**Design Pattern**

**Q-1: What is Desigen Pattern**

**A:** A design pattern is a **well-proved solution** for solving the specific problem/task. We must use the design patterns **during the analysis and requirement phase of SDLC**(Software Development Life Cycle).

Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.

#### Categorization of design patterns:

Basically, design patterns are categorized into two parts:

1. Core java (or JSE) Design Patterns.
2. JEE Design Patterns.

**Q-2: What is Singleton Design Pattern . also write the code.**

**A:** Singleton Pattern says that just**"define a class that has only one instance**

**and provides a global point of access to it".**

A class must ensure that only single instance should be created and single object can be used by all other classes.

**Advantage of Singleton design pattern**

Saves memory because object is not created at each request. Only single instance is reused again and again.

This has advantages in memory management, and for Java, in garbage collection. Moreover, restricting the number of instances may be necessary or desirable for technological or business reasons--for example, we may only want a single instance of a pool of database connections.

**Usage of Singleton design pattern**

Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

**How to create Singleton design pattern?**

To create the singleton class, we need to have static member of class, private constructor and static factory method.

**1-Static member:** It gets memory only once because of static, itcontains the instance of the Singleton class.

**2-Private constructor:** It will prevent to instantiate the Singleton class from outside the class.

**3-Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

**Ex:**

Ex->

**class** Demo

{

**private** **static** Demo *obj*=**null**;

**static**

{

*obj* = **new** Demo();

}

**private** Demo()

{

System.*out*.println("Demo");

}

**public** **static** Demo getObject()

{

**return** *obj*;

}

}

**public** **class** Demo1 {

**public** **static** **void** main(String[] args) {

//Demo d1 = new Demo();

Demo d2 = Demo.*getObject*();

Demo d3 = Demo.*getObject*();

System.*out*.println(d2);

System.*out*.println(d3);

}

}

# Q-58: How can be create only 5 object of the class

A:

**public** **class** MSInt

{

**private** **static** MSInt *instance* = **null**;

**private** **static** **int** *count* = 0;

**private** MSInt()

{

System.out.println("MSINT");

}

**public** **static** MSInt getInstance()

{

**if**(*count* < 5){

*instance* = **new** MSInt();

*count*++;

**return** *instance*;

}

**else**

{

**return** **null**;

}

}

**public** **static** **void** main(String[] args)

{

MSInt m1 = MSInt.*getInstance*();

MSInt m2 = MSInt.*getInstance*();

MSInt m3 = MSInt.*getInstance*();

MSInt m4 = MSInt.*getInstance*();

MSInt m5 = MSInt.*getInstance*();

MSInt m6 = MSInt.*getInstance*();

System.*out*.println("m1"+m1);

System.*out*.println("m2"+m2);

System.*out*.println("m3"+m3);

System.*out*.println("m4"+m4);

System.*out*.println("m5"+m5);

System.*out*.println("m6"+m6);

}

# }

**Q-3: What is Factory Desigen Pattern.also write the code.**

**A:** A Factory Pattern  **define an interface or abstract class for creating an**

**object but let the subclasses decide which class to instantiate.**

In other words, subclasses are responsible to create the instance of the class.

#### Advantage of Factory Design Pattern

1-Factory Method Pattern allows the sub-classes to choose the type of objects to create.

2-It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

#### Usage of Factory Design Pattern

1-When a class doesn't know what sub-classes will be required to create

2-When a class wants that its sub-classes specify the objects to be created.

3-When the parent classes choose the creation of objects to its sub-classes.



**Ex:**

**Step 1->**

Create an interface.

Shape.java

public interface Shape

{

void draw();

}

**Step 2->**

Create concrete classes implementing the same interface.

**1-Rectangle.java**

public class Rectangle implements Shape

{

@Override

public void draw()

{

System.out.println("Inside Rectangle::draw() method.");

}

}

**2-Square.java**

public class Square implements Shape

{

@Override

public void draw()

{

System.out.println("Inside Square::draw() method.");

}

}

**3-Circle.java**

public class Circle implements Shape

{

@Override

public void draw()

{

System.out.println("Inside Circle::draw() method.");

}

}

**Step 3->**

Create a Factory to generate object of concrete class based on given information.

