**Design Patterns**

Design Patterns are very popular among software developers. A design pattern is a well described solution to a common software problem.

Some of the benefits of using design patterns are:

1. Design Patterns are already defined and provides industry standard approach to solve a recurring problem, so it saves time if we sensibly use the design pattern.
2. Using design patterns promotes reusability that leads to more robust and highly maintainable code. It helps in reducing total cost of ownership (TCO) of the software product.
3. Since design patterns are already defined, it makes our code easy to understand and debug. It leads to faster development and new members of team understand it easily.

Java Design Patterns are divided into three categories – creational, structural, and behavioral design patterns. This post serves as an index for all the java design patterns articles I have written so far.

1. **Creational Design Patterns** Creational design patterns provide solution to instantiate a object in the best possible way for specific situations.

* Singleton Pattern Factory Pattern Abstract Factory Pattern Builder Pattern Prototype Pattern

1. **Structural Design Patterns** Structural patterns provide different ways to create a class structure, for example using inheritance and composition to create a large object from small objects.

* Adapter Pattern, Composite Pattern, Proxy Pattern, Flyweight Pattern, Facade Pattern, Bridge Pattern, Decorator Pattern

1. **Behavioral Design Patterns** Behavioral patterns provide solution for the better interaction between objects and how to provide lose coupling and flexibility to extend easily.

* Template Method Pattern, Mediator Pattern, Chain of Responsibility Pattern, Observer Pattern, Strategy Pattern, Command Pattern, State Pattern, Visitor Pattern, Interpreter Pattern, Iterator Pattern, Memento Pattern

**Singleton Pattern,** Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine. The singleton class must provide a global access point to get the instance of the class. Singleton pattern is used for logging, drivers objects, caching and thread pool.

Singleton design pattern is also used in other design patterns like Abstract Factory, Builder, Prototype, Facade etc. Singleton design pattern is used in core java classes also, for example java.lang.Runtime, java.awt.Desktop.

* Private constructor to restrict instantiation of the class from other classes.
* Private static variable of the same class that is the only instance of the class.
* Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

1. Eager initialization
2. Static block initialization
3. Lazy Initialization
4. Thread Safe Singleton
5. Bill Pugh Singleton Implementation
6. Using Reflection to destroy Singleton Pattern
7. Enum Singleton
8. Serialization and Singleton

**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. Here is the implementation of static initialization singleton class.

package com.journaldev.singleton;

public class EagerInitializedSingleton {

**private static final EagerInitializedSingleton instance** = new EagerInitializedSingleton();

    //private constructor to avoid client applications to use constructor

**private EagerInitializedSingleton(){}**

**public static** EagerInitializedSingleton getInstance(){

        return instance;

    }

}

If your singleton class is not using a lot of resources, this is the approach to use. But in most of the scenarios, Singleton classes are created for resources such as File System, Database connections etc and we should avoid the instantiation until unless client calls the **getInstance** method. Also this method doesn’t provide any options for exception handling.

**Static block initialization,** Static block initialization implementation is similar to eager initialization, except that instance of class is created in the static block that provides option for exception handling.

package com.journaldev.singleton;

public class StaticBlockSingleton {

    private static StaticBlockSingleton instance;

    private StaticBlockSingleton(){}

    //static block initialization for exception handling

    static{

        try{

            instance = new StaticBlockSingleton();

        }catch(Exception e){

            throw new RuntimeException("Exception occured in creating singleton instance");

        }

    }

    public static StaticBlockSingleton getInstance(){

        return instance;

    }

}

Both eager initialization and static block initialization creates the instance even before it’s being used and that is not the best practice to use. So in further sections, we will learn how to create Singleton class that supports lazy initialization.

**Lazy Initialization,** Lazy initialization method to implement Singleton pattern creates the instance in the global access method. Here is the sample code for creating Singleton class with this approach.

package com.journaldev.singleton;

public class LazyInitializedSingleton {

    private static LazyInitializedSingleton instance;

    private LazyInitializedSingleton(){}

    public static LazyInitializedSingleton getInstance(){

        if(instance == null){

            instance = new LazyInitializedSingleton();

        }

        return instance;

    }

}

The above implementation works fine incase 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. In next section, we will see different ways to create a thread-safe singleton class.

**Thread Safe Singleton,** The easier way to create a thread-safe singleton class is to make the global access method synchronized, so that only one thread can execute this method at a time. General implementation of this approach is like the below class.

package com.journaldev.singleton;

public class ThreadSafeSingleton {

    private static ThreadSafeSingleton instance;

    private ThreadSafeSingleton(){}

    public static synchronized ThreadSafeSingleton getInstance(){

        if(instance == null){

            instance = new ThreadSafeSingleton();

        }

        return instance;

    }

}

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). 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.

Below code snippet provides the double checked locking implementation.

public static ThreadSafeSingleton getInstanceUsingDoubleLocking(){

    if(instance == null){

        synchronized (ThreadSafeSingleton.class) {

            if(instance == null){

                instance = new ThreadSafeSingleton();

            }

        }

    }

    return instance;

}

**Bill Pugh Singleton Implementation,** Prior to Java 5, java memory model had a lot of issues and above approaches used to fail in certain scenarios where too many threads try to get the instance of the Singleton class simultaneously. So Bill Pugh came up with a different approach to create the Singleton class using a inner static helper class. The Bill Pugh Singleton implementation goes like this;

package com.journaldev.singleton;

public class BillPughSingleton {

    private BillPughSingleton(){}

    private static class SingletonHelper{

    private static final BillPughSingleton INSTANCE = new BillPughSingleton();

    }

    public static BillPughSingleton getInstance(){

        return SingletonHelper.INSTANCE;

    }

}

Notice the private inner static class that contains the instance of the singleton class. When the singleton class is loaded, SingletonHelper class is not loaded into memory and only when someone calls the getInstance method, this class gets loaded and creates the Singleton class instance.

This is the most widely used approach for Singleton class as it doesn’t require synchronization. I am using this approach in many of my projects and it’s easy to understand and implement also.

**Using Reflection to destroy Singleton Pattern,** Reflection can be used to destroy all the above singleton implementation approaches. Let’s see this with an example class.

package com.journaldev.singleton;

import java.lang.reflect.Constructor;

public class ReflectionSingletonTest {

    public static void main(String[] args) {

        EagerInitializedSingleton instanceOne = EagerInitializedSingleton.getInstance();

        EagerInitializedSingleton instanceTwo = null;

        try {

            Constructor[] constructors = EagerInitializedSingleton.class.getDeclaredConstructors();

            for (Constructor constructor : constructors) {

                //Below code will destroy the singleton pattern

                constructor.setAccessible(true);

                instanceTwo = (EagerInitializedSingleton) constructor.newInstance();

                break;

            }

        } catch (Exception e) {

            e.printStackTrace();

        }

        System.out.println(instanceOne.hashCode());

        System.out.println(instanceTwo.hashCode());

    }

 }

When you run the above test class, you will notice that hashCode of both the instances are not same that destroys the singleton pattern. Reflection is very powerful and used in a lot of frameworks like Spring and Hibernate, do check out Java Reflection Tutorial.

