable and Updateable

C# is automatic scalable and updateable programming language. For updating our application we delete the old files and update them with new ones.

### Component Oriented

C# is component oriented programming language. It is the predominant software development methodology used to develop more robust and highly scalable applications.

### Structured Programming Language

C# is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

### Rich Library

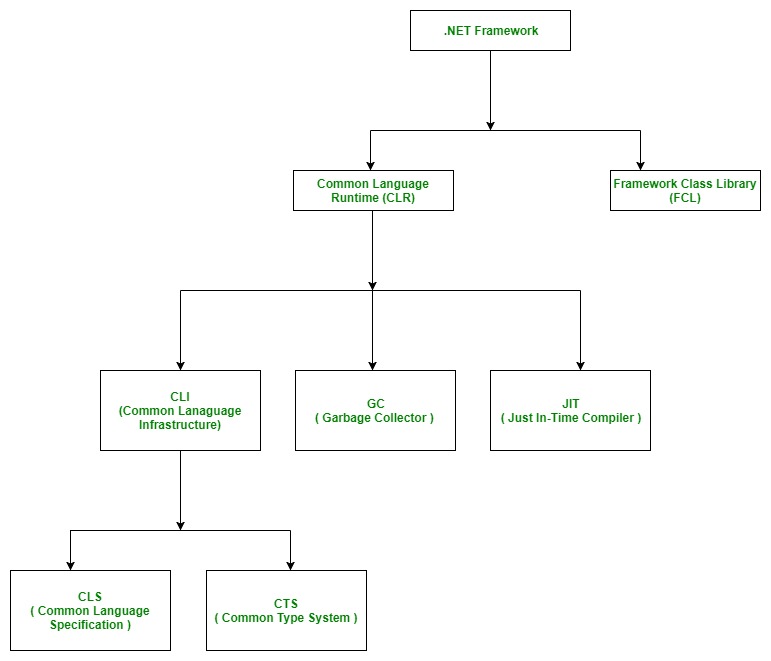
C# provides a lot of inbuilt functions that makes the development fast.

### Fast Speed

The compilation and execution time of C# language is fast.

**Programming Languages which are designed and developed by Microsoft are:**

* C#.NET
* VB.NET
* C++.NET
* J#.NET
* F#.NET
* JSCRIPT.NET
* WINDOWS POWERSHELL



# DotNet Framework Class Library

The Framework Class Library or FCL provides the system functionality in the [**.NET Framework**](https://www.geeksforgeeks.org/introduction-to-net-framework/) as it has various classes, data types, interfaces, etc. to perform multiple functions and build different types of applications such as desktop applications, web applications, mobile applications, etc. **The Framework Class Library is integrated with the**[**Common Language Runtime (CLR)**](https://www.geeksforgeeks.org/common-language-runtime-clr-in-c-sharp/)**of the .NET framework** and is used by all the .NET languages such as [**C#**](https://www.geeksforgeeks.org/csharp-programming-language/), F#, Visual Basic .NET, etc.

| **NAMESPACE** | **DESCRIPTION** |
| --- | --- |
| Accessibility | The Accessibility namespace is a part of the managed wrapper for the COM accessibility interface. |
| Microsoft.Activities | The Microsoft.Activities namespace provides support for Windows Workflow Foundation applications. |
| Microsoft.CSharp | The Microsoft.CSharp namespace has support for compilation and code generation for the C# source code. |
| Microsoft.JScript | The Microsoft.JScript namespace has support for compilation and code generation for the JScript source code. |
| Microsoft.VisualBasic | The Microsoft.VisualBasic namespace has support for compilation and code generation for the VisualBasic source code. |
| System | The System namespace has base classes for definition of interfaces, data types, events, event handlers, attributes, processing exceptions etc. |
| System.Activities | The System.Activities namespace handles the creation and working with activities in the Window Workflow Foundation using various classes. |
| System.Collections | The System.Collections namespace has multiple standard, specialized, and generic collection objects that are defined using various types. |
| System.Configuration | The System.Configuration namespace handles configuration data using various types. This may include data in machine or application configuration files. |
| System.Data | The System.Data namespace accesses and manages data from various sources using different classes. |
| System.Drawing | The System.Drawing namespace handles GDI+ basic graphics functionality. Various child namespaces also handle vector graphics functionality, advanced imaging functionality, etc. |
| System.Globalization | The System.Globalization namespace handles language, country, calendars used, format patterns for dates, etc. using various classes. |
| System.IO | The System.IO namespaces support IO like data read/write into streams, data compression, communicate using named pipes etc. using various types. |
| System.Linq | The System.Linq namespace supports Language-Integrated Query (LINQ) using various types. |
| System.Media | The System.Media namespace handles sound files and accessing the sounds provided by the system using various classes. |
| System.Net | The System.Net namespace provides an interface for network protocols, cache policies for web resources, composing and sending e-mail etc. using various classes. |
| System.Reflection | The System.Reflection namespace gives a managed view of loaded methods, types, fields, etc. It can also create and invoke types dynamically. |
| System.Security | The System.Security namespace has the .NET security system and permissions. Child namespaces provide authentication, crytographic services etc. |
| System.Threading | The System.Threading namespace allows multithreaded programming using various types. |
| XamlGeneratedNamespace | The XamlGeneratedNamespace has compiler-generated types that are not used directly from the code. |

# Garbage Collection in C# | .NET Framework

Automatic memory management is made possible by **Garbage Collection in .NET Framework**. When a class object is created at runtime, certain memory space is allocated to it in the heap memory. However, after all the actions related to the object are completed in the program, the memory space allocated to it is a waste as it cannot be used. In this case, garbage collection is very useful as it automatically releases the memory space after it is no longer required.

Garbage collection will always work on **Managed Heap** and internally it has an Engine which is known as the **Optimization Engine**.

**Benefits of Garbage Collection**

* Garbage Collection succeeds in allocating objects efficiently on the heap memory using the generations of garbage collection.
* Manual freeing of memory is not needed as garbage collection automatically releases the memory space after it is no longer required.
* Garbage collection handles memory allocation safely so that no objects use the contents of another object mistakenly.

class MyClass : IDisposable

{

public MyClass() //default ctor

{

this.iNumber = 0;

System.Console.WriteLine("ctor:MyClass {0}", iNumber);

}

public MyClass(Int32 iNumber) // specialized ctor

{

this.iNumber = iNumber;

System.Console.WriteLine("ctor:MyClass {0}", iNumber);

}

~MyClass() // dtor or finalize

{

System.Console.WriteLine("dtor:~MyClass {0}", iNumber);

}

public void Dispose() // helper finalize function

{

// here you can free the resources you allocated explicitly

System.GC.SuppressFinalize(this);

}

private int iNumber;

}

class main

{

static void Main()

{

MyClass myClass1 = new MyClass();

MyClass myClass2 = new MyClass(19);

myClass1.Dispose(); // myClass1 is explicitly exposed.

System.GC.Collect();

System.GC.WaitForPendingFinalizers();// myClass2 is implicitly exposed by GC.

Console.ReadLine();

}

}

**.NET supports two kind of coding**

1. Managed Code
2. Unmanaged Code

* ***The code, which is developed in .NET framework, is known as managed code.***
* ***This code is directly executed by CLR with help of managed code execution.***
* ***The code, which is developed outside .NET, Framework is known as unmanaged code.***
* ***Applications that do not run under the control of the CLR are said to be unmanaged, and certain languages such as C++ can be used to write such applications,***

## **'using' Statement**

using statement ensures object dispose, in short, it gives a comfort way of use of IDisposable objects. When an Object goes out of scope, Dispose method will get called automatically, basically using block does the same thing as 'TRY...FINALLY' block. To demonstrate it, create a class with IDisposable implementation (it should have Dispose() method), 'using' statement calls 'dispose' method even if exception occurs.

class testClass : IDisposable

{

public void Dispose()

{

*// Dispose objects here*

*// clean resources*

Console.WriteLine(0);

}

}

*//call class*

class Program

{

static void Main()

{

*// Use using statement with class that implements Dispose.*

using (testClass objClass = new testClass())

{

Console.WriteLine(1);

}

Console.WriteLine(2);

}

}

*//output*

1

0

2

*//it is same as below TRY...Finally code*

{

clsDispose\_Fin objClass = new clsDispose\_Fin();

try

{

*//code goes here*

}

finally

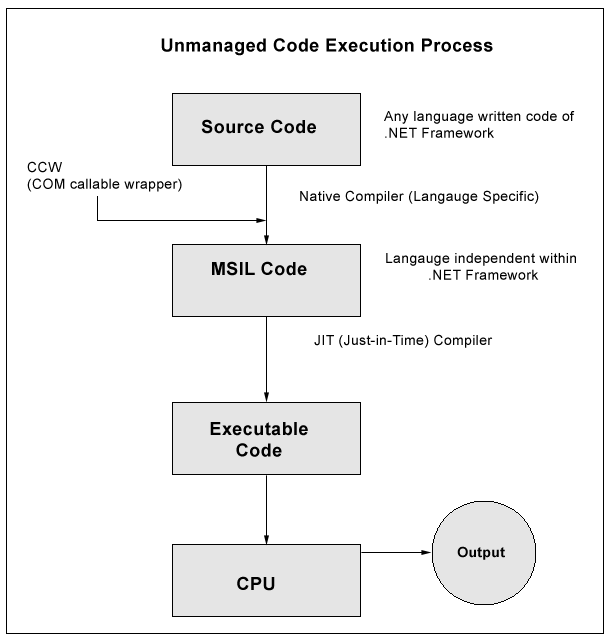
{

if (objClass != null)

((IDisposable)objClass).Dispose();

}

}



C# Data Type:

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

* Value types
* Reference types
* Pointer types

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Represents** | **Range** | **Default Value** |
| Dbool | Boolean value | True or False | False |
| byte | 8-bit unsigned integer | 0 to 255 | 0 |
| char | 16-bit Unicode character | U +0000 to U +ffff | '\0' |
| decimal | 128-bit precise decimal values with 28-29 significant digits | (-7.9 x 1028 to 7.9 x 1028) / 100to 28 | 0.0M |
| double | 64-bit double-precision floating point type | (+/-)5.0 x 10-324 to (+/-)1.7 x 10308 | 0.0D |
| float | 32-bit single-precision floating point type | -3.4 x 1038 to + 3.4 x 1038 | 0.0F |
| int | 32-bit signed integer type | -2,147,483,648 to 2,147,483,647 | 0 |

When a value type is converted to object type, it is called **boxing**

when an object type is converted to a value type, it is called **unboxing**.

## **Reference Type**

**Built-in** reference types are: **object**, **dynamic,** and **string**.

# C# Implicitly-Typed Local Variable - var

C# 3.0 introduced the implicit typed local variable "var". Var can only be defined in a method as a local variable.

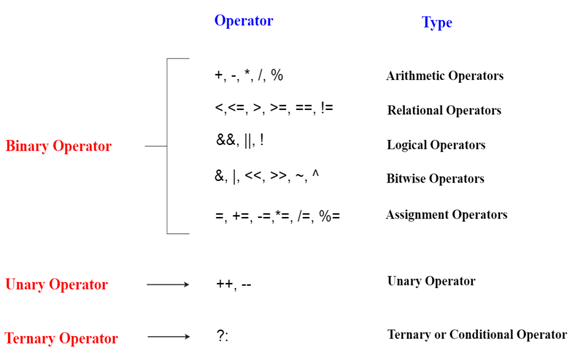
Example: Explicitly Typed Variable

int i = 100;// explicitly typed

var j = 100; // implicitly typed

var j; // Compile-time error: Implicitly-typed variables must be initialized

* **Implicit type conversion** − These conversions are performed by C# in a type-safe manner. For example, are conversions from smaller to larger integral types and conversions from derived classes to base classes.
* **Explicit type conversion** − These conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.



* **Implicit Casting** (automatically) - converting a smaller type to a larger type size  
  char -> int -> long -> float -> double
* **Explicit Casting** (manually) - converting a larger type to a smaller size type  
  double -> float -> long -> int -> char

### Array

### C# Single Dimensional Array

**int**[] arr = **new** **int**[5];//creating array

# C# Multidimensional Arrays

**int**[,] arr=**new** **int**[3,3];//declaration of 2D array

arr[0,1]=10;//initialization

arr[1,2]=20;

arr[2,0]=30;

0 10 0

0 0 20

30 0 0

**int**[,,] arr=**new** **int**[3,3,3];//declaration of 3D array

### Declaration of Jagged array

**int**[][] arr = **new** **int**[2][];

arr[0] = **new** **int**[4];

arr[1] = **new** **int**[6];

arr[0] = **new** **int**[4] { 11, 21, 56, 78 };

arr[1] = **new** **int**[6] { 42, 61, 37, 41, 59, 63 };

Jagged Array

Example 1 :

int[][] intJaggedArray = new int[2][];

intJaggedArray[0] = new int[3]{1, 2, 3};

intJaggedArray[1] = new int[2]{4, 5 };

Console.WriteLine(intJaggedArray[0][0]); // 1

Console.WriteLine(intJaggedArray[0][2]); // 3

Console.WriteLine(intJaggedArray[1][1]); // 5

Example 2 :

int[][,] intJaggedArray = new int[3][,];

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

intJaggedArray[1] = new int[2, 2] { { 3, 4 }, { 5, 6 } };

intJaggedArray[2] = new int[2, 2];

Console.WriteLine(intJaggedArray[0][1,1]); // 4

Console.WriteLine(intJaggedArray[1][1,0]); // 5

Console.WriteLine(intJaggedArray[1][1,1]); // 6

# C# Params

In C#, **params** is a keyword which is used to specify a parameter that takes variable number of arguments. It is useful when we don't know the number of arguments prior. Only one params keyword is allowed and no additional parameter is permitted after params keyword in a function declaration.

**class** Program

    {

        // User defined function

**public** **void** Show(**params** **int**[] val) // Params Paramater

        {

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

            {

                Console.WriteLine(val[i]);

            }

        }

        // Main function, execution entry point of the program

**static** **void** Main(**string**[] args)

        {

            Program program = **new** Program(); // Creating Object

            program.Show(2,4,6,8,10,12,14); // Passing arguments of variable length

        }

    }

# C# Destructor

A destructor works opposite to constructor, It destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

Note: C# destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.

C# FileStream

using System.IO;

### writing single byte into file

FileStream f = **new** FileStream("e:\\b.txt", FileMode.OpenOrCreate);//creating file stream

f.WriteByte(65);//writing byte into stream

f.Close();//closing stream

### reading all bytes from file

FileStream f = **new** FileStream("e:\\b.txt", FileMode.OpenOrCreate);

**int** i = 0;

**while** ((i = f.ReadByte()) != -1)

{

         Console.Write((**char**)i);

}

f.Close();

C# StreamWriter

FileStream f = new FileStream("e:\\output.txt", FileMode.Create);

StreamWriter s = new StreamWriter(f);

s.WriteLine("hello c#");

s.Close();

f.Close();

# C# StreamReader

FileStream f = new FileStream("e:\\output.txt", FileMode.OpenOrCreate);

StreamReader s = new StreamReader(f);

string line=s.ReadLine();

Console.WriteLine(line);

s.Close();

f.Close();

### C# TextWriter Example

using (TextWriter writer = File.CreateText("e:\\f.txt"))

{

writer.WriteLine("Hello C#");

writer.WriteLine("C# File Handling by JavaTpoint");

}

using (TextReader tr = File.OpenText("e:\\f.txt"))

{

Console.WriteLine(tr.ReadToEnd());

}

### C# BinaryWriter Example

using (BinaryWriter writer = new BinaryWriter(File.Open(fileName, FileMode.Create)))

{

writer.Write(2.5);

writer.Write("this is string data");

writer.Write(true);

}

### C# BinaryReader Example

using (BinaryReader reader = new BinaryReader(File.Open("e:\\binaryfile.dat", FileMode.Open)))

{

Console.WriteLine("Double Value : " + reader.ReadDouble());

Console.WriteLine("String Value : " + reader.ReadString());

Console.WriteLine("Boolean Value : " + reader.ReadBoolean());

}

### C# FileInfo Properties

|  |  |
| --- | --- |
| **Properties** | **Description** |
| Attributes | It is used to get or set the attributes for the current file or directory. |
| CreationTime | It is used to get or set the creation time of the current file or directory. |
| Directory | It is used to get an instance of the parent directory. |
| DirectoryName | It is used to get a string representing the directory's full path. |
| Exists | It is used to get a value indicating whether a file exists. |
| FullName | It is used to get the full path of the directory or file. |
| Length | It is used to get the size in bytes of the current file. |
| Name | It is used to get the name of the file. |

### C# FileInfo Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| AppendText() | It is used to create a StreamWriter that appends text to the file represented by this instance of the FileInfo. |
| CopyTo(String) | It is used to copy an existing file to a new file. |
| Create() | It is used to create a file. |
| CreateText() | It is used to create a StreamWriter that writes a new text file. |
| Decrypt() | It is used to decrypt a file that was encrypted by the current account using the Encrypt method. |
| Delete() | It is used to permanently delete a file. |
| Encrypt() | It is used to encrypt a file so that only the account used to encrypt the file can decrypt it. |
| MoveTo(String) | It is used to move a specified file to a new specified location. |
| Open(FileMode) | It is used to open a file in the specified mode. |
| OpenRead() | It is used to create a read-only FileStream. |
| OpenText() | It is used to create a StreamReader with UTF8 encoding that reads from an existing text file. |
| OpenWrite() | It is used to create a write-only FileStream. |
| Refresh() | It is used to refresh the state of the object. |
| Replace(String,String) | It is used to replace the contents of a specified file with the file described by the current FileInfo object. |
| ToString() | It is used to return the path as a string |

### C# FileInfo Example

// Specifying file location

string loc = "F:\\abc.txt";

// Creating FileInfo instance

FileInfo file = new FileInfo(loc);

// Creating an empty file

file.Create();

### C# FileInfo Example: writing to the file

// Specifying file location

string loc = "F:\\abc.txt";

// Creating FileInfo instance

FileInfo file = new FileInfo(loc);

// Creating an file instance to write

StreamWriter sw = file.CreateText();

// Writing to the file

sw.WriteLine("This text is written to the file by using StreamWriter class.");

sw.Close();

### C# FileInfo Example: Reading text from the file

// Specifying file to read

string loc = "F:\\abc.txt";

// Creating FileInfo instance

FileInfo file = new FileInfo(loc);

// Opening file to read

StreamReader sr = file.OpenText();

string data = "";

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

{

Console.WriteLine(data);

}

### C# DirectoryInfo Example:

// Provide directory name with complete location.

DirectoryInfo directory = new DirectoryInfo(@"F:\javatpoint");

try

{

// Check, directory exist or not.

if (directory.Exists)

{

Console.WriteLine("Directory already exist.");

return;

}

// Creating a new directory.

directory.Create();

Console.WriteLine("The directory is created successfully.");

}

### C# DirectoryInfo Example: Deleting Directory

DirectoryInfo directory = new DirectoryInfo(@"F:\javatpoint");

try

{

// Deleting directory

directory.Delete();

Console.WriteLine("The directory is deleted successfully.");

}

## C# Binary Serialization

using System.Runtime.Serialization.Formatters.Binary

FileStream stream = new FileStream("e:\\sss.txt", FileMode.OpenOrCreate);

BinaryFormatter formatter=new BinaryFormatter();

Student s = new Student(101, "sonoo");

formatter.Serialize(stream, s);

stream.Close();

**C# Deserialization**

FileStream stream = new FileStream("e:\\sss.txt", FileMode.OpenOrCreate);

BinaryFormatter formatter=new BinaryFormatter();

Student s=(Student)formatter.Deserialize(stream);

Console.WriteLine("Rollno: " + s.rollno);

Console.WriteLine("Name: " + s.name);

stream.Close();

## ***Generic Class***

Example 1:

1. **using** System.Collections.Generic;
2. **namespace** Generic
3. {
4. **public** **class** ArrayClass<T>
5. {
6. **object** Data;
7. **public** **void** Push(**object** Obj)
8. {
9. Data= Obj;
10. }
11. **public** **object** Pop()
12. {
13. **return** Data;
14. }
15. }
16. }
17. **public** **class** SubClass
18. {
19. **public** **static** **void** MyClass()
20. {
21. ArrayClass<**int**> intArray = **new** ArrayClass<**int**>();
22. intArray.Push(10);
23. **object** intReturnValue = intArray.Pop();
24. Console.Write("   {0}", intReturnValue);
25. ArrayClass<**string**> strArray = **new** ArrayClass<**string**>();
26. strArray.Push("Hello word");
27. **object** strReturnValue = strArray.Pop();
28. Console.Write("   {0}", strReturnValue);
29. Console.ReadLine();
30. }
31. }

Advantages of Generics:

Generics provide type safety without the overhead of multiple implementations. Generics eliminates boxing and unboxing.

