Core Java Design Patterns

In core java, there are mainly three types of design patterns, which are further divided into their sub-parts:

## **1. Creational Design Pattern**

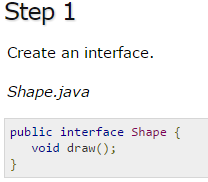
1. **Factory Pattern: -**Factory pattern is one of most used design pattern in Java. In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

## **Implementation**

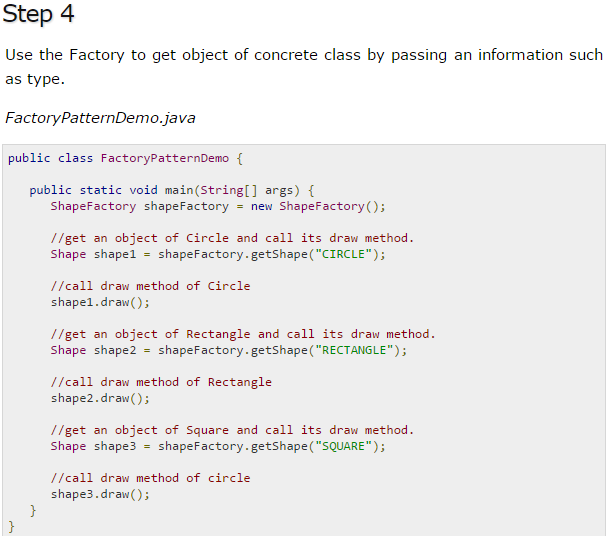
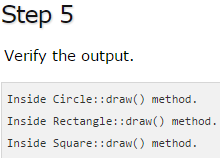
We're going to create a Shape interface and concrete classes implementing theShape interface. A factory class ShapeFactory is defined as a next step.

FactoryPatternDemo, our demo class will use ShapeFactory to get a Shapeobject. It will pass information (CIRCLE / RECTANGLE / SQUARE) toShapeFactory to get the type of object it needs.







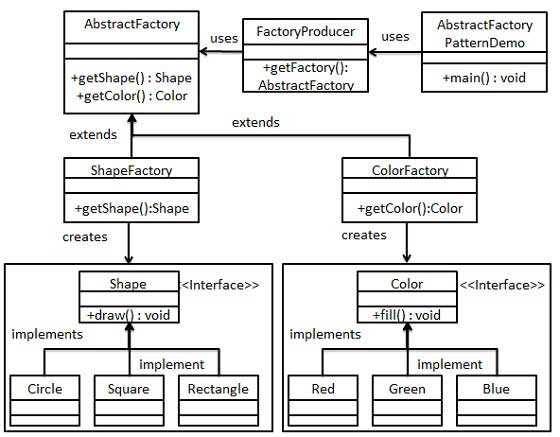
 

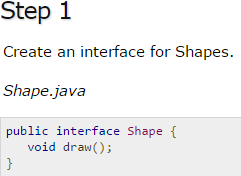
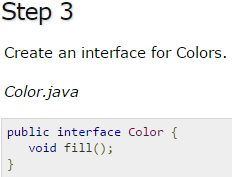
1. **Abstract Factory Pattern: -** In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

## **Implementation**

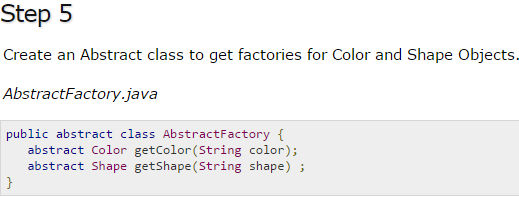
We are going to create a Shape and Color interfaces and concrete classes implementing these interfaces. We create an abstract factory classAbstractFactory as next step. Factory classes ShapeFactory and ColorFactoryare defined where each factory extends AbstractFactory. A factory creator/generator class FactoryProducer is created.

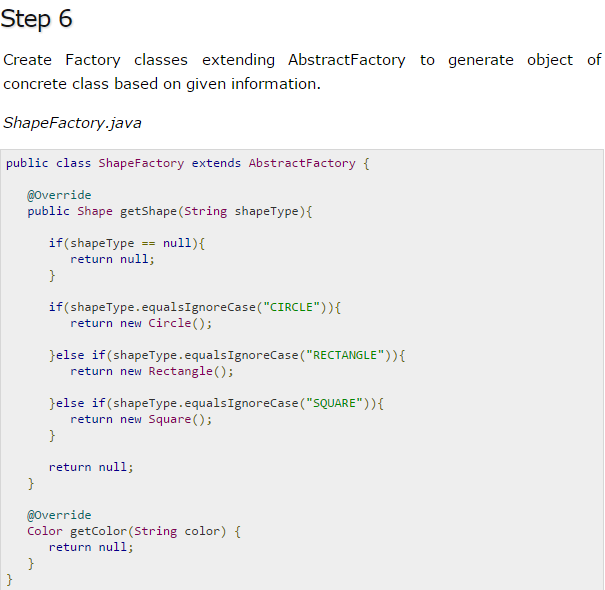
AbstractFactoryPatternDemo, our demo class uses FactoryProducer to get aAbstractFactory object. It will pass information (CIRCLE / RECTANGLE / SQUAREfor Shape) to AbstractFactory to get the type of object it needs. It also passes information (RED / GREEN / BLUE for Color) to AbstractFactory to get the type of object it needs.



