<https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/>

# 10 Subtle Best Practices when Coding Java

Average [Josh Bloch Effective Java](http://www.oracle.com/technetwork/articles/javase/bloch-effective-08-qa-140880.html) rule.

here contains less common situations involving [API / SPI](http://stackoverflow.com/q/2954372/521799) design that may have a big effect nontheless.

I have encountered these things while writing and maintaining [jOOQ](http://www.jooq.org/), an [internal DSL](https://blog.jooq.org/2012/01/05/the-java-fluent-api-designer-crash-course/)modelling SQL in Java. Being an internal DSL, jOOQ challenges Java compilers and generics to the max, [combining generics, varargs and overloading](http://stackoverflow.com/q/5361513/521799) in a way that Josh Bloch probably wouldn’t recommend for the “average API”.

### 1. Remember C++ destructors

Remember [C++ destructors](http://msdn.microsoft.com/en-us/library/6t4fe76c.aspx)? No? Then you might be lucky as you never had to debug through any code leaving memory leaks due to allocated memory not having been freed after an object was removed. Thanks Sun/Oracle for implementing garbage collection!

But nonetheless, destructors have an interesting trait to them. It often makes sense to free memory in the inverse order of allocation. Keep this in mind in Java as well, when you’re operating with destructor-like semantics:

* When using @Before and @After JUnit annotations
* When allocating, freeing JDBC resources
* When calling super methods

There are various other use cases. Here’s a concrete example showing how you might implement some event listener SPI:

|  |
| --- |
| @Override  public void beforeEvent(EventContext e) {      super.beforeEvent(e);      // Super code before my code  }    @Override  public void afterEvent(EventContext e) {      // Super code after my code      super.afterEvent(e);  } |

Another good example showing why this can be important is the [infamous Dining Philosophers problem](http://stackoverflow.com/q/8274098/521799). More info about the dining philosophers can be seen in this awesome post:  
<http://adit.io/posts/2013-05-11-The-Dining-Philosophers-Problem-With-Ron-Swanson.html>

**The Rule**: Whenever you implement logic using before/after, allocate/free, take/return semantics, think about whether the after/free/return operation should perform stuff in the inverse order.

### 2. Don’t trust your early SPI evolution judgement

Providing an SPI to your consumers is an easy way to allow them to inject custom behaviour into your library / code. Beware, though, that your SPI evolution judgement may trick you into thinking that you’re [*(not) going to need that additional parameter*](http://en.wikipedia.org/wiki/You_aren't_gonna_need_it). True, no functionality [should be added early](http://www.extremeprogramming.org/rules/early.html). But once you’ve published your SPI and once you’ve decided following [semantic versioning](http://semver.org/), you’ll really regret having added a silly, one-argument method to your SPI when you realise that you might need another argument in some cases:

|  |
| --- |
| interface EventListener {      // Bad      void message(String message);  } |

What if you also need a message ID and a message source? API evolution will prevent you from adding that parameter easily, to the above type. Granted, with Java 8, you could add a defender method, to “defend” you bad early design decision:

|  |
| --- |
| interface EventListener {      // Bad      default void message(String message) {          message(message, null, null);      }      // Better?      void message(          String message,          Integer id,          MessageSource source      );  } |

Note, unfortunately, the defender method [cannot be made final](https://blog.jooq.org/2011/12/18/java-8-virtual-extension-methods/).

But much better than polluting your SPI with dozens of methods, use a [context object (or argument object)](https://blog.jooq.org/2012/04/10/the-visitor-pattern-re-visited/) just for this purpose.

|  |
| --- |
| interface MessageContext {      String message();      Integer id();      MessageSource source();  }   interface EventListener {      // Awesome!      void message(MessageContext context);  } |

You can evolve the MessageContext API much more easily than the EventListener SPI as fewer users will have implemented it.

**The Rule**: Whenever you specify an SPI, consider using context / parameter objects instead of writing methods with a fixed amount of parameters.[[tweet this](http://twitter.com/home?status=Rule+No+2:+Consider+using+parameter+objects+instead+of+methods+with+a+fixed+amount+of+parameters.+By+@JavaOOQ+https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/)](http://twitter.com/home?status=Rule+No+2:+Consider+using+parameter+objects+instead+of+methods+with+a+fixed+amount+of+parameters.+By+@JavaOOQ+https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/)

**Remark**: It is often a good idea to also communicate results through a dedicated MessageResult type, which can be constructed through a builder API. This will add even more SPI evolution flexibility to your SPI.

### 3. Avoid returning anonymous, local, or inner classes

Swing programmers probably have a couple of keyboard shortcuts to generate the code for their hundreds of anonymous classes. In many cases, creating them is nice as you can locally adhere to an interface, without going through the “hassle” of thinking about a full SPI subtype lifecycle.

But you should not use anonymous, local, or inner classes too often for a simple reason: They keep a reference to the outer instance. And they will drag that outer instance to wherevery they go, e.g. to some scope outside of your local class if you’re not careful. This can be a major source for memory leaks, as your whole object graph will suddenly entangle in subtle ways.

**The Rule**: Whenever you write an anonymous, local or inner class, check if you can make it static or even a regular top-level class. Avoid returning anonymous, local or inner class instances from methods to the outside scope.

**Remark**: There has been some clever practice around double-curly braces for simple object instantiation:

|  |  |
| --- | --- |
| 1  2  3  4 | new HashMap<String, String>() {{      put("1", "a");      put("2", "b");  }} |

This leverages Java’s instance initializer as [specified by the JLS §8.6](http://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.6). Looks nice (maybe a bit weird), but is really a bad idea. What would otherwise be a completely independent HashMap instance now keeps a reference to the outer instance, whatever that just happens to be. Besides, you’ll create an additional class for the class loader to manage.

