<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!