# 싱글톤 할때 방법을 바꿔라

|  |
| --- |
| package com.java;   public class SingleToneNew {  private SingleToneNew() {  System.*out*.println("create!!");  }    public static class Singleton {  private static SingleToneNew *instance* = new SingleToneNew();  }   public static SingleToneNew getInstance () {  System.*out*.println("get instance");  return Singleton.*instance*; // return null;  }   } 이렇게 하면 getInstance할때 그때 Singleton class가 생성되면서 static instance가 생성되고 상수로 박킨다.sub static class는 처음 호출될때 이때 생성된다 |

# java singleton pattern (싱글톤 패턴)

<https://blog.seotory.com/post/2016/03/java-singleton-pattern>

예전 블로그에서도 singleton 에 대한 글을 쓴적이 있다. 그때는 매우 단순하게 적었으나 이번에는 조금 방대할 것이다. 단일 인스턴스를 다양하게 만들 수 있는 방법을 예제로 통해 한번 알아 보도록 하자.

**singleton 이란?**

프로그래밍 세계에 OOP 의 개념이 생기면서 객체 자체에 대한 많은 연구와 패턴(pattern)들이 생겨났다. singleton pattern은 인스턴스가 사용될 때에 똑같은 인스턴스를 만들어 내는 것이 아니라, 동일 인스턴스를 사용하게끔 하는 것이 기본 전략이다. 프로그램상에서 동일한 커넥션 객체를 만든다던지, 하나만 사용되야하는 객체를 만들때 매우 유용하다. singleton pattern은 4대 디자인 패턴에 들어갈 정도로 흔히 쓰이는 패턴이다. 물론 core java(java.lang.Runtime, java.awt.Desktop 등등)에서도 singleton pattern이 사용된다.

**Eager initialization**

아래가 가장 기본적인 singleton pattern이다. 전역 변수로 instance를 만드는데 private static을 이용한다. static이 붙은 클래스변수는 인스턴스화에 상관없이 사용이 가능하게 된다. 하지만 앞의 private 접근제어자로 인해 EagerInitialization.instance로의 접근은 불가능하다. 이런 상태에서 생성자를 private로 명시한다. 생성자를 private로 붙이게되면, new 키워드를 사용할 수 없게된다. 즉 다른 클래스에서 EagerInitialization instance = new EagerInitialization(); 이런 방법을 통한 인스턴스 생성은 불가능해진다. 결국 외부 클래스가 EagerInitialization 클래스의 인스턴스를 가질 수 있는 방법은 11번째 라인에 있는 getInstance() method를 사용하는 수 밖에 없다.

public class EagerInitialization {

// private static 로 선언.

private static EagerInitialization instance = new EagerInitialization();

// 생성자

private EagerInitialization () {

System.out.println( "call EagerInitialization constructor." );

}

// 조회 method

public static EagerInitialization getInstance () {

return instance;

}

public void print () {

System.out.println("It's print() method in EagerInitialization instance.");

System.out.println("instance hashCode > " + instance.hashCode());

}

}

위의 단순한 singleton pattern은 리소스가 작은 프로그램일때엔 고도화 대상이 아니다. 하지만 프로그램의 크기가 커져서 수 많은 클래스에서 위와 같은 singleton pattern을 사용한다고 가정해보자. 3번째 라인의 new EagerInitialization();으로 인해 클래스가 load 되는 시점에 인스턴스를 생성시키는데 이마저도 부담스러울 수가 있다. 또한 이 소스는 EagerInitialization 클래스가 인스턴스화 되는 시점에 어떠한 에러처리도 할 수가 없다.

**static block initialization**

public class StaticBlockInitalization {

private static StaticBlockInitalization instance;

private StaticBlockInitalization () {}

static {

try {

System.out.println("instance create..");

instance = new StaticBlockInitalization();

} catch (Exception e) {

throw new RuntimeException("Exception creating StaticBlockInitalization instance.");

}

}

public static StaticBlockInitalization getInstance () {

return instance;

}

public void print () {

System.out.println("It's print() method in StaticBlockInitalization instance.");

System.out.println("instance hashCode > " + instance.hashCode());

}

}

static 초기화블럭을 이용하면 클래스가 로딩 될 때 최초 한번 실행하게 된다. 특히나 초기화블럭을 이용하면 logic을 담을 수 있기 때문에 복잡한 초기변수 셋팅이나 위와 같이 에러처리를 위한 구문을 담을 수 있다. 첫 번째 패턴보다 좋아보이지만 인스턴스가 사용되는 시점에 생성되는 것은 아니다.