### 4. Start writing SAMs now!

Java 8 is knocking on the door. And with [Java 8 come lambdas](http://cr.openjdk.java.net/~briangoetz/lambda/sotc3.html), whether you like them or not. Your API consumers may like them, though, and you better make sure that they can make use of them as often as possible. Hence, unless your API accepts simple “scalar” types such as int, long, String, Date, let your API accept SAMs as often as possible.

What’s a SAM? A SAM is a Single Abstract Method [Type]. Also known as a [functional interface](http://java.dzone.com/articles/introduction-functional-1), soon to be annotated with the [@FunctionalInterface annotation](http://download.java.net/lambda/b78/docs/api/java/lang/FunctionalInterface.html). This goes well with rule number 2, where EventListener is in fact a SAM. The best SAMs are those with single arguments, as they will further simplify writing of a lambda. Imagine writing

|  |
| --- |
| listeners.add(c -> System.out.println(c.message())); |

Instead of

|  |
| --- |
| listeners.add(new EventListener() {      @Override      public void message(MessageContext c) {          System.out.println(c.message()));      }  }); |

Imagine XML processing through [jOOX](https://code.google.com/p/joox/), which features a couple of SAMs:

|  |
| --- |
| $(document)      // Find elements with an ID      .find(c -> $(c).id() != null)      // Find their  child elements      .children(c -> $(c).tag().equals("order"))      // Print all matches      .each(c -> System.out.println($(c))) |

**The Rule**: Be nice with your API consumers and write SAMs / Functional interfaces already now.[[tweet this](http://twitter.com/home?status=Rule+No+4:+Use+SAMs+or+functional+interfaces+already+NOW.+By+@JavaOOQ+https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/)](http://twitter.com/home?status=Rule+No+4:+Use+SAMs+or+functional+interfaces+already+NOW.+By+@JavaOOQ+https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/)

**Remarks**: A couple of interesting blog posts about Java 8 Lambdas and improved Collections API can be seen here:

* <http://blog.informatech.cr/2013/04/10/java-optional-objects/>
* <http://blog.informatech.cr/2013/03/25/java-streams-api-preview/>
* <http://blog.informatech.cr/2013/03/24/java-streams-preview-vs-net-linq/>
* <http://blog.informatech.cr/2013/03/11/java-infinite-streams/>

### 5. Avoid returning null from API methods

I’ve blogged about Java’s [NULLs](https://blog.jooq.org/2013/08/12/what-if-every-object-was-an-array-no-more-nullpointerexceptions/) once or twice. I’ve also blogged about Java 8’s introduction of [Optional](https://blog.jooq.org/2013/04/11/on-java-8s-introduction-of-optional/). These are interesting topics both from an academic and from a practical point of view.

While NULLs and NullPointerExceptions will probably stay a major pain in Java for a while, you can still design your API in a way that users will not run into any issues. Try to avoid returning null from API methods whenever possible. Your API consumers should be able to chain methods whenever applicable:

|  |  |
| --- | --- |
| 1 | initialise(someArgument).calculate(data).dispatch(); |

In the above snippet, none of the methods should ever return null. In fact, using null’s semantics (the absence of a value) should be rather exceptional in general. In libraries like [jQuery](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/jquery.com) (or [jOOX](https://code.google.com/p/joox/), a Java port thereof), nulls are completely avoided as you’re [always operating on iterable objects](https://blog.jooq.org/2013/08/12/what-if-every-object-was-an-array-no-more-nullpointerexceptions/). Whether you match something or not is irrelevant to the next method call.

Nulls often arise also because of lazy initialisation. In many cases, lazy initialisation can be avoided too, without any significant performance impact. In fact, lazy initialisation should be used carefully, only. If large data structures are involved.

**The Rule**: Avoid returning nulls from methods whenever possible. Use null only for the “uninitialised” or “absent” semantics.

### 6. Never return null arrays or lists from API methods

While there are some cases when returning nulls from methods is OK, there is absolutely no use case of returning null arrays or null collections! Let’s consider the hideous [java.io.File.list()](http://docs.oracle.com/javase/7/docs/api/java/io/File.html" \l "list()) method. It returns:

An array of strings naming the files and directories in the directory denoted by this abstract pathname. The array will be empty if the directory is empty. Returns null if this abstract pathname does not denote a directory, or if an I/O error occurs.

Hence, the correct way to deal with this method is

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | File directory = // ...    if (directory.isDirectory()) {      String[] list = directory.list();        if (list != null) {          for (String file : list) {              // ...          }      }  } |

Was that null check really necessary? Most I/O operations produce IOExceptions, but this one returns null. Null cannot hold any error message indicating why the I/O error occurred. So this is wrong in three ways:

* Null does not help in finding the error
* Null does not allow to distinguish I/O errors from the File instance not being a directory
* Everyone will keep forgetting about null, here

In collection contexts, the notion of “absence” is best implemented by empty arrays or collections. Having an “absent” array or collection is hardly ever useful, except again, for lazy initialisation.