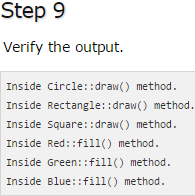
 





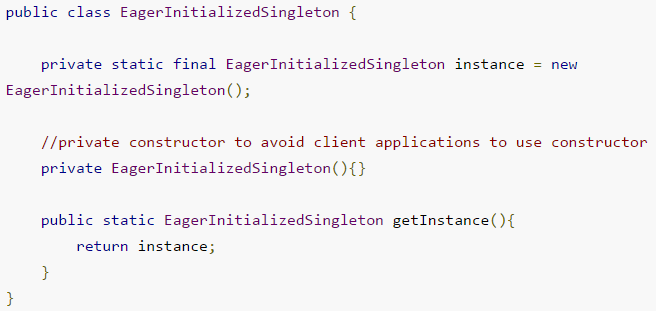
 

1. **Singleton Pattern**

Singleton is a class which has only one instance in whole application and provides a getInstance() method to access the singleton instance. There are many classes in JDK which is implemented using Singleton pattern like java.lang.Runtime which provides getRuntime() method to get access of it and used to get free memory and total memory in Java.

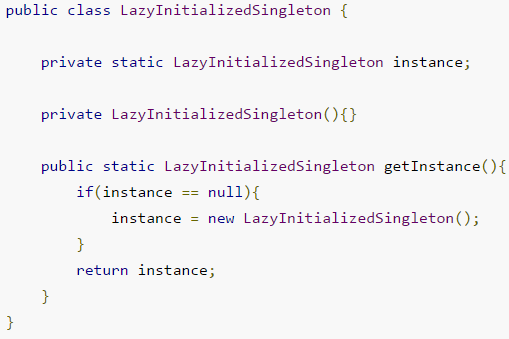
Another example is a utility classes like Popup in GUI application, if you want to show popup with message you can have one PopUp class on whole GUI application and anytime just get its instance, and call show() with message.

**Eager-initialization:**  
In eager initialization, the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a singleton class but it has a drawback that instance is created even though client application might not be using it.



**Lazy-initialization:**

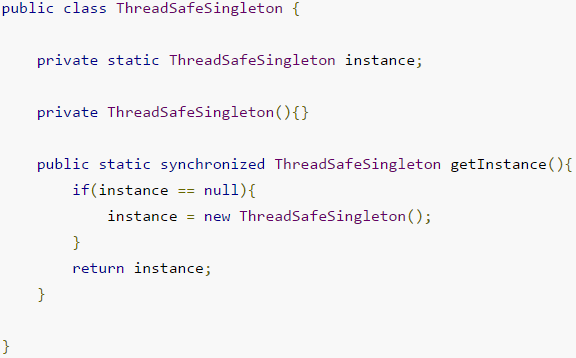
Lazy initialization method to implement Singleton pattern creates the instance in the global access method.



The above implementation works fine in case of single threaded environment but when it comes to multithreaded systems, it can cause issues if multiple threads are inside the if loop at the same time. It will destroy the singleton pattern and both threads will get the different instances of singleton class.

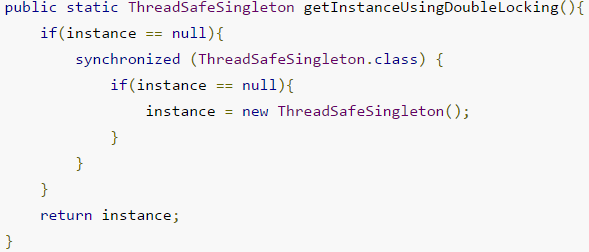
**Thread Safe Singleton:**

The easier way to create a thread-safe singleton class is to make the global access method [synchronized](http://www.journaldev.com/1061/thread-safety-in-java), so that only one thread can execute this method at a time.