**lazy initialization**

이제 클래스 인스턴스가 사용되는 시점에 인스턴스를 만드는 singleton pattern을 배워보도록 하자. 아래 소스의 lazy initialization pattern은 필요할때 인스턴스를 생성시키는 것이 핵심이다.

public class LazyInitialization {

private static LazyInitialization instance;

private LazyInitialization () {}

public static LazyInitialization getInstance () {

if ( instance == null )

instance = new LazyInitialization();

return instance;

}

public void print () {

System.out.println("It's print() method in LazyInitialization instance.");

System.out.println("instance hashCode > " + instance.hashCode());

}

}

new LazyInitialization(); 가 어디에 선언되었는지 주목해보자. getInstance() method 안에서 사용되었다. if문을 이용해 instance가 null 인 경우에만 new를 사용해 객체를 생성하였다. 최초 사용시점에만 인스턴스화 시키기 때문에 프로그램이 메모리에 적재되는 시점에 부담이 많이 줄게된다. 하지만 여전히 문제는 남아있다. 만약 프로그램이 muilti thread 방식이라면 위와 같은 singleton pattern은 안전하지 않다. 동일 시점에 getInstance() method를 호출하면 인스턴스가 두번 생길 위험이 있다.

**thread safe initalization**

위에서 문제가 되었던 muilit thread문제를 해결하기 위해 synchronized(동기화)를 사용하여 singleton pattern을 구현한다. 여러 thread들이 동시에 접근해서 인스턴스를 생성시키는 위험은 없어졌다. 하지만 수 많은 thread 들이 getInstance() method 를 호출하게 되면 높은 cost 비용으로 인해 프로그램 전반에 성능저하가 일어난다.

public class ThreadSafeInitalization {

private static ThreadSafeInitalization instance;

private ThreadSafeInitalization () {}

public static synchronized ThreadSafeInitalization getInstance () {

if (instance == null)

instance = new ThreadSafeInitalization();

return instance;

}

public void print () {

System.out.println("It's print() method in ThreadSafeInitalization instance.");

System.out.println("instance hashCode > " + instance.hashCode());

}

}

**initialization on demand holder idiom**

미국 메릴랜드 대학의 컴퓨터 과학 연구원인 Bill pugh 가 기존의 java singleton pattern이 가지고 있는 문제들을 해결 하기 위해 새로운 singleton pattern을 제시하였다. Initialization on demand holder idiom기법이다. 이것은 jvm 의 class loader의 매커니즘과 class의 load 시점을 이용하여 내부 class를 생성시킴으로 thread 간의 동기화 문제를 해결한다.

public class InitializationOnDemandHolderIdiom {

private InitializationOnDemandHolderIdiom () {}

private static class Singleton {

private static final InitializationOnDemandHolderIdiom instance = new InitializationOnDemandHolderIdiom();

}

public static InitializationOnDemandHolderIdiom getInstance () {

System.out.println("create instance");

return Singleton.instance;

}

}

initialization on demand holder idiom 역시 lazy initialization이 가능하며 모든 java 버젼과, jvm에서 사용이 가능하다. 현재 java 에서 singleton 을 생성시킨다고 하면 거의 위의 방법을 사용한다고 보면 된다.

**enum initialization**

Joshua Bloch가 작성한 effective java 책에서 enum singleton 방법이 소개 되었다.

public enum EnumInitialization {

INSTANCE;

static String test = "";

public static EnumInitialization getInstance() {

test = "test";

return INSTANCE;

}

}

enum 이 singleton pattern 으로 사용될 수 있는 이유는 아래와 같다.

* INSTANCE 가 생성될 때, multi thread 로 부터 안전하다. (추가된 methed 들은 safed 하지 않을 수도 있다.)
* 단 한번의 인스턴스 생성을 보장한다.
* 사용이 간편하다.
* enum value는 자바 프로그램 전역에서 접근이 가능하다.