**The Rule**: Arrays or Collections should never be null.

### 7. Avoid state, be functional

What’s nice about HTTP is the fact that it is stateless. All relevant state is transferred in each request and in each response. This is essential to the naming of REST: [Representational State Transfer](http://en.wikipedia.org/wiki/Representational_State_Transfer). This is awesome when done in Java as well. Think of it in terms of rule number 2 when methods receive stateful parameter objects. Things can be so much simpler if state is transferred in such objects, rather than manipulated from the outside. Take JDBC, for instance. The following example fetches a cursor from a stored procedure:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | CallableStatement s =    connection.prepareCall("{ ? = ... }");    // Verbose manipulation of statement state:  s.registerOutParameter(1, cursor);  s.setString(2, "abc");  s.execute();  ResultSet rs = s.getObject(1);    // Verbose manipulation of result set state:  rs.next();  rs.next(); |

These are the things that make JDBC such an awkward API to deal with. Each object is incredibly stateful and hard to manipulate. Concretely, there are two major issues:

* It is very hard to correctly deal with stateful APIs in multi-threaded environments
* It is very hard to make stateful resources globally available, as the state is not documented

Theatrical poster for Forrest Gump, Copyright © 1994 by [Paramount Pictures](http://www.paramount.com/). All Rights Reserved. It is believed that the above usage fulfils what is known as [Fair Use](http://en.wikipedia.org/wiki/Fair_use)

**The Rule**: Implement more of a functional style. Pass state through method arguments. Manipulate less object state.[[tweet this](http://twitter.com/home?status=Rule+No+7:+Avoid+state.+Implement+more+of+a+functional+style.+By+@JavaOOQ+https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/)](http://twitter.com/home?status=Rule+No+7:+Avoid+state.+Implement+more+of+a+functional+style.+By+@JavaOOQ+https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/)

### 8. Short-circuit equals()

This is a low-hanging fruit. In large object graphs, you can gain significantly in terms of performance, if all your objects’ equals() methods dirt-cheaply compare for identity first:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Override  public boolean equals(Object other) {      if (this == other) return true;        // Rest of equality logic...  } |

Note, other short-circuit checks may involve null checks, which should be there as well:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Override  public boolean equals(Object other) {      if (this == other) return true;      if (other == null) return false;        // Rest of equality logic...  } |

**The Rule**: Short-circuit all your equals() methods to gain performance.[[tweet this](http://twitter.com/home?status=Rule+No+8:+Short-circuit+all+your+equals+methods+to+gain+performance.+By+@JavaOOQ+https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/)](http://twitter.com/home?status=Rule+No+8:+Short-circuit+all+your+equals+methods+to+gain+performance.+By+@JavaOOQ+https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/)

### 9. Try to make methods final by default

Some will disagree on this, as making things final by default is quite the opposite of what Java developers are used to. But if you’re in full control of all source code, there’s absolutely nothing wrong with making methods final by default, because:

* If you do need to override a method (do you really?), you can still remove the final keyword
* You will never accidentally override any method anymore

This specifically applies for static methods, where “overriding” (actually, shadowing) hardly ever makes sense. I’ve come across a very bad example of shadowing static methods in [Apache Tika](http://tika.apache.org/), recently. Consider:

* [TaggedInputStream.get(InputStream)](http://tika.apache.org/1.4/api/org/apache/tika/io/TaggedInputStream.html#get(java.io.InputStream))
* [TikaInputStream.get(InputStream)](http://tika.apache.org/1.4/api/org/apache/tika/io/TikaInputStream.html#get(java.io.InputStream))

TikaInputStream extends TaggedInputStream and shadows its static get() method with quite a different implementation.

Unlike regular methods, static methods don’t override each other, as the call-site binds a static method invocation at compile-time. If you’re unlucky, you might just get the wrong method accidentally.

**The Rule**: If you’re in full control of your API, try making as many methods as possible final by default.[[tweet this](http://twitter.com/home?status=Rule+No+9:+Stay+in+control+of+your+API+by+making+as+many+methods+as+possible+final+by+default.+By+@JavaOOQ+https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/)](http://twitter.com/home?status=Rule+No+9:+Stay+in+control+of+your+API+by+making+as+many+methods+as+possible+final+by+default.+By+@JavaOOQ+https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/)

### 10. Avoid the method(T…) signature

There’s nothing wrong with the occasional “accept-all” varargs method that accepts an Object... argument:

|  |  |
| --- | --- |
| 1 | void acceptAll(Object... all); |

Writing such a method brings a little JavaScript feeling to the Java ecosystem. Of course, you probably want to restrict the actual type to something more confined in a real-world situation, e.g. String.... And because you don’t want to confine too much, you might think it is a good idea to replace Object by a generic T:

|  |  |
| --- | --- |
| 1 | void acceptAll(T... all); |

But it’s not. T can always be inferred to Object. In fact, you might as well just not use generics with the above methods. More importantly, you may think that you can overload the above method, but you cannot:

|  |  |
| --- | --- |
| 1  2 | void acceptAll(T... all);  void acceptAll(String message, T... all); |

This looks as though you could optionally pass a String message to the method. But what happens to this call here?

|  |  |
| --- | --- |
| 1 | acceptAll("Message", 123, "abc"); |

The compiler will infer <? extends Serializable & Comparable<?>> for T, which makes the call ambiguous!

So, whenever you have an “accept-all” signature (even if it is generic), you will never again be able to typesafely overload it. API consumers may just be lucky enough to “accidentally” have the compiler chose the “right” most specific method. But they may as well be tricked into using the “accept-all” method or they may not be able to call any method at all.

**The Rule**: Avoid “accept-all” signatures if you can. And if you cannot, never overload such a method.[[tweet this](http://twitter.com/home?status=Rule+No+10:+Avoid+accept+all+signatures+if+you+can.+By+@JavaOOQ+https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/)](http://twitter.com/home?status=Rule+No+10:+Avoid+accept+all+signatures+if+you+can.+By+@JavaOOQ+https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/)

### Conclusion

Java is a beast. Unlike other, fancier languages, it has evolved slowly to what it is today. And that’s probably a good thing, because already at the speed of development of Java, there are hundreds of caveats, which can only be mastered through years of experience.

[](http://www.jooq.org/)

Stay tuned for more top 10 lists on the subject!

