Java API Best Practices

WRITTEN BY JONATHAN GILES
SENIOR CLOUD DEVELOPER ADVOCATE, MICROSOFT

INTRODUCTION

As engineers we write code every day, and it is inconceivable that this code would ever exist in a vacuum, isolated from all other software ever written. Never has the metaphor, "standing on the shoulders of giants," been more apt than it is today in software engineering, with GitHub, Stack Overflow, Maven Central, and all other directories of code, support, and software libraries available at our fingertips.

Software is built from Application Programming Interfaces (APIs) — we make use of the JDK and numerous dependencies brought in from tools, such as Maven or Gradle, on a daily basis. If you were to ask a room full of software engineers if they were API developers, their response is usually that no, they are not. This is incorrect! Anyone who has ever crafted a public class or public method should consider themselves an API developer. The term "crafting" is used deliberately here. Too often software engineering gets wrapped up in the formality of engineering, but API design is as much, if not moreso, an art form that requires creativity and a gut feeling to be developed over many years, rather than an exact science.

API CHARACTERISTICS

There are many criteria through which an API can be characterized, six of which are introduced below, and which form the threads that we will cover in more depth throughout this refcard.

1. UNDERSTANDABLE

How often as an engineer have you downloaded a library as a Maven dependency, and then wondered which class gets you started with using the API? An API should not be considered successful if a developer cannot intuitively understand how to use it.

Developers should give consideration to the entry points into their API. Complete documentation is useful and helps developers understand the bigger picture, but ideally, we want to ensure minimal friction for the developer, and therefore we should provide developers with the minimal steps to get started at the very top of our documentation. From here, good APIs try to expose their functionality through this entry point, to hold the developer's hand as they make use of the primary use cases enabled by the API. More advanced functionality can then be discovered through external documentation as necessary.

2. WELL-DOCUMENTED

Because we expect others to use our APIs, we should put in the effort to document it. Our focus in this Refcard is on high-quality, detailed JavaDoc content.

Microsoft Azure

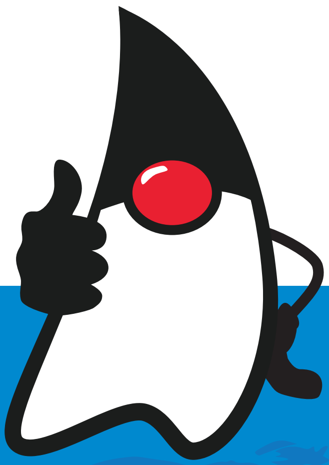
Java on Azure

The Home for Enterprise Java

Bring your Java applications and make them Cloud-scale.

Transform your monoliths and
modernize with cloud-native development.

Learn more: azure.com/java



54 Azure regions - more than any cloud provider

Loved by Developers | Trusted by Enterprises | Supported by Partners

3. CONSISTENT

A good API should not surprise its users, and one way we can fail at this is by not being consistent. When we speak of consistency, we mean ensuring that we repeat the same concepts in our API, rather than introduce different concepts in an adhoc fashion. Examples include:

- All of our methods should have the form `getXYZ()` or `xyz()`, but not both forms.
- If there are two methods (one a convenience overload of the other, e.g. one taking `Object...` (that is, a var-args of `Object`) and the other taking `Collection<? extends Object>`), that overload should be available everywhere.

The point here is to establish a team-wide vocabulary and "cheat sheet" that we use to apply a veneer of consistency across our entire SDK.

4. FIT FOR PURPOSE

In developing an API, we must ensure that we target it at the right level for the intended user. This can be thought of in two ways:

1. Do only one thing, and do it right.
2. Understand your user and their goals.

A good example is the Collections APIs included with the JDK. The user should be able to store and retrieve items from a collection very easily, without first defining a reallocation threshold, configuring a factory, specifying a growth policy, a hash collision policy, a comparison policy, a load factor, a caching policy, and so on. Developers should never be expected to know the inner working of a collections framework to benefit from its functionality.