Generic Method

1. **using** System.Collections.Generic;
3. **public** **class** ArrayList
4. {
5. **static** **void** Swap<T>(**ref** T lhs, **ref** T rhs)
6. {
7. T temp;
8. temp = lhs;
9. lhs = rhs;
10. rhs = temp;
11. }
12. }

The following method shows how to call the method above.

1. **public** **static** **void** TestSwap()
2. {
3. **int** a = 1;
4. **int** b = 2;
5. Swap<**int**>(**ref** a, **ref** b);
6. System.Console.WriteLine(a + " " + b);
7. Console.ReadLine();
8. }

class CompareGenericClass<T>

{

public bool Compare(T x, T y)

{

if (x.Equals(y)) return true;

else return false;

}

}

class Program

{

static void Main(string[] args)

{

CompareGenericClass<int> oIntcompare = new CompareGenericClass<int>();

bool integerresult = oIntcompare.Compare(5, 6);

Console.WriteLine("Generic int comapre result:" + integerresult);

}

}

class Generic<T, U> where T : class where U : struct

{

public T GenericVariableFirst

{

get;

set;

}

public U GenericVariableSecond

{

get;

set;

}

}

class Program

{

static void Main(string[] args)

{

Generic<string, int> g = new Generic<string, int>();

g.GenericVariableFirst = "CodesDope";

g.GenericVariableSecond = 10;

Console.WriteLine(g.GenericVariableFirst);

Console.WriteLine(g.GenericVariableSecond);

}

}

## **Boxing and Unboxing**

1. **int** a = 20;
2. **object** b = a; //boxing
3. **int** c = (**int**)b; // unboxing

Generic Collection

Array List

1. ArrayList obj = **new** ArrayList();
2. obj.Add(50);
3. obj.Add("Dog");
4. obj.Add(**new** TestClass());
5. **foreach**(**int** i **in** obj)
6. {
7. Console.WriteLine(i);
8. }
9. ArrayList  obj = **new** ArrayList();
10. obj.Add(50);    //boxing- convert value type to reference type
11. **int** x= (**int**)obj[0]; //unboxing
12. **foreach**(**int** i **in** obj)
13. {
14. Console.WriteLine(i);   // unboxing
15. }

## **Queues**

|  |  |
| --- | --- |
| System.Collection.Queue Members | Definition |
| Enqueue() | Add an object to the end of the queue. |
| Dequeue() | Removes an object from the beginning of the queue. |
| Peek() | Return the object at the beginning of the queue without removing it. |

//Defines a Queue.

Queue qObj = new Queue();

//adding string values into collection

qObj.Enqueue("Tom");

qObj.Enqueue("Harry");

qObj.Enqueue("Maria");

qObj.Enqueue("john");

//displaying collections

while(qObj.Count !=0 )

{

Console.WriteLine(qObj.Dequeue());

}

**Stacks**

|  |  |
| --- | --- |
| System.Collection.Stack Members | Definition |
| Contains() | Returns true if a specific element is found in the collection. |
| Clear() | Removes all the elements of the collection. |
| Peek() | Previews the most recent element on the stack. |
| Push() | It pushes elements onto the stack. |
| Pop() | Return and remove the top elements of the stack. |

## Loose Coupling

Loose coupling is preferred since through it changing one class will not affect another class. It reduces dependencies on a class.

Writing loosely coupled code has the following advantages −

* One module won’t break other modules
* Enhances testability

## Tight Coupling

In Tight Coupling, the classes and objects are dependent on each other and therefore reduce re-usability of code.

**Tight Coupling** means one class is dependent on another class.  
**Loose Coupling** means one class is dependent on interface rather than class.

**Tight Coupling**

1. **public** **class** CustomerRepository {
2. **private** **readonly** Database database;
3. **public** CustomerRepository(Database database) {
4. **this**.database = database;
5. }
6. **public** **void** Add(**string** CustomerName) {
7. database.AddRow("Customer", CustomerName);
8. }
9. }
10. **public** **class** Database {
11. **public** **void** AddRow(**string** Table, **string** Value) {}
12. }

**Loose Coupling**

1. **public** **class** CustomerRepository {
2. **private** **readonly** IDatabase database;
3. **public** CustomerRepository(IDatabase database) {
4. **this**.database = database;
5. }
6. **public** **void** Add(**string** CustomerName) {
7. database.AddRow("Customer", CustomerName);
8. }
9. }
10. **public** **interface** IDatabase {
11. **void** AddRow(**string** Table, **string** Value);
12. }
13. **public** **class** SqlDatabase: IDatabase {
14. **public** **void** AddRow(**string** Table, **string** Value) {
15. //Logic to add new customer in sql table
16. }
17. }
18. **public** **class** XMLDatabase: IDatabase {
19. **public** **void** AddRow(**string** Table, **string** Value) {
20. //Logic to add new customer in XML Document
21. }
22. }

## 

## **C# Exception Classes**

System.DivideByZeroException

System.NullReferenceException

System.InvalidCastException

System.IO.IOException

System.FieldAccessException

# C# Exception Filters

It is introduced in version C# 6.0. It allows us to specify condition along with a catch block.

C# provides when keyword to apply a condition (or filter) along with catch block.

### C# Exception Filter Syntax

1. **protected** **void** Page\_Load(**object** sender, EventArgs e)
2. {
3. **try**
4. {
5. //Calling the page which is not exixt on this path
6. Response.Redirect("~/Admin/Login.aspx");
7. }
8. **catch** (HttpException hex) **if**(hex.GetHashCode==400)
9. {
10. lblMessage.Text="This Page is Not Found !";
11. }
12. **catch** (HttpException hex) **if**(hex.GetHashCode==500)
13. {
14. lblMessage.Text="Internal Server Error Occured !";
15. }
16. }

catch (ArgumentException e) when (e.ParamName == "?"){ }

try

{

int[] a = new int[5];

a[10] = 12;

}

catch(Exception e) when(e.GetType().ToString() == "System.IndexOutOfRangeException")

{

// Executing some other task

SomeOtherTask();

}

# C# User-Defined Exceptions

C# allows us to create user-defined or custom exception. It is used to make the meaningful exception. To do this, we need to inherit Exception class.

public class InvalidAgeException : Exception

{

public InvalidAgeException(String message)

: base(message)

{

}

}

try

{

// validate(12);

}

catch (InvalidAgeException e)

{

Console.WriteLine(e);

}

Nested Try Catch

static void Main(string[] args)

{

// Here, number is greater

// than divisor.

int[] number = { 8, 17, 24, 5, 25 };

int[] divisor = { 2, 0, 0, 5 };

// Outer try block

// Here IndexOutOfRangeException occurs

// due to which program may terminates

try

{

for (int j = 0; j < number.Length; j++)

{

// Inner try block

// Here DivideByZeroException caught

// and allow the program to continue

// its execution

try

{

Console.WriteLine("Number: " + number[j] +

"\nDivisor: " + divisor[j] +

"\nQuotient: " + number[j] / divisor[j]);

}

// Catch block for inner try block

catch (DivideByZeroException)

{

Console.WriteLine("Inner Try Catch Block");

}

}

}

// Catch block for outer try block

catch (IndexOutOfRangeException)

{

Console.WriteLine("Outer Try Catch Block");

}

}

Example :

public class ExceptionCls

{

public void Divide()

{

int x = 0;

int div = 0;

try

{

try

{

div = 100 / x;

}

catch(DivideByZeroException e)

{

Console.WriteLine(e.Message);

}

Console.WriteLine("Okay");

}

catch(Exception ex)

{

Console.WriteLine(ex.Message);

}

}

}

OutPut :

Exception divide by zero

Okay

**throw :**If we use "throw" statement, it preserve original error stack information. In exception handling "throw" with empty parameter is also called re-throwing the last exception.  
  
**throw ex :**If we use "throw ex" statement, stack trace of exception will be replaced with a stack trace starting at the re-throw point. It is used to intentionally hide stack trace information.

1. **catch** (Exception ex)
2. {
3. // do some stuff here
4. **throw**; // a) continue ex
5. **throw** **new** MyException("failed", ex); // b) wrap
6. **throw** **new** MyException("failed"); // c)  replace
7. **throw** ex; // d)  reset stack-trace
8. }

Dependency Injection

## **Constructor Injection**

public interface IService

{

void Serve();

}

public class Service1 : IService

{

public void Serve() { Console.WriteLine("Service1 Called"); }

}

public class Service2 : IService

{

public void Serve() { Console.WriteLine("Service2 Called"); }

}

public class Client

{

private IService \_service;

public Client(IService service)

{

this.\_service = service;

}

public void ServeMethod() { this.\_service.Serve(); }

}

class Program

{

static void Main(string[] args)

{

//creating object

Service1 s1 = new Service1();

//passing dependency

Client c1 = new Client(s1);

//TO DO:

Service2 s2 = new Service2();

//passing dependency

c1 = new Client(s2);

//TO DO:

}

}

Property Injection

public interface IService

{

void Serve();

}

public class Service1 : IService

{

public void Serve() { Console.WriteLine("Service1 Called"); }

}

public class Service2 : IService

{

public void Serve() { Console.WriteLine("Service2 Called"); }

}

public class Client

{

private IService \_service;

public IService Service

{

set { this.\_service = value; }

}

public void ServeMethod() { this.\_service.Serve(); }

}

class Program

{

static void Main(string[] args)

{

//creating object

Service1 s1 = new Service1();

Client client = new Client();

client.Service = s1; //passing dependency

//TO DO:

Service2 s2 = new Service2();

client.Service = s2; //passing dependency

//TO DO:

}

}

XML Document

How to Load and Read XML in Csharp

using System.Xml.Linq;

<object>

<name>Sphere</name>

<material>Steel</material>

<device Id="01">

<model>Model 1</model>

<color>Red</color>

</device>

<device Id="02">

<model>Model 2</model>

<color>Blue</color>

</device>

</object>

XDocument doc = XDocument.Load(@"C:\Users\ABC\Documents\Visual Studio 2012\ConsoleApplication2\ConsoleApplication2\XMLFile1.xml");

foreach (var itemElement in doc.Elements("object"))

{

Console.WriteLine(itemElement.Element("name").Value);

Console.WriteLine(itemElement.Element("material").Value);

foreach (var device in itemElement.Elements("device"))

{

Console.WriteLine(device.Element("model").Value);

Console.WriteLine(device.Element("color").Value);

}

}

How to save XML

1. //Create the XmlDocument.
2. XmlDocument doc = **new** XmlDocument();
3. doc.LoadXml(("<Student type='regular' Section='B'><Name>Tommy
4. ex</Name></Student>"));
5. //Save the document to a file.
6. doc.Save("C:\\std.xml");

## **Types of Collections in C#**

* **System.Collections.Generic** classes
* **System.Collections** classes (Now deprecated)

## **C# Anonymous Methods**

public delegate void AnonymousFun();

static void Main(string[] args)

{

AnonymousFun fun = delegate () {

Console.WriteLine("This is anonymous function");

};

fun();

}

***Method Overloading***

The **advantage** of method overloading is that it increases the readability of the program because you don't need to use different names for same action.

1. Method overloading is also called  early binding or compile time polymorphism or static binding.
2. The compiler automatically calls the required method or the function by checking number of parameters and their type, which are passed into that method.
3. If the number of parameters and type doesn't match by any method signatures, then it will give the compile time error.

Example 1 : below code execute properly

class Program

{

int k;

public int Method(int i,int j)

{

return k;

}

public int Method(double i, double j)

{

return k;

}

static void Main(string[] args)

{

Program P = new Program();

P.Method(1, 2);

P.Method(1.0, 2.2);

}

}

***Example 2*** : below code execute properly

class Program

{

int k;

public int Method(int i,int j)

{

return k;

}

public void Method(double i, double j)

{

}

static void Main(string[] args)

{

Program P = new Program();

P.Method(1, 2);

P.Method(1.0, 2.2);

}

}

Example 3 : Below code not execute properly

class Program

{

int k;

public int Method(int i,int j)

{

return k;

}

public void Method(int i, int j)

{

}

static void Main(string[] args)

{

Program P = new Program();

P.Method(1, 2);

P.Method(1, 2);

}

}

Example 4 : below code execute properly

class Program

{

int k;

public int Method(int i,int j)

{

return k;

}

public void Method(out int i,out int j)

{

i = 1;

j = 2;

}

static void Main(string[] args)

{

Program P = new Program();

P.Method(1, 2);

int i, j;

P.Method(out i, out j);

}

}

|  |  |
| --- | --- |
| Ref | Out |
| The parameter or argument must be initialized first before it is passed to ref. | It is not compulsory to initialize a parameter or argument before it is passed to an out. |
| It is not required to assign or initialize the value of a parameter (which is passed by ref) before returning to the calling method. | A called method is required to assign or initialize a value of a parameter (which is passed to an out) before returning to the calling method. |
| Passing a parameter value by Ref is useful when the called method is also needed to modify the pass parameter. | Declaring a parameter to an out method is useful when multiple values need to be returned from a function or method. |
| It is not compulsory to initialize a parameter value before using it in a calling method. | A parameter value must be initialized within the calling method before its use. |

***Interface:***

1. **How to declare Interface as private**
2. **How to declare variable in Interface**
3. **How to call sealed class**

public static class StaticClass

{

private interface IInstanceClassInternal

{

int OldValue { get; set; }

}

public sealed class InstanceClass : IInstanceClassInternal

{

int IInstanceClassInternal.OldValue { get; set; }

public int Value;

}

public static void Backup(InstanceClass ic)

{

((IInstanceClassInternal)ic).OldValue = ic.Value;

}

public static void Restore(InstanceClass ic)

{

ic.Value = ((IInstanceClassInternal)ic).OldValue;

}

}

**How to declare same method name in two interface**

interface G1

{

void mymethod();

}

interface G2

{

void mymethod();

}

class Geeks : G1, G2

{

void G1.mymethod()

{

Console.WriteLine("GeeksforGeeks");

}

void G2.mymethod()

{

Console.WriteLine("GeeksforGeeks");

}

}

static void Main(string[] args)

{

G1 obj = new Geeks();

obj.mymethod();

G2 ob = new Geeks();

ob.mymethod();

}

***Base Keyword***

***Example 1 :***

public class Person

{

protected string ssn = "444-55-6666";

protected string name = "John L. Malgraine";

public virtual void GetInfo()

{

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

Console.WriteLine("SSN: {0}", ssn);

}

}

class Employee : Person

{

public string id = "ABC567EFG";

public override void GetInfo()

{

// Calling the base class GetInfo method:

base.GetInfo();

Console.WriteLine("Employee ID: {0}", id);

}

}

class TestClass

{

static void Main()

{

Employee E = new Employee();

E.GetInfo();

}

}

/\*

Output

Name: John L. Malgraine

SSN: 444-55-6666

Employee ID: ABC567EFG

\*/

***Example 2:***

public class BaseClass

{

int num;

public BaseClass()

{

Console.WriteLine("in BaseClass()");

}

public BaseClass(int i)

{

num = i;

Console.WriteLine("in BaseClass(int i)");

}

public int GetNum()

{

return num;

}

}

public class DerivedClass : BaseClass

{

// This constructor will call BaseClass.BaseClass()

public DerivedClass() : base()

{

}

// This constructor will call BaseClass.BaseClass(int i)

public DerivedClass(int i) : base(i)

{

}

static void Main()

{

DerivedClass md = new DerivedClass();

DerivedClass md1 = new DerivedClass(1);

}

}

/\*

Output:

in BaseClass()

in BaseClass(int i)

\*/

***Example 3:***

public class BaseClass

{

int num;

public BaseClass()

{

Console.WriteLine("in BaseClass()");

}

public BaseClass(int i)

{

num = i;

Console.WriteLine("in BaseClass(int i)");

}

public int GetNum()

{

return num;

}

}

public class DerivedClass : BaseClass

{

// This constructor will call BaseClass.BaseClass()

public DerivedClass()

{

}

// This constructor will call BaseClass.BaseClass(int i)

public DerivedClass(int i)

{

}

static void Main()

{

DerivedClass md = new DerivedClass();

DerivedClass md1 = new DerivedClass(1);

}

}

/\*

Output:

in BaseClass()

in BaseClass()

\*/

public class ClsA

{

public void Add()

{

Console.WriteLine("Base Add");

}

}

public class ClsB : ClsA

{

public void Remove()

{

Console.WriteLine("Derive Remove");

}

}

static void Main(string[] args)

{

ClsB objb = new ClsB();

objb.Add();

objb.Remove();

}

Output

Base Add

Derive Remove

**How to call Private Constructor in c#**

public class Bob

{

public String Surname { get; set; }

private Bob()

{

Console.WriteLine("private Constructor");

}

public Bob(string surname)

{

Surname = surname;

}

}

static void Main(string[] args)

{

var constructor = typeof(Bob).GetConstructor(BindingFlags.NonPublic | BindingFlags.Instance, null, new Type[0], null);

var instance = (Bob)constructor.Invoke(null);

}

**How to Access Private Method**

public class AccessSpecifier

{

public void AddNumber()

{

Add();

}

private void Add()

{

int a, b;

a = b = 5;

int c = a + b;

Console.WriteLine(c.ToString());

}

}

static void Main(string[] args)

{

AccessSpecifier obj = new AccessSpecifier();

obj.AddNumber();

}

* **The internal access specifier hides its member variables and methods from other classes and objects, that is resides in other namespace.**
* **The variable or classes that are declared with internal can be access by any member within application.**
* **It is the default access specifiers for a class in C# programming**
* **enum:**The default and only access modifier supported is public.
* **class:**The default access for a class is internal. It may be explicitly defined using any of the access modifiers.
* **interface:**The default and only access modifier supported is public.
* **struct:**The default access is private with public and internal supported as well.