* [124Click to share on Facebook (Opens in new window)124](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?share=facebook&nb=1)
* [Click to share on LinkedIn (Opens in new window)](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?share=linkedin&nb=1)
* [Click to share on Twitter (Opens in new window)](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?share=twitter&nb=1)
* [Click to share on Reddit (Opens in new window)](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?share=reddit&nb=1)
* [More](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/)

# Post navigation

[**PREVIOUS**](https://blog.jooq.org/2013/08/19/do-you-put-trust-in-vendors-nosql-promises/)

[**NEXT**](https://blog.jooq.org/2013/08/21/the-myth-about-slow-sql-join-operations/)

## 53 thoughts on “10 Subtle Best Practices when Coding Java”

1. As for the SPI thing in point #2. While this is true that creating SPI and for that matter creating API is more an art than engineering I would not state those techniques without critic. When you create any xPI (unless x = chocolate) you can give more or less freedom to the user. Giving less freedom will help the casual user avoid wrong use during compile time. After all that is, for example, why you implement JOOX and JOOQ as fluent API and not just simple method chaining. If done wrong being too restrictive may prevent backward compatible development of the xPI. On the loose end you can pass Map objects as parameters and then you have the total freedom what and how you implement. But this was more than 15 years ago when I was programming Perl and I did not start Java in 2006 to do it the Perl style. So better let’s not get that far, or you may face in an imaginary project that I will reject your git pull request.

What you propose is somewhere in the middle. Sounds reasonable, but certain circumstances may prove it too restrictive or too loose. Java API was regarded as fairly good at the time. Nowadays we ask many questions, like “why exception is not an interface”, “why there is no immutableX in JDK extended by the interface X (Set, Map etc)” and my questions similar.

At the bottom line, there is only one thing sure: whatever xPI you design, it will not be a good one in the long run. The only question is: how long will it be good. But then if it is good for a long time, perhaps because there is no use and development of it, which would also be sad. You just never know.

[*Peter Verhas*](http://tifyty.wordpress.com/) [, August 20, 2013 at 18:21](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-2510)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=2510#respond)

* + After all that is, for example, why you implement JOOX and JOOQ as fluent API and not just simple method chaining.

Note, jOOX performs method chaining, just like jQuery.

why exception is not an interface

Now THAT would be awesome :-)

At the bottom line, there is only one thing sure: whatever xPI you design, it will not be a good one in the long run. The only question is: how long will it be good.

Nicely put. Obviously, with [semantic versioning](http://semver.org/), you’re allowed to break your xPI in each major release. So that should define the timespan of how long an xPI should remain “good” before you need to alter it incompatibly.

[*lukaseder*](https://lukaseder.wordpress.com/) [, August 21, 2013 at 09:06](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-2512)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=2512#respond)

* + - This is not about semantic versioning. With semantic versioning you are allowed to break your xPI in each major release. very true. However without it you are also allowed. Semver is only a tool to deliver the message to the users of the xPI what changes there are between releases.

Breaking backward compatibility is a marketing decision. (Marketing in the wider meaning, including market research.) What are the advantages of the new xPI and what are the disadvantages of backward compatibility flaws. There will be customers jumping on the train at the very first moment (hopefully not falling off under the wheels), and others using the old xPI for a long time before changing. Some may churn stopping to use your product on the long run. You should have a model and estimate your income and costs measured in dollars in case of a commercial product or in other measurable quantity in case of OS project and try to maximize the difference (profit). You have to make the new version easy to accommodate to change for from the old xPI in the mutual interest for the change among your customers. If your change is radical, hard to accomplish be prepared and have costs allocated in due time to support the old version for a long time.

There are many factors, and they are more marketing that software engineering.

EXAMPLE: There is log4j still being the most widely used logging framework. And there is slf4j and logback, developed by the same person, and much more advanced than the last version of log4j. Users still use the old version. Why? I see two reasons:

1. Because log4j is the standard logging framework and management does not like us to use something that is not widely used. (Why they did not name it log4j version whatever the next version is? Probably a marketing mistake.)

2. Log4j is good enough, there is no demanding need for the change.

3. API is not fully compatible, needs some work and (more important!) understanding some new structure. Should I waste my brain for such technology as logging? Logging is just the least important tool that I use and it should just work.

[*Peter Verhas*](http://tifyty.wordpress.com/) [, August 21, 2013 at 09:30](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-2513)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=2513#respond)

* + - * True words. As always, it’s about the added value to customers. This includes incompatible changes that will result in excess work and potential malfunction at the use site. “Cleaning up dirty internals” is not a good reason to publish a new version with a new API. Improving performance by 80% or adding 70 new features is.

Some companies are incredibly successful in making their customers believe that previous versions are no longer supported and they indeed need the latest minor version because of an improved button rendering and a better exposé animation and they even pay for it (Apple).

Other companies may have made an early mistake in their sales process and need to support their legacy for ages providing patches, security updates, service packs and what not (Microsoft, Sun Microsystems with the JVM).

And yes, then, there are the popular yet “irrelevant” log4j-style libraries which have come to be taken for granted. I’ve had an interesting talk with Thomas Müller, the [H2 developer](http://www.h2database.com/), about attaining market share. It’s just extremely hard, especially if you’re in a business where everything has been said.

[*lukaseder*](https://lukaseder.wordpress.com/) [, August 21, 2013 at 09:42](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-2514)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=2514#respond)

1. Awesome post. Although I already follow these principles for sometime, it’s always good to see some programmers that we respect following the same thing.

Keep the good work o/

Caesar Ralf [, August 23, 2013 at 18:26](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-2535)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=2535#respond)

1. Ad. 9 – what about mocking? many decent mocking frameworks do not play nicely with final methods

cheers! (and thanks for this very interesting article!)