**Enum Singleton,** To overcome this situation with Reflection, Joshua Bloch suggests the use of Enum to implement Singleton design pattern as Java ensures that any enum value is instantiated only once in a Java program. Since Java Enum values are globally accessible, so is the singleton. The drawback is that the enum type is somewhat inflexible; for example, it does not allow lazy initialization.

package com.journaldev.singleton;

public enum EnumSingleton {

    INSTANCE;

    public static void doSomething(){

        //do something

    }

}

**Serialization and Singleton,** Sometimes in distributed systems, we need to implement Serializable interface in Singleton class so that we can store it’s state in file system and retrieve it at later point of time. Here is a small singleton class that implements Serializable interface also.

package com.journaldev.singleton;

import java.io.Serializable;

public class SerializedSingleton implements Serializable{

    private static final long serialVersionUID = -7604766932017737115L;

    private SerializedSingleton(){}

    private static class SingletonHelper{

        private static final SerializedSingleton instance = new SerializedSingleton();

    }

    public static SerializedSingleton getInstance(){

        return SingletonHelper.instance;

    }

  }

The problem with above serialized singleton class is that whenever we deserialize it, it will create a new instance of the class. Let’s see it with a simple program.

package com.journaldev.singleton;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInput;

import java.io.ObjectInputStream;

import java.io.ObjectOutput;

import java.io.ObjectOutputStream;

public class SingletonSerializedTest {

    public static void main(String[] args) throws FileNotFoundException, IOException, ClassNotFoundException {

        SerializedSingleton instanceOne = SerializedSingleton.getInstance();

        ObjectOutput out = new ObjectOutputStream(new FileOutputStream(

                "filename.ser"));

        out.writeObject(instanceOne);

        out.close();

        //deserailize from file to object

        ObjectInput in = new ObjectInputStream(new FileInputStream(

                "filename.ser"));

        SerializedSingleton instanceTwo = (SerializedSingleton) in.readObject();

        in.close();

        System.out.println("instanceOne hashCode="+instanceOne.hashCode());

        System.out.println("instanceTwo hashCode="+instanceTwo.hashCode());

    }

}

instanceOne hashCode=2011117821

instanceTwo hashCode=109647522

So it destroys the singleton pattern, to overcome this scenario all we need to do it provide the implementation of readResolve() method.

protected Object readResolve() {

    return getInstance();

}

After this you will notice that hashCode of both the instances are same in test program.

**Factory Design Pattern,** Factory design pattern is used when we have a super class with multiple sub-classes and based on input, we need to return one of the sub-class. This pattern take out the responsibility of instantiation of a class from client program to the factory class. Let’s first learn how to implement factory pattern in java and then we will learn it’s benefits and we will see its usage in JDK.

**Super Class ,** Super class in factory pattern can be an interface, abstract class or a normal java class. For our example, we have super class as abstract class with overridden toString() method for testing purpose**.**

package com.journaldev.design.model;

public abstract class Computer {

    public abstract String getRAM();

    public abstract String getHDD();

    public abstract String getCPU();

    @Override

    public String toString(){

        return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+", CPU="+this.getCPU();

    }

}

### Sub Classes, Let’s say we have two sub-classes PC and Server with below implementation.

package com.journaldev.design.model;

public class PC extends Computer {

    private String ram;

    private String hdd;

    private String cpu;

    public PC(String ram, String hdd, String cpu){

        this.ram=ram;

        this.hdd=hdd;

        this.cpu=cpu;

    }

    @Override

    public String getRAM() {

        return this.ram;

    }

    @Override

    public String getHDD() {

        return this.hdd;

    }

    @Override

    public String getCPU() {

        return this.cpu;

    }

}

Notice that both the classes are extending Computer class.

package com.journaldev.design.model;

public class Server extends Computer {

    private String ram;

    private String hdd;

    private String cpu;

    public Server(String ram, String hdd, String cpu){

        this.ram=ram;

        this.hdd=hdd;

        this.cpu=cpu;

    }

    @Override

    public String getRAM() {

        return this.ram;

    }

    @Override

    public String getHDD() {

        return this.hdd;

    }

    @Override

    public String getCPU() {

        return this.cpu;

    }

}

**Factory Class,** Now that we have super classes and sub-classes ready, we can write our factory class. Here is the basic implementation.

package com.journaldev.design.factory;

import com.journaldev.design.model.Computer;

import com.journaldev.design.model.PC;

import com.journaldev.design.model.Server;

public class ComputerFactory {

    public static Computer getComputer(String type, String ram, String hdd, String cpu){

        if("PC".equalsIgnoreCase(type)) return new PC(ram, hdd, cpu);

        else if("Server".equalsIgnoreCase(type)) return new Server(ram, hdd, cpu);

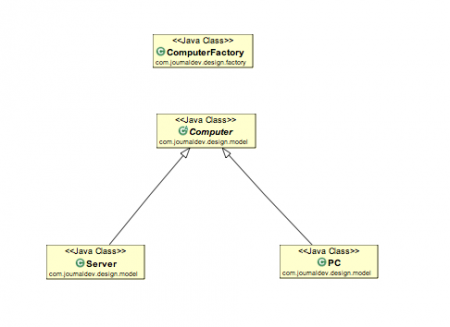
        return null;

    }

}

We can keep Factory class Singleton or we can keep the method that returns the subclass as static.

1. Notice that based on the input parameter, different subclass is created and returned. We can keep Factory class [Singleton](http://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-with-examples) or we can keep the method that returns the subclass as [static](http://www.journaldev.com/1365/static-in-java-methods-variables-block-class).
2. Notice that based on the input parameter, different subclass is created and returned.



Here is a simple test client program that uses above factory pattern implementation.

package com.journaldev.design.test;

import com.journaldev.design.abstractfactory.PCFactory;

import com.journaldev.design.abstractfactory.ServerFactory;

import com.journaldev.design.factory.ComputerFactory;

import com.journaldev.design.model.Computer;

public class TestFactory {

    public static void main(String[] args) {

        Computer pc = ComputerFactory.getComputer("pc","2 GB","500 GB","2.4 GHz");

        Computer server = ComputerFactory.getComputer("server","16 GB","1 TB","2.9 GHz");

        System.out.println("Factory PC Config::"+pc);

        System.out.println("Factory Server Config::"+server);

    }

}

Factory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz

Factory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz

## Benefits of Factory Pattern

1. Factory pattern provides approach to code for interface rather than implementation.
2. Factory pattern removes the instantiation of actual implementation classes from client code, making it more robust, less coupled and easy to extend. For example, we can easily change PC class implementation because client program is unaware of this.
3. Factory pattern provides abstraction between implementation and client classes through inheritance.

## Factory Pattern Examples in JDK

1. java.util.Calendar, ResourceBundle and NumberFormat getInstance() methods uses Factory pattern.
2. valueOf() method in wrapper classes like Boolean, Integer etc.

**Abstract Factory,** Abstract Factory is one of the Creational pattern and almost similar to Factory Pattern except the fact that its more like factory of factories.

If you are familiar with factory design pattern in java, you will notice that we have a single Factory class that returns the different sub-classes based on the input provided and factory class uses if-else or switch statement to achieve this.

In Abstract Factory pattern, we get rid of if-else block and have a factory class for each sub-class and then an Abstract Factory class that will return the sub-class based on the input factory class. At first it seems confusing but once you see the implementation, its really easy to grasp and understand the minor difference between Factory and Abstract Factory pattern.