5. RESTRAINED

Creating a new API can happen almost too quickly, but we should remember that for every new API, we are potentially committing to a lifetime of support.

The actual cost of our API decisions depends a lot on our project and its community — some projects are happy to make breaking changes constantly, whereas others (such as the JDK itself) are eager to break as little as possible. Most projects fall in the middle ground, adopting a semantic versioning approach that carefully deprecates an API before removing it only in major releases.

Some projects even go so far as to include various markers to indicate experimental, beta, or preview functionality that makes it available in releases for feedback before a final API is locked down. A common approach is to introduce this new, experimental functionality with the `@Deprecated` annotation, and to then remove the annotation when the API is considered ready.

6. EVOLVABLE

For every API decision we make, we are potentially backing ourselves into another corner. As we make API decisions, we must take the time to view them in the wider context of the future releases of the SDK.

API AS A CONTRACT

API has to be thought of as a contract — when we make an API available to other developers, we are promising them a certain functionality. It is often hard for engineers to leave their code alone, as they constantly imagine new and better approaches (even when not working). Thinking that we should revise our API, to attempt to make it better, should only be done after much consideration, as we risk introducing breaking changes and bugs to the developers downstream from us.

In the build up to the 1.0.0 release of our API, we should feel a sense of freedom to experiment. In some projects, the point at which we reach 1.0.0 is the point in which our API becomes locked down (at least until the 2.0.0 release). However, in other projects, there can be more flexibility with their obligations, continuing to experiment and refine their API on an ongoing basis. In fact, with a wise deprecation process, it is not too hard to evolve an API in a backwards-compatible way over a long period of time, without restricting the ability to introduce better approaches.

THE IMPORTANCE OF JUSTIFICATION

The easiest API to maintain is no API at all, and it is therefore extremely important to require justification for every method and class that forms part of our API. During the process of designing our APIs, we should frequently find ourselves asking the following question: "Is this really required?" We should assure ourselves that the API is paying for itself, returning vital functionality in return for its continued existence.

EATING YOUR OWN DOGFOOD

As API developers we have to be careful that the API we are creating is indeed useful for its stated purpose. We have to have developer empathy to look through the eyes of the developer, rather than as ourselves. The best way to be assured of this is to "eat your own dog food." In other words, it is important to ensure that the API is used throughout its development, not only by yourself, but also — and more importantly — by trusted "real world" users.

The value of bringing in "real world" users is to help prevent ourselves from losing our sense of restraint, and adding enhancements based solely on our advanced understanding of the API. "Real world" users help to balance this, to ensure we only fix what is considered truly broken.

When we write applications using our API, we should use this time to look for code that does not work cleanly (or with unclear intentions), places where there is duplicate or redundant code, and places where our API forces the developer to work at a level of abstraction that is either too low-level or too high-level.

API DOCUMENTATION

There are two kinds of documents that are critical to developers when working with an SDK: JavaDoc, and more in-depth articles (tutorials on how to get started, etc, such as those Microsoft publishes for Java on Azure). Both of these are equally important to developers, but it is important to understand that they serve different purposes. In this Refcard we will cover JavaDoc, as it is more relevant to our interests as API developers.

JavaDoc should act as the specification of the API. Engineers responsible for writing APIs should consider it part of their job to ensure that a JavaDoc is complete, with class-level and method-level overviews, specifying the expected inputs, outputs, exceptional circumstances, and other details. While this documentation acts as the specification, it is important that it does not become an overly detailed guide to the programmer, or discuss how the implementation works.

In an ideal world, the effort to create a high quality JavaDoc would go a step further to also include code snippets that users can copy/paste into their own software to kick start their own development. These code snippets need not be long screeds of code - it is best if they are constrained to no more than five to ten lines. These code snippets can be added to the JavaDoc of the relevant class or method over time, as users start to ask questions on the API.