**using reflection to destroy singleton**

위에서 여러 방법으로 singleton을 만들어 보았으니 이번엔 java의 reflection을 이용하여 singleton을 깨뜨려 보는법도 배워보자. 누군가 작성한 코드를 원본 수정없이 작업해야 할때 이용될 수 있을 것이다.

public class UsingReflectionToDestroySingleton {

public static void main (String[] args) {

EagerInitialization instance = EagerInitialization.getInstance();

EagerInitialization instance2 = null;

try {

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

for ( Constructor constructor : constructors ) {

constructor.setAccessible(true);

instance2 = (EagerInitialization)constructor.newInstance();

}

} catch (Exception e) {

}

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

System.out.println(instance2.hashCode());

}

}

위의 코드를 실행해보면 아래 System.out.println();의 두 라인에서 찍히는 hachCode()값이 다른 것을 확인 할 수 있다. java의 reflection은 매우 강력하다. 설령 class 의 생성자가 private 일지라도 강제로 가져와서 새로운 인스턴스 생성이 가능하다. 결국 singleton pattern을 깨뜨리는 것이다. 이 외에도 reflection을 여러곳에서 사용할 수 있으니 알아두는 것이 좋다.

<https://dzone.com/articles/the-java-8-api-design-principles>

# API Design with Java 8

### Learn to be a better Java programmer by mastering Java 8 API design, exposing a well-designed API, making sure that client code can use lambdas, and more.

[](https://dzone.com/users/1390709/minborg.html)by

[Per-Åke Minborg](https://dzone.com/users/1390709/minborg.html)

·

Sep. 30, 16 · [Java Zone](https://dzone.com/java-jdk-development-tutorials-tools-news)

Like [(91)](https://dzone.com/articles/the-java-8-api-design-principles)

Comment (25)

Save

[Tweet](https://dzone.com/articles/the-java-8-api-design-principles)

27.00k Views

Join the DZone community and get the full member experience.

[Join For Free](https://dzone.com/articles/the-java-8-api-design-principles)

Check out this [8-step guide](https://dzone.com/go?i=124041&u=http%3A%2F%2Fpages.zeroturnaround.com%2FRocket-Powered-White-Paper_Rocket-Powered-White-Paper.html%3Futm_source%3Ddzone%26utm_medium%3Djavazone_partner_resources%26utm_campaign%3Drocketpowered) to see how you can increase your productivity by skipping slow application redeploys and by implementing application profiling, as you code! Brought to you in partnership with [ZeroTurnaround](https://dzone.com/go?i=124041&u=http%3A%2F%2Fpages.zeroturnaround.com%2FRocket-Powered-White-Paper_Rocket-Powered-White-Paper.html%3Futm_source%3Ddzone%26utm_medium%3Djavazone_partner_resources%26utm_campaign%3Drocketpowered).

[This article is featured in the new DZone Guide to Modern Java, Volume II. Get your free copy for more insightful articles, industry statistics, and more.](http://bit.ly/2dCvGIo)

Anyone that writes Java code is an API designer! It does not matter if the coders share their code with others or not, the code is still used; either by others, by themselves or both. Thus, it becomes important for all Java developer to know the fundamentals of good API design.

A good API design requires careful thinking and a lot of experience. Luckily, we can learn from other clever people like Ference Mihaly, whose [blog post](https://theamiableapi.com/2012/01/16/java-api-design-checklist/) inspired me to write this Java 8 API addendum. We relied heavily on his checklist when we designed the Speedment API. (I encourage you all to read his guide.)

Getting it right from the start is important because once an API is published, a firm commitment is made to the people who are supposed to use it. As Joshua Bloch once said: “Public APIs, like diamonds, are forever. You have one chance to get it right, so give it your best.” A well-designed

API combines the best of two worlds, a firm and precise commitment combined with a high degree of implementation flexibility, eventually benefiting both the API designers and the API users.

Why use a checklist? Getting the API right (i.e. defining the visible parts of a collection of Java classes) can be much harder than writing the implementation classes that makes up the actual work behind the API. It is really an art that few people master. Using a checklist allows the reader to avoid the most obvious mistakes, become a better programmer and save a lot of time.

API designers are strongly encouraged to put themselves in the client code perspective and to optimize that view in terms of simplicity, ease-of-use, and consistency — rather than thinking about the actual API implementation. At the same time, they should try to hide as many implementation details as possible.

## Do not Return Null to Indicate the Absence of a Value

Arguably, inconsistent null handling (resulting in the ubiquitous NullPointerException) is the single largest source of Java applications’ errors historically. Some developers regard the introduction of the null concept as one of the worst mistakes ever made in the computer science domain. Luckily, the first step of alleviating Java’s null handling problem was introduced in Java 8 with the advent of the Optional class. Make sure a method that can return a no-value returns an Optional instead of null.

This clearly signals to the API users that the method may or may not return a value. Do not fall for the temptation to use null over Optional for performance reasons. Java 8’s escape analysis will optimize away most Optional objects anyway. Avoid using Optionals in parameters and fields.

