**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);

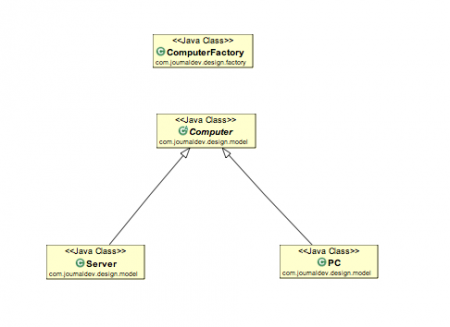
        return null;

    }

}

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.