The value of JavaDoc extends beyond offering it to other developers — it can also help us. This is because JavaDoc gives us a filtered view of our SDK by only showing API that is intended for public use. If we establish a routine of regularly generating JavaDoc we can review our API for issues such as missing JavaDoc, leaking implementation classes or external dependencies, and other things that aren't what we expect.

As most Java projects are based on Maven or Gradle, generating JavaDocs for a project is typically as simple as running `mvn javadoc:javadoc` or `gradle javadoc`, respectively. Getting into the habit of generating this documentation (especially with settings configured to fail on any error or warning) on a regular basis ensures that we are always able to spot API issues early, and remind ourselves of areas of our API that need more JavaDoc content to be written.

JAVADOC FOR BEHAVIORAL CONTRACTS

One underutilized aspect of JavaDoc is to use it to specify behavioral contracts. An example of a behavioral contract is the `Arrays.sort()` method, which guarantees that it is "stable" (that is, equal elements are not reordered). There is no way to easily specify this guarantee as part of the API itself (aside from making our API unwieldy, e.g. `Arrays.stableSort()`), but JavaDoc is an ideal location for this.

However, if we add behavioral contracts as part of our JavaDoc, this then becomes as much a part of our API as the API itself. We cannot change this behavioral contract with the same level of consideration, as it will potentially cause downstream issues for your users.

JAVADOC TAGS

JavaDoc ships with a number of tags such as `@link`, `@param`, and `@return`, which provide more context to the JavaDoc tooling, and enables a richer experience when HTML output is generated. It is extremely useful when writing JavaDoc content to keep these in the back of your mind, to ensure that they are all used when relevant. To understand when to use each of these tags, refer to the "Tag Comments" section of the Java Platform, Standard Edition Tools Reference documentation.

CONSISTENCY

Very rarely these days does software get developed by a single person, and even if it did, the human condition is so fickle that what they deem is great one day may be considered dead wrong the next. Fortunately, as we design APIs, we have a clear record of the decisions we have made in the form of our public API, and it can be quite easy to spot when something is deviating from this forming convention.

The short-term benefit to having a consistent API is that we reduce the risk of frustrating our users, and the long-term benefit is that a consistent API ensures that when an end user arrives at a new section of your API, that they are more readily able to intuit how it should be used.

Some of the more important considerations around consistency include:

RETURN TYPES

Ideally all APIs that must return a collection should be consistent, using only a few classes rather than all possible ones. A good subset of collections to return might be, for example, `List`, `Set`, and `Iterator` (and in this case, never `Collection`, `Iterable`, and `Stream` - but note that these are all valid return types - just in this example they were chosen to not be in the subset). Relatedly, if the API takes pain to not return null in most cases (for a certain return type), it is best advised to never return null for that return type.

METHOD NAMING PATTERNS

Developers rely on their IDE to auto-complete their input, so consider the importance of API naming, to ensure related concepts appear adjacent to each other in the users auto-complete popup lists. An API should have a well-established set of terms and reuse them all the time in: type names, method names, argument names, constants, etc. Method names like `Type.of(...)`, `Type.valueOf()`, `Type.toXYZ()`, `Type.from(...)`, etc. should be used consistently, and never mixed. Our goal is to have a team-wide vocabulary that we use throughout our API.

ARGUMENT ORDER

APIs that overload methods to accept different numbers or types of arguments should always ensure that the ordering is consistent and logical. In some cases the arguments we pass into a method form some logical grouping, and in these cases it may make sense to introduce an intermediate argument type that can wrap these arguments. This reduces the risk that subsequent releases of our API will need to overload the method to accept more arguments. This also helps to aid our goal of API evolvability.

MINIMIZE API

It is the natural instinct of an API developer to want to write the biggest API they can — to offer more convenience rather than less - but this leads to two concerns:

1. It can lead to API overload: developers are required to scan through and understand more of an API than is necessary to complete their job.
2. The more of an API we expose, the greater the maintenance burden we place on our future selves.