Like our factory pattern post, we will use the same super class and sub-classes.

package com.journaldev.design.model;

public abstract class Computer {

    public abstract String getRAM();

    public abstract String getHDD();

    public abstract String getCPU();

    @Override

    public String toString(){

        return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+", CPU="+this.getCPU();

    }

}

package com.journaldev.design.model;

public class PC extends Computer {

    private String ram;

    private String hdd;

    private String cpu;

    public PC(String ram, String hdd, String cpu){

        this.ram=ram;

        this.hdd=hdd;

        this.cpu=cpu;

    }

    @Override

    public String getRAM() {

        return this.ram;

    }

    @Override

    public String getHDD() {

        return this.hdd;

    }

    @Override

    public String getCPU() {

        return this.cpu;

    }

}

package com.journaldev.design.model;

public class Server extends Computer {

    private String ram;

    private String hdd;

    private String cpu;

    public Server(String ram, String hdd, String cpu){

        this.ram=ram;

        this.hdd=hdd;

        this.cpu=cpu;

    }

    @Override

    public String getRAM() {

        return this.ram;

    }

    @Override

    public String getHDD() {

        return this.hdd;

    }

    @Override

    public String getCPU() {

        return this.cpu;

    }

}

**Factory Classes for Each sub-class**, First of all we need to create a Abstract Factory interface or abstract class.

package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;

public interface ComputerAbstractFactory {

    public Computer createComputer();

}

Notice that createComputer() method is returning an instance of super class Computer. Now our factory classes will implement this interface and return their respective sub-class.

package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;

import com.journaldev.design.model.PC;

public class PCFactory implements ComputerAbstractFactory {

    private String ram;

    private String hdd;

    private String cpu;

    public PCFactory(String ram, String hdd, String cpu){

        this.ram=ram;

        this.hdd=hdd;

        this.cpu=cpu;

    }

    @Override

    public Computer createComputer() {

        return new PC(ram,hdd,cpu);

    }

}

Similarly we will have a factory class for Server sub-class.

package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;

import com.journaldev.design.model.Server;

public class ServerFactory implements ComputerAbstractFactory {

    private String ram;

    private String hdd;

    private String cpu;

    public ServerFactory(String ram, String hdd, String cpu){

        this.ram=ram;

        this.hdd=hdd;

        this.cpu=cpu;

    }

    @Override

    public Computer createComputer() {

        return new Server(ram,hdd,cpu);

    }

}

Now we will create a consumer class that will provide the entry point for the client classes to create sub-classes.

package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;

public class ComputerFactory {

    public static Computer getComputer(ComputerAbstractFactory factory){

        return factory.createComputer();

    }

}

Notice that its a simple class and getComputer method is accepting ComputerAbstractFactory argument and returning Computer object. At this point the implementation must be getting clear.

Lets write a simple test method and see how to use the abstract factory to get the instance of sub-classes.

package com.journaldev.design.test;

import com.journaldev.design.abstractfactory.PCFactory;

import com.journaldev.design.abstractfactory.ServerFactory;

import com.journaldev.design.factory.ComputerFactory;

import com.journaldev.design.model.Computer;

public class TestDesignPatterns {

    public static void main(String[] args) {

        testAbstractFactory();

    }

    private static void testAbstractFactory() {

        Computer pc = com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new PCFactory("2 GB","500 GB","2.4 GHz"));

        Computer server = com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new ServerFactory("16 GB","1 TB","2.9 GHz"));

        System.out.println("AbstractFactory PC Config::"+pc);

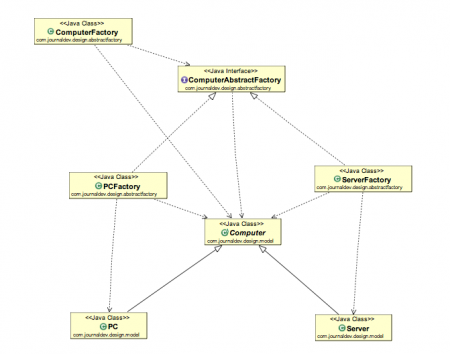
        System.out.println("AbstractFactory Server Config::"+server);

    }

}

AbstractFactory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz

AbstractFactory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz



### Benefits of Abstract Factory Pattern

* Abstract Factory pattern provides approach to code for interface rather than implementation.
* Abstract Factory pattern is “factory of factories” and can be easily extended to accommodate more products, for example we can add another sub-class Laptop and a factory LaptopFactory.
* Abstract Factory pattern is robust and avoid conditional logic of Factory pattern.

### Abstract Factory Pattern Examples in JDK

* javax.xml.parsers.DocumentBuilderFactory#newInstance()
* javax.xml.transform.TransformerFactory#newInstance()
* javax.xml.xpath.XPathFactory#newInstance()

**Builder Design Pattern,** Builder design pattern is a creational design pattern like [Factory Pattern](http://www.journaldev.com/1392/factory-design-pattern-in-java) and [Abstract Factory Pattern](http://www.journaldev.com/1418/abstract-factory-design-pattern-in-java). This pattern was introduced to solve some of the problems with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.

There are three major issues with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.

1. Too Many arguments to pass from client program to the Factory class that can be error prone because most of the time, the type of arguments are same and from client side its hard to maintain the order of the argument.
2. Some of the parameters might be optional but in Factory pattern, we are forced to send all the parameters and optional parameters need to send as NULL.
3. If the object is heavy and its creation is complex, then all that complexity will be part of Factory classes that is confusing.

We can solve the issues with large number of parameters by providing a constructor with required parameters and then different setter methods to set the optional parameters but the problem with this is that the Object state will be inconsistent until unless all the attributes are set explicitly.

Builder pattern solves the issue with large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object.

**Builder Pattern Implementation,**

1. First of all you need to create a [static nested class](http://www.journaldev.com/996/java-nested-classes-java-inner-class-static-nested-class-local-inner-class-and-anonymous-inner-class) and then copy all the arguments from the outer class to the Builder class. We should follow the naming convention and if the class name is Computer then builder class should be named as ComputerBuilder.
2. The Builder class should have a public constructor with all the required attributes as parameters.
3. Builder class should have methods to set the optional parameters and it should return the same Builder object after setting the optional attribute.
4. The final step is to provide a build() method in the builder class that will return the Object needed by client program. For this we need to have a private constructor in the Class with Builder class as argument.

Here is the sample code where we have a Computer class and ComputerBuilder class to build it.

package com.journaldev.design.builder;

public class Computer {

    //required parameters

    private String HDD;

    private String RAM;

    //optional parameters

    private boolean isGraphicsCardEnabled;

    private boolean isBluetoothEnabled;

    public String getHDD() {

        return HDD;

    }

    public String getRAM() {

        return RAM;

    }

    public boolean isGraphicsCardEnabled() {

        return isGraphicsCardEnabled;

    }

    public boolean isBluetoothEnabled() {

        return isBluetoothEnabled;

    }

    private Computer(ComputerBuilder builder) {

        this.HDD=builder.HDD;

        this.RAM=builder.RAM;

        this.isGraphicsCardEnabled=builder.isGraphicsCardEnabled;

        this.isBluetoothEnabled=builder.isBluetoothEnabled;

    }

    //Builder Class

    public static class ComputerBuilder{

        // required parameters

        private String HDD;

        private String RAM;

        // optional parameters

        private boolean isGraphicsCardEnabled;

        private boolean isBluetoothEnabled;

        public ComputerBuilder(String hdd, String ram){

            this.HDD=hdd;

            this.RAM=ram;

        }

        public ComputerBuilder setGraphicsCardEnabled(boolean isGraphicsCardEnabled) {

            this.isGraphicsCardEnabled = isGraphicsCardEnabled;

            return this;

