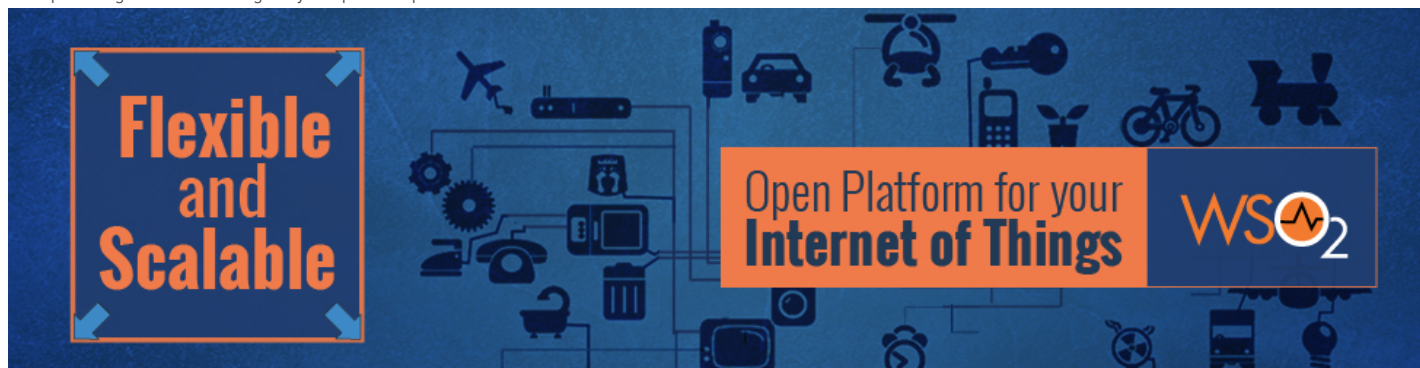


Enterprise Integration Zone is brought to you in partnership with:



Welcome back, *Sinha.sonal.kumar*.

► **IoT is HoT! The Internet of Things will revolutionize your world.**



MAINAK GOSWAMI

Bio

Website

@idiotechie



Singleton Design Pattern – An Introspection w/ Best Practices

02.13.2013 | 266550 VIEWS | Like {52} Tweet {18} g+ {17} SHARE 14

The [Enterprise Integration Zone](#) is brought to you in partnership with [WSO2](#). Learn more about [WSO2's API Management](#).

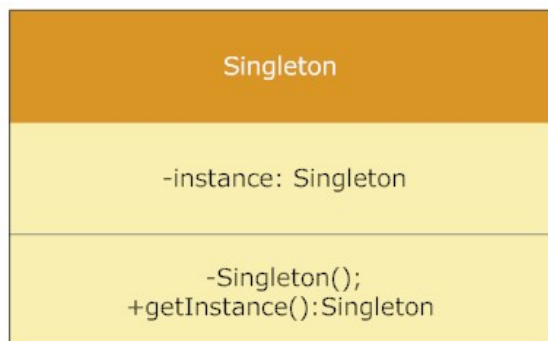
Definition:

Singleton is a part of **Gang of Four design** pattern and it is categorized under creational design patterns.

In this article we are going to take a deeper look into the usage of the Singleton pattern. It is one of the most simple design pattern in terms of the modelling but on the other hand this is one of the most controversial pattern in terms of complexity of usage.

In Java the Singleton pattern will ensure that there is only one instance of a class is created in the Java Virtual Machine. It is used to provide global point of access to the object. In terms of practical use Singleton patterns are used in logging, caches, thread pools, configuration settings, device driver objects. Design pattern is often used in conjunction with [Factory design pattern](#). This pattern is also used in Service Locator JEE pattern.

Structure:



Singleton Class Diagram

 [Publish an Article](#)

✉ Share a Tip

Connect with DZone



RELATED MICROZONE RESOURCES

Open Platform for IoT - A Reference Architecture for Getting Started and Scaling

Try This Winning Combination: IoT + Data, Big Data and Real Time Analytics

Your Thing Is Pwned - Addressing Security Challenges in IoT

Here's How API management Can Help with Your IoT

Get Back Control - Devise Management for Connected Devices



DZONE'S GUIDE TO ENTERPRISE INTEGRATION


Discover best practices and the most useful tools for building the ideal integration architecture.