All API developers should start by understanding the critical use cases required for their API, and design their API to support these use cases. They should fight the urge to add more convenience (thinking to themselves that by adding a new method will save developers from writing a few more lines of code).

Having said this, it is important to clarify that convenience APIs serve a critically important role in any good API, especially in servicing our goal of having an understandable API. The challenge is in determining what should be accepted as valuable, and what should be rejected as not having enough value to "pay for itself."

An example of a good convenience API in the JDK is the `List.add(Object)` method, to avoid developers always having to call `List.add(int, Object)`.

When discussing this topic with Stuart Marks, an engineer on the JDK team at Oracle, he provided the following insight:

On the other hand I've seen APIs that really get bogged down in "conveniences." Here's a hypothetical example. Suppose you have an API that has `bar()` *and* `foo()` *operations. They are useful individually, but they are quite frequently used together. So you might have a* `bf()` *operation that does both. OK so far.*

Now suppose you add a `mumble()` *operation. It sort of sticks out to have to call* `bf()` *and* `mumble()` *separately, so maybe you need more convenience APIs — like* `bfm()`*. Well, what if you don't need* `foo()`*? How about* `bm()`*? And throw in* `fm()` *for good measure. Now you have seven methods, more than half of which are just combinations of the fundamental three operations. Maybe this is good, maybe not; it certainly has potential to bloat the APIs. At a certain point there are enough 'convenience' methods that they tend to outweigh the basic operations.*

Now this is mainly a matter of style. You could just do the basic operations and let the user compose them. This is the JDK style. Or you could provide ALL the combinations, so that once the user knows they system, any combination they could possibly need is already there. For an example of the latter, see Eclipse Collections.

PREVENTING LEAKS

It is important to ensure that implementation classes, and classes belonging to external dependencies, do not "leak" out into the public API as either return types from the public API or argument types into the public API. Appropriate measures should be taken to ensure these classes are hidden.

There are two main ways to hide implementation classes:

1. Put implementation classes into packages under an `impl` package. All classes under this package can then be excluded from the JavaDoc output, and documented to not be part of the API for developers to use. If this API were being developed under JDK 9 or later, a module could be defined that excludes this `impl` package from being exported (and therefore, it will not be available to developers at all).
2. Make implementation classes "package-private" (i.e. have no modifier on the class). This means that the classes are not part of the public API, and also are not able to be used by developers.

When we leak external dependencies through our API, we have unintentionally increased the surface area of our API to include all the APIs that the leaked dependency class(es) expose, and we've also made our API beholden to the whims of an API that is outside of our control. If we discover that we are exposing external dependencies in our API, we should consider if this is a desirable outcome, or if we should move to counteract this. The options that exist include removing the offending API that leaks the external dependency, and writing a wrapper class around the leaked class so that the actual class is not exposed.

UNDERSTAND PROTECTED

The `protected` keyword in Java is often misunderstood, and perhaps, overused. In short, `protected` members are for communicating with subclasses, and `public` members are for communicating with callers.

There are certainly cases where it can be a very useful tool in the API developers toolkit, but unless it is designed into a class from the start, it is often used incorrectly, leading to classes that appear to be extensible, but in practice fail to be so. In fact, sometimes the `protected` keyword can act as a form of virus, infecting the class as users of the API demand more and more of its `private` methods be made `protected` (or `public`!) to enable their requirements to be met.

In addition, API developers need to understand that `protected` methods form as much a part of their public API footprint as public API does. This is often misunderstood by beginner API developers, to their eventual detriment.

INTENTIONAL INHERITANCE

As an API developer, we must strike a balance between offering developers the functionality and flexibility that they need to perform their jobs and the ability for ourselves to evolve our API over time. One way to ensure we, as API developers, retain some level of control is to make use of the `final` keyword. By making our classes or methods `final`, we are signalling to developers that, at this point in time, they cannot consider extending or overriding these particular classes and methods.