**Double Checked Locking Singleton:**

Above implementation works fine and provides thread-safety but it reduces the performance because of cost associated with the synchronized method, although we need it only for the first few threads who might create the separate instances (Read: [Java Synchronization](http://www.journaldev.com/1061/thread-safety-in-java)). To avoid this extra overhead every time, **double checked locking** principle is used. In this approach, the synchronized block is used inside the if condition with an additional check to ensure that only one instance of singleton class is created.



<http://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples>

<http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html>

1. **Prototype Pattern**

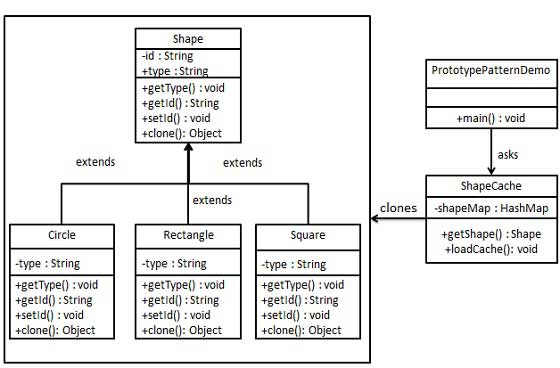
Prototype pattern refers to creating duplicate object while keeping performance in mind.

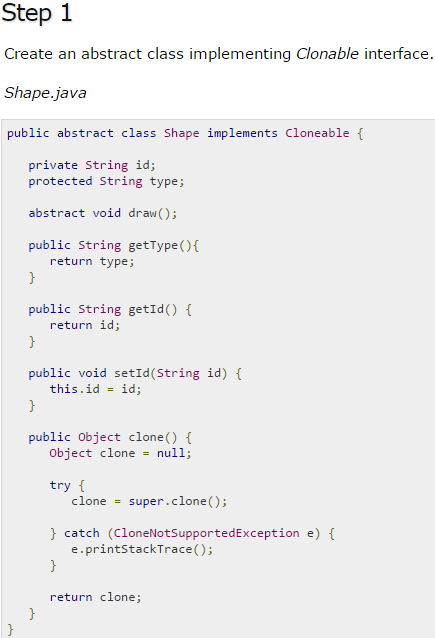
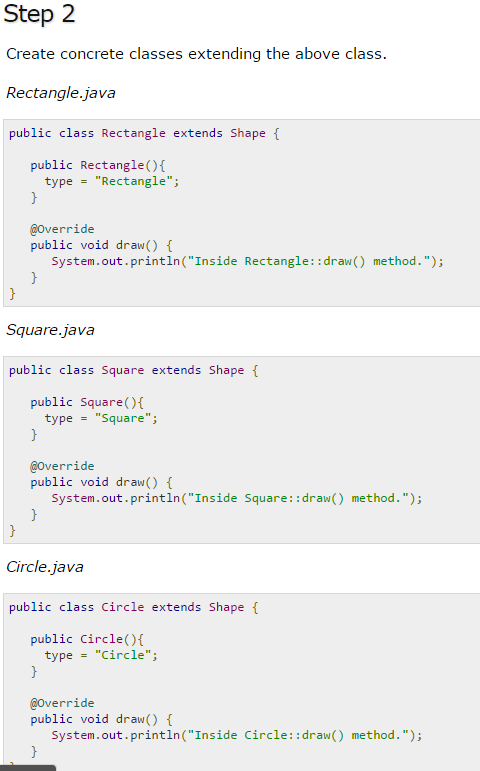
This pattern involves implementing a prototype interface which tells to create a clone of the current object. This pattern is used when creation of object directly is costly. For example, an object is to be created after a costly database operation. We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls.

