* **Delegate:**

public delegate void SimpleDelegate();

class TestDelegate

{

public static void MyFunc()

{

Console.WriteLine("I was called by delegate.");

}

public static void Main()

{

SimpleDelegate simpleDelegate = new SimpleDelegate(MyFunc);

simpleDelegate();

}

}

OUTPUT:

I was called by delegate.

============================================

delegate int NumberChanger(int n);

class TestDelegate

{

static int num = 10;

public static int AddNum(int p)

{

num += p;

return num;

}

public static void Main()

{

NumberChanger ncd = new NumberChanger(AddNum);

int result = ncd(25);

Console.WriteLine(result);

}

}

OUTPUT:

35

* **Explicit Interface:**

public interface IFileLogger

{

void LogError();

}

public interface IDatabaseLogger

{

void LogError();

}

public class CustomLogger: IFileLogger,IDatabaseLogger

{

public void LogError()

{

Console.WriteLine("Explicit Interface Error!");

}

}

class MyClass

{

public static void Main()

{

CustomLogger customLogger = new CustomLogger();

IFileLogger fileLogger = new CustomLogger();

IDatabaseLogger databaseLogger = new CustomLogger();

customLogger.LogError();

fileLogger.LogError();

databaseLogger.LogError();

}

}

OUTPUT:

Explicit Interface Error!

Explicit Interface Error!

Explicit Interface Error!

=======================================================

public interface IFileLogger

{

void LogError();

}

public interface IDatabaseLogger

{

void LogError();

}

public class CustomLogger: IFileLogger,IDatabaseLogger

{

public void LogError()

{

Console.WriteLine("Explicit Interface Error!");

}

void IFileLogger.LogError()

{

Console.WriteLine("Log Error to File!");

}

void IDatabaseLogger.LogError()

{

Console.WriteLine("Log Error to Database!");

}

}

class MyClass

{

public static void Main()

{

CustomLogger customLogger = new CustomLogger();

IFileLogger fileLogger = new CustomLogger();

IDatabaseLogger databaseLogger = new CustomLogger();

customLogger.LogError();

fileLogger.LogError();

databaseLogger.LogError();

}

}

OUTPUT:

Explicit Interface Error!

Log Error to File!

Log Error to Database!

* **Dependency Injection:**

1. Constructor Injection

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

public interface IEmailService

{

void SendMail(string emailAddress, string message)

}

public class OutlookEmailService: IEmailService

{

public void SendMail(string emailAddress, string message)

{

//Send an email using outlook

}

}

public class UserLogic

{

private IEmailService \_emailService;

public UserLogic(IEmailSevice emailService)

{

\_emailService = emailService;

}

public void Register(string emailAddress, string password)

{

\_emailService.SendMail(emailAddress, authResult.ConfirmationMessage);

}

}

2. Method Injection

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

public interface IEmailService

{

void SendMail(string emailAddress, string message)

}

public class OutlookEmailService: IEmailService

{

public void SendMail(string emailAddress, string message)

{

//Send an email using outlook

}

}

public class UserLogic

{

private IEmailService \_emailService;

public UserLogic()

{

\_emailService = new OutlookEmailService();

}

public void Register(string emailAddress, string password)

{

\_emailService.SendMail(emailAddress, authResult.ConfirmationMessage);

}

}

==================================================

1. Constructor Injection

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

public interface text

{

void print();

}

class format : text

{

public void print()

{

Console.WriteLine(" here is text format");

}

}

// constructor injection

public class constructorinjection

{

private text \_text;

public constructorinjection(text t1)

{

\_text = t1;

}

public void output()

{

\_text.print();

}

}

* **Why oops is not supported multiple inheritance.**

1. One problem occurs when two parent classes have data members or methods of the same name. It is difficult to resolve which is being referenced by the sub-class.

2. The Diamond of Dread:

We have a class A, then B and C both inherit from A. And someone then decides that D must inherit both from B and C.

The parent class A was present twice in its grandchild class D, having something go silently wrong and crash.