Download the Guide

-
- The diagram illustrates the architecture of an 'Open platform for Connected Things'. At the center is a hub labeled 'Open platform for Connected Things'. Surrounding this hub are six interconnected components, each represented by an icon and a label:
- DEVICE MANAGEMENT** (Top): Represented by a gear and a device icon.
 - SECURITY** (Right): Represented by a shield and a lock icon.
 - CONNECTIVITY AND COMMUNICATIONS** (Bottom Right): Represented by a network of nodes and a signal icon.
 - DATA COLLECTION, ANALYSIS, AND ACTUATION** (Bottom Left): Represented by a document and a bar chart icon.
 - SCALABILITY** (Left): Represented by a cluster of nodes.
 - Open platform for Connected Things** (Center): The central hub of the architecture.
- The entire diagram is set against a dark blue background with a light blue circular pattern radiating from the center.


Spotlight Features

The Best of DZone: Jan. 14 - Jan. 21






QUIZ: What's Your Developer Personality?



A rebuttal against terrible interview advice



NOT SURE IF ACTUAL VALUE
ON PAPER YOU ARE

The New Agile: Why Is Agile So Popular?

[illegible]

When this class will be called from the client side using

```
1. public static SingletonExample getSingletonInstance() {
2.     if (null == singletonInstance) {
3.         singletonInstance = new SingletonExample();
4.         System.out.println("Creating new instance");
5.     }
6.     return singletonInstance;
7. }
```

```
1. SingletonExample.getSingletonInstance().printSingleton();
2. SingletonExample.getSingletonInstance().printSingleton();
3. SingletonExample.getSingletonInstance().printSingleton();
4. SingletonExample.getSingletonInstance().printSingleton();
```

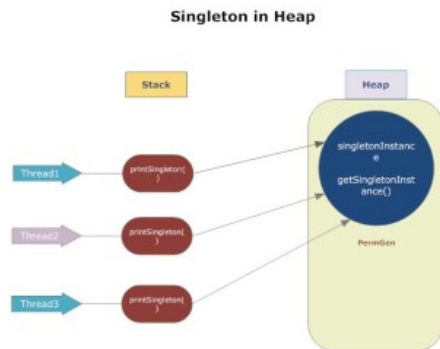
<http://java.dzone.com/articles/singleton-design-pattern-%E2%80%93>

- ```
1. Creating new instance
2. Inside print Singleton
3. Inside print Singleton
4. Inside print Singleton
5. Inside print Singleton
```



## Implementation Example: Double check locking

The above code works absolutely fine in a single threaded environment and processes the result faster because of lazy initialization. However the above code might create some abrupt behaviour in the results in a multithreaded environment as in this situation multiple threads can possibly create multiple instance of the same SingletonExample class if they try to access the *getSingletonInstance()* method at the same time. Imagine a practical scenario where we have to create a log file and update it or while using a shared resource like Printer. To prevent this we must use some locking mechanism so that the second thread cannot use this *getInstance()* method until the first thread has completed the process.



## Singleton in heap

In Figure 3 we show how multiple threads access the singleton instance. Since the singleton instance is a static class variable and is stored in the PermGen space of the heap. This applies to `getSingletonInstance()` instance method as well since it is static too. In the multithreading environment to prevent each thread to create another instance of singleton object and thus creating concurrency issue we will need to use locking mechanism. This can be achieved by synchronized keyword. By using this synchronized keyword we prevent Thread2 or Thread3 to access the singleton instance while Thread1 is inside the method `getSingletonInstance()`.

From code perspective it means:

```
1. public static synchronized SingletonExample getSingletonInstance() {
2. if (null == singletonInstance) {
3. singletonInstance = new SingletonExample();
4. }
5. return singletonInstance;
6. }
```



So this means that every time the `getSingletonInstance()` is called it gives us an additional overhead . To prevent this expensive operation we will use **double checked locking** so that the synchronization happens only during the first call and we limit this expensive operation to happen only once. It is only required for:

```
1. singletonInstance = new SingletonExample();
```



Code example:

```
01. public static volatile SingletonExample getSingletonInstance(
02. if (null == singletonInstance) {
03. synchronized (SingletonExample.class){
04. if (null == singletonInstance) {
05. singletonInstance = new SingletonExample();
06. }
07. }
08. }
09. return singletonInstance;
10. }
```



In the above code snippet imagine that multiple threads come concurrently and try to create the new instance. In such a situation, there may be three or more threads waiting on the synchronized block to get access. Since we have used `synchronized` only one thread will be given access. All the remaining threads which were waiting on the synchronized block will be given access when the first thread exits this block. However, when the remaining concurrent thread enters the synchronized block, they are prevented from entering further due to the double check : null check. Since the first thread has already created an instance, no other thread will enter this loop.