***Static Class***

static class Class\_Name

{

// static data members

// static method

}

Example

static class Author

{

public static string A\_name = "Ankita";

public static string L\_name = "CSharp";

public static int T\_no = 84;

public static void details()

{

Console.WriteLine("The details of Author is:");

}

}

public class GFG

{

static public void Main()

{

Author.details();

Console.WriteLine("Author name : {0} ", Author.A\_name);

Console.WriteLine("Language : {0} ", Author.L\_name);

Console.WriteLine("Total number of articles : {0} ",

Author.T\_no);

}

}

Output:

The details of Author is:

Author name : Ankita

Language : CSharp

Total number of articles : 84

Cannot Inherit Static class in c#

// declaring a static class

public static class GFG

{

static void display()

{

Console.WriteLine("Static Method of class GFG");

}

}

// trying to inherit the class GFG

// it will give error as static

// class can't be inherited

class GFG2 : GFG

{

public static void Main(String[] args)

{

}

}

**Difference between static and non-static class**

|  |  |
| --- | --- |
| **STATIC CLASS** | **NON-STATIC CLASS** |
| Static class is defined using static keyword. | Non-Static class is not defined by using static keyword. |
| In static class, you are not allowed to create objects. | In non-static class, you are allowed to create objects using new keyword. |
| The data members of static class can be directly accessed by its class name. | The data members of non-static class is not directly accessed by its class name. |
| Static class always contains static members. | Non-static class may contain both static and non-static methods. |
| Static class does not contain an instance constructor. | Non-static class contains an instance constructor. |
| Static class cannot inherit from another class. | Non-static class can be inherited from another class. |

***Sealed Class***

* ***It is possible to create sealed class object***

public sealed class SealClass

{

public void Print()

{

Console.WriteLine("This method created in sealed class");

}

}

static void Main(string[] args)

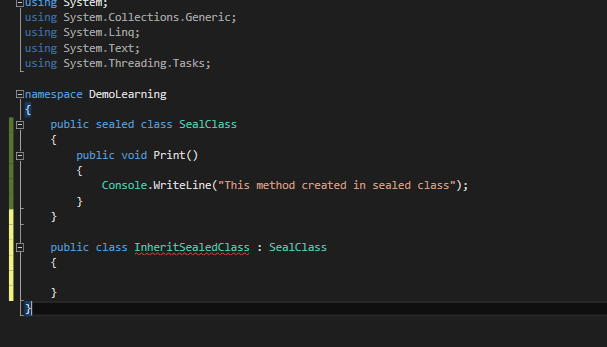
{

SealClass obj2 = new SealClass();

obj2.Print();

}

* This is not possible to inherit sealed class



***Abstract***

|  |  |
| --- | --- |
| **Abstract Class** | **Interface** |
| We cannot create an instance of the this class. | Interface can be only variable type and not instance. |
| It can have constructor. | It cannot have constructor. |
| It can be derived to some other class. | It is created to be derived by other class. |
| It can have implementation(non abstract) of one or more methods. | It cannot have function definition. |
| Concrete class can implement only one abstract class | Concrete class can implement many interfaces |
| It can or cannot contain the abstract methods | It should only have the method signatures. |
| It can have private, protected, internal data members. | All the members are public by default. |
| It cannot be derived to a structure. | It can be derived by a structure. |

abstract class A

{

protected A()

{ Console.WriteLine("Abstract class constructor"); }

}

//Derived class

class B : A

{

public B() { Console.WriteLine("Derived class constructor"); }

}

class Program

{

static void Main(string[] args)

{

B obj = new B();

}

}

Output :

Abstract class constructor

Derived class constructor

Example : 2

// abstract class 'GeeksForGeeks'

public abstract class GeeksForGeeks

{

// abstract method 'gfg()'

public abstract void gfg();

}

// class 'GeeksForGeeks' inherit

// in child class 'Geek1'

public class Geek1 : GeeksForGeeks

{

// abstract method 'gfg()'

// declare here with

// 'override' keyword

public override void gfg()

{

Console.WriteLine("class Geek1");

}

}

// class 'GeeksForGeeks' inherit in

// another child class 'Geek2'

public class Geek2 : GeeksForGeeks

{

// same as the previous class

public override void gfg()

{

Console.WriteLine("class Geek2");

}

}

public static void Main()

{

// 'g' is object of class

// 'GeeksForGeeks' class '

// GeeksForGeeks' cannot

// be instantiate

GeeksForGeeks g;

// instantiate class 'Geek1'

g = new Geek1();

// call 'gfg()' of class 'Geek1'

g.gfg();

// instantiate class 'Geek2'

g = new Geek2();

// call 'gfg()' of class 'Geek2'

g.gfg();

}

OUTPUT :

class Geek1

class Geek2

Example : 3

abstract class AreaClass

{

abstract public int Area();

}

class Square : AreaClass

{

int side = 0;

public Square(int n)

{

side = n;

}

public override int Area()

{

return side \* side;

}

}

class gfg

{

public static void Main()

{

Square s = new Square(6);

Console.WriteLine("Area = " + s.Area());

}

}

Example 3 :

abstract class AbstractClass

{

public int AddTwoNumbers(int Num1, int Num2)

{

return Num1 + Num2;

}

public abstract int MultiplyTwoNumbers(int Num1, int Num2);

}

class Derived : AbstractClass

{

public override int MultiplyTwoNumbers(int Num1, int Num2)

{

return Num1 \* Num2;

}

}

// Driver Class

class geek

{

// Main Method

public static void Main()

{

// Instance of the derived class

Derived d = new Derived();

Console.WriteLine("Addition : {0}\nMultiplication :{1}",

d.AddTwoNumbers(4, 6),

d.MultiplyTwoNumbers(6, 4));

}

}

Example 4:

Abstract class support public, static and protected constructor

public abstract class A

{

protected A()

{

Console.WriteLine("Abstract class protected constructor");

}

public abstract void Add();

}

public class B : A

{

public B()

{

Console.WriteLine("Public class public constructor");

}

public override void Add()

{

Console.WriteLine("override method");

}

}

***Implicitly Calling: Interface***

namespace Learning

{

public interface IEmployee

{

string GetDay();

int GetMonth();

}

public class Employee : IEmployee

{

public string GetDay()

{

return DateTime.Now.DayOfWeek.ToString();

}

public int GetMonth()

{

return DateTime.Now.Month;

}

}

class Program

{

static void Main(string[] args)

{

Employee e = new Employee();

e.GetMonth();

e.GetDay();

}

}

}

***Explicitly calling: Interface***

namespace Learning

{

public interface IEmployee

{

string GetDay();

int GetMonth();

}

public class Employee : IEmployee

{

string IEmployee.GetDay()

{

throw new NotImplementedException();

}

int IEmployee.GetMonth()

{

throw new NotImplementedException();

}

}

class Program

{

static void Main(string[] args)

{

IEmployee e = new Employee();

e.GetDay();

e.GetMonth();

}

}

}

## **Singleton class vs. Static methods**

The following conpares Singleton class vs. Static methods:

1. A Static Class cannot be extended whereas a singleton class can be extended.
2. A Static Class can still have instances (unwanted instances) whereas a singleton class prevents it.
3. A Static Class cannot be initialized with a STATE (parameter), whereas a singleton class can be.
4. A Static class is loaded automatically by the CLR when the program or namespace containing the class is loaded.

## **Advantages of Singleton Design Pattern**

The advantages of a Singleton Pattern are:

1. Singleton pattern can be implemented interfaces.
2. It can be also inherited from other classes.
3. It can be lazy-loaded.
4. It has Static Initialization.
5. It can be extended into a factory pattern.
6. It helps to hide dependencies.

***Singleton Class***

public class Singleton

{

private static int counter = 0;

private static Singleton instance = null;

public static Singleton GetInstance

{

get

{

if (instance == null)

instance = new Singleton();

return instance;

}

}

private Singleton()

{

counter++;

Console.WriteLine("Counter Value " + counter.ToString());

}

public void PrintDetails(string message)

{

Console.WriteLine(message);

}

public class DerivedSingleton : Singleton

{

}

}

class Program

{

static void Main(string[] args)

{

Singleton fromTeachaer = Singleton.GetInstance;

fromTeachaer.PrintDetails("From Teacher");

Singleton fromStudent = Singleton.GetInstance;

fromStudent.PrintDetails("From Student");

/\*

\* Instantiating singleton from a derived class.

\* This violates singleton pattern principles.

\*/

Singleton.DerivedSingleton derivedObj = new Singleton.DerivedSingleton();

derivedObj.PrintDetails("From Derived");

Singleton.DerivedSingleton derivedObj1 = new Singleton.DerivedSingleton();

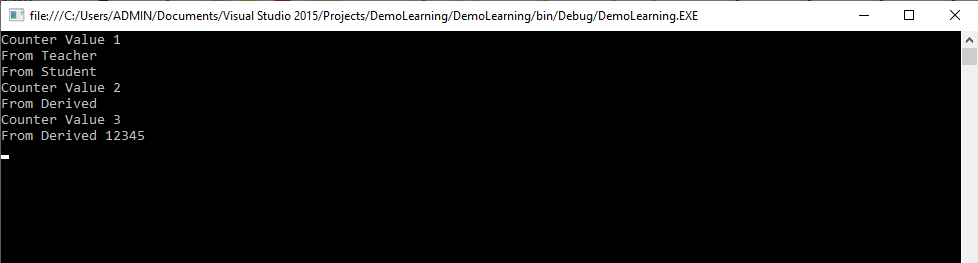
derivedObj1.PrintDetails("From Derived 12345");

Console.ReadLine();

}

}

Output:



Example 2:

class Singleton

{

private static Singleton \_instance;

// Constructor is 'protected'

protected Singleton()

{

}

public static Singleton Instance()

{

// Uses lazy initialization.

// Note: this is not thread safe.

if (\_instance == null)

{

\_instance = new Singleton();

Console.WriteLine("Instance created");

}

return \_instance;

}

}

class Program

{

static void Main(string[] args)

{

// Constructor is protected -- cannot use new

Singleton s1 = Singleton.Instance();

Singleton s2 = Singleton.Instance();

// Test for same instance

if (s1 == s2)

{

Console.WriteLine("Objects are the same instance");

}

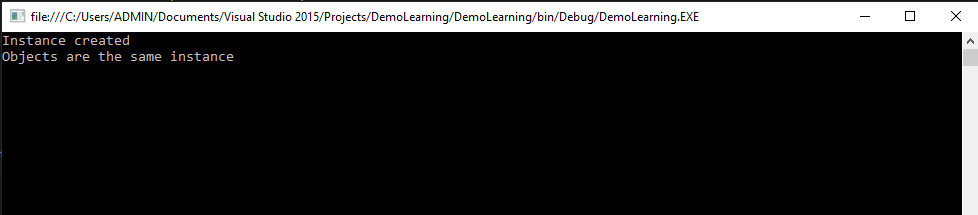
// Wait for user

Console.ReadKey();

}

}

Output:



Singleton Class

public class SingletonExample

{

private static SingletonExample instance;

private SingletonExample () {}

public static SingletonExample Instance

{

get

{

if (instance == null)

{

instance = new SingletonExample ();

}

return instance;

}

}

}

Abstract Factory Design Pattern

abstract class CreditCard

{

public abstract string CardType { get; }

public abstract int CreditLimit { get; set; }

public abstract int AnnualCharge { get; set; }

}

class MoneyBackCreditCard : CreditCard

{

private readonly string \_cardType;

private int \_creditLimit;

private int \_annualCharge;

public MoneyBackCreditCard(int creditLimit, int annualCharge)

{

\_cardType = "MoneyBack";

\_creditLimit = creditLimit;

\_annualCharge = annualCharge;

}

public override string CardType

{

get { return \_cardType; }

}

public override int CreditLimit

{

get { return \_creditLimit; }

set { \_creditLimit = value; }

}

public override int AnnualCharge

{

get { return \_annualCharge; }

set { \_annualCharge = value; }

}

}

class TitaniumCreditCard : CreditCard

{

private readonly string \_cardType;

private int \_creditLimit;

private int \_annualCharge;

public TitaniumCreditCard(int creditLimit, int annualCharge)

{

\_cardType = "Titanium";

\_creditLimit = creditLimit;

\_annualCharge = annualCharge;

}

public override string CardType

{

get { return \_cardType; }

}

public override int CreditLimit

{

get { return \_creditLimit; }

set { \_creditLimit = value; }

}

public override int AnnualCharge

{

get { return \_annualCharge; }

set { \_annualCharge = value; }

}

}

abstract class CardFactory

{

public abstract CreditCard GetCreditCard();

}

class MoneyBackFactory : CardFactory

{

private int \_creditLimit;

private int \_annualCharge;

public MoneyBackFactory(int creditLimit, int annualCharge)

{

\_creditLimit = creditLimit;

\_annualCharge = annualCharge;

}

public override CreditCard GetCreditCard()

{

return new MoneyBackCreditCard(\_creditLimit, \_annualCharge);

}

}

class TitaniumFactory : CardFactory

{

private int \_creditLimit;

private int \_annualCharge;

public TitaniumFactory(int creditLimit, int annualCharge)

{

\_creditLimit = creditLimit;

\_annualCharge = annualCharge;

}

public override CreditCard GetCreditCard()

{

return new TitaniumCreditCard(\_creditLimit, \_annualCharge);

}

}

class Program

{

static void Main(string[] args)

{

CardFactory factory = null;

Console.Write("Enter the card type you would like to visit: ");

string car = Console.ReadLine();

switch (car.ToLower())

{

case "moneyback":

factory = new MoneyBackFactory(50000, 0);

break;

case "titanium":

factory = new TitaniumFactory(100000, 500);

break;

default:

break;

}

CreditCard creditCard = factory.GetCreditCard();

Console.WriteLine("\nYour card details are below : \n");

Console.WriteLine("Card Type: {0}\nCredit Limit: {1}\nAnnual Charge: {2}",

creditCard.CardType, creditCard.CreditLimit, creditCard.AnnualCharge);

Console.ReadKey();

}

}

***Partial Class***

**Advantages:**

* With the help of partial class multiple developers can work simultaneously on the same class in different files.
* With the help of partial class concept you can split the UI of design code and the business logic code to read and understand the code.
* When you were working with automatically generated code, the code can be added to the class without having to recreate the source file like in Visual studio.
* You can also maintain your application in an efficient manner by compressing large classes into small ones.

class Program

{

static void Main(string[] args)

{

Geeks objGeek = new Geeks("Avadhut",1);

objGeek.Display();

}

}

public partial class Geeks

{

private string Author\_name;

private int Total\_articles;

public Geeks(string a, int t)

{

this.Author\_name = a;

this.Total\_articles = t;

}

}

public partial class Geeks

{

public void Display()

{

Console.WriteLine("Author's name is : " + Author\_name);

Console.WriteLine("Total number articles is : " + Total\_articles);

Console.ReadLine();

}

}

Example 2:

public partial class Geeks

{

public Geeks()

{

Console.WriteLine("Constructor 1");

}

}

public partial class Geeks

{

static Geeks()

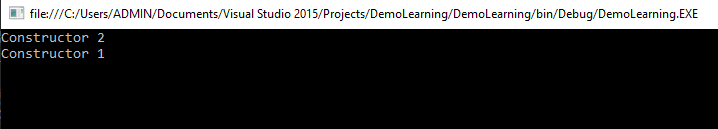
{

Console.WriteLine("Constructor 2");

}

}

Output:



Structure:

## **Difference between structs and classes**

|  |  |
| --- | --- |
| **Structs** | **classes** |
| structs are value type | classes are reference type |
| structs are stored in stack or a inline | classes are stored on managed heap |
| structs doesn't support inheritance | classes support inheritance |
| But handing of constructor is different in structs. The complier supplies a default no-parameter constructor, which your are not permitted to replace | Constructors are fully supported in classes |

public interface aa

{

// no access specifier is given in interface methods (by defualt they are public)

double Increment();

void DisplayValues();

}

public struct Student : aa

{

int id;

int zipcode;

double salary;

public Student(int id, int zipcode, double salary)

{

this.id = id;

this.zipcode = zipcode;

this.salary = salary;

}

public void DisplayValues()

{

Console.WriteLine("ID: " + this.id.ToString());

Console.WriteLine("Zipcode : " + this.zipcode.ToString());

Console.WriteLine("Salary : " + this.salary.ToString());

}

public double Increment()

{

return (this.salary += 1000.00);

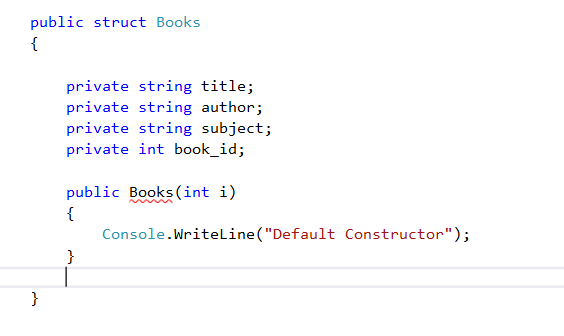
}

}

* Struct cannot have a default constructor (a constructor without parameters) or a destructor.
* Structs are value types and are copied on assignment.
* Structs are value types while classes are reference types.
* Structs can be instantiated without using a new operator.
* A struct cannot inherit from another struct or class, and it cannot be the base of a class. All structs inherit directly from System.ValueType, which inherits from System.Object.
* Struct cannot be a base class. So, Struct types cannot abstract and are always implicitly sealed.

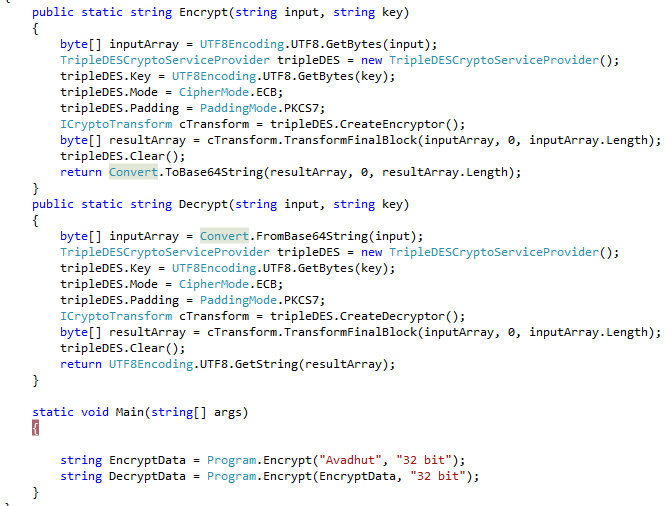
Question :-

**Inside struct member and parametrized constructor have created.wil it be work?**



How to Encrypt Data and Decrypt Data : Using UTF8Encoding we can possible to convert data to encrypt to decrypt format.

using System.Security.Cryptography;



## **What is Serialization?**

Serialization is the process of bringing an object into a form that it can be written on stream. It's the process of converting the object into a form so that it can be stored on a file, database, or memory; or, it can be transferred across the network. Its main purpose is to save the state of the object so that it can be recreated when needed.

## **What is Deserialization?**

As the name suggests, deserialization is the reverse process of serialization. It is the process of getting back the serialized object so that it can be loaded into memory. It resurrects the state of the object by setting properties, fields etc.

First, we go and import the name space for serialization.

***Using System.Runtime.Serialization;***

Then Import the formatter

***using System.Runtime.Serialization.Formatters.Binary;***

**Types**

* Binary Serialization
* XML Serialization
* JSON Serialization

Advantages of serialization

1. Passing an object from one application to another.
2. Passing an object from one domain to another.

Binary Serialization :

1. Employee emp = **new** Employee {
2. name = textBoxName.Text,
3. phone = textBoxPhone.Text,
4. dob = dateTimePickerDoB.Value,
5. department = textBoxDepartment.Text,
6. salary = Convert.ToInt32(textBoxSalary.Text),
7. additionalInfo = "We don't want it to serialize"
8. };
9. BinaryFormatter bf = **new** BinaryFormatter();
11. FileStream fsout = **new** FileStream("employee.binary", FileMode.Create, FileAccess.Write, FileShare.None);
12. **try**
13. {
14. **using** (fsout)
15. {
16. bf.Serialize(fsout, emp);
17. label6.Text = "Object Serialized";
18. }
19. }

Binary Deserialization:

1. Employee emp = **new** Employee();
3. BinaryFormatter bf = **new** BinaryFormatter();
5. FileStream fsin = **new** FileStream("employee.binary", FileMode.Open, FileAccess.Read, FileShare.None);
6. **try**
7. {
8. **using** (fsin)
9. {
10. emp = (Employee) bf.Deserialize(fsin);
11. label6.Text = "Object Deserialized";
13. textBoxName.Text = emp.name;
14. textBoxPhone.Text = emp.phone;
15. dateTimePickerDoB.Value = emp.dob;
16. textBoxDepartment.Text = emp.department;
17. textBoxSalary.Text = emp.salary.ToString();
18. }
19. }

Indexer :

**Important points to remember on indexers:**

* Indexers are always created with **this** keyword.
* Parameterized property are called indexer.
* Indexers are implemented through get and set accessors for the [ ] operator.
* ref and out parameter modifiers are not permitted in indexer.
* The formal parameter list of an indexer corresponds to that of a method and at least one parameter should be specified.
* Indexer is an instance member so can't be static but property can be static.
* Indexers are used on group of elements.
* Indexer is identified by its signature where as a property is identified it's name.
* Indexers are accessed using indexes where as properties are accessed by names.
* Indexer can be overloaded.

