# **Design Pattern**

Design patterns represent the best practices used by experienced object-oriented software developers. **Design patterns are solutions to general problems that software developers faced during software development**.

## What is Gang of Four (GOF)?

In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development.

These authors are collectively known as **Gang of Four (GOF)**.

## Types of Design Patterns

As per the design pattern reference book **Design Patterns - Elements of Reusable Object-Oriented Software** , there are 23 design patterns which can be classified in three categories: Creational, Structural and Behavioral patterns. We'll also discuss another category of design pattern: J2EE design patterns.

|  |  |
| --- | --- |
| **S.N.** | **Pattern & Descrip­­tion** |
| 1 | **Creational Patterns** **These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator.** This gives program more flexibility in deciding which objects need to be created for a given use case. |
| 2 | **Structural Patterns** **These design patterns concern class and object composition.** Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities. |
| 3 | **Behavioral Patterns** **These design patterns are specifically concerned with communication between objects.** |
| 4 | **J2EE Patterns** T**hese design patterns are specifically concerned with the presentation tier**. These patterns are identified by Sun Java Center. |

**1)Singleton Design Pattern**

**Java Singleton Pattern** is one of the **Gangs of Four Design patterns** and comes in the **Creational** [**Design Pattern**](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) category

* **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](https://www.journaldev.com/977/logger-in-java-logging-example), drivers objects, caching and [thread pool](https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice).

To implement Singleton pattern, we have different approaches but all of them have following common concepts.

* **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.**

In further sections, we will learn different approaches of Singleton pattern implementation and design concerns with the implementation.

1. [Eager initialization](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#eager-initialization)
2. [Static block initialization](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#static-block-initialization)
3. [Lazy Initialization](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#lazy-initialization)
4. [Thread Safe Singleton](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#thread-safe-singleton)
5. [Bill Pugh Singleton Implementation](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#bill-pugh-singleton)
6. [Using Reflection to destroy Singleton Pattern](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#reflection-and-singleton)
7. [Enum Singleton](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#enum-singleton)
8. [Serialization and Singleton](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#serialization-and-singleton)
9. Clone & Singleton

### **1.1) 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.

### **1.2) Static block initialization**

[Static block](https://www.journaldev.com/1365/static-keyword-in-java) initialization implementation is similar to eager initialization, except that instance of class is created in the static block that provides option for [exception handling](https://www.journaldev.com/1696/exception-handling-in-java).

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

### **1.3) 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](https://www.journaldev.com/1061/thread-safety-in-java) singleton class

### **1.4) Thread Safe Singleton**

The easier way to create a thread-safe singleton class is to make the global access method [synchronized](https://www.journaldev.com/1061/thread-safety-in-java), 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](https://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.

**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;

}

### **1.5) 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](https://www.journaldev.com/996/java-inner-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.**

### **1.6) 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

### **1.7) 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](https://www.journaldev.com/716/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

}

}

### **1.8) 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());

}

}

Output of the above program is;

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

**2) Decorator Design Pattern**

**Decorator pattern allows adding new functionality an existing object without altering its structure.** **This type of design pattern comes under structural pattern** as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

**3) Strategy Design Pattern**

In Strategy pattern, a class behavior or its algorithm can be changed at run time. **This type of design pattern comes under behavior pattern.**

**4) Chain of Responsibility Design Pattern**

As the name suggest, the chain of responsibility pattern creates a chain of receiver objects for a request. This pattern decouples sender and receiver of a request based on type of request. **This pattern comes under behavioral patterns**.

**In this pattern, normally each receiver contains reference to another receiver.** If one object cannot handle the request then it passes the same to the next receiver and so on.

**5) Builder Design Pattern**

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. The problem with this approach 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.

Let’s see how we can implement builder design pattern in java.

1. First of all you need to create a [static nested class](https://www.journaldev.com/996/java-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 Computerthen builder class should be named as ComputerBuilder.
2. Java Builder class should have a public constructor with all the required attributes as parameters.
3. Java 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.

**Refer :**

[**https://www.javatpoint.com/design-patterns-in-java**](https://www.javatpoint.com/design-patterns-in-java)