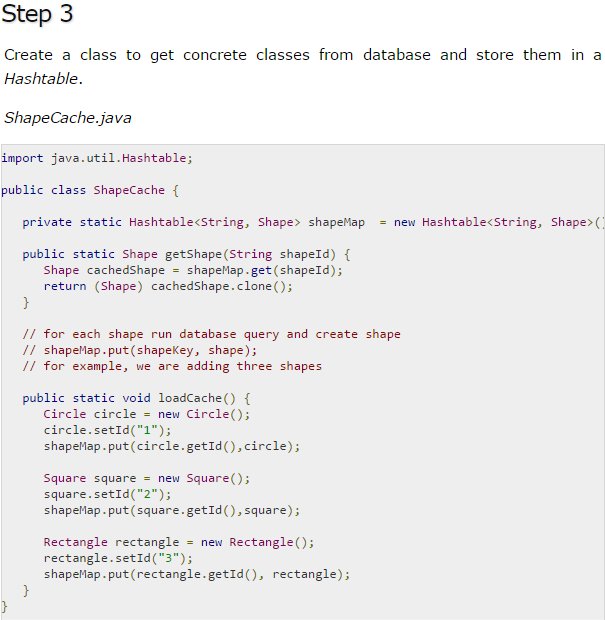
**Implementation**

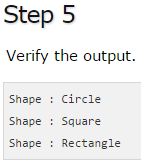
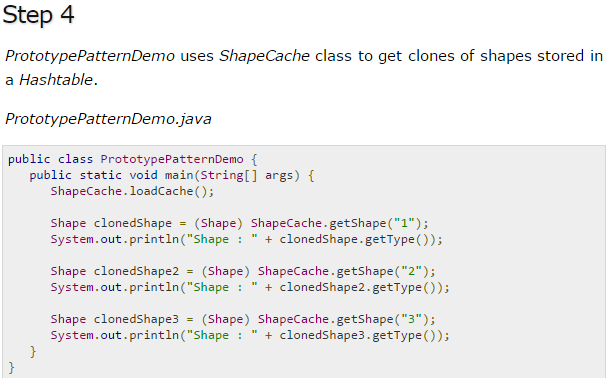
We're going to create an abstract class *Shape* and concrete classes extending the *Shape* class. A class *ShapeCache* is defined as a next step which stores shape objects in a *Hashtable* and returns their clone when requested.

*PrototypPatternDemo*, our demo class will use *ShapeCache* class to get a *Shape* object.







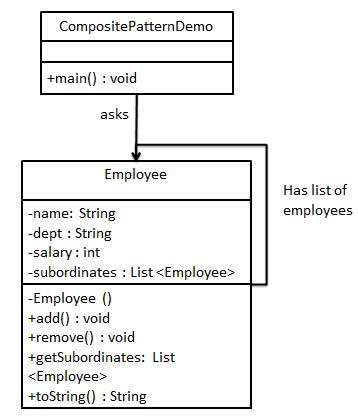
## **2. Structural Design Pattern**

1. Adapter Pattern
2. Bridge Pattern
3. **Composite Pattern:**

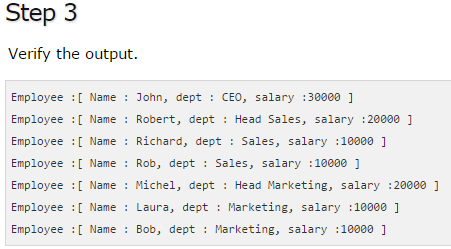
Composite pattern is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy. This pattern creates a class that contains group of its own objects.

**Implementation**

We have a class *Employee* which acts as composite pattern actor class.*CompositePatternDemo*, our demo class will use *Employee* class to add department level hierarchy and print all employees.







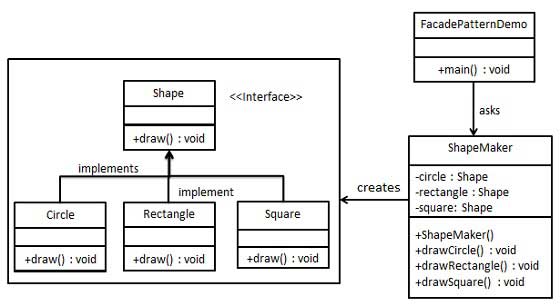
1. **Facade Pattern:**

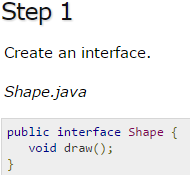
Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

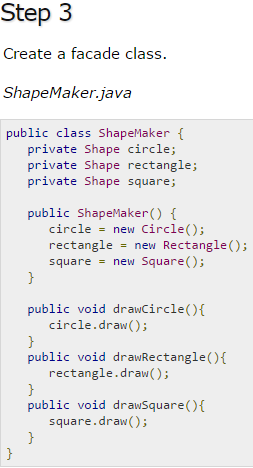
**Implementation**

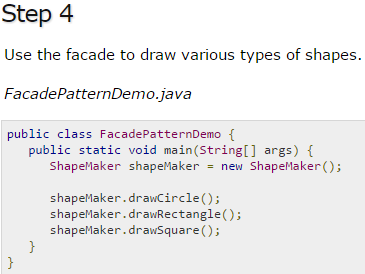
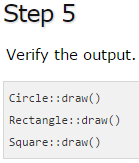
We are going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A facade class *ShapeMaker* is defined as a next step.

*ShapeMaker* class uses the concrete classes to delegate user calls to these classes. *FacadePatternDemo*, our demo class, will use *ShapeMaker* class to show the results.





1. Proxy Pattern

## **3. Behavioral Design Pattern**

1. Chain Of Responsibility Pattern
2. Observer Pattern