The reason why `final` is valuable to API developers is due to the fact that sometimes, our APIs are not flawless, and instead of getting in touch to help fix things, many developers want to try to work around our flaws to patch their version, so that they can move on to the next problem. With this approach, they only create new problems for themselves, and ultimately, for us as API developers. Ideally, when a user of our API encounters a `final` class or method, they reach out to us to discuss their needs, which can lead to an even better API. The `final` keyword, after all, can always be removed in a subsequent release, but it is a wise idea to not make something `final` after it has already been released.

BACKWARDS COMPATIBILITY

This refcard, up until now, has skirted around the issue of how exactly to evolve an API. The basic advice is that adding a new API is generally fine, but removing or changing an existing API is not. The reason for this is essentially because adding an API is (generally) backwards compatible, whereas removing or changing existing an API is backwards incompatible. In other words, when we remove or change existing an API, we run the risk of breaking our users when they upgrade to our next release if they relied on this API that no longer exists in the form their code expects.

Sometimes we must make backwards incompatible changes, for example if we made a mistake in our API design or if we overlooked some aspect of the requirements that requires a different approach. The challenge is to do this in a way that is clearly communicated to our users whenever possible. Making use of the `@Deprecated` annotation (and related `@deprecated` JavaDoc tag) is a good first step, but this only works when we have a clearly articulated policy regarding when we permit breaking changes in a release. A common approach to this is to subscribe to the semantic versioning policy of only making incompatible API changes in major releases (that is, in the versioning scheme MAJOR.MINOR.PATCH, where the MAJOR value is incremented). In this approach, anything that you plan to change or remove is marked as deprecated, but not removed or modified until the next major release. If this policy is to be used, it is important to communicate this outwardly, so that users of your API can be certain of this plan.

On the other side of the coin is the case where there are accidental API breaks, introduced by developers who are unaware of the implications of a change they have made. This happens more often than ideal, and is more often than not very hard to notice. Tools exist to watch API changes and to notify you of backwards incompatibilities that have been introduced. Revapi is one such tool that continues to prove its worth on projects that I have worked on at Microsoft.

Compatibility concerns relate to far more than just naming and method signatures. Equally important, but outside of the scope of this Refcard, is that changes to behavior (i.e. implementation) can also lead to breaking changes. In fact, it has been said that every change is an incompatible change, since even a bug fix might break a user who relies on that bug!

Why should we care about backwards compatibility? Simply because breaking compatibility is really painful to our user base. There are examples of projects that have suffered quite severe consequences by playing too fast and loose with their approach to backwards compatibility.

DON'T RETURN NULL

Sir Tony Hoare called the invention of the null reference (something he created) his "billion-dollar mistake." In Java we have become so accustomed to handling some error conditions by returning `null` that it is second nature to null check everything, but in many cases there are better options, rather than returning `null`. Refer to the table below for some common examples:

RETURN TYPE	NON-NULL RETURN VALUE
String	" " (An empty string)
List/Set Map/ Iterator	Use the `Collections` class, e.g. `Collections.emptyList()`
Stream	`Stream.empty()`
Array	Return an empty, zero-lenght array
All other types	Consider using `Optional` (but read the `Optional` section below first)

By guaranteeing to return non-null values to callers of our API, our users can opt to not include the noisiness of the null check in their code base. It is important however, that should this approach be taken, that we ensure that it is applied consistently across an entire API. It is very easy to erode trust in an API if it fails to consistently apply patterns (and in failing to do so, cause the user to encounter unexpected null pointer exceptions).

UNDERSTAND WHEN TO USE OPTIONAL

Java 8 introduced `Optional` as a way of lessening the possibility of null pointer exceptions, because when a method returns `Optional`, it guarantees that it will be non-null. It is then up to the consumer of the API to determine if the returned `Optional` contains an element, or is empty. In other words, an `Optional<T>` can be thought of as a container for at most one element.

