**Proxy Pattern**

* **Provides the control for accessing the original object".**
* **we can perform many operations like hiding the information of original object, on demand loading etc.**
* **Proxy pattern is also known as Surrogate or Placeholder.**

#### **Advantage of Proxy Pattern**

* **It provides the protection to the original object from the outside world.**

#### **Example**

#### **Step 1: Create an OfficeInternetAccess interface.**

**public interface OfficeInternetAccess {**

**public void grantInternetAccess();**

**}**

#### **Step 2 :Create a RealInternetAccess class that will implement OfficeInternetAccess interface for granting the permission to the specific employee.**

**public class RealInternetAccess implements OfficeInternetAccess {**

**private String employeeName;**

**public RealInternetAccess(String empName) {**

**this.employeeName = empName;**

**}**

**@Override**

**public void grantInternetAccess() {**

**System.out.println("Internet Access granted for employee: "+ employeeName);**

**}**

**}**

#### **Step 3 : Create a ProxyInternetAccess class that will implement OfficeInternetAccess interface for providing the object of RealInternetAccess class.**

**public class ProxyInternetAccess implements OfficeInternetAccess {**

**private String employeeName;**

**private RealInternetAccess  realaccess;**

**public ProxyInternetAccess(String employeeName) {**

**this.employeeName = employeeName;**

**}**

**@Override**

**public void grantInternetAccess()**

**{**

**if (getRole(employeeName) > 4)**

**{**

**realaccess = new RealInternetAccess(employeeName);**

**realaccess.grantInternetAccess();**

**}**

**else**

**System.out.println("No Internet access granted. Your job level is below 5");**

**}**

**}**

**public int getRole(String emplName) {**

**// Check role from the database based on Name and designation**

**// return job level or job designation.**

**return 9;**

**}**

**}**

#### **Step 4: Now, Create a ProxyPatternClient class that can access the internet actually.**

**public class ProxyPatternClient {**

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

**{**

**OfficeInternetAccess access = new ProxyInternetAccess("Ashwani Rajput");**

**access.grantInternetAccess();**

**}**

**}**

#### **Iterator Pattern**

* **Iterator Pattern is used to access the elements of an aggregate object sequentially without exposing its underlying implementation".**
* **The Iterator pattern is also known as Cursor.**
* **In collection framework, we are now using Iterator that is preferred over Enumeration.**

#### **Advantage of Iterator Pattern**

* **It supports variations in the traversal of a collection.**
* **It simplifies the interface to the collection.**

#### **Usage of Iterator Pattern:**

**It is used:**

* **When you want to access a collection of objects without exposing its internal representation.**
* **When there are multiple traversals of objects need to be supported in the collection.**

#### **Step 1: Create a Iterartor interface.**

**public interface Iterator {**

**public boolean hasNext();**

**public Object next();**

**}**

#### **Step 2: Create a Container interface.**

**public interface Container {**

**public Iterator getIterator();**

**}// End of the Iterator interface.**

#### **Step 3: Create a CollectionofNames class that will implement Container interface.**

**public class CollectionofNames implements Container {**

**public String name[]={"Ashwani Rajput", "Soono Jaiswal","Rishi Kumar","Rahul Mehta","Hemant Mishra"};**

**@Override**

**public Iterator getIterator() {**

**return new CollectionofNamesIterate() ;**

**}**

**private class CollectionofNamesIterate implements Iterator{**

**int i;**

**@Override**

**public boolean hasNext() {**

**if (i<name.length){**

**return true;**

**}**

**return false;**

**}**

**@Override**

**public Object next() {**

**if(this.hasNext()){**

**return name[i++];**

**}**

**return null;**

**}**

**}**

**}**

**}**

#### **Step 4: Create a IteratorPatternDemo class.**

**public class IteratorPatternDemo {**

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

**CollectionofNames cmpnyRepository = new CollectionofNames();**

**for(Iterator iter = cmpnyRepository.getIterator(); iter.hasNext();){**

**String name = (String)iter.next();**

**System.out.println("Name : " + name);**

**}**

**}**

**}**

**TEMPLATE PATTERN**

**In Template pattern, an abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by an abstract class. This pattern comes under behavior pattern category.**