        }

        public ComputerBuilder setBluetoothEnabled(boolean isBluetoothEnabled) {

            this.isBluetoothEnabled = isBluetoothEnabled;

            return this;

        }

        public Computer build(){

            return new Computer(this);

        }

    }

}

Notice that Computer class has only getter methods and no public constructor, so the only way to get a Computer object is through the ComputerBuilder class. Here is a test program showing how to use Builder class to get the object.

package com.journaldev.design.test;

import com.journaldev.design.builder.Computer;

public class TestBuilderPattern {

    public static void main(String[] args) {

        //Using builder to get the object in a single line of code and

                //without any inconsistent state or arguments management issues

        Computer comp = new Computer.ComputerBuilder(

                "500 GB", "2 GB").setBluetoothEnabled(true)

                .setGraphicsCardEnabled(true).build();

    }

}

### Builder Design Pattern Example in JDK

* java.lang.StringBuilder#append() (unsynchronized)
* java.lang.StringBuffer#append() (synchronized)

**Prototype Pattern,** Prototype pattern is one of the Creational Design pattern, so it provides a mechanism of object creation. Prototype pattern is used when the Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing. So this pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. This pattern uses java cloning to copy the object.

It would be easy to understand this pattern with an example, suppose we have an Object that loads data from database. Now we need to modify this data in our program multiple times, so its not a good idea to create the Object using new keyword and load all the data again from database. So the better approach is to clone the existing object into a new object and then do the data manipulation.

Prototype design pattern mandates that the Object which you are copying should provide the copying feature. It should not be done by any other class. However whether to use shallow or deep copy of the Object properties depends on the requirements and its a design decision.

Here is a sample program showing implementation of Prototype pattern.

package com.journaldev.design.prototype;

import java.util.ArrayList;

import java.util.List;

public class Employees implements Cloneable{

    private List<String> empList;

    public Employees(){

        empList = new ArrayList<String>();

    }

    public Employees(List<String> list){

        this.empList=list;

    }

    public void loadData(){

        //read all employees from database and put into the list

        empList.add("Pankaj");

        empList.add("Raj");

        empList.add("David");

        empList.add("Lisa");

    }

    public List<String> getEmpList() {

        return empList;

    }

    @Override

    public Object clone() throws CloneNotSupportedException{

            List<String> temp = new ArrayList<String>();

            for(String s : this.getEmpList()){

                temp.add(s);

            }

            return new Employees(temp);

    }

}

Notice that the clone method is overridden to provide a deep copy of the employees list.

package com.journaldev.design.test;

import java.util.List;

import com.journaldev.design.prototype.Employees;

public class PrototypePatternTest {

    public static void main(String[] args) throws CloneNotSupportedException {

        Employees emps = new Employees();

        emps.loadData();

        //Use the clone method to get the Employee object

        Employees empsNew = (Employees) emps.clone();

        Employees empsNew1 = (Employees) emps.clone();

        List<String> list = empsNew.getEmpList();

        list.add("John");

        List<String> list1 = empsNew1.getEmpList();

        list1.remove("Pankaj");

        System.out.println("emps List: "+emps.getEmpList());

        System.out.println("empsNew List: "+list);

        System.out.println("empsNew1 List: "+list1);

    }

}

emps HashMap: [Pankaj, Raj, David, Lisa]

empsNew HashMap: [Pankaj, Raj, David, Lisa, John]

empsNew1 HashMap: [Raj, David, Lisa]

If the object cloning was not provided, every time we need to make database call to fetch the employee list and then do the manipulations that would have been resource and time consuming.

**Adapter Pattern,** Adapter design pattern is one of the structural design pattern and its used so that two unrelated interfaces can work together. The object that joins these unrelated interface is called an Adapter. As a real life example, we can think of a mobile charger as an adapter because mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India). So the mobile charger works as an adapter between mobile charging socket and the wall socket.

We will try to implement multi-adapter using adapter design pattern in this tutorial.

So first of all we will have two classes – Volt (to measure volts) and Socket (producing constant volts of 120V).

package com.journaldev.design.adapter;

public class Volt {

    private int volts;

    public Volt(int v){

        this.volts=v;

    }

    public int getVolts() {

        return volts;

    }

    public void setVolts(int volts) {

        this.volts = volts;

    }

}

package com.journaldev.design.adapter;

public class Socket {

    public Volt getVolt(){

        return new Volt(120);

    }

}

Now we want to build an adapter that can produce 3 volts, 12 volts and default 120 volts. So first of all we will create an adapter interface with these methods.

package com.journaldev.design.adapter;

public interface SocketAdapter {

    public Volt get120Volt();

    public Volt get12Volt();

    public Volt get3Volt();

}

### Two Way Adapter Pattern

While implementing Adapter pattern, there are two approaches – class adapter and object adapter, however both these approaches produce same result.

1. **Class Adapter** – This form uses [**java inheritance**](http://www.journaldev.com/644/inheritance-in-java-example) and extends the source interface, in our case Socket class.
2. **Object Adapter** – This form uses [**Java Composition**](http://www.journaldev.com/1325/what-is-composition-in-java-java-composition-example) and adapter contains the source object.

**Class Adapter Implementation**

Here is the class adapter approach implementation of our adapter.

package com.journaldev.design.adapter;

//Using inheritance for adapter pattern

public class SocketClassAdapterImpl extends Socket implements SocketAdapter{

    @Override

    public Volt get120Volt() {

        return getVolt();

    }

    @Override

    public Volt get12Volt() {

        Volt v= getVolt();

        return convertVolt(v,10);

    }

    @Override

    public Volt get3Volt() {

        Volt v= getVolt();

        return convertVolt(v,40);

    }

    private Volt convertVolt(Volt v, int i) {

        return new Volt(v.getVolts()/i);

    }

}

**Object Adapter Implementation**, Here is the Object adapter implementation of our adapter.

package com.journaldev.design.adapter;

public class SocketObjectAdapterImpl implements SocketAdapter{

    //Using Composition for adapter pattern

    private Socket sock = new Socket();

    @Override

    public Volt get120Volt() {

        return sock.getVolt();

    }

    @Override

    public Volt get12Volt() {

        Volt v= sock.getVolt();

        return convertVolt(v,10);

    }

    @Override

    public Volt get3Volt() {

        Volt v= sock.getVolt();

        return convertVolt(v,40);

    }

    private Volt convertVolt(Volt v, int i) {

        return new Volt(v.getVolts()/i);

    }

}

Notice that both the adapter implementations are almost same and they implement the SocketAdapterinterface. The adapter interface can also be an [abstract class](http://www.journaldev.com/1582/abstract-class-in-java-with-example).