1. **class** Program
2. {
3. **class** IndexerClass
4. {
5. **private** **string**[] names = **new** **string**[10];
6. **public** **string** **this**[**int** i]
7. {
8. **get**
9. {
10. **return** names[i];
11. }
12. **set**
13. {
14. names[i] = value;
15. }
17. }
18. **static** **void** Main(**string**[] args)
19. {
20. IndexerClass Team = **new** IndexerClass();
21. Team[0] = "Rocky";
22. Team[1] = "Teena";
23. Team[2] = "Ana";
24. Team[3] = "Victoria";
25. Team[4] = "Yani";
26. Team[5] = "Mary";
27. Team[6] = "Gomes";
28. Team[7] = "Arnold";
29. Team[8] = "Mike";
30. Team[9] = "Peter";
31. **for** (**int** i = 0; i < 10; i++)
32. {
33. Console.WriteLine(Team[i]);
34. }
35. Console.ReadKey();
36. }
37. }

## **Difference between Indexers and Properties**

|  |  |
| --- | --- |
| Indexers | Properties |
| Indexers are created with this keyword. | Properties don't require this keyword. |
| Indexers are identified by signature. | Properties are identified by their names. |
| Indexers are accessed using indexes. | Properties are accessed by their names. |
| Indexer are instance member, so can't be static. | Properties can be static as well as instance members. |
| A get accessor of an indexer has the same formal parameter list as the indexer. | A get accessor of a property has no parameters. |
| A set accessor of an indexer has the same formal parameter list as the indexer, in addition to the value parameter. | A set accessor of a property contains the implicit value parameter. |

Attribute:

An **attribute** is a declarative tag that is used to convey information to runtime about the behaviors of various elements like classes, methods, structures, enumerators, assemblies etc. in your program. You can add declarative information to a program by using an attribute.

[obsolete] : Method need to use or not.

1. **class** MyClass
2. {
3. [Obsolete("Use method SaySomething1")]
4. **public** **void** SaySomething(){ Console.WriteLine("Execute saysomething()");}
5. **public** **void** SaySomething1(){ Console.WriteLine("Execute saysomething1()");}
6. }

[serialization]

[NonSerialization]

[Web Method]

Custom Attribute: - creation

// AttributeUsage specifies the usage

// of InformationAttribute

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Constructor |

AttributeTargets.Method, AllowMultiple = true)]

// InformationAttribute is a custom attribute class

// that is derived from Attribute class

class InformationAttribute : Attribute

{

public string InformationString { get; set; }

}

// InformationAttribute is used in student class

[Information(InformationString = "Class")]

public class student

{

private int rollno;

private string name;

[Information(InformationString = "Constructor")]

public student(int rollno, string name)

{

this.rollno = rollno;

this.name = name;

}

[Information(InformationString = "Method")]

public void display()

{

Console.WriteLine("Roll Number: {0}", rollno);

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

}

}

<https://www.c-sharpcorner.com/UploadFile/84c85b/using-attributes-with-C-Sharp-net/>

**What is a Delegate?**

Delegate is similar to a function pointer in C & C++ but the delegates are user defined types in C#. Make a note that delegates are not a member of class, but similar to a class. These are the backbone for events.

**When to use delegates?**

I believe that a lot of people can answer the "what is a delegate?" question in interviews but are not able to explain when to use it. No worries! Let me tell you a few important points about delegates.

1. These are used to represent or refer to one or more functions.
2. These can only be used to define call-back methods.
3. In order to consume a delegate, we need to create an object to delegate.

**How many types of delegates are present?**

There are two types of delegates available.

1. Single Cast Delegate
2. Multi Cast Delegate
3. **class** Program {
4. **static** **void** Display(string S) {
5. Console.WriteLine("My Name is :" + S);
6. }
7. delegate **void** X(string a);
8. **static** **void** Main(string[] args) {
9. X objD = **new** X(Display);
10. objD("Rathrola Prem Kumar");
11. Console.Read();
12. }
13. }

1. **class** Program {
2. **public** **void** Display(string S) {
3. Console.WriteLine("My Designation is :" + S);
4. }
5. **public** delegate **void** Delegate(string a);
6. **class** DelegateDemo {
7. **static** **void** Main() {
8. Program obj1 = **new** Program();
9. Delegate objD = **new** Delegate(obj1.Display);
10. objD("Technical Specialist");
11. Console.Read();
12. }
13. }
14. }

MulitcasteDelegate

1. **class** Program {
2. **public** **void** Add(**int** x, **int** y) {
3. Console.WriteLine("Sum is:" + (x + y));
4. }
5. **public** **void** Subtract(**int** x, **int** y) {
6. Console.WriteLine("Difference is:" + (x - y));
7. }
8. **public** **void** Multiply(**int** x, **int** y) {
9. Console.WriteLine("Product is:" + (x \* y));
10. }
11. **public** **void** Divide(**int** x, **int** y) {
12. Console.WriteLine("Quotient is:" + (x / y));
13. }
14. }
15. **public** delegate **void** MultiCastDelegate(**int** a, **int** b);
16. **class** ClsDelegate {
17. **static** **void** Main() {
18. Program obj1 = **new** Program();
19. MultiCastDelegate objD = **new** MultiCastDelegate(obj1.Multiply);
20. objD += obj1.Add;
21. objD += obj1.Substract;
22. objD += obj1.Divide;
23. objD(40, 10);
24. objD -= obj1.Add;
25. objD -= obj1.Divide;
26. objD(50, 10);
27. Console.ReadLine();
28. }
29. }

## **Array of Delegates**

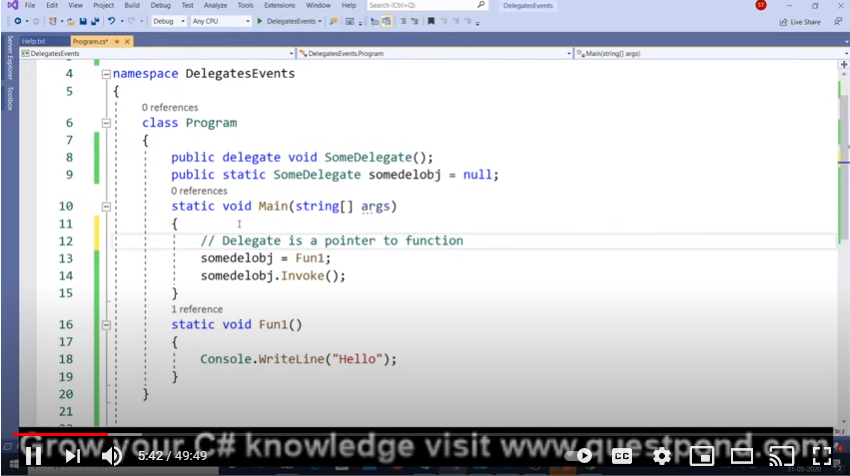
1. **namespace** Delegates
2. {
3. **public** **class** Operation
4. {
5. **public** **static** **void** Add(**int** a, **int** b)
6. {
7. Console.WriteLine("Addition={0}",a + b);
8. }
10. **public** **static** **void** Multiple(**int** a, **int** b)
11. {
12. Console.WriteLine("Multiply={0}", a \* b);
13. }
14. }
16. **class** Program
17. {
18. **delegate** **void** DelOp(**int** x, **int** y);
20. **static** **void** Main(**string**[] args)
21. {
22. // Delegate instantiation
23. DelOp[] obj =
24. {
25. **new** DelOp(Operation.Add),
26. **new** DelOp(Operation.Multiple)
27. };
29. **for** (**int** i = 0; i < obj.Length; i++)
30. {
31. obj[i](2, 5);
32. obj[i](8, 5);
33. obj[i](4, 6);
34. }
35. Console.ReadLine();
36. }
37. }
38. }

## **Anonymous Methods**

1. **namespace** Delegates
2. {
3. **class** Program
4. {
5. // Delegate Definition
6. **delegate** **void** operation();
8. **static** **void** Main(**string**[] args)
9. {
10. // Delegate instantiation
11. operation obj = **delegate**
12. {
13. Console.WriteLine("Anonymous method");
14. };
15. obj();
17. Console.ReadLine();
18. }
19. }
20. }

## **Events**

1. **namespace** Delegates
2. {
3. **public** **delegate** **void** DelEventHandler();
5. **class** Program
6. {
7. **public** **static** **event** DelEventHandler add;
9. **static** **void** Main(**string**[] args)
10. {
11. add += **new** DelEventHandler(USA);
12. add += **new** DelEventHandler(India);
13. add += **new** DelEventHandler(England);
14. add.Invoke();
16. Console.ReadLine();
17. }
18. **static** **void** USA()
19. {
20. Console.WriteLine("USA");
21. }
23. **static** **void** India()
24. {
25. Console.WriteLine("India");
26. }
28. **static** **void** England()
29. {
30. Console.WriteLine("England");
31. }
32. }
33. }



Thread

using System.Threading.Tasks;

public static void StartupA(object parameters)

{

string Name = Convert.ToString(parameters);

Console.WriteLine(Name);

}

public static void StartupB(string Thread , string Expression)

{

Console.WriteLine(Thread + ":" + Expression);

}

public static void ThreadStart()

{

Console.WriteLine("Thread Start");

}

static void Main(string[] args)

{

Thread t = new Thread(new ParameterizedThreadStart(StartupA));

t.Name = "Paramatertize Thread";

t.Start(t.Name);

// You can also use an anonymous delegate to do this.

Thread t2 = new Thread(delegate()

{

ThreadStart();

});

t2.Start();

//// Or lambda expressions if you are using C# 3.0

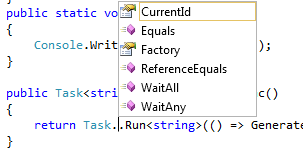
Thread t3 = new Thread(() => StartupB("Thread", "Lambda Expression"));

t3.Start();

Console.ReadLine();

}

Task Property:



async / await Tutorial in C#

Nowadays, Asynchronous programming is very popular with the help of the async and await keywords in C#. When we are dealing with UI and on button click, we use a long running method like reading a large file or something else which will take a long time, in that case, the entire application must wait to complete the whole task. In other words, if any process is blocked in a synchronous application, the entire application gets blocked and our application stops responding until the whole task completes.

Asynchronous programming is very helpful in this condition. By using Asynchronous programming, the Application can continue with the other work that does not depend on the completion of the whole task.

We will get all the benefits of traditional Asynchronous programming with much less effort by the help of async and await keywords.

Suppose, we are using two methods as Method1 and Method2 respectively and both the methods are not dependent on each other and Method1 is taking a long time to complete its task. In Synchronous programming, it will execute the first Method1 and it will wait for completion of this method and then it will execute Method2. Thus, it will be a time intensive process even though both the methods are not depending on each other.

We can run all the methods parallelly by using the simple thread programming but it will block UI and wait to complete all the tasks. To come out of this problem, we have to write too many codes in traditional programming but if we will simply use the async and await keywords, then we will get the solutions in much less code.

public static void DoSynchronousWork()

{

// You can do whatever work is needed here

Console.WriteLine("1. Doing some work synchronously");

}

static async Task DoSomethingAsync() //A Task return type will eventually yield a void

{

await GetStringAsync(); // we are awaiting the Async Method GetStringAsync

Console.WriteLine("3. Async task ended...");

}

static async Task GetStringAsync()

{

Console.WriteLine("2. Async task has started...");

}

static void DoSynchronousWorkAfterAwait()

{

//This is the work we can do while waiting for the awaited Async Task to complete

Console.WriteLine("7. While waiting for the async task to finish, we can do some unrelated work...");

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

{

for (var j = i; j <= 5; j++)

{

Console.Write("\*");

}

Console.WriteLine();

}

}

static void Main(string[] args)

{

DoSynchronousWork();

var someTask = DoSomethingAsync();

DoSynchronousWorkAfterAwait();

someTask.Wait(); //this is a blocking call, use it only on Main method

Console.ReadLine();

}

 Thread.Sleep(200);

The Sleep() method suspends the current thread for the specified milliseconds

The Abort() method is used to terminate the thread.

Thread t1 = **new** Thread(**new** ThreadStart(mt.Thread1));

 t1.Start();

t1.Abort();

# C# Threading Example: Join() method

It causes all the calling threads to wait until the current thread (joined thread) is terminated or completes its task.

MyThread mt = new MyThread();

Thread t1 = new Thread(new ThreadStart(mt.Thread1));

Thread t2 = new Thread(new ThreadStart(mt.Thread1));

Thread t3 = new Thread(new ThreadStart(mt.Thread1));

t1.Start();

t1.Join();

t2.Start();

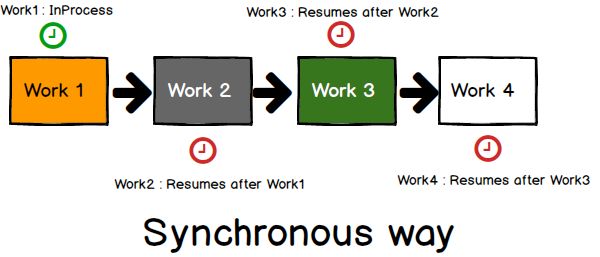
t3.Start();

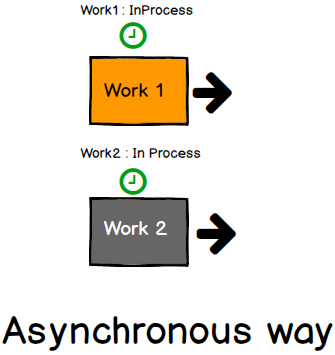
static void Main(string[] args)  
{   
     ThreadStart ts = new ThreadStart(run);  
     // create new thread  
     Thread thrd = new Thread(ts);  
     // start thread  
     thrd.Start();  
     Thread.Sleep(1000);

for (int t = 10; t > 0; t--)  
     {  
         Console.WriteLine("Main Thread value is :" + t);  
         Thread.Sleep(1000);  
     }  
  
     Console.WriteLine("Good Bye!!!I'm main Thread");  
     Console.ReadLine();

}

static void run()  
{  
      for (int i = 0; i < 10; i++)  
      {  
          Console.WriteLine("Sub Thread value is : " + i);  
          Thread.Sleep(1000);  
      }  
      Console.WriteLine("Good Bye!!!I'm Sub Thread");  
}





How to call store procedure in c# with input and output parameter

1. SqlCommand cmd = **new** SqlCommand("spuserdetail", con);
2. cmd.CommandType = CommandType.StoredProcedure;
3. cmd.Parameters.AddWithValue("@UserName", UserName);
4. cmd.Parameters.AddWithValue("@Password", Password);
5. cmd.Parameters.AddWithValue("@Email", Email);
6. cmd.Parameters.AddWithValue("@Country", Country);
7. cmd.Parameters.Add("@ERROR", SqlDbType.Char, 500);
8. cmd.Parameters["@ERROR"].Direction = ParameterDirection.Output;
9. cmd.ExecuteNonQuery();
10. message = (string) cmd.Parameters["@ERROR"].Value;

How to retrieve connection string from web config file

ConfigurationManger.ConnectionString[“DBMS”].connectionstring

using (SqlConnection cn = new SqlConnection(cnString))

    {

     cn.Open();

        using (SqlDataAdapter da = new SqlDataAdapter(sql, cn))

        {

         da.SelectCommand.CommandTimeout = 120;

            DataSet ds = new DataSet();

            da.Fill(ds);

            return ds.Tables[0];

        }

    }

//Execute stored procedure

ObjectParameter newkey = new ObjectParameter("newKey", typeof(char));

db.RF\_GetNewKey("A", "B", "S",newkey);

//get new key output from stored procedure RF\_GetNewKey

string myKey=newkey.Value.ToString();

How to read excel file in c#

var fileName = @"C:\ExcelFile.xlsx";

var connectionString = "Provider=Microsoft.ACE.OLEDB.12.0;Data Source=" + fileName + ";Extended Properties=\"Excel 12.0;IMEX=1;HDR=NO;TypeGuessRows=0;ImportMixedTypes=Text\""; ;

using (var conn = new OleDbConnection(connectionString))

{

conn.Open();

var sheets = conn.GetOleDbSchemaTable(System.Data.OleDb.OleDbSchemaGuid.Tables, new object[] { null, null, null, "TABLE" });

using (var cmd = conn.CreateCommand())

{

cmd.CommandText = "SELECT \* FROM [" + sheets.Rows[0]["TABLE\_NAME"].ToString() + "] ";

var adapter = new OleDbDataAdapter(cmd);

var ds = new DataSet();

adapter.Fill(ds);

}

}

How to create Excel file

public static void exportToExcel(DataSet source, string fileName)

{

const string endExcelXML = "</Workbook>";

const string startExcelXML = "<xml version>\r\n<Workbook " +

"xmlns=\"urn:schemas-microsoft-com:office:spreadsheet\"\r\n" +

" xmlns:o=\"urn:schemas-microsoft-com:office:office\"\r\n " +

"xmlns:x=\"urn:schemas- microsoft-com:office:" +

"excel\"\r\n xmlns:ss=\"urn:schemas-microsoft-com:" +

"office:spreadsheet\">\r\n <Styles>\r\n " +

"<Style ss:ID=\"Default\" ss:Name=\"Normal\">\r\n " +

"<Alignment ss:Vertical=\"Bottom\"/>\r\n <Borders/>" +

"\r\n <Font/>\r\n <Interior/>\r\n <NumberFormat/>" +

"\r\n <Protection/>\r\n </Style>\r\n " +

"<Style ss:ID=\"BoldColumn\">\r\n <Font " +

"x:Family=\"Swiss\" ss:Bold=\"1\"/>\r\n </Style>\r\n " +

"<Style ss:ID=\"StringLiteral\">\r\n <NumberFormat" +

" ss:Format=\"@\"/>\r\n </Style>\r\n <Style " +

"ss:ID=\"Decimal\">\r\n <NumberFormat " +

"ss:Format=\"0.0000\"/>\r\n </Style>\r\n " +

"<Style ss:ID=\"Integer\">\r\n <NumberFormat " +

"ss:Format=\"0\"/>\r\n </Style>\r\n <Style " +

"ss:ID=\"DateLiteral\">\r\n <NumberFormat " +

"ss:Format=\"mm/dd/yyyy;@\"/>\r\n </Style>\r\n " +

"</Styles>\r\n ";

System.IO.StreamWriter excelDoc = null;

excelDoc = new System.IO.StreamWriter(fileName);

int sheetCount = 1;

excelDoc.Write(startExcelXML);

foreach (DataTable table in source.Tables)

{

int rowCount = 0;

excelDoc.Write("<Worksheet ss:Name=\"" + table.TableName + "\">");

excelDoc.Write("<Table>");

excelDoc.Write("<Row>");

for (int x = 0; x < table.Columns.Count; x++)

{

excelDoc.Write("<Cell ss:StyleID=\"BoldColumn\"><Data ss:Type=\"String\">");

excelDoc.Write(table.Columns[x].ColumnName);

excelDoc.Write("</Data></Cell>");

}

excelDoc.Write("</Row>");

foreach (DataRow x in table.Rows)

{

rowCount++;

//if the number of rows is > 64000 create a new page to continue output

if (rowCount == 64000)

{

rowCount = 0;

sheetCount++;

excelDoc.Write("</Table>");

excelDoc.Write(" </Worksheet>");

excelDoc.Write("<Worksheet ss:Name=\"" + table.TableName + "\">");

excelDoc.Write("<Table>");

}

excelDoc.Write("<Row>"); //ID=" + rowCount + "

for (int y = 0; y < table.Columns.Count; y++)

{

System.Type rowType;

rowType = x[y].GetType();

switch (rowType.ToString())

{

case "System.String":

string XMLstring = x[y].ToString();

XMLstring = XMLstring.Trim();

XMLstring = XMLstring.Replace("&", "&");

XMLstring = XMLstring.Replace(">", ">");

XMLstring = XMLstring.Replace("<", "<");

excelDoc.Write("<Cell ss:StyleID=\"StringLiteral\">" +

"<Data ss:Type=\"String\">");

excelDoc.Write(XMLstring);

excelDoc.Write("</Data></Cell>");

break;

case "System.DateTime":

