**Fundamental concepts of Object Oriented Programming.**

* **Encapsulation** – Encapsulation enables a group of properties, methods and other members to be considered a single unit or object.

**Public** – Visible in the current and referencing assembly.

**Private** – Visible inside the current class.

**Protected** – Visible inside the current as well as in the derived class.

**Internal** – Visible inside the containing assembly.

**Internal protected** – Visible inside the containing assembly, also in the descendent of the current class.

* **Abstract class** – An abstract class is a special type of class that cannot be instantiated. An abstract class is designed to be inherited by subclasses that either implement or override its methods.

It is a class whose object cannot be instantiated. The class can only be inherited. It should contain at least one method abstract. It is denoted by the keyword *abstract.*

1. **Public abstract** **class** NonAbstractMethod
2. {
3. **public** **string** getDcn()
4. {
5. **return** "PS20100301A0012";
6. }
7. **public** **abstract** **void** getSeqID();
8. }
9. **Public class** Utilize : NonAbstractMethod
10. {
11. **public** **override** **void** getSeqID()
12. {
13. //Method Logic
14. }
15. }

* **Inheritance** –Inheritance is the ability to create a new class with attributes and behaviours from an existing class. The newly created class is the derived (or child) class and the existing class is the base (or parent) class.

public class Animal

{

    public void See()

    {

        Console.WriteLine("All animals can see");

    }

  public void Hear()

    {

        Console.WriteLine("All animals have ears");

    }

}

public class Cat : Animal

{

    public void Talk()

    {

        Console.WriteLine("A cat purrs");

    }

}

public static void Main(string[] args)

    {

        Cat obj = new Cat();

        obj.Hear();

obj.See();

     }

* **Polymorphism** – The name means, one name, many forms. It is achieved by having multiple methods with the same name but different implementations.

**Overloading** (Compile Time Polymorphism / Static Binding / Early Binding)

Public void show\_method(int a, int b)

{

//do something

}

Public void show\_method(string str1, string str2)

{

//do something

}

**Overriding** (Runtime Time Polymorphism / Dynamic Binding / Late Binding) 

Public class Net

{

public virtual void Act()

{

Console.WriteLine(‘’Net Act");

}

}

Public class Perl : Net

{

public override void Act()

{

Console.WriteLine("Perl Act");

}

}

* **Interface** – An Interface is a class with no implementation. The only thing that it contains is the declaration of methods, properties, and events.

Public interface IZoo

{

Public void See();

Public string Hear();

}

public abstract class Animal : IZoo

{

    public abstract void Talk()

    {

        Console.WriteLine("All animals can talk");

    }

    public void See()

    {

        Console.WriteLine("All animals can see");

    }

  public void Hear()

    {

        Console.WriteLine("All animals have ears");

    }

}

* **Partial class** – C# provides the ability to have a single class implementation in multiple files using the partial modifier [keyword](https://www.tutorialsteacher.com/csharp/csharp-keywords).

**classfile1.cs**

public partial class MyPartialClass

{

public void Method1(int val)

{

Console.WriteLine(val);

}

}

**classfile2.cs**

public partial class MyPartialClass

{

public void Method2(int val)

{

Console.WriteLine(val);

}

}

* **Sealed class** – Sealed classes are used to restrict the inheritance feature of object oriented programming. Once a class is defined as a sealed class, this class cannot be inherited. To access the member of sealed class, we need to create the object of class.

sealed class SealedClass

{

    public int Add(int a, int b)

    {

        return a + b;

    }

}

Public static void Main(string[] args)

    {

        SealedClass slc = new SealedClass();

        int total = slc.Add(6, 4);

        Console.WriteLine("Total = " + total.ToString());

    }

* **Static class -** A static class can only contain static data members, static methods, and a static constructor. It is not allowed to create objects of the static class. Static classes are sealed by default; means it does not allow inheritance.