Here is a test program to consume our adapter implementation.

package com.journaldev.design.test;

import com.journaldev.design.adapter.SocketAdapter;

import com.journaldev.design.adapter.SocketClassAdapterImpl;

import com.journaldev.design.adapter.SocketObjectAdapterImpl;

import com.journaldev.design.adapter.Volt;

public class AdapterPatternTest {

    public static void main(String[] args) {

        testClassAdapter();

        testObjectAdapter();

    }

    private static void testObjectAdapter() {

        SocketAdapter sockAdapter = new SocketObjectAdapterImpl();

        Volt v3 = getVolt(sockAdapter,3);

        Volt v12 = getVolt(sockAdapter,12);

        Volt v120 = getVolt(sockAdapter,120);

        System.out.println("v3 volts using Object Adapter="+v3.getVolts());

        System.out.println("v12 volts using Object Adapter="+v12.getVolts());

        System.out.println("v120 volts using Object Adapter="+v120.getVolts());

    }

    private static void testClassAdapter() {

        SocketAdapter sockAdapter = new SocketClassAdapterImpl();

        Volt v3 = getVolt(sockAdapter,3);

        Volt v12 = getVolt(sockAdapter,12);

        Volt v120 = getVolt(sockAdapter,120);

        System.out.println("v3 volts using Class Adapter="+v3.getVolts());

        System.out.println("v12 volts using Class Adapter="+v12.getVolts());

        System.out.println("v120 volts using Class Adapter="+v120.getVolts());

    }

    private static Volt getVolt(SocketAdapter sockAdapter, int i) {

        switch (i){

        case 3: return sockAdapter.get3Volt();

        case 12: return sockAdapter.get12Volt();

        case 120: return sockAdapter.get120Volt();

        default: return sockAdapter.get120Volt();

        }

    }

}

v3 volts using Class Adapter=3

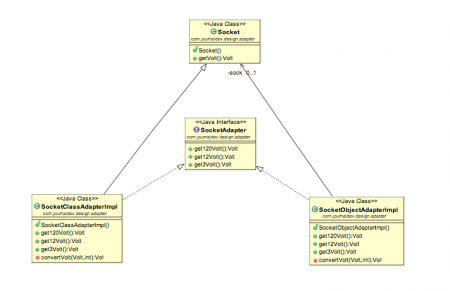
v12 volts using Class Adapter=12

v120 volts using Class Adapter=120

v3 volts using Object Adapter=3

v12 volts using Object Adapter=12

v120 volts using Object Adapter=120



### Adapter Pattern Example in JDK

* java.util.Arrays#asList()
* java.io.InputStreamReader(InputStream) (returns a Reader)
* java.io.OutputStreamWriter(OutputStream) (returns a Writer)

**Composite Pattern,**

Composite pattern is one of the Structural design pattern and is used when we have to represent a part-whole hierarchy. When we need to create a structure in a way that the objects in the structure has to be treated the same way, we can apply composite design pattern.

Lets understand it with a real life example – A diagram is a structure that consists of Objects such as Circle, Lines, Triangle etc and when we fill the drawing with color (say Red), the same color also gets applied to the Objects in the drawing. Here drawing is made up of different parts and they all have same operations.

Composite Pattern consists of following objects.

**Base Component** – Base component is the interface for all objects in the composition, client program uses base component to work with the objects in the composition. It can be an interface or an abstract class with some methods common to all the objects.

**Leaf** – Defines the behaviour for the elements in the composition. It is the building block for the composition and implements base component. It doesn’t have references to other Components.

**Composite** – It consists of leaf elements and implements the operations in base component.

Here I am applying composite design pattern for the drawing scenario.

### Base Component Base component defines the common methods for leaf and composites, we can create a class Shape with a method draw(String fillColor) to draw the shape with given color.

package com.journaldev.design.composite;

public interface Shape {

    public void draw(String fillColor);

}

**Leaf Objects,** Leaf implements base component and these are the building block for the composite. We can create multiple leaf objects such as Triangle, Circle etc.

package com.journaldev.design.composite;

public class Triangle implements Shape {

    @Override

    public void draw(String fillColor) {

        System.out.println("Drawing Triangle with color "+fillColor);

    }

}

package com.journaldev.design.composite;

public class Circle implements Shape {

    @Override

    public void draw(String fillColor) {

        System.out.println("Drawing Circle with color "+fillColor);

    }

}

**Composite,** A composite object contains group of leaf objects and we should provide some helper methods to add or delete leafs from the group. We can also provide a method to remove all the elements from the group.

package com.journaldev.design.composite;

import java.util.ArrayList;

import java.util.List;

public class Drawing implements Shape{

    //collection of Shapes

    private List<Shape> shapes = new ArrayList<Shape>();

    @Override

    public void draw(String fillColor) {

        for(Shape sh : shapes)

        {

            sh.draw(fillColor);

        }

    }

    //adding shape to drawing

    public void add(Shape s){

        this.shapes.add(s);

    }

    //removing shape from drawing

    public void remove(Shape s){

        shapes.remove(s);

    }

    //removing all the shapes

    public void clear(){

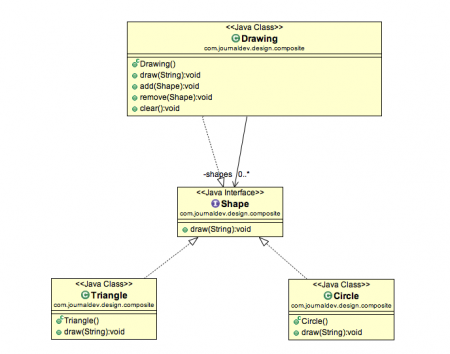
        System.out.println("Clearing all the shapes from drawing");

        this.shapes.clear();

    }

}

Notice that composite also implements component and behaves similar to leaf except that it can contain group of leaf elements.



Our composite pattern implementation is ready and we can test it with a client program.

package com.journaldev.design.test;

import com.journaldev.design.composite.Circle;

import com.journaldev.design.composite.Drawing;

import com.journaldev.design.composite.Shape;

import com.journaldev.design.composite.Triangle;

public class TestCompositePattern {

    public static void main(String[] args) {

        Shape tri = new Triangle();

        Shape tri1 = new Triangle();

        Shape cir = new Circle();

        Drawing drawing = new Drawing();

        drawing.add(tri1);

        drawing.add(tri1);

        drawing.add(cir);

        drawing.draw("Red");

        drawing.clear();

        drawing.add(tri);

        drawing.add(cir);

        drawing.draw("Green");

    }

}

Drawing Triangle with color Red

Drawing Triangle with color Red

Drawing Circle with color Red

Clearing all the shapes from drawing

Drawing Triangle with color Green

Drawing Circle with color Green

### Important Points about Composite Pattern

* Composite pattern should be applied only when the group of objects should behave as the single object.
* Composite pattern can be used to create a tree like structure.

java.awt.Container#add(Component) is a great example of Composite pattern in java and used a lot in Swing

**Proxy Pattern**

Proxy Design pattern is one of the Structural design pattern and in my opinion one of the simplest pattern to understand. Proxy pattern intent according to GoF is:

***“Provide a surrogate or placeholder for another object to control access to it.”***

The definition itself is very clear and proxy pattern is used when we want to provide controlled access of a functionality. Let’s say we have a class that can run some command on the system. Now if we are using it, its fine but if we want to give this program to a client application, it can have severe issues because client program can issue command to delete some system files or change some settings that you don’t want. Here a proxy class can be created to provide controlled access of the program.