//Excel has a specific Date Format of YYYY-MM-DD followed by

//the letter 'T' then hh:mm:sss.lll Example 2005-01-31T24:01:21.000

//The Following Code puts the date stored in XMLDate

//to the format above

DateTime XMLDate = (DateTime)x[y];

string XMLDatetoString = ""; //Excel Converted Date

XMLDatetoString = XMLDate.Year.ToString() +

"-" +

(XMLDate.Month < 10 ? "0" +

XMLDate.Month.ToString() : XMLDate.Month.ToString()) +

"-" +

(XMLDate.Day < 10 ? "0" +

XMLDate.Day.ToString() : XMLDate.Day.ToString()) +

"T" +

(XMLDate.Hour < 10 ? "0" +

XMLDate.Hour.ToString() : XMLDate.Hour.ToString()) +

":" +

(XMLDate.Minute < 10 ? "0" +

XMLDate.Minute.ToString() : XMLDate.Minute.ToString()) +

":" +

(XMLDate.Second < 10 ? "0" +

XMLDate.Second.ToString() : XMLDate.Second.ToString()) +

".000";

excelDoc.Write("<Cell ss:StyleID=\"DateLiteral\">" +

"<Data ss:Type=\"DateTime\">");

excelDoc.Write(XMLDatetoString);

excelDoc.Write("</Data></Cell>");

break;

case "System.Boolean":

excelDoc.Write("<Cell ss:StyleID=\"StringLiteral\">" +

"<Data ss:Type=\"String\">");

excelDoc.Write(x[y].ToString());

excelDoc.Write("</Data></Cell>");

break;

case "System.Int16":

case "System.Int32":

case "System.Int64":

case "System.Byte":

excelDoc.Write("<Cell ss:StyleID=\"Integer\">" +

"<Data ss:Type=\"Number\">");

excelDoc.Write(x[y].ToString());

excelDoc.Write("</Data></Cell>");

break;

case "System.Decimal":

case "System.Double":

excelDoc.Write("<Cell ss:StyleID=\"Decimal\">" +

"<Data ss:Type=\"Number\">");

excelDoc.Write(x[y].ToString());

excelDoc.Write("</Data></Cell>");

break;

case "System.DBNull":

excelDoc.Write("<Cell ss:StyleID=\"StringLiteral\">" +

"<Data ss:Type=\"String\">");

excelDoc.Write("");

excelDoc.Write("</Data></Cell>");

break;

default:

throw (new Exception(rowType.ToString() + " not handled."));

}

}

excelDoc.Write("</Row>");

}

excelDoc.Write("</Table>");

excelDoc.Write(" </Worksheet>");

sheetCount++;

}

excelDoc.Write(endExcelXML);

excelDoc.Close();

}

## **Create and Access Tuples**

1. // Create a 3-tuple
2. var author = **new** Tuple<**string**, **string**, **int**>("Mahesh Chand", "ADO.NET Programming", 2003);
4. // Display author info
5. System.Console.WriteLine("Author {0} wrote his first book titled {1} in {2}.", author.Item1, author.Item2, author.Item3);

## **Tuples in methods**

1. ***public******void****SetTupleMethod(Tuple<****string****,****string****,****int****> tupleAuthor)*
2. *{*
3. *var author2 = tupleAuthor;*
4. *Console.WriteLine("Author:{0}, Title:{1}, Year:{2}.",*
5. *author2.Item1, author2.Item2, author2.Item3);*
6. *}*

1. ts.SetTupleMethod(**new** Tuple<**string**, **string**, **int**>(
2. "Mike Gold", "Code UML", 2005));

## **Return Tuples**

1. **public** **static** Tuple<**string**, **string**, **int**> GetTupleMethod()
2. {
3. // Create a 3-tuple and return it
4. var author = **new** Tuple<**string**, **string**, **int**>(
5. "Mahesh Chand", "Programming C#", 2002);
6. **return** author;
7. }
8. var author2 = TupleSamples.GetTupleMethod();
9. Console.WriteLine("Author:{0}, Title:{1}, Year:{2}.", author2.Item1, author2.Item2, author2.Item3);

You’re now ready to use C# 7.0 tuples. (Value Tuple )

1. // tuple return type
2. **public** (**string**, **string**, **long**) TupleReturnLiteral(**long** id)
3. {
5. **string** name = **string**.Empty;
6. **string** title = **string**.Empty;
7. **long** year = 0;
9. **if** (id == 1000)
10. {
11. name = "Mahesh Chand";
12. title = "ADO.NET Programming";
13. year = 2003;
14. }
16. // tuple literal
17. **return** (name, title, year);
19. }

<https://www.c-sharpcorner.com/article/tuples-in-c-sharp/>

IEnumerable is interface contain to below method

public interface IEnumerable

{

[DispId(-4)]

IEnumerator GetEnumerator();

}

IEnumerator IS Inherit to is interface contain to below method

public interface IEnumerator

{

object Current { get; }

bool MoveNext();

void Reset();

}

| Non-generic Collections | Usage |
| --- | --- |
| [ArrayList](https://www.tutorialsteacher.com/csharp/csharp-arraylist) | ArrayList stores objects of any type like an array. However, there is no need to specify the size of the ArrayList like with an array as it grows automatically. |
| [SortedList](https://www.tutorialsteacher.com/csharp/csharp-sortedlist) | SortedList stores key and value pairs. It automatically arranges elements in ascending order of key by default. C# includes both, generic and non-generic SortedList collection. |
| [Stack](https://www.tutorialsteacher.com/csharp/csharp-stack) | Stack stores the values in LIFO style (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values. C# includes both, generic and non-generic Stack. |
| [Queue](https://www.tutorialsteacher.com/csharp/csharp-queue) | Queue stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection. C# includes generic and non-generic Queue. |
| [Hashtable](https://www.tutorialsteacher.com/csharp/csharp-hashtable) | Hashtable stores key and value pairs. It retrieves the values by comparing the hash value of the keys. |
| BitArray | BitArray manages a compact array of bit values, which are represented as Booleans, where true indicates that the bit is on (1) and false indicates the bit is off (0). |

## **Important Methods of Static File Class**

| Method | Usage |
| --- | --- |
| AppendAllLines | Appends lines to a file, and then closes the file. If the specified file does not exist, this method creates a file, writes the specified lines to the file, and then closes the file. |
| AppendAllText | Opens a file, appends the specified string to the file, and then closes the file. If the file does not exist, this method creates a file, writes the specified string to the file, then closes the file. |
| AppendText | Creates a StreamWriter that appends UTF-8 encoded text to an existing file, or to a new file if the specified file does not exist. |
| Copy | Copies an existing file to a new file. Overwriting a file of the same name is not allowed. |
| Create | Creates or overwrites a file in the specified path. |
| CreateText | Creates or opens a file for writing UTF-8 encoded text. |
| Decrypt | Decrypts a file that was encrypted by the current account using the Encrypt method. |
| Delete | Deletes the specified file. |
| Encrypt | Encrypts a file so that only the account used to encrypt the file can decrypt it. |
| Exists | Determines whether the specified file exists. |
| GetAccessControl | Gets a FileSecurity object that encapsulates the access control list (ACL) entries for a specified file. |
| Move | Moves a specified file to a new location, providing the option to specify a new file name. |
| Open | Opens a FileStream on the specified path with read/write access. |
| ReadAllBytes | Opens a binary file, reads the contents of the file into a byte array, and then closes the file. |
| ReadAllLines | Opens a text file, reads all lines of the file, and then closes the file. |
| ReadAllText | Opens a text file, reads all lines of the file, and then closes the file. |
| Replace | Replaces the contents of a specified file with the contents of another file, deleting the original file, and creating a backup of the replaced file. |
| WriteAllBytes | Creates a new file, writes the specified byte array to the file, and then closes the file. If the target file already exists, it is overwritten. |
| WriteAllLines | Creates a new file, writes a collection of strings to the file, and then closes the file. |
| WriteAllText | Creates a new file, writes the specified string to the file, and then closes the file. If the target file already exists, it is overwritten. |

## **Modifier Keywords**

| Modifier keywords |
| --- |
| abstract |
| async |
| const |
| event |
| extern |
| New |
| override |
| Partial |
| Readonly |
| Sealed |
| Static |
| Unsafe |
| Virtual |
| Volatile |

## **Access Modifier Keywords:**

| Access Modifiers | Usage |
| --- | --- |
| public | The Public modifier allows any part of the program in the same assembly or another assembly to access the type and its members. |
| private | The Private modifier restricts other parts of the program from accessing the type and its members. Only code in the same class or struct can access it. |
| internal | The Internal modifier allows other program code in the same assembly to access the type or its members. This is default access modifiers if no modifier is specified. |
| protected | The Protected modifier allows codes in the same class or a class that derives from that class to access the type or its members. |

Example: Explicitly Typed Variable

int i = 100;// explicitly typed

var j = 100; // implicitly typed

## **Const**

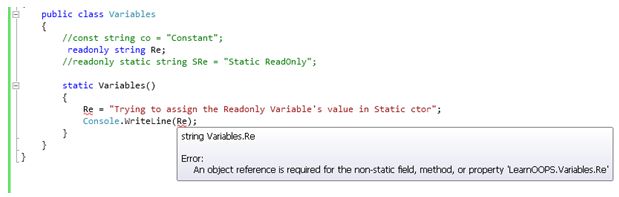
1. Const is nothing but "constant", a variable of which the value is constant but at compile time.
2. it's mandatory to assign a value to it.
3. By default a const is static and we cannot change the value of a const variable throughout the entire program.

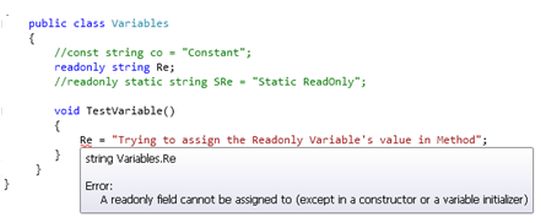


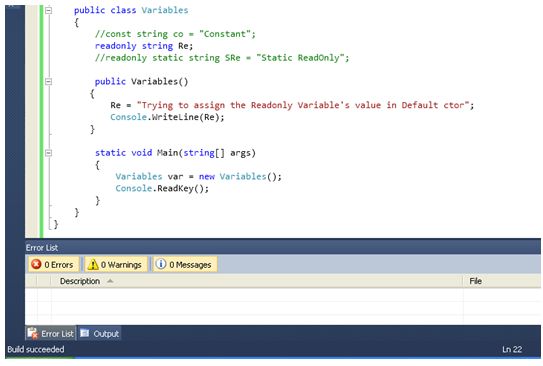


## **Readonly**

Readonly is the keyword whose value we can change during runtime or we can assign it at run time but only through the non-static constructor. Not even a method. Let's see:







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();

}

}

static void Main(string[] args)

{

A a1 = new A();

a1.show();

B b1 = new B();

b1.show();

A a2 = new B();

a2.show();

}

OUTPUT :

Hello: Base Class!

Hello: Derived Class!

Hello: Base Class!

class A

{

public void show()

{

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

// Console.ReadLine();

}

}

class B : A

{

Public void show()

{

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

// Console.ReadLine();

}

}

static void Main(string[] args)

{

A a1 = new A();

a1.show();

B b1 = new B();

b1.show();

A a2 = new B();

a2.show();

}

OUTPUT :

Hello: Base Class!

Hello: Derived Class!

Hello: Base Class!

class A

{

public virtual void show()

{

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

// Console.ReadLine();

}

}

class B : A

{

public override void show()

{

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

// Console.ReadLine();

}

}

static void Main(string[] args)

{

A a1 = new A();

a1.show();

B b1 = new B();

b1.show();

A a2 = new B();

a2.show();

}

OUTPUT :

Hello: Base Class!

Hello: Derived Class!

Hello: Derived Class!

# Concurrent Dictionary

*TryAdd(TKey, TValue)*

This method is used to add the item in ConcurrentDictionary:

1. **static** **void** Main(string[] args) {
2. ConcurrentDictionary < string, string > \_myConcuDict = **new** ConcurrentDictionary < string, string > ();
3. \_myConcuDict.TryAdd("1", "A");
4. \_myConcuDict.TryAdd("2", "B");
5. \_myConcuDict.TryAdd("3", "C");
6. }
7. **static** **void** Main(string[] args) {
8. ConcurrentDictionary < string, string > \_myConcuDict = **new** ConcurrentDictionary < string, string > ();
9. //returns true
10. bool r1 = \_myConcuDict.TryAdd("1", "A");
11. //returns true
12. bool r2 = \_myConcuDict.TryAdd("2", "B");
13. //returns false;
14. bool r3 = \_myConcuDict.TryAdd("1", "C");
15. }

**TryGetValue(TKey, TValue)**

In this method, we will get an item by the given key:

1. **static** **void** Main(string[] args) {
2. ConcurrentDictionary < string, string > \_myConcuDict = **new** ConcurrentDictionary < string, string > ();
3. bool firstItem = \_myConcuDict.TryAdd("1", "A");
4. bool secondItem = \_myConcuDict.TryAdd("2", "B");
5. string item1;
6. string item2;
7. //returns true
8. bool isItemExists1 = \_myConcuDict.TryGetValue("1", out item1);
9. Console.WriteLine(item1); // "A"
10. //returns false
11. bool isItemExists2 = \_myConcuDict.TryGetValue("3", out item2);
12. Console.WriteLine(item2); // it will print blank
13. }

**TryRemove(TKey, TValue)**

ConcurrentDictionary also provides the ability to remove a specific item by a given key.

1. **static** **void** Main(string[] args) {
2. ConcurrentDictionary < string, string > \_myConcuDict = **new** ConcurrentDictionary < string, string > ();
3. \_myConcuDict.TryAdd("1", "A");
4. \_myConcuDict.TryAdd("2", "B");
5. \_myConcuDict.TryAdd("3", "C");
6. \_myConcuDict.TryAdd("4", "D");
7. string itmRemove;
8. bool r1 = dictionary.TryRemove("2", out itmRemove); //Returns true
9. Console.WriteLine(removedItem); // "B"
10. }

**TryUpdate(TKey, TValue, TValue)**

This method is used to update the item if given a key whose value matches the condition.

1. **static** **void** Main(string[] args) {
2. ConcurrentDictionary < string, string > \_myConcuDict = **new** ConcurrentDictionary < string, string > ();
3. \_myConcuDict.TryAdd("1", "A");
4. \_myConcuDict.TryAdd("2", "B");
5. \_myConcuDict.TryAdd("3", "C");
6. \_myConcuDict.TryAdd("4", "D");
7. string newItem;
8. bool returnTrue = dictionary.TryUpdate("1", "P", "A"); //Returns true
9. dictionary.TryGetValue("1", out newItem);
10. Console.WriteLine(newValue); // "P"
11. bool returnsFalse = dictionary.TryUpdate("2", "Q", "CDD"); //Returns false
12. dictionary.TryGetValue("2", out newItem);
13. //it will return old value like B, Because of given condition is not matched.
14. Console.WriteLine(newValue); //B
15. }

**Clear()**

This method is used to remove all items from ConcurrentDictionary.

1. \_myConcuDict.Clear();

**ContainsKey(TKey)**

Sometimes, we are required to check if a particular key item exists or not in a given collection.

So here, ConcurrentDictionary also provides a method to check it.

1. //It will return true, because of givent key exist in ConcurrentDictionary
2. **bool** r1 = dictionary.ContainsKey("1");
3. //It will return false, because of givent key not exist in ConcurrentDictionary
4. **bool** r2 = dictionary.ContainsKey("5");

For that, ConcurrentDictionary provides various casting methods to do this.

* ToArray
* ToDictionary
* ToList

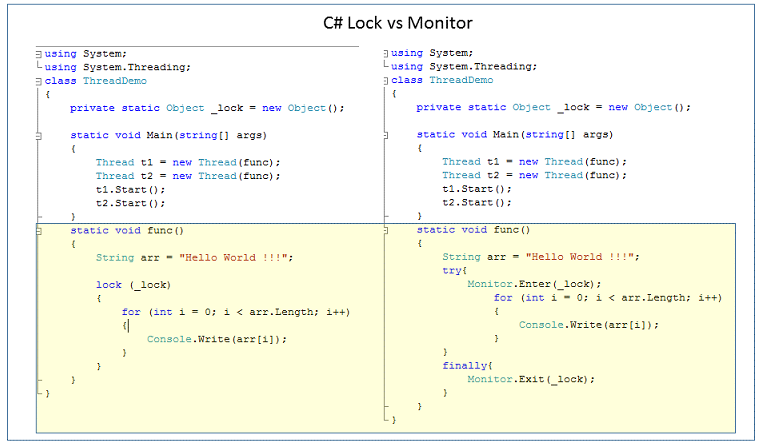
1. **static** **void** Main(string[] args) {
2. ConcurrentDictionary < string, string > \_myConcuDict = **new** ConcurrentDictionary < string, string > ();
3. \_myConcuDict.TryAdd("1", "A");
4. \_myConcuDict.TryAdd("2", "B");
5. \_myConcuDict.TryAdd("3", "C");
6. \_myConcuDict.TryAdd("4", "D");
7. // Convert in Array
8. KeyValuePair < string, string > [] keyValues = \_myConcuDict.ToArray();
9. // Convert in Dictionary
10. Dictionary < string, string > myDict = \_myConcuDict.ToDictionary(w => w.Key, m => m.Value);
11. // Convert in List
12. List < KeyValuePair < string, string >> lst = \_myConcuDict.ToList();
13. }

***Monitor and Lock***

Lock Vs Monitor in C# multithreading: Difference between monitor and lock in C# is that **lock** internally wraps the Enter and Exit methods in a **try**…**finally** block with exception handling. Whereas for [Monitor class in C#](https://www.interviewsansar.com/2016/06/19/what-is-monitor-class-object-in-csharp-threading/), we use try and finally block explicitly to release lock properly.

Lock = Monitor + try finally.

Below is the source code of lock vs Monitor class used in below C# thread example.



**Secondly,** **Monitor class has extra option that is signalling option**, that is used to communicate/signal to other threads using wait(), pulse() and pulseAll() methods.

C# Monitor.wait(): A thread wait for other threads to notify.

**Scheduled Task** -

1. When activity to be carried out on some fixed/predefined schedule.
2. It take less memory and resources of OS.
3. Not required installation.
4. It can have UI (eg. Send reminder mail to defaulters)

**Windows Service**-

1. When a continue monitoring is required.
2. It makes OS busy by consuming more.
3. Require install/uninstallation while changing version.
4. No UI at all (eg. Process a mail as soon as it arrives)

Windows Service Step :

1. Select Window Service Project
2. On Program.cs below code is available

static void Main()

{

ServiceBase[] ServicesToRun;

ServicesToRun = new ServiceBase[]

{

new Service1()

};

ServiceBase.Run(ServicesToRun);

}

1. Open ProjectFileName.cs (Desiger)

Below code

1. **protected** **override** **void** OnStart(**string**[] args)
2. {
3. LogService("Service is Started");
4. timeDelay.Enabled = **true**;
5. }
6. **protected** **override** **void** OnStop()
7. {
8. LogService("Service Stoped");
9. timeDelay.Enabled = **false**;
10. }

**Install Process**

1. Now return to the Scheduler.cs [Design] and right-click on the editor window then click "Add Installer".
2. Then you can see that there will be a new file called "ProjectInstaller.cs" as shown in the following.
3. Right-click on the "serviceInstaller1" and click "Properties".
4. Change the ServiceName to "Test Windows Service" (or your own name) and StartType to "Manual" (or you can choose "Automatic" if you need this service to be automatic).
5. Right-click the serviceProcessInstaller1, go to the properties window and change "Account" to "LocalSystem".
6. Right-click the serviceProcessInstaller1, go to the properties window and change "Account" to "LocalSystem".

## **Installing the Windows Service**

Go to "Start" >> "All Programs" >> "Microsoft Visual Studio 2012" >> "Visual Studio Tools" then click "Developer Command Prompt for VS2012".