**ShapeFactory.java**

public class ShapeFactory

{

//use getShape method to get object of type shape

public Shape getShape(String shapeType)

{

if(shapeType == null){

return null;

}

if(shapeType.equalsIgnoreCase("CIRCLE")){

return new Circle();

} else if(shapeType.equalsIgnoreCase("RECTANGLE")){

return new Rectangle();

} else if(shapeType.equalsIgnoreCase("SQUARE")){

return new Square();

}

return null;

}

}

**Step 4->**

Use the Factory to get object of concrete class by passing an information such as type.

**FactoryPatternDemo.java**

public class FactoryPatternDemo

{

public static void main(String[] args)

{

ShapeFactory shapeFactory = new ShapeFactory();

Shape shape1 = shapeFactory.getShape("CIRCLE");

shape1.draw();

Shape shape2 = shapeFactory.getShape("RECTANGLE");

shape2.draw();

Shape shape3 = shapeFactory.getShape("SQUARE");

shape3.draw();

}

}

**Q-4: What is Prototype Desigen Pattern.also write the code.**

**A:** Prototype Pattern says that **cloning of an existing object instead of creating new one and can also be customized as per the requirement**.

**Advantage of Prototype Pattern**

The main advantages of prototype pattern are as follows:

1. It reduces the need of sub-classing.
2. It hides complexities of creating objects.
3. The clients can get new objects without knowing which type of object it will be.
4. It lets you add or remove objects at runtime.

How to implement Prototype Pattern

1. We are going to create **an interface Prototype** that contains a method **getClone()** of **Prototype type.**
2. Then, we create **a concrete class EmployeeRecord** which implements **Prototype interface** that does the cloning of EmployeeRecord object.
3. **PrototypeDemo class** will uses this concrete class **EmployeeRecord.**

Ex:

**interface** Prototype

{

**public** Prototype getClone();

}

**class** Employee **implements** Prototype

{

**private** **int** id;

**private** String name;

**private** **int** salary;

**private** String address;

**public** Employee(){ }

**public** Employee(**int** id, String name, **int** salary, String address) {

**this**.id = id;

**this**.name = name;

**this**.salary = salary;

**this**.address = address;

}

**public** **void** showRecord()

{

System.*out*.println(id+"\t"+name+"\t"+salary+"\t"+address); }

@Override

**public** Prototype getClone()

{

**return** **new** Employee(id,name,salary,address);

}

}

**class** PrototypeDemo

{

**public** **static** **void** main(String[] args)

{

Employee e1=**new** Employee(11,"subhag",12345,"BTM");

e1.showRecord();

Employee e2=(Employee) e1.getClone();

e2.showRecord();

}

}

**Q-6: FrontController Desigen Pattern.**

**A:** The front controller design pattern is used to provide a centralized request handling mechanism so that all requests will be handled by a single handler. This handler can do the authentication/ authorization/ logging or tracking of request and then pass the requests to corresponding handlers

Note: we will see the without Front Controller Desigen Pattern

**without Front Controller Desigen Pattern**

**Presentation Controller Model**

**Layer Layer**  **Layer**

\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Some**

**Model**

**component**

**LoginServlet.java**

public void service(req,res){

1-collect data

2-use model to interact DB

3-Return the result page to

Forward

}

**Login.jsp**

<form action=”login.jlc”>

………

</form>

**Register.jsp**

**DB**

**RegisterServlet.java**

public void service(req,res){

1-collect data

2-use model to interact DB

3-Return the result page to

Forward

}

<form action=”register.jlc”>

………

</form>

**Frogotpw.jsp**

**ForgotPWServlet.java**

public void service(req,res)){

1-collect data

2-use model to interact DB

3-Return the result page to

Forward

}

<form action=”forgot.jlc”>

………

</form>

**Project Question**

**Explation of the Problem->**

1-in this for every JSP,we are creating the every servlet.for **Login.jsp** – it call **LoginServlet** . for Register.jsp –it will call **RegisterServlet** in this code duplication problem arise.

2-So Application needs a Single Entry Point(centeralized access point) for request handling

3-without a central access point,control code that is common across multiple requests duplicate in multiple servlet

4-Having control code in multiple Servlets is difficult to maintain and a single code change made be made in multiple Servlets.