**Main Class** Since we code Java in terms of interfaces, here is our interface and its implementation class

package com.journaldev.design.proxy;

public interface CommandExecutor {

    public void runCommand(String cmd) throws Exception;

}

package com.journaldev.design.proxy;

import java.io.IOException;

public class CommandExecutorImpl implements CommandExecutor {

    @Override

    public void runCommand(String cmd) throws IOException {

                //some heavy implementation

        Runtime.getRuntime().exec(cmd);

        System.out.println("'" + cmd + "' command executed.");

    }

}

### Proxy Class, Now we want to provide only admin users to have full access of above class, if the user is not admin then only limited commands will be allowed. Here is our very simple proxy class implementation.

package com.journaldev.design.proxy;

public class CommandExecutorProxy implements CommandExecutor {

    private boolean isAdmin;

    private CommandExecutor executor;

    public CommandExecutorProxy(String user, String pwd){

        if("Pankaj".equals(user) && "J@urnalD$v".equals(pwd)) isAdmin=true;

        executor = new CommandExecutorImpl();

    }

    @Override

    public void runCommand(String cmd) throws Exception {

        if(isAdmin){

            executor.runCommand(cmd);

        }else{

            if(cmd.trim().startsWith("rm")){

                throw new Exception("rm command is not allowed for non-admin users.");

            }else{

                executor.runCommand(cmd);

            }

        }

    }

}

### Proxy Pattern Client Test Program

package com.journaldev.design.test;

import com.journaldev.design.proxy.CommandExecutor;

import com.journaldev.design.proxy.CommandExecutorProxy;

public class ProxyPatternTest {

    public static void main(String[] args){

        CommandExecutor executor = new CommandExecutorProxy("Pankaj", "wrong\_pwd");

        try {

            executor.runCommand("ls -ltr");

            executor.runCommand(" rm -rf abc.pdf");

        } catch (Exception e) {

            System.out.println("Exception Message::"+e.getMessage());

        }

    }

Output of above test program is:

'ls -ltr' command executed.

Exception Message::rm command is not allowed for non-admin users.

Proxy pattern common uses are to control access or to provide a wrapper implementation for better performance. Java RMI whole package uses proxy pattern.

**Flyweight Pattern,** According to GoF, flyweight design pattern intent is:

***“Use sharing to support large numbers of fine-grained objects efficiently”***

Flyweight design pattern is a **Structural design pattern** like [Facade pattern](http://www.journaldev.com/1557/facade-pattern-in-java-example-tutorial), [Adapter Pattern](http://www.journaldev.com/1487/adapter-design-pattern-in-java-example-tutorial) and [Decorator pattern](http://www.journaldev.com/1540/decorator-pattern-in-java-example-tutorial). Flyweight design pattern is used when we need to create a lot of Objects of a class. Since every object consumes memory space that can be crucial for low memory devices, such as mobile devices or embedded systems, flyweight design pattern can be applied to reduce the load on memory by sharing objects.

Before we apply flyweight design pattern, we need to consider following factors:

* The number of Objects to be created by application should be huge.
* The object creation is heavy on memory and it can be time consuming too.
* The object properties can be divided into intrinsic and extrinsic properties, extrinsic properties of an Object should be defined by the client program.

To apply flyweight pattern, we need to divide Object property into intrinsic and extrinsic properties. Intrinsic properties make the Object unique whereas extrinsic properties are set by client code and used to perform different operations. For example, an Object Circle can have extrinsic properties such as color and width.

For applying flyweight pattern, we need to create a Flyweight factory that returns the shared objects. For our example, lets say we need to create a drawing with lines and Ovals. So we will have an interface Shapeand its concrete implementations as Line and Oval. Oval class will have intrinsic property to determine whether to fill the Oval with given color or not whereas Line will not have any intrinsic property.

package com.journaldev.design.flyweight;

import java.awt.Color;

import java.awt.Graphics;

public interface Shape {

    public void draw(Graphics g, int x, int y, int width, int height,

            Color color);

}

package com.journaldev.design.flyweight;

import java.awt.Color;

import java.awt.Graphics;

public class Line implements Shape {

    public Line(){

        System.out.println("Creating Line object");

        //adding time delay

        try {

            Thread.sleep(2000);

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

    @Override

    public void draw(Graphics line, int x1, int y1, int x2, int y2,

            Color color) {

        line.setColor(color);

        line.drawLine(x1, y1, x2, y2);

    }

}

package com.journaldev.design.flyweight;

import java.awt.Color;

import java.awt.Graphics;

public class Oval implements Shape {

    //intrinsic property

    private boolean fill;

    public Oval(boolean f){

        this.fill=f;

        System.out.println("Creating Oval object with fill="+f);

        //adding time delay

        try {

            Thread.sleep(2000);

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

    @Override

    public void draw(Graphics circle, int x, int y, int width, int height,

            Color color) {

        circle.setColor(color);

        circle.drawOval(x, y, width, height);

        if(fill){

            circle.fillOval(x, y, width, height);

        }

    }

}

Notice that I have intensionally introduced delay in creating the Object of concrete classes to make the point that flyweight pattern can be used for Objects that takes a lot of time while instantiated.

**Flyweight Factory**

The flyweight factory will be used by client programs to instantiate the Object, so we need to keep a map of Objects in the factory that should not be accessible by client application. Whenever client program makes a call to get an instance of Object, it should be returned from the HashMap, if not found then create a new Object and put in the Map and then return it. We need to make sure that all the intrinsic properties are considered while creating the Object.

Our flyweight factory class looks like below code.

package com.journaldev.design.flyweight;

import java.util.HashMap;

public class ShapeFactory {

    private static final HashMap<ShapeType,Shape> shapes = new HashMap<ShapeType,Shape>();

    public static Shape getShape(ShapeType type) {

        Shape shapeImpl = shapes.get(type);

        if (shapeImpl == null) {

            if (type.equals(ShapeType.OVAL\_FILL)) {

                shapeImpl = new Oval(true);

            } else if (type.equals(ShapeType.OVAL\_NOFILL)) {

                shapeImpl = new Oval(false);

            } else if (type.equals(ShapeType.LINE)) {

                shapeImpl = new Line();

            }

            shapes.put(type, shapeImpl);

        }

        return shapeImpl;

    }

    public static enum ShapeType{

        OVAL\_FILL,OVAL\_NOFILL,LINE;

    }

}

Notice the use of Java Enum for type safety, Java Composition (shapes map) and Factory pattern in getShape method.

**Flyweight Pattern Client Example**

package com.journaldev.design.flyweight;

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Container;

import java.awt.Graphics;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;

import com.journaldev.design.flyweight.ShapeFactory.ShapeType;

public class DrawingClient extends JFrame{

    private static final long serialVersionUID = -1350200437285282550L;

    private final int WIDTH;

    private final int HEIGHT;

    private static final ShapeType shapes[] = { ShapeType.LINE, ShapeType.OVAL\_FILL,ShapeType.OVAL\_NOFILL };

    private static final Color colors[] = { Color.RED, Color.GREEN, Color.YELLOW };

    public DrawingClient(int width, int height){

        this.WIDTH=width;

        this.HEIGHT=height;

        Container contentPane = getContentPane();

        JButton startButton = new JButton("Draw");

        final JPanel panel = new JPanel();

        contentPane.add(panel, BorderLayout.CENTER);

        contentPane.add(startButton, BorderLayout.SOUTH);

        setSize(WIDTH, HEIGHT);

        setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

        setVisible(true);

        startButton.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent event) {

                Graphics g = panel.getGraphics();

                for (int i = 0; i < 20; ++i) {

                    Shape shape = ShapeFactory.getShape(getRandomShape());

                    shape.draw(g, getRandomX(), getRandomY(), getRandomWidth(),

                            getRandomHeight(), getRandomColor());

                }

            }

        });

    }

    private ShapeType getRandomShape() {

        return shapes[(int) (Math.random() \* shapes.length)];

    }

    private int getRandomX() {

        return (int) (Math.random() \* WIDTH);

    }

    private int getRandomY() {

        return (int) (Math.random() \* HEIGHT);

    }

    private int getRandomWidth() {

        return (int) (Math.random() \* (WIDTH / 10));

    }

    private int getRandomHeight() {

        return (int) (Math.random() \* (HEIGHT / 10));

    }

    private Color getRandomColor() {

        return colors[(int) (Math.random() \* colors.length)];

    }

    public static void main(String[] args) {

        DrawingClient drawing = new DrawingClient(500,600);

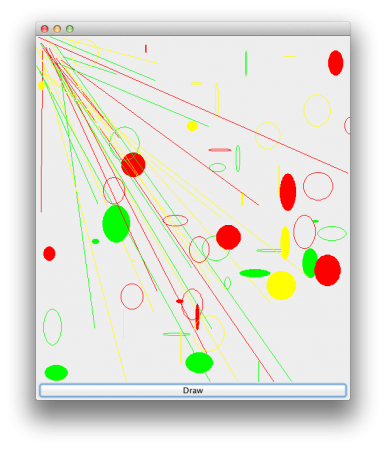
    }

}

I have used random number generation to generate different type of Shapes in our frame.

