Design Pattern

Factory Method Pattern

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.

```
package com.factorydesignpattern;

public class CommonUtils {

	static String language;

	public static CommonUtils getUtils() {

		switch (language.toLowerCase()) {
		case "english": return new EnglishLanguageUtils();
		case "spanish": return new SpanishUtils();
		default: return new EnglishLanguageUtils();
		}
	}

	public void Test() {
		System.out.println("Inside Common Util");
	}
}
```

```java
package com.factorydesignpattern;

public class EnglishLanguageUtils extends CommonUtils {

	@Override
	public void Test(){
		System.out.println("Inside English Utils");
	}
}
```

```java
package com.factorydesignpattern;

public class EnglishLanguageUtils extends CommonUtils {

	@Override
	public void Test(){
		System.out.println("Inside English Utils");
	}
}
```

```java
package com.factorydesignpattern;

public class SpanishUtils extends CommonUtils {

}
```

```java
package com.factorydesignpattern;

public class ClientCode {

	public static void main(String[] args) {
		CommonUtils.language="Spanish";
		CommonUtils.getUtils().Test();
	}

}
```

Singleton Pattern

Singleton Pattern says that just"define a class that has only one instance and provides a global point of access to it".

In other words, a class must ensure that only single instance should be created and single object can be used by all other classes.

There are two forms of singleton design pattern

- **Early Instantiation: creation of instance at load time.**
- **Lazy Instantiation: creation of instance when required.**

Advantage of Singleton design pattern

- 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.

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.

- **Static member:** It gets memory only once because of static, it contains the instance of the Singleton class.
- **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
- **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

`Types:

- Early Instatiation:
  The class instance is created at the time of declaring the static data member,[Created at the time of Class Loading].

```
public class EarlyInstantiation {
	public static void main(String[] args) {
		CreateDriver.getDriver();
	}
}

class CreateDriver{

	private static CreateDriver driver=new CreateDriver();

	//As it is having private constructor we
	//cannot create the object outside it
	private CreateDriver() {
	}

	public static CreateDriver getDriver() {
		return driver;
	}
}
```

- Lazy Initialization:
  Here we create the instance of the class in a **synchronized method** or **synchronized block,**so

```
package com.singleondesign;

public class LazyInstantiation {
	public static void main(String[] args) {
		BankAccount.createBankAccount().calculateTotal(2000);
	}
}

class BankAccount{
	private static BankAccount bankAccount;
	private static int openingSum;
	private BankAccount() {
		openingSum=2000;
	}

	public static BankAccount createBankAccount() {
		if(bankAccount==null) {
			synchronized (BankAccount.class) {
				bankAccount= new BankAccount();
			}
		}
		return bankAccount;
	}

	public void calculateTotal(int initialAmt) {
		int totalamt=openingSum+2000;
		System.out.println(totalamt);
	}

}
```

instance of the class is created when required.

Prototype Design 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.

```
class Employees implements Cloneable{
	private List<String> emplist;

	//Default Constructor
	public Employees() {
		emplist=new ArrayList<String>();
	}

	//Parameterized Constructor to set employee list
	public Employees(List<String> list) {
		this.emplist=list;
	}

	//read all employees from database and put into the list
	public void loadData() {
		emplist.add("Ram");
		emplist.add("Karan");
		emplist.add("Raj");
	}

	public List<String> getEmployeeList(){
		return emplist;
	}

	@Override
	protected Object clone() throws CloneNotSupportedException {
		List<String> tempList=new ArrayList<>();
		for(String s:this.getEmployeeList()) {
			tempList.add(s);
		}
		return new Employees(tempList);
	}
}
```

```
public class ProtypePatternTest {
	public static void main(String[] args) throws CloneNotSupportedException {
		Employees emp=new Employees();
		emp.loadData(); //Loads database values

		Employees empNew1=(Employees) emp.clone();
		Employees empNew2=(Employees) emp.clone();

		List<String> list1=empNew1.getEmployeeList();
		list1.add("Kiran");

		List<String> list2=empNew2.getEmployeeList();
		list2.remove("Ram");

		System.out.println("Original List :"+emp.getEmployeeList());
		System.out.println("empNew1 List :"+list1);
		System.out.println("empNew2 List :"+list2);
	}
```

```
Original List :[Ram, Karan, Raj]
empNew1 List :[Ram, Karan, Raj, Kiran]
empNew2 List :[Karan, Raj]
```