Type the following command:

cd <physical location of your TestWindowService.exe file>

in my case it is :

cd C:\Sandbox\WindowServices\TestWindowService\TestWindowService\bin\Debug

Next type the following command:

InstallUtil.exe “TestWindowService.exe”

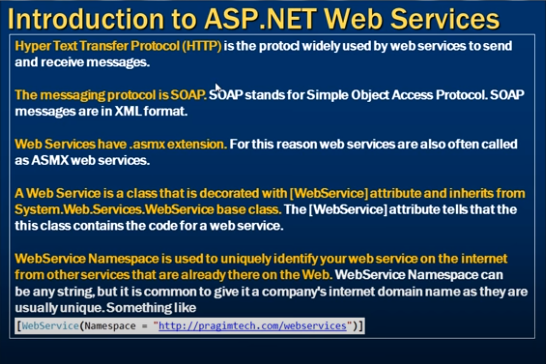
## **Uninstalling a Windows Service**

If you want to uninstall your service, fire the below command.

1. Syntax InstallUtil.exe -u + Your copied path + \your service name + .exe
2. Our path InstallUtil.exe -u C:\Users\Faisal-Pathan\source\repos\MyFirstService\MyFirstService\bin\Debug\MyFirstService.exe

<https://www.c-sharpcorner.com/UploadFile/naresh.avari/develop-and-install-a-windows-service-in-C-Sharp/>

***If we need to expose web service method then we need to decorator with [WEBMethod] attribute***

******

**What is a Web Service?**

A web service is a web-based functionality accessed using the protocols of the web to be used by the web applications and uses standard XML Messaging for communication.

XML is used to encode all communications to a Web service. For example, a client invokes a Web service by sending an XML message, then waits for a corresponding XML response. Because all communication is in XML, Web services are not tied to any one operating system or programming.

**What is the Serialization used for WebService?**

Web Services uses XML Serialization.

**What is WSDL?**

WSDL is an acronym for Web Services Description Language. It is a format to describe what a Web Service is going to offer, generally Operations, Definition and Service bindings.

**http://localhost:3501/ApplicationName?WSDL**

**What is the Web service protocol stack?**

The Web service protocol stack is an evolving set of protocols used to define, discover, and implement Web services. The core protocol stack consists of four layers:

**Service Transport:** This layer is responsible for transporting messages between applications. Currently, this includes HTTP, SMTP, FTP, and newer protocols, such as Blocks Extensible Exchange Protocol (BEEP).

**What is the extension for Web Service?**

.asmx

**What is the namespace for Web Service?**

System.Web.Services

How to Consume web service in web application

* 1. Add service reference in project
  2. On button click event

ServiceName.ServiceNameSoapClient client=new ServiceName.ServiceNameSoapClient()

Client.FunctionName(Parameter);

How Proxy class generate :Visual Studio generate proxy class using WSDL(web service description language)document of web service.

WSDL contain following point.

* 1. All method expose by web service
  2. The parameter and type
  3. The return type of this method

Proxy class will serialize the parameter,prepare a SOPA request message and sent it to the web service.

Web service execute method and retruns a SOAP response message to the proxy.Proxy class deSerialize the SOAP response message and hand to client application

What attribute are use in web service

[WebMethod(EnableSession=true)]

Description : use to specify a description for the web service method

[WebMethod(EnableSession=true,Description=”Add two method”)]

[WebMethod(EnableSession=true,Description=”Add two method”,CacheDuration=20)]

If function overload in web service

[WebMethod(MessageName=”Add2Number”)]

And set below as None

[WebServiceBinding(ConformsTo = WsiProfiles.None)]

## SOAP vs. REST

Let' have a quick overview of SOAP and REST before we do a deep dive into the key differences between them.

**SOAP** – SOAP is a protocol which was designed before REST and came into the picture. The main idea behind designing SOAP was to ensure that programs built on different platforms and programming languages could exchange data in an easy manner.

**REST** – This was designed specifically for working with components such as media components, files, or even objects on a particular hardware device. Any web service that is defined on the principles of REST can be called a RestFul web service. A Restful service would use the normal HTTP verbs of GET, POST, PUT and DELETE for working with the required components.

Web API :

Note : If package manger is not working then we have execute script on power shell

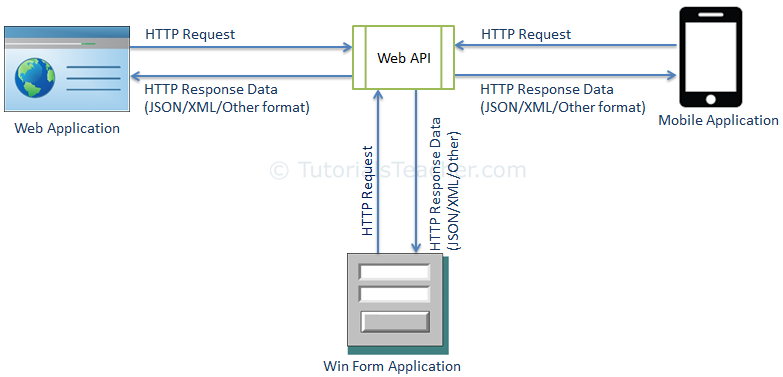
Setp 1 : open windows PowerShell in administrator

Setp 2 : execute comment SET-ExecutionPolicy AllSigned

Setp 3 : Open package manger console

## **ASP.NET Web API**

The ASP.NET Web API is an extensible framework for building HTTP based services that can be accessed in different applications on different platforms such as web, windows, mobile etc.



## **ASP.NET Web API Characteristics**

1. ASP.NET Web API is an ideal platform for building RESTful services.
2. ASP.NET Web API is built on top of ASP.NET and supports ASP.NET request/response pipeline
3. ASP.NET Web API maps HTTP verbs to method names.
4. ASP.NET Web API supports different formats of response data. Built-in support for JSON, XML, BSON format.

| **Web API Version** | **Supported .NET Framework** | **Coincides with** | **Supported in** |
| --- | --- | --- | --- |
| **Web API 1.0** | **.NET Framework 4.0** | **ASP.NET MVC 4** | **VS 2010** |
| **Web API 2 - Current** | **.NET Framework 4.5** | **ASP.NET MVC 5** | **VS 2012, 2013** |

| **Web API** | **WCF** |
| --- | --- |
| **Open source and ships with .NET framework.** | **Ships with .NET framework** |
| **Supports only HTTP protocol.** | **Supports HTTP, TCP, UDP and custom transport protocol.** |
| **Maps http verbs to methods** | **Uses attributes based programming model.** |
| **Uses routing and controller concept similar to ASP.NET MVC.** | **Uses Service, Operation and Data contracts.** |

**When to choose WCF?**

* Choose WCF if you use .NET Framework 3.5. Web API does not support .NET 3.5 or below.
* Choose WCF if your service needs to support multiple protocols such as HTTP, TCP, Named pipe.

**When to choose ASP.NET Web API?**

* Choose Web API if you are using .NET framework 4.0 or above.
* Choose Web API if you want to build a service that supports only HTTP protocol.
* Choose Web API to build RESTful HTTP based services.

**REST Constraint**

* GET
* POST
* PUT
* DELETE

Media Type formatter

config.Formatters.JsonFormatter.SerializerSettings.Formatting = Newtonsoft.Json.Formatting.Indented;

config.Formatters.JsonFormatter.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();

public JsonMediaTypeFormatter JsonFormatter { get; }

public class JsonMediaTypeFormatter : MediaTypeFormatter

{

public JsonSerializerSettings SerializerSettings { get; set; }

}

**How to return only JSON from ASP.Net WEB Api service.**

config.Formatters.Remove(config.Formatters.XMLFormatter);

**How to return only XML from ASP.Net WEB Api service.**

config.Formatters.Remove(config.Formatters.JSONFormatter);

* **Use [FormBody] attribute to force web api to get simple types from the request body.**
* **Use [FormBody] attribute to force web api to get complex types from URI.**

# How to read data from XML String and insert in to table in SQL Server

Create PROCEDURE [dbo].[SP\_Insert\_MultipleRows] (  
 @xmlData XML ,  
 @retValue varchar(20) OUTPUT  
)

AS  
BEGIN  
SET @retValue='Failed';

INSERT INTO  [Employee](  
[id],  
[firstName],  
[lastName],  
[company]  
)

SELECT  
COALESCE([Table].[Column].value('ID[1]', 'int'),0) as 'ID',  
[Table].[Column].value('FirstName [1]', 'varchar(20)') as ' FirstName ',  
[Table].[Column].value(' LastName[1]', 'varchar(20)') as ' LastName',  
[Table].[Column].value(' Company [1]', 'varchar(50)') as ' Company'

 FROM @xmlData.nodes('/ Customers / customer') as [Table]([Column])  
IF(@@ROWCOUNT > 0 )  
  SET @retValue='SUCCESS';  
  
  
END

Execute

Declare @retValue1 varchar(50);  
Declare @XmlStr XML;  
SET @XmlStr='<Customers>  
 <customer>  
    <ID>111589</ID>  
    <FirstName>name1</FirstName>  
    <LastName>Lname1</LastName>  
    <Company>ABC</Company>  
  </customer>  
  <customer>  
    <ID>12345</ID>  
    <FirstName>name2</FirstName>  
    <LastName>Lname2</LastName>  
    <Company>ABC</Company>  
  </customer>  
  <customer>  
    <ID>14567</ID>  
    <FirstName>name3</FirstName>  
    <LastName>Lname3</LastName>  
    <Company>DEF</Company>  
  </customer>  
</Customers>';

EXEC [SP\_Insert\_MultipleRows] @xmlData=@XmlStr,@retValue=@retValue1 OUTPUT  
print @retValue1



**Stored Procedure for reading data from XML String.  
  
XML String**  
  
<DataSet>  
  <tblEmp>  
    <name>Vishal</name>  
    <designation>Developer</designation>l  
  </tblEmp>  
  <tblEmp>  
    <name>Jibin</name>  
    <designation>System Analyst</designation>l  
  </tblEmp>  
</DataSet>  
  
**Stored Procedure to read XML string**

CREATE PROC [dbo].[USP\_READXMLString]

(

   @XMLDOC2    XML

)

AS

   BEGIN

         SET NOCOUNT ON

         DECLARE @HANDLE INT

         EXEC SP\_XML\_PREPAREDOCUMENT  @HANDLE OUTPUT,@XMLDOC2

         SELECT \* FROM OPENXML(@HANDLE, '/DataSet/tblEmp', 2)

         WITH (name VARCHAR(50),designation VARCHAR(50))

END

## **Types of DML Triggers**

Insert Trigger

1. CREATE TRIGGER trgAfterInsert on Employee\_Demo
2. FOR INSERT
3. AS
4. declare @empid int,
   1. @empname varchar(55),
   2. @empsal decimal(10,2),
   3. @audit\_action varchar(100);
5. select @empid=i.Emp\_ID from inserted i;
6. select @empname=i.Emp\_Name from inserted i;
7. select @empsal=i.Emp\_Sal from inserted i;
8. set @audit\_action='Inserted Record -- After Insert Trigger.';

insert into Employee\_Demo\_Audit(Emp\_ID,Emp\_Name,Emp\_Sal,Audit\_Action,Audit\_Timestamp)

1. values (@empid,@empname,@empsal,@audit\_action,getdate());
2. PRINT 'AFTER INSERT trigger fired.'

Update Trigger

1. CREATE TRIGGER trgAfterUpdate ON dbo.Employee\_Demo
2. FOR UPDATE
3. AS
4. declare @empid int, @empname varchar(55), @empsal decimal(10,2), @audit\_action varchar(100);
5. select @empid=i.Emp\_ID from inserted i;
6. select @empname=i.Emp\_Name from inserted i;
7. select @empsal=i.Emp\_Sal from inserted i; if update(Emp\_Name)
8. set @audit\_action='Update Record --- After Update Trigger.';
9. if update (Emp\_Sal)
10. set @audit\_action='Update Record --- After Update Trigger.';
11. insert intoEmployee\_Demo\_Audit(Emp\_ID,Emp\_Name,Emp\_Sal,Audit\_Action,Audit\_Timestamp)
12. values (@empid,@empname,@empsal,@audit\_action,getdate());
13. PRINT 'AFTER UPDATE trigger fired.'
14. --Output will be

Delete Trigger

1. -- Create trigger on table Employee\_Demo for Delete statement
2. CREATE TRIGGER trgAfterDelete ON dbo.Employee\_Demo
3. FOR DELETE
4. AS
5. declare @empid int, @empname varchar(55), @empsal decimal(10,2), @audit\_action varchar(100); select @empid=d.Emp\_ID FROM deleted d;
6. select @empname=d.Emp\_Name from deleted d;
7. select @empsal=d.Emp\_Sal from deleted d;
8. select @audit\_action='Deleted -- After Delete Trigger.';
9. insert into Employee\_Demo\_Audit (Emp\_ID,Emp\_Name,Emp\_Sal,Audit\_Action,Audit\_Timestamp)
10. values (@empid,@empname,@empsal,@audit\_action,getdate());
11. PRINT 'AFTER DELETE TRIGGER fired.'
12. --Output will be

WEB API

Web API configuration in **WebApiConfig.cs** file and WebApiConfig registration in Application\_Start (Global.aspx.cs)

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

GlobalConfiguration.Configure(WebApiConfig.Register);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

}

Below code in WebApiConfig.cs

public static void Register(HttpConfiguration config)

{

// Web API configuration and services

// Configure Web API to use only bearer token authentication.

config.SuppressDefaultHostAuthentication();

config.Filters.Add(new HostAuthenticationFilter(OAuthDefaults.AuthenticationType));

// Web API routes

config.MapHttpAttributeRoutes();

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

//config.Formatters.Add(new CustomJsonFormatter());

//Start Retrun Json

config.Formatters.Remove(config.Formatters.XmlFormatter);

config.Formatters.JsonFormatter.SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/html"));

//End

//Start Retrun Xml

// config.Formatters.Remove(config.Formatters.JsonFormatter);

//End

//config.Formatters.Remove(config.Formatters.JsonFormatter);

//config.Formatters.XmlFormatter.UseXmlSerializer = true;

// var jsonpFormatter = new JsonMediaTypeFormatter(config.Formatters.JsonFormatter);

// config.Formatters.Insert(0, jsonpFormatter);

EnableCorsAttribute cor = new EnableCorsAttribute("\*", "\*", "\*");

config.EnableCors(cor);

}

[FormBody] - Simple Type

[FormUri] - Complex Type

public HttpResponseMessage Put([FromBody] int id, [FromUri] Employee employee)

{

}

HTP Status Code :

## 204 No Content

* The windows Authentication provider lets you authenticates users based on their windows accounts. This provider uses IIS to perform the authentication and then passes the authenticated identity to your code. This is the default provided for ASP.net.
* The passport authentication provider uses Microsoft's passport service to authenticate users.
* The forms authentication provider uses custom HTML forms to collect authentication information and lets you use your own logic to authenticate users. The user's credentials are stored in a cookie for use during the session
* <authentication mode="windows">
* authentication mode="passport">
* <authentication mode="forms">

***Public and Shared Assembly***

**Creating A shared assembly**

**Step 1:** Generate a key file. Open a VS command prompt. Go into your folder and generate a key file as:

1. <drive>:\<folder> sn -k key.snk

**Step 2:** create a project and associate a key file to it before compilation so that the generated assembly will be strongly named.

Open a new project of type class library and name its Assembly; under class1 write the following:

1. **public** **string** sayhello()
2. {
3. **return** "hello from shared assembly";
4. }

To associate a key file we generated with the project, open the project properties and select the "signing" tab on the LHS which displays a CheckBox as "sign the assembly" select it that displays ComboBox below it from it a select browse and select key.snk from its physical location then compile the project using build which will generate assembly Assembly.dll that is strongly named.

**Step 3:** copy the assembly into GAC

.Net provides a command-line utility to be used as shown in the following:

1. Gacutil -I | -u <assembly name> I:install u:uninstall

Open a VS command prompt; go to the location where the Assembly.dll is present and write the following:

1. <drive>:\<folder>\sAssembly\ sAssembly\\bin\Debug>gacutil -I Assembly.dll

**Step 4:** Testing

Open a new project add a reference to Assembly.dll and write the following code for the button click event.

1. sAssembly.Classs1 obj=**new** sAssembly.Class1();
2. Messagebox.Show(obj.sayhello());

Run the project and verify under the bin/debug folder of the current project where we will not find a copy of the Assembly.dll as it is a shared assembly.

**5) Define what is Where clause and Let clause?**

* **Where clause**: It allows adding some conditional filters to the query.
* **Let clause**: It allows defining a variable and assigning it a value calculated from the data values.

**6) Explain why SELECT clause comes after FROM clause in LINQ?**

With other programming language and C#, LINQ is used, it requires all the variables to be declared first. “FROM” clause of LINQ query defines the range or conditions to select records. So, FROM clause must appear before SELECT in LINQ.

**14) Mention what is the role of DataContext classes in LINQ?**

DataContext class acts as a bridge between [SQL Server](https://www.guru99.com/sql-server-questions.html) database and the LINQ to SQL. For accessing the database and also for changing the data in the database, it contains connections string and the functions.

## **C# Extension Method**

1. **using** System;
2. **using** System.Text;
4. **namespace** ClassLibExtMethod
5. {
6. **public** **class** Class1
7. {
8. **public** **string** Display()
9. {
10. **return** ("I m in Display");
11. }
13. **public** **string** Print()
14. {
15. **return** ("I m in Print");
16. }
17. }
18. }
19. **using** System;
20. **using** System.Text;
21. **using** ClassLibExtMethod;
23. **namespace** ExtensionMethod1
24. {
25. **public** **static** **class** XX
26. {
27. **public** **static** **void** NewMethod(**this** Class1 ob)
28. {
29. Console.WriteLine("Hello I m extended method");
30. }
31. }
33. **class** Program
34. {
35. **static** **void** Main(**string**[] args)
36. {
37. Class1 ob = **new** Class1();
38. ob.Display();
39. ob.Print();
40. ob.NewMethod();
41. Console.ReadKey();
42. }
43. }
44. }

## **Benefits of extension methods**

* Extension methods allow existing classes to be extended without relying on inheritance or having to change the class's source code.
* If the class is sealed than there in no concept of extending its functionality. For this a new concept is introduced, in other words extension methods.
* This feature is important for all developers, especially if you would like to use the dynamism of the C# enhancements in your class's design.

Lazy loading and Eager loading and Explicit loading

**Eager Loading**

For example, you have a User table and a UserDetails table (related entity to User table), then you will write the code given below. Here, we are loading the user with the Id equal to userId along with the user details.

1. User usr = dbContext.Users.Include(a => a.UserDetails).FirstOrDefault(a => a.UserId == userId);

If you have multiple level of child entities, then you can load, using the query given below.

1. User usr = dbContext.Users.Include(a => a.UserDetails.Select(ud => ud.Address)).FirstOrDefault(a => a.UserId == userId);

**Lazy Loading**  
It is the default behavior of an Entity Framework, where a child entity is loaded only when it is accessed for the first time. It simply delays the loading of the related data, until you ask for it.

For example, when we run the query given below, UserDetails table will not be loaded along with the User table.

1. User usr = dbContext.Users.FirstOrDefault(a => a.UserId == userId);

It will only be loaded when you explicitly call for it, as shown below.

1. UserDeatils ud = usr.UserDetails; // UserDetails are loaded here

**Explicit Loading**  
There are options to disable Lazy Loading in an Entity Framework. After turning Lazy Loading off, you can still load the entities by explicitly calling the Load method for the related entities. There are two ways to use Load method Reference (to load single navigation property) and Collection (to load collections), as shown below.

1. User usr = dbContext.Users.FirstOrDefault(a => a.UserId == userId);
2. dbContext.Entry(usr).Reference(usr => usr.UserDetails).Load();

With Eager Loading, all the data is retrieved in a single query, which can then be cached to improve the Application performance. With Eager Loading, we are trading memory consumption for the database round trips.