If you run above client program, you will notice the delay in creating first Line Object and Oval objects with fill as true and false. After that the program executes quickly since its using the shared objects.

After clicking “Draw” button multiple times, the frame looks like below image.



And you will see following output in command line confirming that Objects are shared.

Creating Line object

Creating Oval object with fill=true

Creating Oval object with fill=false

Thats all for flyweight pattern, we will look into more design patterns in future posts. If you liked it, please share your thoughts in comments section and share it with others too.

**Flyweight Pattern Example in JDK**

All the wrapper classes valueOf() method uses cached objects showing use of Flyweight design pattern. The best example is Java String class String Pool implementation.

**Important Points**

In our example, the client code is not forced to create object using Flyweight factory but we can force that to make sure client code uses flyweight pattern implementation but its a complete design decision for particular application.

Flyweight pattern introduces complexity and if number of shared objects are huge then there is a trade of between memory and time, so we need to use it judiciously based on our requirements.

Flyweight pattern implementation is not useful when the number of intrinsic properties of Object is huge, making implementation of Factory class complex.

**Facade Pattern,** Facade Pattern is one of the Structural design patterns (such as Adapter pattern and Decorator pattern) and used to help client applications to easily interact with the system.

“***Provide a unified interface to a set of interfaces in a subsystem. Facade Pattern defines a higher-level interface that makes the subsystem easier to use.”***

Suppose we have an application with set of interfaces to use MySql/Oracle database and to generate different types of reports, such as HTML report, PDF report etc. So we will have different set of interfaces to work with different types of database. Now a client application can use these interfaces to get the required database connection and generate reports. But when the complexity increases or the interface behavior names are confusing, client application will find it difficult to manage it. So we can apply Facade pattern here and provide a [wrapper](http://www.journaldev.com/1002/java-wrapper-classes-tutorial-with-examples) interface on top of the existing interface to help client application.

**Set of Interfaces**

package com.journaldev.design.facade;

import java.sql.Connection;

public class MySqlHelper {

    public static Connection getMySqlDBConnection(){

        //get MySql DB connection using connection parameters

        return null;

    }

    public void generateMySqlPDFReport(String tableName, Connection con){

        //get data from table and generate pdf report

    }

    public void generateMySqlHTMLReport(String tableName, Connection con){

        //get data from table and generate pdf report

    }

}

package com.journaldev.design.facade;

import java.sql.Connection;

public class OracleHelper {

    public static Connection getOracleDBConnection(){

        //get Oracle DB connection using connection parameters

        return null;

    }

    public void generateOraclePDFReport(String tableName, Connection con){

        //get data from table and generate pdf report

    }

    public void generateOracleHTMLReport(String tableName, Connection con){

        //get data from table and generate pdf report

    }

}

**Facade Interface,**

package com.journaldev.design.facade;

import java.sql.Connection;

public class HelperFacade {

    public static void generateReport(DBTypes dbType, ReportTypes reportType, String tableName){

        Connection con = null;

        switch (dbType){

        case MYSQL:

            con = MySqlHelper.getMySqlDBConnection();

            MySqlHelper mySqlHelper = new MySqlHelper();

            switch(reportType){

            case HTML:

                mySqlHelper.generateMySqlHTMLReport(tableName, con);

                break;

            case PDF:

                mySqlHelper.generateMySqlPDFReport(tableName, con);

                break;

            }

            break;

        case ORACLE:

            con = OracleHelper.getOracleDBConnection();

            OracleHelper oracleHelper = new OracleHelper();

            switch(reportType){

            case HTML:

                oracleHelper.generateOracleHTMLReport(tableName, con);

                break;

            case PDF:

                oracleHelper.generateOraclePDFReport(tableName, con);

                break;

            }

            break;

        }

    }

    public static enum DBTypes{

        MYSQL,ORACLE;

    }

    public static enum ReportTypes{

        HTML,PDF;

    }

}

Client Program,

package com.journaldev.design.test;

import java.sql.Connection;

import com.journaldev.design.facade.HelperFacade;

import com.journaldev.design.facade.MySqlHelper;

import com.journaldev.design.facade.OracleHelper;

public class FacadePatternTest {

    public static void main(String[] args) {

        String tableName="Employee";

        //generating MySql HTML report and Oracle PDF report without using Facade

        Connection con = MySqlHelper.getMySqlDBConnection();

        MySqlHelper mySqlHelper = new MySqlHelper();

        mySqlHelper.generateMySqlHTMLReport(tableName, con);

        Connection con1 = OracleHelper.getOracleDBConnection();

        OracleHelper oracleHelper = new OracleHelper();

        oracleHelper.generateOraclePDFReport(tableName, con1);

        //generating MySql HTML report and Oracle PDF report using Facade

        HelperFacade.generateReport(HelperFacade.DBTypes.MYSQL, HelperFacade.ReportTypes.HTML, tableName);

        HelperFacade.generateReport(HelperFacade.DBTypes.ORACLE, HelperFacade.ReportTypes.PDF, tableName);

    }

}

As you can see that using Facade interface is a lot easier and cleaner way and avoid having a lot of logic at client side. JDBC Driver Manager class to get the database connection is a wonderful example of facade pattern.

### Important Points

* Facade pattern is more like a helper for client applications, it doesn’t hide subsystem interfaces from the client. Whether to use Facade or not is completely dependent on client code.
* Facade pattern can be applied at any point of development, usually when the number of interfaces grow and system gets complex.
* Subsystem interfaces are not aware of Facade and they shouldn’t have any reference of the Facade interface.
* Facade pattern should be applied for similar kind of interfaces, its purpose is to provide a single interface rather than multiple interfaces that does the similar kind of jobs.
* We can use [Factory pattern](http://www.journaldev.com/1392/factory-design-pattern-in-java) with Facade to provide better interface to client systems.

Thats all for Facade pattern, stay tuned for more design pattern articles.

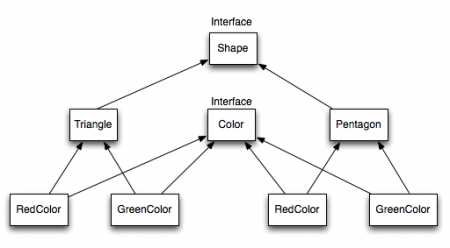
**Bridge Pattern ,**

When we have interface hierarchies in both interfaces as well as implementations, then builder design pattern is used to decouple the interfaces from implementation and hiding the implementation details from the client programs. Like Adapter pattern, its one of the Structural design pattern.