Public static class Author

{

    public static string Name = "Bappaditya";

    public static void Details()

    {

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

    }

}

public static void Main(String[] args)

{

        Console.WriteLine(Author.Name);

        Author.Details();

     }

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

Public class Program

{

static void MethodA()

{

Console.WriteLine("Static method");

}

void MethodB()

{

Console.WriteLine("Instance method");

}

}

Public static void Main()

{

Program.MethodA();

Program programInstance = new Program();

programInstance.MethodB();

}

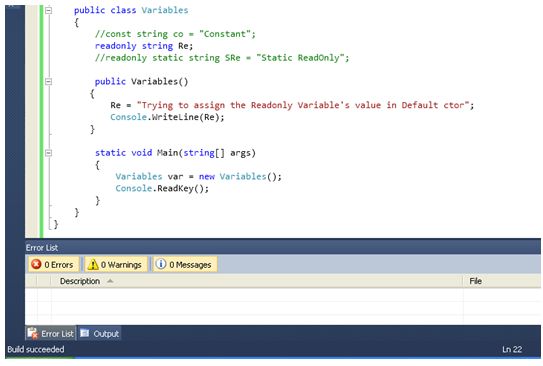
* **Difference between static, const and readonly**

**Const**  
Constant fields are defined at the time of declaration in the code snippet, because once they are defined they can't be modified. By default, a constant is static, so you can't define them static from your side.

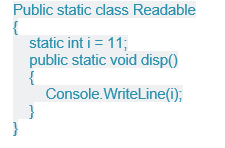




**Readonly**A Readonly field can be initialized either at the time of declaration or within the constructor of the same class. We can also change the value of a Readonly at runtime or assign a value to it at runtime (but in a non-static constructor only).



**Static**  
The static keyword is used to declare a static member. Static field can be initialized at the time of declaration or anywhere within the class.



* **Difference between ref and out**

**Ref Keyword**While using the Ref keyword we need to initialize the parameter before passing to metod.

int x=10;

Foo(ref int x); // OK

int y;

Foo(ref int y); // Error: y should be initialized before calling the method

1. **int** intRef = 2;
2. **private** **static** **int** MethodCall(**ref** **int** intRef)
3. {
4. **return** intRef;
5. }

**Out Keyword**  
  
While using Out keyword we need to initialize the parameter before returning from the calling method.

1. **int** intOut;
2. **private** **static** **int** MethodCall(**out** **int** intOut)
3. {
4. **return** intOut = 3;
5. }

* **How to restrict a method to be inherited?**

Making the method Static method.

* **What is the difference between Continue and Break Statement**

Break statement breaks the loop. It makes the control of the program to exit from the loop. Continue statement makes the control of the program to exit only the current iteration. It does not break the loop.

* **Where do we use interface and abstract class?**

C# does not support multiple inheritances, interfaces are mainly used to implement the multiple inheritances. As a class can implement more than one interface and only inherit from one abstract class. An interface is mainly used only when we do not require the implementation of methods or functionalities.

|  |  |
| --- | --- |
| **ABSTRACT CLASS** | **INTERFACE** |
| It contains both declaration and definition part. | It contains only a declaration part. |
| Multiple inheritance is not achieved by abstract class. | Multiple inheritance is achieved by interface. |
| It contain [constructor](https://www.geeksforgeeks.org/c-sharp-constructors/). | It does not contain [constructor](https://www.geeksforgeeks.org/c-sharp-constructors/). |
| It can contain static members. | It does not contain static members. |
| It can contain different types of access modifiers like public, private, protected etc. | It only contains public access modifier because everything in the interface is public. |
| The performance of an abstract class is fast. | The performance of interface is slow because it requires time to search actual method in the corresponding class. |
| A class can only use one abstract class. | A class can use multiple interface. |
| Abstract class can contain methods, fields, constants, etc. | Interface can only contain methods. |

| **DISPOSE** | **FINALIZE** |
| --- | --- |
| The method dispose( ) is defined in the interface IDisposable interface. | The method finalize( ) id defined in java.lang.object class. |
| public void dispose( ){ // dispose code here } | protected void finalize( ){ // finalization code here } |
| The method dispose( ) is invoked by the user. | The method finalize( ) is invoked by the garbage collector. |
| Method dispose( ) is used to free unmanaged resources whenever it is invoked. | Method finalize( ) is used to free unmanaged resources before the object is destroyed. |
| The method dispose( ) is to be implemented whenever there is a close( ) method. | The method finalize( ) is to be implemented for unmanaged resources. |
| The method dispose( ) is declared as public. | The method finalize( ) is declared as private. |
| The method dispose( ) is faster and instantly disposes an object. | The method finalize is slower as compared to dispose. |

* **Define ABC concept in WCF**

**Address:**Where exactly the service hosted? In other words, what is the location of the service? An address could be an IP Address, server name, URL and so on.