[*tomekkaczanowski*](http://practicalunittesting.com/) [, September 9, 2013 at 23:50](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-2725)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=2725#respond)

* + As I said – some will disagree on this ;-)

With respect to mocking, from my [jOOQ](http://www.jooq.org/) experience, I can say this:

* + - I have never felt the urge for mocking. I personally prefer testing a blackbox, which probably qualifies as integration testing rather than unit testing
    - jOOQ’s API is implemented through interfaces. They can still be mocked, even if the implementation is mostly final
    - The example mostly referred to static final methods, where mocking is hard anyway (which is why I like “blackbox-testing”)

[*lukaseder*](https://lukaseder.wordpress.com/) [, September 10, 2013 at 07:28](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-2731)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=2731#respond)

1. It might help if you actually tell us what SPI means. One of these things?

<http://www.acronymfinder.com/SPI.html>

Dan [, February 28, 2014 at 14:06](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-14746)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=14746#respond)

* + SPI usually means [Service Provider Interface](http://en.wikipedia.org/wiki/Service_provider_interface). This is commonly used to provide an API with extensions / services. This Stack Overflow question may further help:

[http://stackoverflow.com/questions/2954372/difference-between-spi-and-api](http://stackoverflow.com/q/2954372/521799)

[*lukaseder*](https://lukaseder.wordpress.com/) [, February 28, 2014 at 14:20](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-14748)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=14748#respond)

1. I don’t understand your point 9 “Try to make methods final by default” for static methods. Even if your static method is final, it doesn’t prevent someone from defiining a method with the same name and signature in a subclass.

Ric [, March 2, 2014 at 10:01](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-14875)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=14875#respond)

* + Sorry, I was wrong in my comment dated March 2, I have just written. Delete it.

Ric [, March 2, 2014 at 10:04](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-14876)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=14876#respond)

* + - No worries. I’ll leave it for future reference

[*lukaseder*](https://lukaseder.wordpress.com/) [, March 2, 2014 at 10:05](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-14877)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=14877#respond)

* + I actually ran into an issue with the example from [Apache Tika](https://tika.apache.org/) once. The subtype shadowed the supertype’s static method(s), “overriding” it with a different semantics. This can lead to very dangerous bugs.

It’s a corner-case, though. If your API is not widely reused, you probably don’t need to worry about making all static methods final as well.

[*lukaseder*](https://lukaseder.wordpress.com/) [, March 2, 2014 at 10:08](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-14879)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=14879#respond)

1. *Pingback:*[*Java 8 Friday: The Dark Side of Java 8 | Java, SQL and jOOQ.*](https://blog.jooq.org/2014/04/04/java-8-friday-the-dark-side-of-java-8/)
2. Nice list. Returning Empty object instead of Null has yield more benefit for us then optional or any other trick, Agree that its not always possible, but whenever possible, I prefer it over Optional. It result in much cleaner and readable code. By the I have also shared [few practices to avoid NPE in Java](http://javarevisited.blogspot.com/2013/05/ava-tips-and-best-practices-to-avoid-nullpointerexception-program-application.html), may be your reader find it useful as well.

JP [, May 4, 2014 at 14:17](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-28323)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=28323#respond)

* + Yes, the “null object pattern” or “empty object instead of null” can help, but it is only possible, if you’re constructing your own types. You cannot apply that technique with, for instance, java.lang.Integer. So, I wonder if from a consistency point of view, this is really a viable solution in an application?

[*lukaseder*](https://lukaseder.wordpress.com/) [, May 5, 2014 at 09:06](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-28511)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=28511#respond)

1. *Pingback:*[*Three-State Booleans in Java | Java, SQL and jOOQ.*](https://blog.jooq.org/2014/05/05/three-state-booleans-in-java/)
2. Great article, thank you. Slight disagreement on “final”, though. The “if you are in full control of all source code” is the tipping point for me. If someone else is ever going to need to use (extend, fix, test, etc.) that software, ‘final’ frequently puts a big orange-painted concrete traffic block in their way. So I guess I’m agreeing with your point, but disagreeing with the premise: we’re never in full control throughout all time of any software — so don’t make it harder for the next guy/gal.

[*Charles Roth*](http://thedance.net/) [, May 5, 2014 at 20:43](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-28645)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=28645#respond)

* + Thanks for the feedback. Yes, that part about final may be a bit controversial. Although, as an API designer, I don’t want API consumers tweaking things that were not meant to be tweaked. I personally prefer the “classic” approach of feature request, careful design and planning and then implementation of a clean SPI. After all, in the case of Open Source, there is always a way around such limitations. Be it overriding a method, or patching its sources.

[*lukaseder*](https://lukaseder.wordpress.com/) [, May 5, 2014 at 22:48](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-28672)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=28672#respond)

* + - Fair enough. Sometimes I have to finagle my way around someone else’s (final) API when I need to mock something in order to unit-test my code. That can be a nuisance. OTOH, if the API Is carefully designed (as I think you’re suggesting, if I understand you correctly) to an interface, then it’s not an issue.

[*Charles Roth*](http://thedance.net/) [, May 5, 2014 at 23:04](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-28677)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=28677#respond)

* + - * OTOH, if the API Is carefully designed (as I think you’re suggesting, if I understand you correctly) to an interface, then it’s not an issue.

Yes, because all the potential mocking “contact points” are well-defined in [SPIs](http://en.wikipedia.org/wiki/Service_provider_interface). In a database API, this could go as far as providing a single [MockConnection](http://www.jooq.org/doc/latest/manual/tools/jdbc-mocking/) to mock the entirety of JDBC.