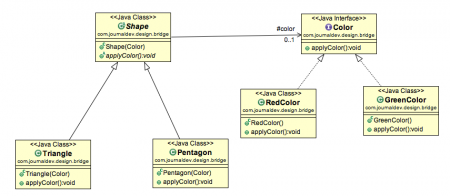
According to GoF bridge design pattern is:

***“Decouple an abstraction from its implementation so that the two can vary independently”***

If we look into this design pattern with example, it will be easy to understand. Lets say we have an interface hierarchy in both interfaces and implementations like below image.



Now we will use bridge design pattern to decouple the interfaces from implementation and the UML diagram for the classes and interfaces after applying bridge pattern will look like below image.



Notice the bridge between Shape and Color interfaces and use of composition in implementing the bridge pattern. Here is the java code for Shape and Color interfaces.

package com.journaldev.design.bridge;

public interface Color {

    public void applyColor();

}

package com.journaldev.design.bridge;

public abstract class Shape {

    //Composition - implementor

    protected Color color;

    //constructor with implementor as input argument

    public Shape(Color c){

        this.color=c;

    }

    abstract public void applyColor();

}

package com.journaldev.design.bridge;

public class Triangle extends Shape{

    public Triangle(Color c) {

        super(c);

    }

    @Override

    public void applyColor() {

        System.out.print("Triangle filled with color ");

        color.applyColor();

    }

}

package com.journaldev.design.bridge;

public class Pentagon extends Shape{

    public Pentagon(Color c) {

        super(c);

    }

    @Override

    public void applyColor() {

        System.out.print("Pentagon filled with color ");

        color.applyColor();

    }

}

package com.journaldev.design.bridge;

public class RedColor implements Color{

    public void applyColor(){

        System.out.println("red.");

    }

}

package com.journaldev.design.bridge;

public class GreenColor implements Color{

    public void applyColor(){

        System.out.println("green.");

    }

}

package com.journaldev.design.test;

import com.journaldev.design.bridge.GreenColor;

import com.journaldev.design.bridge.Pentagon;

import com.journaldev.design.bridge.RedColor;

import com.journaldev.design.bridge.Shape;

import com.journaldev.design.bridge.Triangle;

public class BridgePatternTest {

    public static void main(String[] args) {

        Shape tri = new Triangle(new RedColor());

        tri.applyColor();

        Shape pent = new Pentagon(new GreenColor());

        pent.applyColor();

    }

}

Triangle filled with color red.

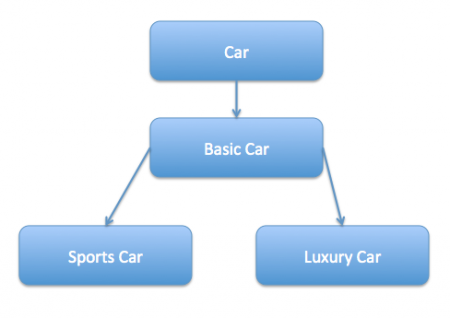
Pentagon filled with color green.

Bridge design pattern can be used when both abstraction and implementation can have different hierarchies independently and we want to hide the implementation from the client application.

**Decorator Pattern,** Decorator design pattern is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behavior. Decorator design pattern is one of the structural design pattern (such as Adapter Pattern, Bridge Pattern, Composite Pattern) and uses abstract classes or interface with composition to implement.

We use inheritance or composition to extend the behavior of an object but this is done at compile time and its applicable to all the instances of the class. We can’t add any new functionality of remove any existing behavior at runtime – this is when Decorator pattern comes into picture.

Suppose we want to implement different kinds of cars – we can create interface Car to define the assemble method and then we can have a Basic car, further more we can extend it to Sports car and Luxury Car. The implementation hierarchy will look like below image.



But if we want to get a car at runtime that has both the features of sports car and luxury car, then the implementation gets complex and if further more we want to specify which features should be added first, it gets even more complex. Now image if we have ten different kind of cars, the implementation logic using inheritance and composition will be impossible to manage. To solve this kind of programming situation, we apply decorator pattern.

We need to have following types to implement decorator design pattern.

**1. Component Interface** – The interface or [abstract class](http://www.journaldev.com/1582/abstract-class-in-java-with-example) defining the methods that will be implemented. In our case Car will be the component interface.

package com.journaldev.design.decorator;

public interface Car {

    public void assemble();

}

**2.Component Implementation** – The basic implementation of the component interface. We can haveBasicCar class as our component implementation.

package com.journaldev.design.decorator;

public class BasicCar implements Car {

    @Override

    public void assemble() {

        System.out.print("Basic Car.");

    }

}

**3.Decorator** – Decorator class implements the component interface and it has a HAS-A relationship with the component interface. The component variable should be accessible to the child decorator classes, so we will make this variable protected.

package com.journaldev.design.decorator;

public class CarDecorator implements Car {

    protected Car car;

    public CarDecorator(Car c){

        this.car=c;

    }

    @Override

    public void assemble() {

        this.car.assemble();

    }

}

**4.Concrete Decorators** – Extending the base decorator functionality and modifying the component behavior accordingly. We can have concrete decorator classes as LuxuryCar and SportsCar.

package com.journaldev.design.decorator;

public class SportsCar extends CarDecorator {

    public SportsCar(Car c) {

        super(c);

    }

    @Override

    public void assemble(){

        car.assemble();

        System.out.print(" Adding features of Sports Car.");

    }

}

package com.journaldev.design.decorator;

public class LuxuryCar extends CarDecorator {

    public LuxuryCar(Car c) {

        super(c);

    }

    @Override

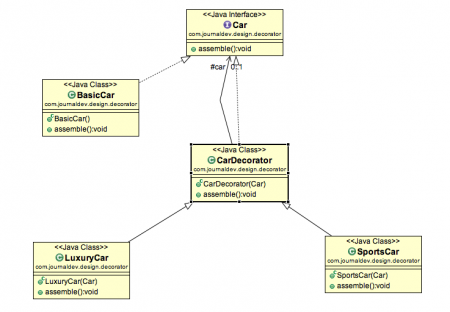
    public void assemble(){

        car.assemble();

        System.out.print(" Adding features of Luxury Car.");

    }

}



**Decorator Pattern Client Program**

package com.journaldev.design.test;

import com.journaldev.design.decorator.BasicCar;

import com.journaldev.design.decorator.Car;

import com.journaldev.design.decorator.LuxuryCar;

import com.journaldev.design.decorator.SportsCar;

public class DecoratorPatternTest {

    public static void main(String[] args) {

        Car sportsCar = new SportsCar(new BasicCar());

        sportsCar.assemble();

        System.out.println("\n\*\*\*\*\*");

        Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new BasicCar()));

        sportsLuxuryCar.assemble();

    }

}

Basic Car. Adding features of Sports Car.

\*\*\*\*\*

Basic Car. Adding features of Luxury Car. Adding features of Sports Car.

### Important Points

* Decorator pattern is helpful in providing runtime modification abilities and hence more flexible. Its easy to maintain and extend when the number of choices are more.
* The disadvantage of decorator pattern is that it uses a lot of similar kind of objects (decorators).
* Decorator pattern is used a lot in [Java IO](http://www.journaldev.com/942/java-io-tutorial) classes, such as [FileReader, BufferedReader](http://www.journaldev.com/867/how-to-read-file-in-java-using-bufferedreader-scanner-files-with-encoding-support-and-filereader) etc.