## **Step 1**

**Create an abstract class with a template method being final.**

***Game.java***

**public abstract class Game {**

**abstract void initialize();**

**abstract void startPlay();**

**abstract void endPlay();**

**//template method**

**public final void play(){**

**//initialize the game**

**initialize();**

**//start game**

**startPlay();**

**//end game**

**endPlay();**

**}**

**}**

## **Step 2**

**Create concrete classes extending the above class.**

***Cricket.java***

**public class Cricket extends Game {**

**@Override**

**void endPlay() {**

**System.out.println("Cricket Game Finished!");**

**}**

**@Override**

**void initialize() {**

**System.out.println("Cricket Game Initialized! Start playing.");**

**}**

**@Override**

**void startPlay() {**

**System.out.println("Cricket Game Started. Enjoy the game!");**

**}**

**}**

***Football.java***

**public class Football extends Game {**

**@Override**

**void endPlay() {**

**System.out.println("Football Game Finished!");**

**}**

**@Override**

**void initialize() {**

**System.out.println("Football Game Initialized! Start playing.");**

**}**

**@Override**

**void startPlay() {**

**System.out.println("Football Game Started. Enjoy the game!");**

**}**

**}**

## **Step 3**

**Use the *Game*'s template method play() to demonstrate a defined way of playing game.**

***TemplatePatternDemo.java***

**public class TemplatePatternDemo {**

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

**Game game = new Cricket();**

**game.play();**

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

**game = new Football();**

**game.play();**

**}**

**}**

**Singleton design pattern**

* **Define a class that has only one instance and provides a global point of access to it".**
* **Forms of singleton design pattern**
  + **Early Instantiation: creation of instance at load time.**
  + **Lazy Instantiation: creation of instance when required.**
* **To create the singleton class, we need to have**
  + **Static member:**
  + **Private constructor:**
  + **Static factory method:**

**1)Early Instantiation of Singleton Pattern**

* **Instance of the class is created at the time of declaring the static data member, so instance of the class is created at the time of classloading.**

**class A{**

**private static A obj=new A();//Early, instance will be created at load time**

**private A(){}**

**public static A getA(){**

**return obj;**

**}**

**public void doSomething(){**

**//write your code**

**}**

**}**

**2) lazy Instantiation of Singleton Pattern**

* **Instance of the class is created in synchronized method or synchronized block, so instance of the class is created when required.**

**class A{**

**private static A obj;**

**private A(){}**

**public static A getA(){**

**if (obj == null){**

**synchronized(Singleton.class){**

**if (obj == null){**

**obj = new Singleton();//instance will be created at request time        }**

**}**

**}**

**return obj;**

**}**

**public void doSomething(){**

**//write your code**

**}**

**}**

**Advantage**

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

**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.**

**Prototype pattern**

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

**This pattern should be followed, if the cost of creating a new object is expensive and resource intensive.**

#### **Advantage of Prototype Pattern**

**The main advantages of prototype pattern are as follows:**

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

#### **Usage of Prototype Pattern**

* **When the classes are instantiated at runtime.**
* **When the cost of creating an object is expensive or complicated.**
* **When you want to keep the number of classes in an application minimum.**
* **When the client application needs to be unaware of object creation and representation.**

**Example**

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

#### **Example of Prototype Design Pattern**

**Let's see the example of prototype design pattern.**

**interface Prototype {**

**public Prototype getClone();**

**}//End of Prototype interface.**

***File: EmployeeRecord.java***

**class EmployeeRecord implements Prototype{**

**private int id;**

**private String name, designation;**

**private double salary;**

**private String address;**

**public EmployeeRecord(){**

**System.out.println("   Employee Records of Oracle Corporation ");**

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

**System.out.println("Eid"+"\t"+"Ename"+"\t"+"Edesignation"+"\t"+"Esalary"+"\t\t"+"Eaddress");**

**}**

**public  EmployeeRecord(int id, String name, String designation, double salary, String address) {**

**this();**

**this.id = id;**

**this.name = name;**

**this.designation = designation;**

**this.salary = salary;**

**this.address = address;**

**}**

**public void showRecord(){**

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

**}**

1. **@Override**
2. **public Prototype getClone() {**
4. **return new EmployeeRecord(id,name,designation,salary,address);**
5. **}**
6. **}//End of EmployeeRecord class.**