[*lukaseder*](https://lukaseder.wordpress.com/) [, May 6, 2014 at 07:57](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-28856)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=28856#respond)

* + - Sadly, I have to disagree with you, I think final is a violation of SOLID, in fact I once tried to extend a Map implementation (to change the Iterator), only to find out I would have had to reimplement the entire class (copy from java source code) because half of the things were final. IMHO, you should never make any “released to the public” API final. here’s the blog I wrote on that subject (and other matters of java privacy ) <http://www.xenoterracide.com/2014/06/java-privacy-broken-by-design.html> I do wish java required the @Override annotation, so that you’d get a compile error without it.

xenoterracide [, September 4, 2014 at 04:16](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-55204)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=55204#respond)

* + - * I understand people’s concerns with “final” methods that they want to override. At some point they want to override an Iterator (Did you consistently override entrySet(), values(), and keySet() behaviour?), at another they want to change things more heavily. Most of the time, what was a tweak at first turns into relied-upon new functionality that will almost inevitably break when the patched library is upgraded.

I personally find this to be specifically true for patches to collections behaviour. I’d always prefer writing my own map over patching someone else’s “just for that one occasion”.

In other words, if you’re willing to patch things on this level – against the design ideas of the author of the library – then you might as well just patch the library source code itself. In most cases and in the long run, however, it should be better to place a feature request with the author such that your new functionality can be officially supported and integration-tested by the vendor.

[*lukaseder*](https://lukaseder.wordpress.com/) [, September 4, 2014 at 07:07](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-55330)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=55330#respond)

* + - * + in the case of Map, submitting patches upstream wouldn’t have worked, IIRC, all I wanted to do was make .add work on values derived collection, it normally doesn’t work because there’s no key in a normal map, in my case, the key is also in the entity, so it could be made to work. We could argue whether this is \*actually\* a good idea.

a better example perhaps is the unextensable at all UUID class, of which I wanted to add base64 encoded behavior to, whilst allowing classes than know how to deal with instanceof UUID to continue to work. because the class is final, you can’t do that, it might be valid to request the behavior in Java, but because there’s no base64 spec for UUID, AFAIK, that’s likely to be rejected.

patching the source code, clone and own is fraught with it’s own problems (now you have to maintain a fork, and merges). Of course any fork should submit patches upstream, but as with my above examples, neither of those would ever, and correctly so, be accepted.

It is my belief that when releasing code to the public, you should allow consumers to do as much as possible without modifying your code (Open Closed Principle). If it is an internal project (such as my work code) then you’d be correct, it doesn’t matter.

xenoterracide [, September 4, 2014 at 15:33](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-55632)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=55632#respond)

all I wanted to do was make .add work on values derived collection […] We could argue whether this is \*actually\* a good idea.

Nope, that certainly isn’t a good idea. :-) And technically, it isn’t related to the final discussion because what you wanted to do was prohibited by the [Map contract](http://docs.oracle.com/javase/8/docs/api/java/util/Map.html#values--) in the first place:

The collection supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Collection.remove, removeAll, retainAll and clear operations. It does not support the add or addAll operations.

…

[…] UUID class, of which I wanted to add base64 encoded behavior to […]

I can see the point, but in principle, there’s no reason not to solve this via a utility method:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class Utils {      public static String encode(UUID uuid) {          // [...]      }  } |

In fact, one might even argue that from an OO design perspective, base64 encoding does not belong in the UUID class as it is completely unrelated. What about url-encoding, hex encoding, binary encoding, rot13 encoding, md5 hashing, sha2 hashing, etc. etc.

No, I think you’re barking up the wrong tree. Inheritance as a tool to patch behaviour is not a long-term solution. Inheritance as a tool to enhance bahviour is even worse. Inheritance was much overrated in the beginning of OO history, and broke a concept that was older than OO: encapsulation. We may agree to disagree here, but I rest my case. Final should be the default in almost every case. [I think that many people nowadays agree](http://whiley.org/2011/12/06/final-should-be-default-for-classes-in-java/).

[*lukaseder*](https://lukaseder.wordpress.com/) [, September 4, 2014 at 17:44](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-55682)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=55682#respond)

Ummm… that very article links to people who disagree! :-)<http://www.ibm.com/developerworks/java/library/j-jtp1029/index.html>

I know we’ve had a very useful conversation about this already, but just to remind other folks on this thread: much of the debate around ‘final’ depends on the context: is this a public API, or is this internal code? In the former context, I agree there are some good arguments for final. In the latter case, final is almost always a BAD idea.

It also goes hand-in-hand with the testing question: I would argue that if you don’t have near-100% unit-test code coverage of your API implementation, then you have absolutely no business making the API final… because it isn’t!

[*Charles Roth*](http://thedance.net/) [, September 4, 2014 at 18:55](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-55722)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=55722#respond)

is this a public API, or is this internal code?

I don’t see the point of the distinction. The only reason why final is generally omitted is because it’s not the default and people are lazy. Well-designed “internal” APIs are just as important as well-designed “public” APIs. Of course, chances are, that you are only using well-designed “public” APIs (because you have a choice), whereas “internal” APIs tend to suck (because the intern / the architect / the drunk ourselves made it so the other day)…

if you don’t have near-100% unit-test code coverage of your API implementation, then you have absolutely no business making the API final… because it isn’t!

That’s intriguing. Would you mind elaborating (genuinely curious)?

[*lukaseder*](https://lukaseder.wordpress.com/) [, September 4, 2014 at 19:39](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-55759)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=55759#respond)

In reverse order…

If “you” (not you personally, of course! :-) haven’t fully tested your API implementation, then that means those of us using your API may well end up having to do the testing… as part of our own testing of our components that use your API.