**Solution of the problem->**

**1-**i wants to apply duplicate control logic. And apply common logic to multiple requests.(we want to apply Single entry point for every jsp)

**2-** Use a Front Controller as the initial Point of contact for handling all related request. The Front Controller centralizes control logic

**Presentation Controller**

**Layer Layer**

\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**JLCControllerServlet.java**

LoginAction loginAction=null

**public void init(….)**{

loginAction=new LoginAction();

…….

}

**public void service(..)**{

String page=” ”;

String uri=req.getRequestURL();

**if(uri.endsWith(“login.jlc”))**{

Page=loginAction.process(req,res)

}

**else** **if(uri.endsWith(“login.jlc”))**{

}

…….

RequestDispatcher rd=

Req.getRequestDispatcher(page);

**Rd.forward(req,res);**

}

…………..

}

**LoginAction.java**

public String process(rq,rs){

1-collect data

2-use model to interact DB

3-Return the result page to

Forward

}

**Model Layer**

**Some**

**Model**

**Component**

**(it is used to interact with DB)**

**Login.jsp**

<form action=”login.jlc”>

………

</form>

**Register.jsp**

**RegisterAction.java**

public String process(rq,rs){

1-collect data

2-use model to interact DB

3-Return the result page to

Forward

}

<form action=”register.jlc”>

………

</form>

**Forgotpw.jsp**

<form action=”forgot.jlc”>

………

</form>

fffffffffffffffffffff

**ForgotPWAction.java**

public String process(rq,rs){

1-collect data

2-use model to interact DB

3-Return the result page to

Forward

}

**3-**As per Front Controller Design Pattern

a-You need to write One and Only one controller component for the entire web application

b- One and Only one controller component is single entry point for the web application

**4-** Controller component may be implemented as Servlet or Filter.

5-Servlet is used as Front Controller component in Struts1,JSF,Spring MVC

6-Filter is used as Front Controller component in Struts2.

7-From the above diagram JLCControllerServlet.java(it will keep common task such as init() and destroy()) get the Action from value(request URI) it is called as front controller.

8- because all requesrt come to a single entry point ->JLCControllerServlet.java

9- here All Servlet is controlled by a Single Servlet called Front Controller.

**Q-5: What is MVC Desigen Pattern.how many layer in this Desigen Pattern.what is the benefits of this Desigen Pattern.**

**A:** Consider the following Requirement

1. if same Persistence logic or Business Logic is written in all the Servlets. This will give the code duplication problem.
2. in future , you want to change Persistence logic or Business logic, you need to modify al the Servlets.this give you the Maintenance problem.
3. When you have multiple Servlet and each Servlet we have a database code so this will give “duplication” problem.

And if we want to change the database. So we have to modified database code in every Servlet, So it will create the “Maintenance” problem

1. if same Presentation Logic is written in all Servlets(may be Presentation Logic related to Header,menu,footer and formatting styles).

This will give code duplication problem

**5-**in future, if you want to change Presentation Logic,you need to modify all Servlets.This will give the Maintenance problem

**So Solution of this Problem is MVC Desigen Pattern**

**MVC Stands for Model-View-Controller**

**View->** Represents the presentation i.e. UI(User Interface).

It is use for sending the request to server and sending the response

to client (HTML , JSP is use for Presentation Purpose)

**Controller->** Controller acts on both model and view. It controls the data flow

into model object and updates the view whenever data changes. It

keeps view and model separate. It Controlling the Request

processing.(Servlet acts as a Controller)

**Model->** interacting with Database(Persistence Logic) or External(Business Logic)

Model represents an object or JAVA POJO carrying data. It can also have

logic to update controller if its data changes.

**View Controller Model**

\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Htlm/jsp**

**Business Layer**

**Integration Layer**

**Persistence Layer**

**Some Controller Class**

**Login.jsp**

Some java Class

Or

Bean class

<form action=”login.jlc”>

………

</form>

**Register.html**

DAO interface

<form action=”register.jlc”>

………

</form>

:

**( Known as Helper**

**Controller )**

Some Servlet Class

**DB**

**:**

**Some more JSP**

DAO Class

Or

POJO Class

**<form action=”…..jlc”>**

**………**

**</form>**