With Lazy Loading, we only retrieve just the amount of data, which we need in a single query. When we need more data related to the initial data, additional queries are issued to the database. This means there are several round trips between the Application Server and the database Server. In general, these database round trips are very often the major performance bottleneck in most Applications. Lesser the round trips, better will be the performance.

SOLID Principle

SOLID stands for Single Responsibility Principle (SRP), Open closed Principle (OSP), Liskov substitution Principle (LSP), Interface Segregation Principle (ISP), and Dependency Inversion Principle (DIP)

### 1. About SOLID

SOLID is basically 5 principles, which will help to create a good software architecture. You can see that all design patterns are based on these principles. SOLID is basically an acronym of the following:

* **S**is single responsibility principle (SRP)
* **O**stands for open closed principle (OCP)
* **L** Liskov substitution principle (LSP)
* **I** interface segregation principle (ISP)
* **D** Dependency injection principle (DIP)

I believe that with pictures, with examples, an article will be more approachable and understandable.

#### 1.1 Single responsibility principle (SRP)

A class should take one responsibility and there should be one reason to change that class. Now what does that mean? I want to share one picture to give a clear idea about this.

Now see this tool is a combination of so many different tools like knife, nail cutter, screw driver, etc. So will you want to buy this tool? I don’t think so. Because there is a problem with this tool, if you want to add any other tool to it, then you need to change the base and that is not good. This is a bad architecture to introduce into any system. It will be better if nail cutter can only be used to cut the nail or knife can only be used to cut vegetables.

Now I want to give one C# example on this principle:

namespace SRP

{

public class Employee

{

public int Employee\_Id { get; set; }

public string Employee\_Name { get; set; }

/// *<summary>*

/// *This method used to insert into employee table*

/// *</summary>*

/// *<param name="em">Employee object</param>*

/// *<returns>Successfully inserted or not</returns>*

public bool InsertIntoEmployeeTable(Employee em)

{

*// Insert into employee table.*

return true;

}

/// *<summary>*

/// *Method to generate report*

/// *</summary>*

/// *<param name="em"></param>*

public void GenerateReport(Employee em)

{

*// Report generation with employee data using crystal report.*

}

}

}

‘Employee’ class is taking 2 responsibilities, one is to take responsibility of employee database operation and another one is to generate employee report. Employee class should not take the report generation responsibility because suppose some days after your customer asked you to give a facility to generate the report in Excel or any other reporting format, then this class will need to be changed and that is not good.

So according to SRP, one class should take one responsibility so we should write one different class for report generation, so that any change in report generation should not affect the ‘Employee’ class.

public class ReportGeneration

{

/// *<summary>*

/// *Method to generate report*

/// *</summary>*

/// *<param name="em"></param>*

public void GenerateReport(Employee em)

{

*// Report reneration with employee data.*

}

}

#### 2.2 Open closed principle (OCP)

Now take the same ‘ReportGeneration’ class as an example of this principle. Can you guess what is the problem with the below class!!

public class ReportGeneration

{

/// *<summary>*

/// *Report type*

/// *</summary>*

public string ReportType { get; set; }

/// *<summary>*

/// *Method to generate report*

/// *</summary>*

/// *<param name="em"></param>*

public void GenerateReport(Employee em)

{

if (ReportType == "CRS")

{

*// Report generation with employee data in Crystal Report.*

}

if (ReportType == "PDF")

{

*// Report generation with employee data in PDF.*

}

}

}

Brilliant!! Yes you are right, too much ‘If’ clauses are there and if we want to introduce another new report type like ‘Excel’, then you need to write another ‘if’. This class should be open for extension but closed for modification. But how to do that!!

public class IReportGeneration

{

/// *<summary>*

/// *Method to generate report*

/// *</summary>*

/// *<param name="em"></param>*

public virtual void GenerateReport(Employee em)

{

*// From base*

}

}

/// *<summary>*

/// *Class to generate Crystal report*

/// *</summary>*

public class CrystalReportGeneraion : IReportGeneration

{

public override void GenerateReport(Employee em)

{

*// Generate crystal report.*

}

}

/// *<summary>*

/// *Class to generate PDF report*

/// *</summary>*

public class PDFReportGeneraion : IReportGeneration

{

public override void GenerateReport(Employee em)

{

*// Generate PDF report.*

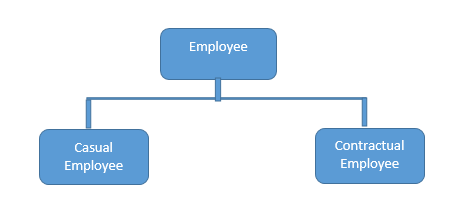
}

}

So if you want to introduce a new report type, then just inherit from IReportGeneration. So IReportGeneration is open for extension but closed for modification.

#### 2.3 Liskov substitution principle (LSP)

This principle is simple but very important to understand. Child class should not break parent class’s type definition and behavior. Now what is the meaning of this!! Ok let me take the same employee example to make you understand this principle. Check the below picture. Employee is a parent class and Casual and Contractual employee are the child classes, inhering from employee class.



Now see the below code:

public abstract class Employee

{

public virtual string GetProjectDetails(int employeeId)

{

return "Base Project";

}

public virtual string GetEmployeeDetails(int employeeId)

{

return "Base Employee";

}

}

public class CasualEmployee : Employee

{

public override string GetProjectDetails(int employeeId)

{

return "Child Project";

}

*// May be for contractual employee we do not need to store the details into database.*

public override string GetEmployeeDetails(int employeeId)

{

return "Child Employee";

}

}

public class ContractualEmployee : Employee

{

public override string GetProjectDetails(int employeeId)

{

return "Child Project";

}

*// May be for contractual employee we do not need to store the details into database.*

public override string GetEmployeeDetails(int employeeId)

{

throw new NotImplementedException();

}

}

Up to this is fine right? Now, check the below code and it will violate the LSP principle.

List<Employee> employeeList = new List<Employee>();

employeeList.Add(new ContractualEmployee());

employeeList.Add(new CasualEmployee());

foreach (Employee e in employeeList)

{

e.GetEmployeeDetails(1245);

}

Now I guess you got the problem. Yes right, for contractual employee, you will get not implemented exception and that is violating LSP. Then what is the solution? Break the whole thing in 2 different interfaces, 1. IProject 2. IEmployee and implement according to employee type.

public interface IEmployee

{

string GetEmployeeDetails(int employeeId);

}

public interface IProject

{

string GetProjectDetails(int employeeId);

}

Now, contractual employee will implement IEmployee not IProject. This will maintain this principle.

#### 2.4 Interface segregation principle (ISP)

This principle states that any client should not be forced to use an interface which is irrelevant to it. Now what does this mean, suppose there is one database for storing data of all types of employees (i.e. Permanent, non-permanent), now what will be the best approach for our interface?

public interface IEmployee

{

bool AddEmployeeDetails();

}

And all types of employee class will inherit this interface for saving data. This is fine right? Now suppose that company one day told to you that they want to read only data of permanent employees. What you will do, just add one method to this interface?

public interface IEmployeeDatabase

{

bool AddEmployeeDetails();

bool ShowEmployeeDetails(int employeeId);

}

But now we are breaking something. We are forcing non-permanent employee class to show their details from database. So, the solution is to give this responsibility to another interface.

public interface IAddOperation

{

bool AddEmployeeDetails();

}

public interface IGetOperation

{

bool ShowEmployeeDetails(int employeeId);

}

And non-permanent employee will implement only IAddOperation and permanent employee will implement both the interface.

#### 2.5Dependency inversion principle (DIP)

This principle tells you not to write any tightly coupled code because that is a nightmare to maintain when the application is growing bigger and bigger. If a class depends on another class, then we need to change one class if something changes in that dependent class. We should always try to write loosely coupled class.

Suppose there is one notification system after saving some details into database.

public class Email

{

public void SendEmail()

{

*// code to send mail*

}

}

public class Notification

{

private Email \_email;

public Notification()

{

\_email = new Email();

}

public void PromotionalNotification()

{

\_email.SendEmail();

}

}

Now Notification class totally depends on Email class, because it only sends one type of notification. If we want to introduce any other like SMS then? We need to change the notification system also. And this is called tightly coupled. What can we do to make it loosely coupled? Ok, check the following implementation.

public interface IMessenger

{

void SendMessage();

}

public class Email : IMessenger

{

public void SendMessage()

{

*// code to send email*

}

}

public class SMS : IMessenger

{

public void SendMessage()

{

*// code to send SMS*

}

}

public class Notification

{

private IMessenger \_iMessenger;

public Notification()

{

\_ iMessenger = new Email();

}

public void DoNotify()

{

\_ iMessenger.SendMessage();

}

}

Still Notification class depends on Email class. Now, we can use dependency injection so that we can make it loosely coupled. There are 3 types to DI, Constructor injection, Property injection and method injection.

##### Constructor Injection

public class Notification

{

private IMessenger \_iMessenger;

public Notification(Imessenger pMessenger)

{

\_ iMessenger = pMessenger;

}

public void DoNotify()

{

\_ iMessenger.SendMessage();

}

}

##### Property Injection

public class Notification

{

private IMessenger \_iMessenger;

public Notification()

{

}

public IMessenger MessageService

{

private get;

set

{

\_ iMessenger = value;

}

}

public void DoNotify()

{

\_ iMessenger.SendMessage();

}

}

##### Method Injection

public class Notification

{

public void DoNotify(IMessenger pMessenger)

{

pMessenger.SendMessage();

}

}

So, SOLID principle will help us to write loosely coupled code which is highly maintainable and less error prone.

**is OPERATOR** The is operator in C# is used to check the object type and it returns a bool value: true if the object is the same type and false if not. or also The “is” operator is used to check whether the run-time type of an object is compatible with a given type or not. For null objects, it returns false e.g

if(obj is AnimalObject)

{

//Then Work

}

**as OPERATOR**

The as operator does the same job of is operator but the difference is instead of bool, it returns the object if they are compatible to that type, else it returns null.In otherwords, The ‘as‘ operator is used to perform conversions between compatible types.

e.g

Type obj = Object as Type;

* The **is** operator is used to check if the run-time type of an object is compatible with the given type or not whereas **as** operator is used to perform conversion between compatible reference types or Nullable types.
* The **is** operator is of boolean type whereas **as** operator is not of boolean type.

@using(Ajax.BeginForm("Action","ControllerName",new AjaxOptions { InsertionMode = InsertionMode.Replace,HttpMethod="Post",UpdateTargetId="Method",OnSuccess="Method"},new { id="Control" }))

{

}

HTML Control bind with model

@Html.ValidationSummary(true, "", new { @class = "text-danger" })

<div class="form-group">

@Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })

<div class="col-md-10">

@Html.TextBoxFor(m => m.Email, new { @class = "form-control" })

@Html.ValidationMessageFor(m => m.Email, "", new { @class = "text-danger" })

</div>

</div>

HTML.DropDownFor

public class MyEggs

{

public Dictionary<int, string> Egg { get; set; }

public MyEggs()

{

Egg = new Dictionary<int, string>()

{

{ 0, "No Preference"},

{ 1, "I hate eggs"},

{ 2, "Over Easy"},

{ 3, "Sunny Side Up"},

{ 4, "Scrambled"},

{ 5, "Hard Boiled"},

{ 6, "Eggs Benedict"}

};

}

@Html.DropDownListFor(m => m.Egg.Keys, new SelectList( Model.Egg , "Key" , "Value"))

HTML.CheckBoxFor

public ActionResult YourActionMethod()

{

CareerForm model = new CareerForm();

model.EmploymentType = new List<CheckBox>

{

new CheckBox { Text = "Fulltime" },

new CheckBox { Text = "Partly" },

new CheckBox { Text = "Contract" }

};

return View(model);

}

@for (int i = 0; i < Model.EmploymentType.Count; i++)

{

@Html.HiddenFor(m => m.EmploymentType[i].Text)

@Html.CheckBoxFor(m => m.EmploymentType[i].Checked, new { id = "YourId" })

}

public SelectList GetAllCountryList()

{

List<Country> objcountry = new List<Country>();

objcountry.Add(new Country { Id = 1, CountryName = "India" });

objcountry.Add(new Country { Id = 2, CountryName = "USA" });

objcountry.Add(new Country { Id = 3, CountryName = "Pakistan" });

objcountry.Add(new Country { Id = 4, CountryName = "Nepal" });

SelectList objselectlist = new SelectList(objcountry, "Id", "CountryName");

return objselectlist;

}

@Html.ListBoxFor(m => m.SelectedCountry, new SelectList(Model.CountryList, "Value", "Text", Model.CountryList.SelectedValue), new { @Id = "lstcountry", @style = "width:200px;height:60px;" })

# ValidateInput Attribute to Prevent CSS Attack in MVC

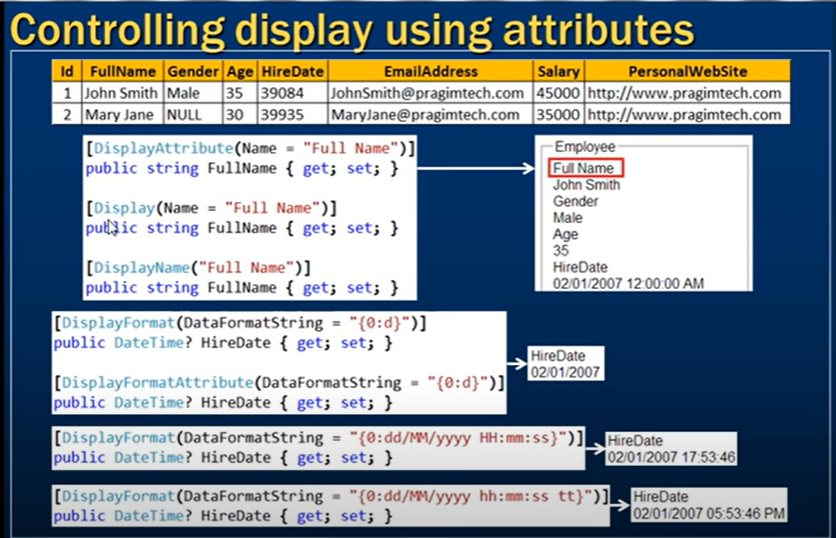
1. **public** **class** personController : Controller
2. {
3. **public** ActionResult Index()
4. {
5. **return** View();
6. }
7. [ValidateInput(**true**)]
8. **public** **void** GetPerson(person p)
9. {
10. }
11. }

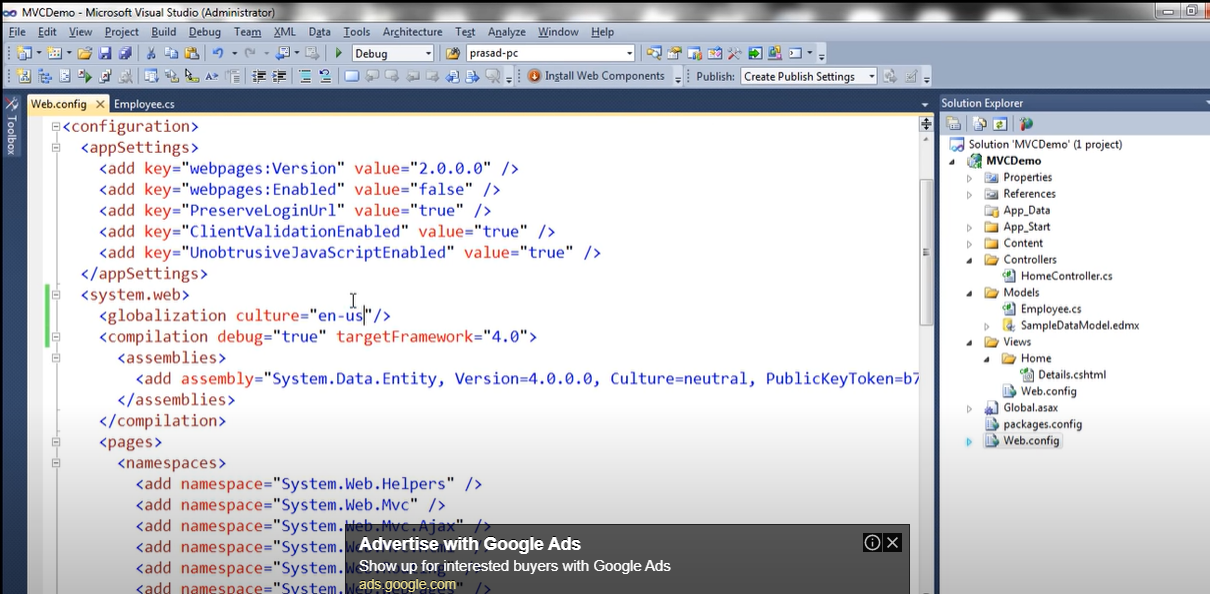
Or we can use the ValidateInput() attribute over the controller.

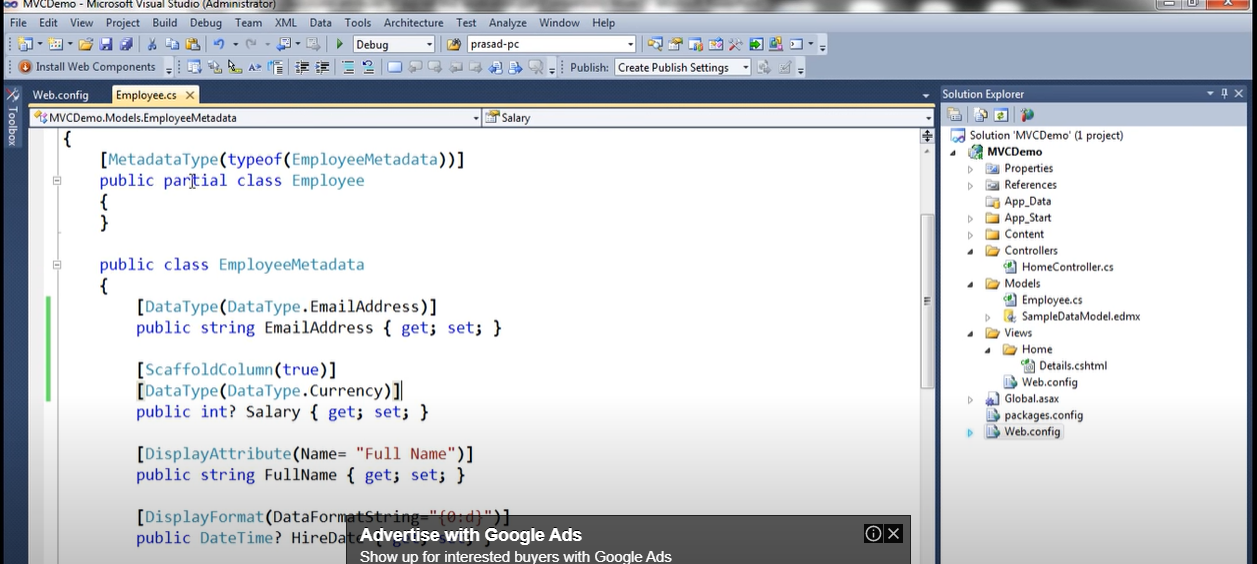
1. [ValidateInput(**true**)]
2. **public** **class** personController : Controller
3. {
4. }

And if we want to allow a HTML element through form input, we can just set the true parameter to false. Then it will allow acceptance of a HTML element as input.  
  
Or  
  
We can use the AllowHtml() attribute of the model property, as in the following to allow a HTML element to a certain property only.