Worse… often in testing our components, we may need to mock out parts of your API. If they are final, that can be a real barrier to making the mocks. (Yes, of course you can do it with PowerMock, but to me that’s already making a deal with the Devil…)

I think the real issue, that can sometimes cause for well-intentioned disagreements, is that Java the language was not designed for testability. So in terms of code purity, final often makes sense, but in terms of testability, it’s a monkey around our necks (metaphor?). What may look like a disagreement about final can often really be a difference in context: “doing the right thing” vs “being able to test that it’s right”. Both sides are right.

[*Charles Roth*](http://thedance.net/) [, September 4, 2014 at 21:23](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-55805)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=55805#respond)

* These are interesting thoughts, and I now better understand your point – even if I still strongly disagree ;-)

then that means those of us using your API may well end up having to do the testing

That, of course, presumes that you (also not “you”, of course ;-) ) are picking random, mediocre (and probably free OSS) APIs without proper evaluation, instead of building upon a potentially much better, possibly commercial API that is already tested.

Worse… often in testing our components, we may need to mock out parts of your API.

[We would certainly disagree on that, too.](https://blog.jooq.org/2014/06/26/stop-unit-testing-database-code/)

Java the language was not designed for testability

I’ve never had issues in that respect. I see all code modules as black boxes. I can easily set up well-known initialised state (e.g. the database), then run all sorts of services on the black box, and see if the black box as a whole returns the right thing, before I reset the state. There’s simply no need for mocks if you have APIs.

Of course, again, if you don’t trust your API suppliers and vendors, that might be another story… But again, there might be another remedy, too.

Since we’re on the jOOQ blog, I frankly don’t think anyone ever remotely considered mocking parts of jOOQ ([and jOOQ is full of final!](https://blog.jooq.org/2013/08/26/silly-metrics-the-most-used-java-keywords/)). We always prefer discussing potential features on the roadmap than giving users the possibility to shoot themselves in the foot with extensions and mocks built upon incomplete understanding of internals that they shouldn’t have been tampering with in the first place.

[*lukaseder*](https://lukaseder.wordpress.com/) [, September 4, 2014 at 21:39](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-55811)

* Cool.

But when you say “…There’s simply no need for mocks if you have APIs.”, I’d like to push back on that.

I have approx 20,000 unit-tests for ~2 million lines of code. I run all unit-tests in < 2 minutes (and it should be FASTER than that), and I do it many times a day. I do it once a sprint with all network access disconnected. (We also have "integration tests" that talk to real services, and selenium tests that exercise the entire stack from the web front end.)

There's simply no way I can run that kind of testing even with a "black box" API setup. I have to mock things. I do a lot of extra work to put wrappers around some of the most basic Java final/static classes, so that I can mock \*them\*.

Again, so much of this kind of discussion depends on context. The narrower, more controlled the context, the more sense 'final' makes. The bigger/messier, the less sense it makes (on the whole, plenty of individual exceptions).

Now let me argue against myself :-) … in a context like jOOQ, I probably don't need mocks. Because if I'm mocking anything, I should be mocking at least one level of abstraction ABOVE what jOOQ is doing for me. (Correct me if I've got this wrong.) Proper separation of layers removes a lot of the problems with testing and mocking, and often the need for mocks at all.

And yes, you got me with “random, mediocre and probably free OSS) API’s..”! Right between the eyes. (Of course, that includes Sun’s, I mean Oracle’s, early implementations of some pretty basic services like ‘File’.)

[*Charles Roth*](http://thedance.net/) [, September 4, 2014 at 21:58](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-55826)

* I have approx 20,000 unit-tests for ~2 million lines of code

That’s impressive! I must admit, I’ve only ever worked on libraries (where this is much easier to achieve) or on very large, 15 year-old legacy systems (where this is completely out of reach – or maybe it wasn’t?). I’d like to see a system like yours in action, personally, one day. I’m really impressed when teams pull that off, thoroughly.

Because if I’m mocking anything, I should be mocking at least one level of abstraction ABOVE what jOOQ is doing for me

I would say so – probably on a service layer. Because all the transactions themselves will probably need to go through the database anyway, as complex state transfer can be very hard to mock – unless you’re writing a new database.

(Of course, that includes Sun’s, I mean Oracle’s, early implementations of some pretty basic services like ‘File’.)

Eh. And Calendar ;-)

[*lukaseder*](https://lukaseder.wordpress.com/) [, September 4, 2014 at 22:16](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-55836)

I suggest that there is a HUGE distinction between internal code and public API’s.

A public API is, at least in one sense, a finished product. Many people outside the author organization are going to use it, depend on it, and depend on it keeping the same contract. Final makes sense for all the reasons you have already elaborated on.

Internal code is (usually in my experience) part of an on-going project. It is “agile”, in the sense of always-changing. It’s not that code (or even the design) sucks… it’s that it’s constantly changing, and it’s SUPPOSED to be constantly changing.

Now there may still be good reasons to make something final — a specific reason why extending (say) a class would be a bad idea. As you(?) and others have said, in that case final should be just PART of documentation about that. IOW, if something is final, in my organization I get to insist that the developer add comments as to WHY it’s final. Otherwise it’s just hubris (or sloppiness, i.e. no documentation or bread-crumb trail left for the next guy or gal).

On the third hand… if it’s internal code, then I can remove the \*&^%\_ing final if I need to :-) which obviates some of my own arguments. I do that frequently, mostly because somebody read that final is good and they used it without understanding.

Hope that helps explain where I’m coming from. I’m enjoying the discussion: much more light than heat! :-)

[*Charles Roth*](http://thedance.net/) [, September 4, 2014 at 21:33](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-55809)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=55809#respond)

* Hmm, I get your point. I like the argumentation about “internal” code being “agile,” and thus less well-designed, or at least less designed for eternity – where final might be more of a precaution.