### Do This:

public Optional<String> getComment() {

return Optional.ofNullable(comment);

}

### Don't Do This:

public String getComment() {

return comment; // comment is nullable

}

## Do not Use Arrays to Pass Values to and From the API

A significant API mistake was made when the Enum concept was introduced in Java 5. We all know that an Enum class has a method called values() that returns an array of all the Enum’s distinct values. Now, because the Java framework must ensure that the client code cannot change the Enum’s values (for example, by directly writing to the array), a copy of the internal array must be produced for each call to the value() method.

This results in poor performance and also poor client code usability. If the Enum would have returned an unmodifiable List, that List could be reused for each call and the client code would have had access to a better and more useful model of the Enum’s values. In the general case, consider exposing a Stream, if the API is to return a collection of elements. This clearly states that the result is read-only (as opposed to a List which has a set() method).

It also allows the client code to easily collect the elements in another data structure or act on them on-the-fly. Furthermore, the API can lazily produce the elements as they become available (e.g. are pulled in from a file, a socket, or from a database). Again, Java 8’s improved escape analysis will make sure that a minimum of objects are actually created on the Java heap.

Do not use arrays as input parameters for methods either, since this — unless a defensive copy of the array is made — makes it possible for another thread to modify the content of the array during method execution.

### Do This:

public Stream<String> comments() {

return Stream.of(comments);

}

## Don't Do This:

public String[] comments() {

return comments; // Exposes the backing array!

}

## Consider Adding Static Interface Methods to Provide a Single Entry Point for Object Creation

Avoid allowing the client code to directly select an implementation class of an interface. Allowing client code to create implementation classes directly creates a much more direct coupling of the API and the client code. It also makes the API commitment much larger, since now we have to maintain all the implementation classes exactly as they can be observed from outside instead of just committing to the interface as such.

Consider adding static interface methods, to allow the client code to create (potentially specialized) objects that implement the interface. For example, if we have an interface Point with two methods int x() and int y(), then we can expose a static method Point.of(int x, int y) that produces a (hidden) implementation of the interface.

So, if x and y are both zero, we can return a special implementation class PointOrigoImpl (with no x or y fields), or else we return another class PointImpl that holds the given x and y values. Ensure that the implementation classes are in another package that are clearly not a part of the API (e.g. put the Point interface in com.company. product.shape and the implementations in com.company.product.internal.shape).

### Do This:

Point point = Point.of(1,2);

### Don't Do This:

Point point = new PointImpl(1,2);

## Favor Composition With Functional Interfaces and Lambdas Over Inheritence

For good reasons, there can only be one super class for any given Java class. Furthermore, exposing abstract or base classes in your API that are supposed to be inherited by client code is a very big and problematic API commitment. Avoid API inheritance altogether, and instead consider providing static interface methods that take one or several lambda parameters and apply those given lambdas to a default internal API implementation class.

This also creates a much clearer separation of concerns. For example, instead of inheriting from a public API class AbstractReader and overriding abstract void handleError(IOException ioe), it is better to expose a static method or a builder in the Reader interface that takes a Consumer<IOException> and applies it to an internal generic ReaderImpl.

## Do This:

Reader reader = Reader.builder()

.withErrorHandler(IOException::printStackTrace)

.build();

### Don't Do This:

Reader reader = new AbstractReader() {

@Override

public void handleError(IOException ioe) {

ioe. printStackTrace();

}

};

## Ensure That You Add the @FunctionalInterface Annotation to Functional Interfaces

Tagging an interface with the @FunctionalInterface annotation signals that API users may use lambdas to implement the interface, and it also makes sure the interface remains usable for lambdas over time by preventing abstract methods from accidently being added to the API later on.

### Do This:

@FunctionalInterface

public interface CircleSegmentConstructor {

CircleSegment apply(Point cntr, Point p, double ang);

// abstract methods cannot be added

}

### Don't Do This:

public interface CircleSegmentConstructor {

CircleSegment apply(Point cntr, Point p, double ang);

// abstract methods may be accidently added later

}

## Avoid Overloading Methods With Functional Interfaces as Parameters

If there are two or more functions with the same name that take functional interfaces as parameters, then this would likely create a lambda ambiguity on the client side. For example, if there are two Point methods add(Function<Point, String> renderer) and add(Predicate<Point> logCondition) and we try to call point.add(p -> p + “ lambda”) from the client code, the compiler is unable to determine which method to use and will produce an error. Instead, consider naming methods according to their specific use.