All the remaining threads that were not lucky to enter the synchronized block along with the first thread will be blocked at the first null check. This mechanism is called **double checked locking** and it provides significant performance benefit and also it is cost effective solution.

### Implementation Example: Volatile Keyword

We can also use the `volatile` keyword to the instance variable declaration.

```
1. private volatile static SingletonExample singletonInstance;
```

The volatile keyword helps as concurrency control tool in a multithreaded environment and provides the latest update in a most accurate manner. However please note that double check locking might not work before Java 5. In such situation we can use early loading mechanism. If we remember about the original sample code we had used lazy loading. In case of early loading we will instantiate the SingletonExample class at the start and this will be referred to the private static instance field.

```
01. public class SingletonExample {
02. private static final SingletonExample singletonInstance = new
 SingletonExample();
03.
04. // SingletonExample prevents any other class from instantiating
05. private SingletonExample() {
06. }
07.
08. // Providing Global point of access
09. public static SingletonExample getSingletonInstance() {
10.
11. return singletonInstance;
12. }
13.
14. public void printSingleton(){
15. System.out.println("Inside print Singleton");
16. }
17. }
```

In this approach the singleton object is created before it is needed. The JVM takes care of the static variable initialization and ensures that the process is thread safe and that the instance is created before the threads tries to access it. In case of Lazy loading the *singletonInstance* is created when the client class calls the *getSingletonInstance()* whereas in case of the early loading the singletonInstance is create when the class is loaded in the memory.

### Implementation Example: Using enum

### Implementing Singleton in Java 5 or above version using Enum:

Enum is thread safe and implementation of Singleton through Enum ensures that your singleton will have only one instance even in a multithreaded environment. Let us see a simple implementation:

```
01. public enum SingletonEnum {
02. INSTANCE;
03. public void doStuff(){
04. System.out.println("Singleton using Enum");
05. }
06. }
07. And this can be called from clients :
08. public static void main(String[] args) {
09. SingletonEnum.INSTANCE.doStuff();
10. }
11. }
```

## FAQs:

**Question:** Why can't we use a static class instead of singleton?

**Answer:**

- One of the key advantages of singleton over static class is that it can implement interfaces and extend classes while the static class cannot (it can extend classes, but it does not inherit their instance members). If we consider a static class it can only be a nested static class as top level class cannot be a static class. Static means that it belongs to a class it is in and not to any instance. So it cannot be a top level class.
- Another difference is that static class will have all its member as static only unlike Singleton.
- Another advantage of Singleton is that it can be lazily loaded whereas static will be initialized whenever it is first loaded.
- Singleton object stores in Heap but, static object stores in stack.
- We can clone the object of Singleton but, we can not clone the static class object.
- Singleton can use the Object Oriented feature of polymorphism but static class cannot.

**Question:** Can the singleton class be subclassed?

**Answer:** Frankly speaking singleton is just a design pattern and it can be subclassed. However it is worth to understand the logic or requirement behind subclassing a singleton class as the child class might not inherit the singleton pattern objective by extending the Singleton class. However the subclassing can be prevented by using the final keyword in the class declaration.

**Question:** Can there be multiple instance of singleton using cloning?

**Answer:** That was a good catch! What do we do now? To prevent the another instance to be created of the singleton instance we can throw exception from inside the clone() method.

**Question:** What is the impact if we are creating another instance of singleton using serialization and deserialization?

**Answer:** When we serialize a class and deserialize it then it creates another instance of the singleton class. Basically as many times as you deserialize the singleton instance it will create multiple instance. Well in this case the best way is to make the singleton as enum. In that way the underlying Java implementation takes care of all the details. If this is not possible then we will need to override the *readObject()* method to return the same singleton instance.

**Question:** Which other pattern works with Singleton?

**Answer:** There are several other pattern like Factory method, builder and prototype pattern which uses Singleton pattern during the implementation.

**Question:** Which classes in JDK uses singleton pattern?

**Answer:** java.lang.Runtime : In every Java application there is only one Runtime instance that allows the application to interface with the environment it is running. The getRuntime is equivalent to the getInstance() method of the singleton class.

## Uses of Singleton design pattern:

Various usages of Singleton Patterns:

- Hardware interface access: The use of singleton depends on the requirements. However practically singleton can be used in case external hardware resource usage limitation required e.g. Hardware printers where the print spooler can be made a singleton to avoid multiple concurrent accesses and creating deadlock.
- Logger : Similarly singleton is a good potential candidate for using in the log files generation. Imagine an application where the logging utility has to produce one log file based on the messages received from the users. If there is multiple client application using this logging utility class they might create multiple instances of this class and it can potentially cause issues during concurrent access to the same logger file. We can use the logger utility class as a singleton and provide a global point of reference.
- Configuration File: This is another potential candidate for Singleton pattern because this has a performance benefit as it prevents multiple users to repeatedly access and read the configuration

file or properties file. It creates a single instance of the configuration file which can be accessed by multiple calls concurrently as it will provide static config data loaded into in-memory objects. The application only reads from the configuration file at the first time and there after from second call onwards the client applications read the data from in-memory objects.