[*lukaseder*](https://lukaseder.wordpress.com/) [, September 4, 2014 at 21:45](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-55815)

* + - * + > what you wanted to do was prohibited by the Map contract in the first place: … “It does not support the add or addAll operations.”

If it was prohibited, the formulation should be “must not” or alike. Compare it to `Collections.unmodifiable\*` saying “and attempts to modify the returned …, result in an `UnsupportedOperationException`.” or even `ImmutableCollection.add` saying “Guaranteed to throw an exception and leave the collection unmodified”.

I can’t see what could break if some overrode `HashMap.keySet` by returning a `ForwardingSet` to its `super` and added `add`. I surely might be wrong.

Maaartinus [, September 23, 2014 at 05:24](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-67230)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=67230#respond)

Imagine you make a FancyMap that implements this, unexpectedly. Now you pass this map to some API expecting Map. The receiving site will be able to safely assume that this Map.values() collection will neverincrease in size.

In other words, the contract on all Map implementations is:

It does not support the add or addAll operations.

So, I don’t think the subtlety of distinguishing does not or must not is significant here.

[*lukaseder*](https://lukaseder.wordpress.com/) [, September 23, 2014 at 07:54](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-67333)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=67333#respond)

1. *Pingback:*[*Beyond average: 10 best practices for coding Java APIs | NerdlyNews*](http://nerdlynews.32hours.com/beyond-average-10-best-practices-for-coding-java-apis/)
2. *Pingback:*[*This is the Final Discussion! | Java, SQL and jOOQ.*](https://blog.jooq.org/2014/09/15/this-is-the-final-discussion/)
3. *Pingback:*[*Top 10 Most Popular Articles on the jOOQ Blog | Java, SQL and jOOQ.*](https://blog.jooq.org/2014/09/22/top-10-most-popular-articles-on-the-jooq-blog/)
4. Leaving the ‘final’ conversation behind, a quick note about your #8 in the original article “short circuit equals”. Java 7 has a very nice “Objects.equals()” static method that makes writing (overriding) your own equals() methods very much easier/more efficient/more testable/etc. The Objects (note plural) class has nice “small” new stuff that’s handy.

[*Charles Roth*](http://thedance.net/) [, September 22, 2014 at 14:44](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-66712)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=66712#respond)

* + Good point. Unfortunately, I myself am most often stuck in Java-6-land, which is why I haven’t thought of Objects at that point.

[*lukaseder*](https://lukaseder.wordpress.com/) [, September 22, 2014 at 14:49](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-66717)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=66717#respond)

1. *Pingback:*[*10 Things You Didn’t Know About Java | Java, SQL and jOOQ.*](https://blog.jooq.org/2014/11/03/10-things-you-didnt-know-about-java/)
2. *Pingback:*[*Add Some Entropy to Your JVM | Java, SQL and jOOQ.*](https://blog.jooq.org/2013/10/17/add-some-entropy-to-your-jvm/)
3. *Pingback:*[*Don’t be “Clever”: The Double Curly Braces Anti Pattern | Java, SQL and jOOQ.*](https://blog.jooq.org/2014/12/08/dont-be-clever-the-double-curly-braces-anti-pattern/)
4. *Pingback:*[*You Will Regret Applying Overloading with Lambdas! | Java, SQL and jOOQ.*](https://blog.jooq.org/2015/01/29/you-will-regret-applying-overloading-with-lambdas/)
5. *Pingback:*[*Top 10 Easy Performance Optimisations in Java | Java, SQL and jOOQ.*](https://blog.jooq.org/2015/02/05/top-10-easy-performance-optimisations-in-java/)
6. Thank you very much for your article. And what about Java 8 lambda expressions? E.g. when you pass lambda expression to some function, you create an anonymous class, that implements java.util.function.Function interface. Does it contain a reference to its enclosing instance and causes memory leaks?

dmitry [, May 15, 2015 at 17:21](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-126765)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=126765#respond)

* + Good question. I believe that lambda expressions aren’t really anonymous classes, and thus do not contain any references to enclosing instances…

[*lukaseder*](https://lukaseder.wordpress.com/) [, May 15, 2015 at 18:15](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-126768)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=126768#respond)

1. *Pingback:*[*What the sun.misc.Unsafe Misery Teaches Us | Java, SQL and jOOQ.*](https://blog.jooq.org/2015/07/20/what-the-sun-misc-unsafe-misery-teaches-us/)
2. *Pingback:*[*Was wir aus der Misere um sun.misc.Unsafe lernen können - jaxenter.de*](https://jaxenter.de/was-wir-aus-der-misere-um-sun-misc-unsafe-lernen-koennen-23390)
3. *Pingback:*[*Top 10 Useful, Yet Paranoid Java Programming Techniques | Java, SQL and jOOQ.*](https://blog.jooq.org/2015/08/11/top-10-useful-yet-paranoid-java-programming-techniques/)
4. Hi and thanks for the article, it’s simple and very useful.

related to point 7. what I understand is that we have to use immutable objects for all communication between objects(return results, method pramas …) in order to avoid state change on one thread. Am’I understand correctly ?

thanks

[*cristeasorin*](http://cristeasorin.wordpress.com/) [, January 25, 2016 at 10:11](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-142594)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=142594#respond)

* + Indeed, making all objects immutable does help in designing simpler, more robust APIs.

[*lukaseder*](https://lukaseder.wordpress.com/) [, January 25, 2016 at 14:23](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/#comment-142598)

[REPLY](https://blog.jooq.org/2013/08/20/10-subtle-best-practices-when-coding-java/?replytocom=142598#respond)

1. *Pingback:*[*10 Features I Wish Java Would Steal From the Kotlin Language | Java, SQL and jOOQ.*](https://blog.jooq.org/2016/03/31/10-features-i-wish-java-would-steal-from-the-kotlin-language/)