**Binding:**It is about how the messages are handled in the service side and the client side.

**Contract**: The contract is an agreement between the client and the server about what the structure and content of messages being exchanged.

* An **address** that indicates where the endpoint can be found / service hosted.
* A **binding** that specifies how a client can communicate with the server.
* A **contract** is just an interface between client and server where client and server communicate each other.
* **Caching:**

Caching is a technique of storing frequently used data/information in memory, so that, when the same data/information is needed next time, it could be directly retrieved from the memory instead of being generated by the application.

Types -

Page Caching

Fragment Caching

Data Caching

**Page Caching:** To cache an entire page's output we need to specify a directive at the top of our page, this directive is the @ OutputCache.

1. <%@ OutputCache Duration = 5 VaryByParam = "ID" %>

**Fragment caching:** In some scenarios we only need to cache only a segment of a page. For example a contact us page in a main page will be the same for all the users and for that there is no need to cache the entire page.

1. <%@ OutputCache Duration = 10 VaryByParam = "None" %>

**Data Caching:** As we know in C# everything is about classes and objects. So ASP.NET supports data caching by treating them as small sets of objects. We can store objects in memory very easily and use them depending on our functionality and needs, anywhere across the page.

1. Cache["Website"] = "CSharpCorner";

* **Trigger:**

Trigger is a Special kind of Stored Procedure or an operation that gets executed automatically when an event occurs in the database.

In SQL Server we can create the following 3 types of triggers:

* + Data Definition Language (DDL) triggers
  + Data Manipulation Language (DML) triggers
  + Logon triggers

<http://sqlhints.com/tag/types-of-triggers/>

* **Difference between Html.Partial() and Html.RenderPartial() in ASP.NET MVC**

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

* **httpVerbs:**

<https://www.restapitutorial.com/lessons/httpmethods.html>

| **HTTP Verb** | **CRUD** | **Entire Collection (e.g. /customers)** | **Specific Item (e.g. /customers/{id})** |
| --- | --- | --- | --- |
| POST | Create | 201 (Created), 'Location' header with link to /customers/{id} containing new ID. | 404 (Not Found), 409 (Conflict) if resource already exists.. |
| GET | Read | 200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists. | 200 (OK), single customer. 404 (Not Found), if ID not found or invalid. |
| PUT | Update/Replace | 405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection. | 200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid. |
| PATCH | Update/Modify | 405 (Method Not Allowed), unless you want to modify the collection itself. | 200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid. |
| DELETE | Delete | 405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable. | 200 (OK). 404 (Not Found), if ID not found or invalid. |

* **Var Vs Dynamic Keyword:**

“**var**” is a statically typed variable. It results in a strongly typed variable, in other words the data type of these variables are inferred at compile time. This is done based on the type of value that these variables are initialized with.

**Example:** var data = 12345;

data =”Welcome”;

**Output:** Cannot implicitly convert type ‘string’ to ‘int’.

“**dynamic**” are dynamically typed variables. This means, their type is inferred at run-time and not the compile time in contrast to var type.

**Example:** dynamic data = 12345;

data =”Welcome”;

**Output:** Welcome.

* **Virtual Keyword:**

The virtual keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class.

* **Difference between window.load and document.ready**

$(window).load event is fired after whole content (including css, images etc.) is loaded.

document.ready is called when DOM is loaded and document structure is ready.

# Difference Between Dispose and Finalize

Dispose and Finalize are invoked to free the unmanaged resources held by an object. The dispose() method is defined inside the interface IDisposable whereas, the method finalize() is defined inside the class object. The main difference between dispose() and finalize() is that the method **dispose**() has to be explicitly invoked by the user, whereas the method **finalize()** is invoked by the garbage collector, just before the object is destroyed.

