* **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 behavior./

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/>