### Do This:

public interface Point {

addRenderer(Function<Point, String> renderer);

addLogCondition(Predicate<Point> logCondition);

}

### Don't Do This:

public interface Point {

add(Function<Point, String> renderer);

add(Predicate<Point> logCondition);

}

## Avoid Overusing Default Methods in Interfaces

Default methods can easily be added to interfaces and sometimes it makes sense to do that. For example, a method that is expected to be the same for any implementing class and that is short and “fundamental” in its functionality, is a viable candidate for a default implementation. Also, when an API is expanded, it sometimes makes sense to provide a default interface method for backward compatibility reasons.

As we all know, functional interfaces contain exactly one abstract method, so default methods provide an escape hatch when additional methods must be added. However, avoid having the API interface evolve to an implementation class by polluting it with unnecessary implementation concerns. If in doubt, consider moving the method logic to a separate utility class and/or place it in the implementing classes.

### Do This:

public interface Line {

Point start();

Point end();

int length();

}

### Don't Do This:

public interface Line {

Point start();

Point end();

default int length() {

int deltaX = start().x() - end().x();

int deltaY = start().y() - end().y();

return (int) Math.sqrt(

deltaX \* deltaX + deltaY \* deltaY

);

}

}

## Ensure That the API Methods Check the Parameter Invariants Before They Are Acted Upon

Historically, people have been sloppy in making sure to validate method input parameters. So, when a resulting error occurs later on, the real reason becomes obscured and hidden deep down the stack trace. Ensure that parameters are checked for nulls and any valid range constrains or preconditions before the parameters are ever used in the implementing classes. Do not fall for the temptation to skip parameter checks for performance reasons.

The JVM will be able to optimize away redundant checking and produce efficient code. Make use of the Objects.requireNonNull() method. Parameter checking is also an important way to enforce the API’s contract. If the API was not supposed to accept nulls but did anyhow, users will become confused.

### Do This:

public void addToSegment(Segment segment, Point point) {

Objects.requireNonNull(segment);

Objects.requireNonNull(point);

segment.add(point);

}

### DON’T DO THIS:

public void addToSegment(Segment segment, Point point) {

segment.add(point);

}

## Do not Simply Call Optional.get()

The API designers of Java 8 made a mistake when they selected the name Optional.get() when it should really have been named Optional.getOrThrow() or something similar instead. Calling get() without checking if a value is present with the Optional.isPresent() method is a very common mistake which fully negates the null elimination features Optional originally promised. Consider using any of the Optional’s other methods such as map(), flatMap() or ifPresent() instead in the API’s implementing classes or ensure that isPresent() is called before any get() is called.

### Do This:

Optional<String> comment = // some Optional value

String guiText = comment

.map(c -> "Comment: " + c)

.orElse("");

### Don't Do This:

Optional<String> comment = // some Optional value

String guiText = "Comment: " + comment.get();

## Consider Separating Your Stream Pipeline on Distinct Lines in Implementing API Classes

Eventually, all APIs will contain errors. When receiving stack traces from API users, it is often much easier to determine the actual cause of the error if a Stream pipeline is split into distinct lines compared to a Stream pipeline that is expressed on a single line. Also, code readability will improve.

### Do This:

Stream.of("this", "is", "secret")

.map(toGreek())

.map(encrypt())

.collect(joining(" "));

### Don't Do This:

Stream.of("this", "is", "secret").map(toGreek()).map(encrypt()).collect(joining(" "));

[For more insights on Jigsaw, reactive microservices, and more get your free copy of the new DZone Guide to Modern Java, Volume II!](http://bit.ly/2dCvGIo)

The Java Zone is brought to you in partnership with [ZeroTurnaround](https://dzone.com/go?i=97821&u=http%3A%2F%2Fpages.zeroturnaround.com%2FRocket-Powered-White-Paper_Rocket-Powered-White-Paper.html%3Futm_source%3Ddzone%26utm_medium%3Djavazone_partner_resources%26utm_campaign%3Drocketpowered). Check out this [8-step guide](https://dzone.com/go?i=97821&u=http%3A%2F%2Fpages.zeroturnaround.com%2FRocket-Powered-White-Paper_Rocket-Powered-White-Paper.html%3Futm_source%3Ddzone%26utm_medium%3Djavazone_partner_resources%26utm_campaign%3Drocketpowered) to see how you can increase your productivity by skipping slow application redeploys and by implementing application profiling, as you code!