The `Optional` return type is best used in select cases when:

1. A result might not be able to be returned, and
2. The API consumer has to perform some different action in this case

Optional provides a number of convenience methods, some of which are highlighted below, in a hypothetical situation where there exists a `public Optional<Car> getFastest(List<Car> cars)` method:

```
// getFastest returns Optional<Car>, but if the cars
// list is empty, it returns Optional.empty(). In this
// case, we can choose to map this to an invalid value.
Car fastestCar = getFastest(cars).orElse(Car.INVALID);

// If the orElse case is expensive to calculate, we can
// also use a Supplier to only generate the alternate
// value if the Optional is empty
Car fastestCar = getFastest(cars).orElseGet(() ->
searchTheWeb());

// We could alternatively throw an exception
Car fastestCar = getFastest(cars).
orElseThrow(MissingCarsException::new);

// We can also provide a lambda expression to operate
// on the value, if it is not empty
getFastest(cars).ifPresent(this::raceCar)
```

The examples above all demonstrate good times for an API to return `Optional`. On the other hand, if most users of an API that returns `Optional` write code as shown below, it is possible to argue that this is an object-oriented version of a null-reference check, and is probably no better (or any more intuitive) than simply declaring that this particular API will return `null` rather than `Optional`.

```
// Whilst it is ok to call get() directly on an
// Optional, you risk a NoSuchElementException if it is
// empty. You can wrap it with an isPresent() call as
// shown below, but if your API is commonly used like
// this, it suggests that Optional might not be the
// right return type
Optional<Car> result = getFastest(cars);
if (result.isPresent()) {
  result.get().startCarRace();
}
```

There are two final rules when it comes to returning an `Optional` in API:

1. Never return `Optional<Collection<T>>` from a method, as this can be more succinctly represented by simply returning `Collection<T>` with an empty collection (as mentioned earlier in this refcard).
2. Never, ever return `null` from a method that has a return type of `Optional`!

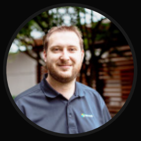
CONCLUSION

This document has covered an array of considerations that all engineers should have in the back of their mind whenever they are writing code that has any public API. At a high level, readers should put this refcard down with an appreciation for the importance of considered API design. If it wasn't already present, readers should start to sense that there is an art form to API design, and that improving our skills in this area comes about through practice and feedback - from our mentors and from our users. As with many art forms, API design succeeds not by seeing how much can be put in, but by seeing how much can be taken out. The challenge therefore is minimalism, it is consistency, it is intentionalism, and above all, it is consideration - for an API, but more importantly, for the end user of the API. We must constantly practice developer empathy to ensure we keep our end users needs sharply in perspective.

Regardless of whether we are writing API for our own consumption, for others in our organization, or more broadly as part of an open source project or commercial library, taking into consideration the values outlined in this refcard will hopefully help to direct readers to a higher quality and more professional outcome. This should not be looked at simply as "more work," but as a challenge to ourselves to fixate on the artistic endeavour of making an enjoyable, functional, productive API for our users.

ACKNOWLEDGEMENTS

Drafts of this refcard were reviewed by a number of people, and their feedback helped improve this refcard immensely. Thanks to the following people: Abhinay Agarwal, Brian Benz, Bruno Borges, Lukas Eder, Stuart Marks, Theresa Nguyen, Kevin Rushforth, Eugene Ryzhikov, Johan Vos, and Ruth Yakubu.



Written by **Jonathan Giles**, Senior Cloud Developer Advocate, Microsoft

Jonathan Giles has focused on Java for a very long time. He spent nine years at Sun Microsystems / Oracle working as a tech lead in the JDK team, honing his API design skills, and today works as a Cloud Developer Advocate at Microsoft, focused on Java, and especially developer experience. He is a Java Champion, frequent conference presenter, author, blogger, Duke's Choice Award winner, open source contributor, and JavaOne Rockstar. You can find him on Twitter @JonathanGiles, and on the web at jonathangiles.net.

DZone, Inc.
150 Preston Executive Dr. Cary, NC 27513
888.678.0399 919.678.0300