1. **public** **class** person
2. {
3. [AllowHtml]
4. **public** **string** personDescription { **get**; **set**; }
5. }

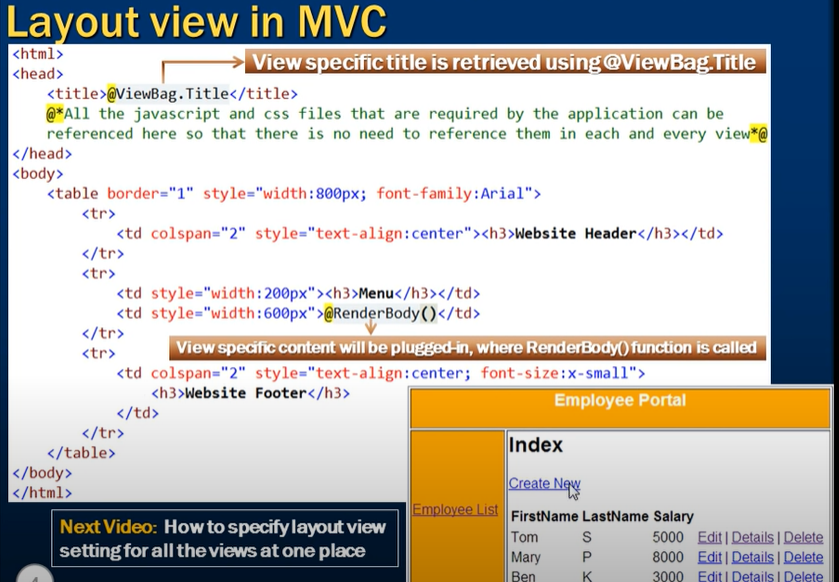


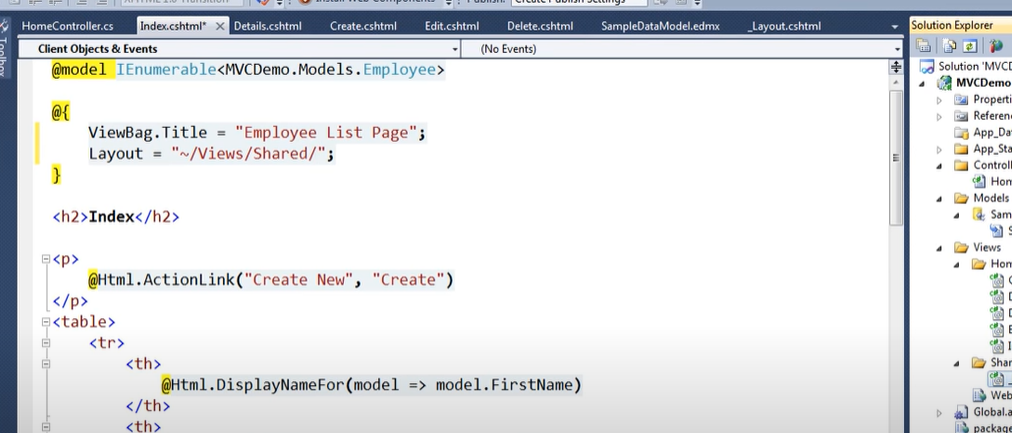




1) In shared folder we have to create layout file

\_Layout.cshtml





## Types of Filters

There are five types of Filters in ASP.NET MVC 5.0:

* Authentication Filters (available in ASP.NET MVC5)
* Authorization Filters
* Action Filters
* Result Filters
* Exception Filters

## Examples of ASP.NET MVC Filters

### Authentication Filters

Introduced with ASP.NET MVC5, an authentication filter is a component that authenticates an HTTP request. Both Web API 2 and MVC 5 support authentication filters. The IAuthenticationFilter interface is used to create CustomAuthentication filter. Refer the following code snippet.

1. namespace MVCFilters
2. {
3. public interface IAuthenticationFilter
4. {
5. void OnAuthentication(AuthenticationContext
6. AuthenticationFilterContext);
8. void OnAuthenticationChallenge(AuthenticationChallengeContext
9. filterContext);
10. }
12. public class CustomAuthenticationAttribute :
13. ActionFilterAttribute, IAuthenticationFilter
14. {
15. public void OnAuthentication(AuthenticationContext
16. filterContext)
17. {
18. // Custom logic for user authentication
19. }
21. public void OnAuthenticationChallenge
22. (AuthenticationChallengeContext filterContext)
23. {
24. // Any Additional tasks on the request
26. }
27. }
28. }

### Authorization Filters

Authorization filters are required to provide an authorization level in the application. The ASP.NET MVC Authorize filter attribute implements the IAuthorizationFilter interface. Also, the AuthorizeAttribute class provides the following methods to override in the CustomAuthorize attribute class.

The following snippet shows how the Interface and the class operate.

1. namespace MVCFilters
2. {
3. public interface IAuthorizationFilter
4. {
5. void OnAuthorization(AuthorizationContext filterCOntext);
6. }
8. public class AuthorizeAttribute : FilterAttribute,
9. IAuthorizationFilter
10. {
11. protected virtual bool AuthorizeCore(HttpContextBase
12. httpContext);
13. protected virtual void HandleUnauthorizedRequest
14. (AuthorizationContext filterContext);
15. public virtual void OnAuthorization(AuthorizationContext
16. filterContext);
17. protected virtual HttpValidationStatus OnCacheAuthorization
18. (HttpContextBase httpContext);
19. }
21. }

### Action Filters

Action filters are used to implement the logic that get executed before or after a controller action executes. The IActionFilter interface is used to create an Action Filter. That interface provides two methods: OnActionExecuting and OnActionExecuted. These methods will be executed before or after an action is executed, respectively.

1. namespace MVCFilters
2. {
3. public interface IActionFilter
4. {
5. void OnActionExecuting(ActionExecutingContext filterContext);
6. void OnActionExecuted(ActionExecutedContext filterContext);
7. }
8. }

ASP.NET MVC provides the following action filters:

* **Output Cache**: Caches the output of a controller action for a specified amount of time. One of the important features of output cache is that it supports an action filter that can be called all the time, even when the action is marked with a cache attribute.
  1. namespace MVCFiltere
  2. {
  3. [LogThis]
  4. [DonutOutputCache(Duration = 5, Order = 100)]
  5. public ActionResult Index()
  7. }
* **Handle Error:** A filter that works when you have enabled the custom errors in web.config. The HandleError attribute could be applied over the action method as well as controller level and global level. The HandleError attribute is generally added in the Global.asax.cs file and registered in the Application\_Start event.
* **Authorize:** This action filter enables developers to restrict access to a particular user or role.

### Result Filters

Result filters contain logic for executing before or after generating the result for an action. The IResultFilter interface is used to create a Result Filter.

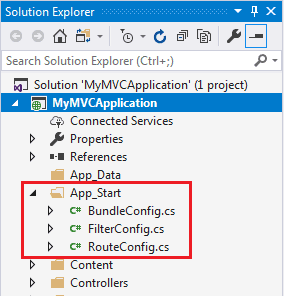
Result filters are performed through the IResultFilter interface. That interface provides two methods: OnResultExecuting and OnResultExecuted. These methods will be executed before or after generating the result for an action.

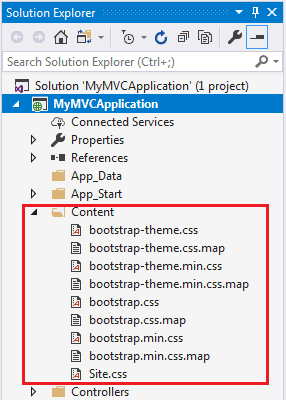
1. namespace MVCFiltere
2. {
3. public interface IResultFilter
4. {
5. void OnResultExecuting(ResultExecutingContext filterContext);
6. void OnResultExecuted(ResultExecutedContext filterContext);
7. }
8. }

### Exception Filters

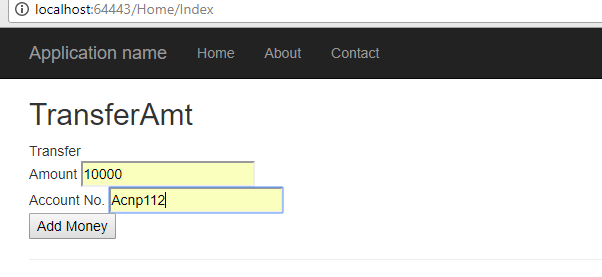
Exception filters are used for handling errors. Exception filters can catch an exception caused by either a controller action or controller action results. Exception filters are also used for logging the exceptions. The IExceptionFilter interface is used to create an Exception Filter which provides the OnException method which will be executed when exception occurs during the action's execution or filter's execution.

1. namespace MVCFiltere
2. {
3. public interface IExceptionFilter
4. {
5. void OnException(ExceptionContext filterContext);
6. }
7. }

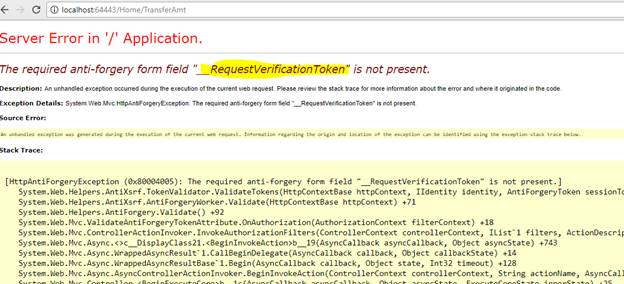




* Now add [ValidateAntiForgeryToken()] to TransferAmt action method as shown below,
  1. [ValidateAntiForgeryToken()]
  2. **public** ActionResult TransferAmt()
  3. {
  4. // Money transfer logic goes here
  5. **return** Content(Request.Form["amount"] + " has been transferred to account " + Request.Form["account"]);
  6. }
* Now if you will run the application



* Click on Add Money, you will get the below error for HttpAntiForgeryException



In the above case it is expecting a verification token which not getting supplied.

* To fix this we need to use @Html.AntiForgeryToken() in view  as  shown below,
  1. <html>
  3. <head>
  4. <title>Transfer money</title>
  5. </head>
  7. <body>
  8. <div> Transfer
  9. <form action="Home/TransferAmt" method=post> Amount <input type="text" name="amount" value="" /><br /> Account No. <input type="text" name="account" value="" /><br /> @Html.AntiForgeryToken() <input type=submit value="Add Money" /> </form>
  10. </div>
  11. </body>
  13. </html>

| Html.Partial() | Html.RenderPartial() |
| --- | --- |
| Html.Partial returns html string. | Html.RenderPartial returns void. |
| Html.Partial injects the html string of the partial view into the main view. | Html.RenderPartial writes html in the response stream. |
| Performance is slow. | Perform is faster compared with HtmlPartial(). |
| Html.Partial() need not to be inside the braces. | Html.RenderPartial must be inside braces @{ }. |

// Razor syntax

@Html.Partial("ViewName")

@{ Html.RenderPartial("ViewName"); }

Exception Handling

<https://www.c-sharpcorner.com/UploadFile/mscratnesh/exception-handling-in-mvc/>

Web API retrun type of IHTTPActionResult

IHttpActionResult : - OK,NotFound,BadRequest

public class ValuesController : ApiController  
  {

    public HttpResponseMessage GetMethod(int id)

    {

        var staff = dbContext.Staff.GetStaff(id);

        if (staff == null)

            return Request.CreateResponse(HttpStatusCode.NotFound);

        // could also throw a HttpResponseException(HttpStatusCode.NotFound)

        return Request.CreateResponse(HttpStatusCode.OK, staff);

    }

}

public class ValuesController : ApiController

{

    public IHttpActionResult GetStaff(int id)

    {

        var staff = dbContext.Staff.GetStaff(id);  
        if (staff == null)

            return NotFound();  
        return Ok(staff);

    }

}

using (var client = new HttpClient())

{

client.BaseAddress=new Uri("http://localhost:64189/api/student");

//HTTP POST

var postTask = client.PostAsJsonAsync<StudentViewModel>("student", student);

postTask.Wait();

var result = postTask.Result;

if (result.IsSuccessStatusCode)

{

return RedirectToAction("Index");

}

}

using (var client = new HttpClient())

{

client.BaseAddress = new Uri("http://localhost:64189/api/");

//HTTP GET

var responseTask = client.GetAsync("student");

responseTask.Wait();

var result = responseTask.Result;

if (result.IsSuccessStatusCode)

{

var readTask = result.Content.ReadAsAsync<IList<StudentViewModel>>();

readTask.Wait();

students = readTask.Result;

}

else //web api sent error response

{

//log response status here..

students = Enumerable.Empty<StudentViewModel>();

ModelState.AddModelError(string.Empty, "Server error. Please contact administrator.");

}

}

using (var client = new HttpClient())

{

client.BaseAddress=new Uri("http://localhost:64189/api/student");

//HTTP POST

Var putTask = client.PutAsJsonAsync<StudentViewModel>("student", student);

putTask.Wait();

var result = putTask.Result;

if (result.IsSuccessStatusCode)

{

return RedirectToAction("Index");

}

}

<https://www.dotnettricks.com/learn/linq/understanding-single-singleordefault-first-and-firstordefault>

public class A

{

public A()

{

Console.WriteLine("A");

}

public void ABC()

{

Console.WriteLine("ABC 1");

}

}

public class B1 : A

{

public B1()

{

Console.WriteLine("B");

}

public void ABC()

{

Console.WriteLine("ABC");

}

}

class Program

{

static void Main(string[] args)

{

B1 obj = new B1();

A obj1 = obj;

obj1.ABC();

}

}

A

B

ABC 1

|  |  |  |
| --- | --- | --- |
| **int.Parse** | **Convert.ToInt32** | **int.TryParse** |
| Syntax | int.Parse(string s) | Convert.ToInt32(bool value) | int.TryParse (string s,out int result) |
| string str1 = "2017";5 | 2017 | 2017 | 2017 |
| string str2 = null; | ArgumentNullException | 0 | Failed to Convert but returns 0 and doesn't throw the exception |
| string str3 = "2017.111111"; | FormatException | FormatException | Failed to Convert but returns 0 and doesn't throw the exception |
| string str4 = "201717171717777777777777777777777777"; | OverflowException | OverflowException | Failed to Convert but returns 0 and doesn't throw the exception |
| bool input00 = false; | Not Supporting to convert | 0 | Failed to Convert but returns 0 and doesn't throw the exception |

The only real difference here is the size. All of the int types here are signed integer values which have varying sizes

* Int16: 2 bytes
* Int32 and int: 4 bytes
* Int64 : 8 bytes

public class classA

{

public classA()

{

Console.WriteLine("Constructor classA");

}

public virtual string Print()

{

return "classA";

}

}

public class classB : classA

{

public classB()

{

Console.WriteLine("Constructor classB");

}

public override string Print()

{

return "classB";

}

}

public class classC : classA

{

public classC()

{

Console.WriteLine("Constructor classC");

}

public new string Print()

{

return "ClassC";

}

}

static void Main(string[] args)

{

classA a = new classC();

Console.WriteLine(a.Print());

}

Output :

Constructor classA

Constructor classC

ClassA

public class classA

{

public classA()

{

Console.WriteLine("Constructor classA");

}

public virtual string Print()

{

return "classA";

}

}

public class classB : classA

{

public classB()

{

Console.WriteLine("Constructor classB");

}

public override string Print()

{

return "classB";

}

}

public class classC : classB

{

public classC()

{

Console.WriteLine("Constructor classC");

}

public new string Print()

{

return "ClassC";

}

}

static void Main(string[] args)

{

classA a = new classC();

Console.WriteLine(a.Print());

}

Output :

Constructor classA

Constructor classB

Constructor classC

ClassB

public class Base

{

public virtual void Func(int x)

{

Console.WriteLine("Base Func(int x)");

}

}

public class Derived : Base

{

public override void Func(int x)

{

Console.WriteLine("Derived Func(int x)");

}

public void Func(object o)

{

Console.WriteLine("Derived Func(object x)");

}

}

class Program

{

static void Main(string[] args)

{

Derived d = new Derived();

int i = 10;

d.Func(i);

Console.ReadKey();

}

}

Output : Derived Func(object x)

dynamic val = num;

Console.WriteLine(val.GetType());

val += num;

Console.WriteLine(val.GetType());

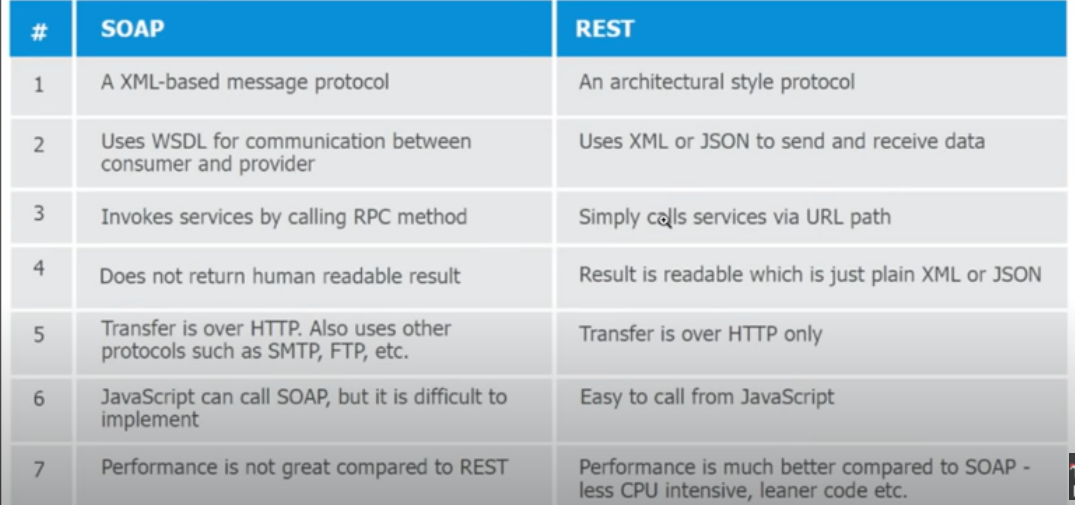
System.Byte

System.Int32

int[] arr = new int[0];

Console.WriteLine(arr[0]);

Index out of range



class A

{

public virtual void show()

{

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

// Console.ReadLine();

}

}

class B : A

{

}

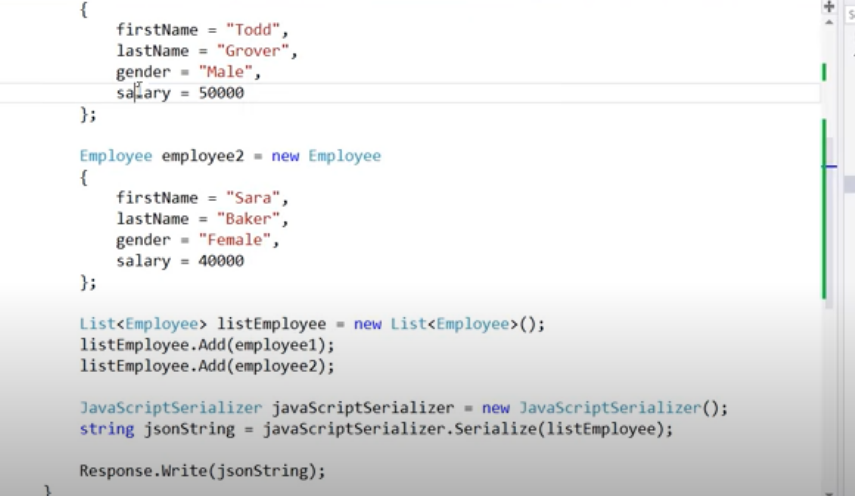
B obj =new B();

Obj.show();

Output :

“Hello : Base Class!”

JavaScript Serialization in c#



JavaScript Deserialization in c#



## **Isolation Levels in SQL Server**

Read Uncommitted

Read Committed

Repeatable Read

Snapshot

Serializable

SELECT \* FROM SourceProducts

SELECT \* FROM TargetProducts

MERGE TargetProducts AS Target

USING SourceProducts AS Source

ON Source.ProductID = Target.ProductID

WHEN NOT MATCHED BY Target THEN

INSERT (ProductID,ProductName, Price)

VALUES (Source.ProductID,Source.ProductName, Source.Price);

MERGE TargetProducts AS Target

USING SourceProducts AS Source

ON Source.ProductID = Target.ProductID

-- For Inserts

WHEN NOT MATCHED BY Target THEN

INSERT (ProductID,ProductName, Price)

VALUES (Source.ProductID,Source.ProductName, Source.Price)

-- For Updates

WHEN MATCHED THEN UPDATE SET

Target.ProductName = Source.ProductName,

Target.Price = Source.Price;