***File: PrototypeDemo.java***

1. **import java.io.BufferedReader;**
2. **import java.io.IOException;**
3. **import java.io.InputStreamReader;**
5. **class PrototypeDemo{**
6. **public static void main(String[] args) throws IOException {**
8. **BufferedReader br =new BufferedReader(new InputStreamReader(System.in));**
9. **System.out.print("Enter Employee Id: ");**
10. **int eid=Integer.parseInt(br.readLine());**
11. **System.out.print("\n");**
13. **System.out.print("Enter Employee Name: ");**
14. **String ename=br.readLine();**
15. **System.out.print("\n");**
17. **System.out.print("Enter Employee Designation: ");**
18. **String edesignation=br.readLine();**
19. **System.out.print("\n");**
21. **System.out.print("Enter Employee Address: ");**
22. **String eaddress=br.readLine();**
23. **System.out.print("\n");**
25. **System.out.print("Enter Employee Salary: ");**
26. **double esalary= Double.parseDouble(br.readLine());**
27. **System.out.print("\n");**
29. **EmployeeRecord e1=new EmployeeRecord(eid,ename,edesignation,esalary,eaddress);**
31. **e1.showRecord();**
32. **System.out.println("\n");**
33. **EmployeeRecord e2=(EmployeeRecord) e1.getClone();**
34. **e2.showRecord();**
35. **}**
36. **}//End of the ProtoypeDemo class.**

# Factory Method Pattern

**A Factory Pattern or Factory Method Pattern says that just 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.**

**The Factory Method Pattern is also known as Virtual Constructor.**

#### **Advantage of Factory Design Pattern**

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

* **When a class doesn't know what sub-classes will be required to create**
* **When a class wants that its sub-classes specify the objects to be created.**
* **When the parent classes choose the creation of objects to its sub-classes.**

**Example**

**Step 1: Create a Plan abstract class.**

**import java.io.\*;**

**abstract class Plan{**

**protected double rate;**

**abstract void getRate();**

**public void calculateBill(int units){**

**System.out.println(units\*rate);**

**}**

**}//end of Plan class.**

**Step 2: Create the concrete classes that extends Plan abstract class.**

**class  DomesticPlan extends Plan{**

**//@override**

**public void getRate(){**

**rate=3.50;**

**}**

**}//end of DomesticPlan class.**

1. **class  DomesticPlan extends Plan{**
2. **//@override**
3. **public void getRate(){**
4. **rate=3.50;**
5. **}**
6. **}//end of DomesticPlan class.**
7. **class  CommercialPlan extends Plan{**
8. **//@override**
9. **public void getRate(){**
10. **rate=7.50;**
11. **}**
12. **/end of CommercialPlan class.**
13. **class  InstitutionalPlan extends Plan{**
14. **//@override**
15. **public void getRate(){**
16. **rate=5.50;**
17. **}**
18. **/end of InstitutionalPlan class.**

**Step 3: Create a GetPlanFactory to generate object of concrete classes based on given information..**

1. **class GetPlanFactory{**
3. **//use getPlan method to get object of type Plan**
4. **public Plan getPlan(String planType){**
5. **if(planType == null){**
6. **return null;**
7. **}**
8. **if(planType.equalsIgnoreCase("DOMESTICPLAN")) {**
9. **return new DomesticPlan();**
10. **}**
11. **else if(planType.equalsIgnoreCase("COMMERCIALPLAN")){**
12. **return new CommercialPlan();**
13. **}**
14. **else if(planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {**
15. **return new InstitutionalPlan();**
16. **}**
17. **return null;**
18. **}**
19. **}//end of GetPlanFactory class.**

**Step 4: Generate Bill by using the GetPlanFactory to get the object of concrete classes by passing an information such as type of plan DOMESTICPLAN or COMMERCIALPLAN or INSTITUTIONALPLAN.**

1. **import java.io.\*;**
2. **class GenerateBill{**
3. **public static void main(String args[])throws IOException{**
4. **GetPlanFactory planFactory = new GetPlanFactory();**
6. **System.out.print("Enter the name of plan for which the bill will be generated: ");**
7. **BufferedReader br=new BufferedReader(new InputStreamReader(System.in));**
9. **String planName=br.readLine();**
10. **System.out.print("Enter the number of units for bill will be calculated: ");**
11. **int units=Integer.parseInt(br.readLine());**
13. **Plan p = planFactory.getPlan(planName);**
14. **//call getRate() method and calculateBill()method of DomesticPaln.**
16. **System.out.print("Bill amount for "+planName+" of  "+units+" units is: ");**
17. **p.getRate();**
18. **p.calculateBill(units);**
19. **}**
20. **}//end of GenerateBill class.**