- **Cache:** We can use the cache as a singleton object as it can have a global point of reference and for all future calls to the cache object the client application will use the in-memory object.

Hands-on:

Let us take a small practical example to understand the Singleton design pattern in more details.

### Problem Statement:

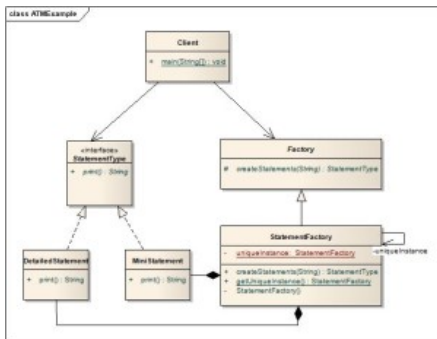
Design a small ATM printing application which can generate multiple types of statements of the transaction including Mini Statement, Detailed statement etc. However the customer should be aware of the creation of these statements. Ensure that the memory consumption is minimized.

### Design Solution:

The above requirement can be addressed using two core Gang of four design pattern – Factory design pattern and Singleton design pattern. In order to generate multiple types of statements for the ATM transactions in the ATM machine we can create a Statement Factory object which will have a factory method of *createStatements()*. The createStatements will create DetailedStatement or MiniStatement objects.

The client object will be completely unaware of the object creation since it will interact with the Factory interface only. We will also create an interface called `StatementType`. This will allow further statement type objects e.g. Credit card statement etc to be added. So the solution is scalable and extensible following the object oriented Open/Closed design principle.

The second requirement of reducing the memory consumption can be achieved by using Singleton design pattern. The Statement Factory class need not be initiated multiple times and a single factory can create multiple statement objects. Singleton pattern will create a single instance of the StatementFactory class thus saving memory.



## ATM example

- **Factory:** Factory is an abstract class which is a single point of contact for the client. All the concrete factory classes needs to implement the abstract factory method.
- **StatementFactory:** This is the Factory implementation class which consist of the creator method. This class extends from the Factory abstract class.This is the main creator class for all the products e.g. Statements.
- **StatementType:** This is a product interface which provides abstraction to the various types of products which needs to be created by the Factory class.
- **DetailedStatement:** This is a concrete implementation of the StatementType interface which will print the detailed statements.

- MiniStatement: This is a concrete implementation of the StatementType interface which will print the mini statements.
- Client: This is the client class which will call the StatementFactory and StatementType and request for object creation.

**Assumption:**

The design solution caters to only single ATM machine.

1 2 next › last »

You are not following comments on this post. [Click to start watching.](#)

Published at DZone with permission of [Mainak Goswami](#), author and DZone MVB. ([source](#))

(Note: Opinions expressed in this article and its replies are the opinions of their respective authors and not those of DZone, Inc.)

Tags: [Design Patterns](#) [Java](#) [Tutorial](#) [Architecture](#)

The *Enterprise Integration Zone* is brought to you in partnership with [WSO2](#). [Learn more about WSO2's API Management.](#)

## Comments



Niklas Efternamn replied on Wed, 2013/02/13 - 8:42am

Your following statement is incorrect: "In Java the Singleton pattern will ensure that there is only one instance of a class is created in the Java Virtual Machine", when using static members. It is only true per *Class Loader*, not *JVM*.

[reply](#)



Stoo Mcstooo replied on Wed, 2013/02/13 - 10:05am

Double checked locking doesn't work well in java. See Java Concurrency in Practice, page 349 which recommends against it. At minimum the singleton should be volatile to work, and even then the performance improvement is negligible.

[reply](#)



Dapeng Liu replied on Wed, 2013/02/13 - 12:03pm

singletons should really be managed by DI containers

[reply](#)



Mainak Goswami replied on Wed, 2013/02/13 - 2:43pm in response to: [Niklas Efternamn](#)

Thanks Niklas. You are correct. Static deals with per class basis so when using static members it will be per class loader basis. However imagine the situation where there is only one application in the JVM which is used by this class loader. So implicitly singleton will have a unique instance throughout the JVM...isn't it? Please