* **Method Overriding:**

<https://www.codeproject.com/Articles/18734/Method-Overriding-in-C>

* **SOLID Principles:**

S: Single Responsibility Principle (SRP)

O: Open closed Principle (OCP)

L: Liskov substitution Principle (LSP)

I: Interface Segregation Principle (ISP)

D: Dependency Inversion Principle (DIP)

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

S: Single Responsibility Principle (SRP):-

Every software module should have only one responsibility and one reason to change. /

Every class should have a single responsibility. And there should be a single reason to change the class.

O: Open/Closed Principle (OCP):-

A software module/class is open for extension and closed for modification. /

Software application source codes should be open for extension but should be closed for modification.

L: Liskov Substitution Principle (LSP):-

The derived classes are extending the base classes without changing their behaviour. /

The derived classes should be perfectly substitutable for their base classes.

I: Interface Segregation Principle (ISP):-

Clients should not be forced to implement interfaces they don't use. /

Clients should not be forced to implement methods which it does not use.

D: Dependency Inversion Principle (DIP):-

High-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions.

Secondly, abstractions should not depend upon details. Details should depend upon abstractions.

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

* **Singletone:**

public sealed class Singleton

{

private static Singleton instance=null;

private Singleton()

{

}

public static Singleton Instance

{

get

{

if (instance==null)

{

instance = new Singleton();

}

return instance;

}

}

}

-------------------Example-----------------------

public sealed class Singleton

{

private static Singleton instance=null;

private Singleton()

{

}

public static Singleton Instance

{

get

{

if (instance==null)

{

instance = new Singleton();

}

return instance;

}

}

public void LogMessage(string message)

{

WriteLine("Message : " + message);

}

}

public static void Main()

{

Singleton fromManager = Singleton.Instance;

fromManager.LogMessage("Good Morning!");

Singleton fromEmployee = Singleton.Instance;

fromEmployee.LogMessage("Good Night!");

}

Output:

Message : Good Morning!

Message : Good Night!

* **Factory Pattern:**

class Program

{

abstract class Position

{

public abstract string Title { get; }

}

class Manager : Position

{

public override string Title

{

get

{

return "Manager";

}

}

}

class Clerk : Position

{

public override string Title

{

get

{

return "Clerk";

}

}

}

class Programmer : Position

{

public override string Title

{

get

{

return "Programmer";

}

}

}

static class Factory

{

/// <summary>

/// Decides which class to instantiate.

/// </summary>

public static Position Get(int id)

{

switch (id)

{

case 0:

return new Manager();

case 1:

return new Clerk();

case 2:

return new Programmer();

default:

return new Clerk();

}

}

}

static void Main()

{

for (int i = 0; i <= 3; i++)

{

var position = Factory.Get(i);

Console.WriteLine("Where id = {0}, position = {1} ", i, position.Title);

}

}

}

OutPut:

Where id = 0, position = Manager

Where id = 1, position = Clerk

Where id = 2, position = Programmer

Where id = 3, position = Clerk

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

interface IGet

{

string ConC(int num);

}

class clsFirst : IGet

{

public string ConC(int num)

{

string Final = "Position First: " + num;

return Final;

}

}

class clsSecond : IGet

{

public string ConC(int num)

{

string Final = "Position Second: " + num;

return Final;

}

}

static class clsFactory

{

static public IGet CreateandReturnObj(int cChoice)

{

IGet ObjSelector = null;

switch (cChoice)

{

case 1:

ObjSelector = new clsFirst();

break;

case 2:

ObjSelector = new clsSecond();

break;

default:

ObjSelector = new clsFirst();

break;

}

return ObjSelector;

}

}

static void Main()

{

IGet ObjIntrface = null;

for (int i = 0; i <= 3; i++)

{

ObjIntrface = clsFactory.CreateandReturnObj(i);

string res = ObjIntrface.ConC(i);

}

}

Output:

Position First: 1

Position Second: 2

Position First: 1

* **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. |

* **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.

* **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.

* **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/>

* **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.