## **ViewData Vs ViewBag Vs TempData**

|  |  |  |
| --- | --- | --- |
| ViewData | ViewBag | TempData |
| It is Key-Value Dictionary collection | It is a type object | It is Key-Value Dictionary collection |
| ViewData is a dictionary object and it is property of ControllerBase class | ViewBag is Dynamic property of ControllerBase class. | TempData is a dictionary object and it is property of controllerBase class. |
| ViewData is Faster than ViewBag | ViewBag is slower than ViewData | NA |
| ViewData is introduced in MVC 1.0 and available in MVC 1.0 and above | ViewBag is introduced in MVC 3.0 and available in MVC 3.0 and above | TempData is also introduced in MVC1.0 and available in MVC 1.0 and above. |
| ViewData also works with .net framework 3.5 and above | ViewBag only works with .net framework 4.0 and above | TempData also works with .net framework 3.5 and above |
| Type Conversion code is required while enumerating | In depth, ViewBag is used dynamic, so there is no need to type conversion while enumerating. | Type Conversion code is required while enumerating |
| Its value becomes null if redirection has occurred. | Same as ViewData | TempData is used to pass data between two consecutive requests. |
| It lies only during the current request. | Same as ViewData | TempData only works during the current and subsequent request |

* **Difference between throw and throw(ex)**

### **Similarities**:

Let’s first see what is similar in both.

1. Both are used to throw exception in catch block to log message
2. Both contains same message of exception

### **Difference**:

Now let’s see what is the difference.

1. throw is used to throw current exception while throw(ex) mostly used to create a wrapper of exception.
2. throw(ex) will reset your stack trace so error will appear from the line where throw(ex) written while throw does not reset stack trace and you will get information about original exception.
3. In MSIL code when you use throw(ex) it will generate code as throw and if you use throw it will create rethrow.

* **What is Garbage Collection**

The Common Language Runtime (CLR) requires that you create objects in the managed heap, but you do not have to bother with cleaning up the memory once the object goes out of the scope or is no longer needed. Garbage collection refers to the strategy to free unused objects or the objects that go out of the scope automatically.

Implicit Garbage Collection is handled by the .Net framework. When object is created then it will be placed in the Generation 0. The garbage collection uses an algorithm which checks the objects in the generation, the objects life time get over then it will be removed from the memory. The two kinds of objects. One is Live Objects and Dead Objects. The Garbage collection algorithm collects all unused objects that are dead objects in the generation. If the live objects running for long time then based on that life time it will be moved to next generation.

* **Generation of Garbage Collection**

1. Generation 0 identifies a newly created object that has never examined by GC.
2. Generation 1 identifies an object that has survived a GC, marked for collection but not removed because there was sufficient heap space.
3. Generation 2 identifies an object that has survived more than one movement of the GC.

GC.Collect() method is used to force a garbage collection of all the generations.  It can also force a garbage collection of a particular generation passed to it as a parameter.

public static void Collect();

public static void Collect(Integer int);

# Improve the Performance of an ASP.Net Application

# <https://www.c-sharpcorner.com/article/tips-and-best-practices-to-improve-asp-net-web-application-performance/>

# <https://www.c-sharpcorner.com/uploadfile/skumaar_mca/tips-to-improve-the-performance-in-Asp-Net-application/>

# Improve the Performance of SQL Server

<https://www.dotnettricks.com/learn/sqlserver/tips-to-improve-sql-server-database-design-and-performance>

<https://www.winwire.com/25-tips-to-improve-sql-query-performance/>

* **What are the data type used in JavaScript**

Six data types that are primitives:

Number

String

Boolean

Null

Undefined

Symbol

* **What is fault Contract**  
    
  A Fault Contract is a way to handle an error/exception in WCF. In C# we can handle the error using try and catch blocks at the client side. The purpose of a Fault Contract is to handle an error by the service class and display in the client side. Whenever the client makes a call to a service class and an unexpected exception, in this case the service class throws an exception to the client using the Fault Contract.

However, if you need to pass user friendly exception messages from the service, you should throw fault exceptions. Fault exceptions are exceptions that are thrown by a WCF service when an exception occurs at runtime.