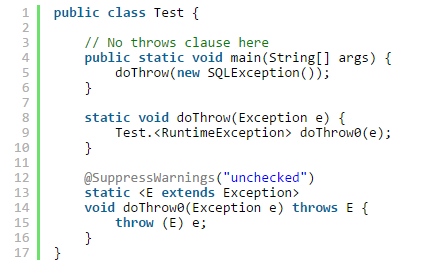
# 10 Things You Didn’t Know About Java

## 1. There is no such thing as a checked exception

That’s right! The JVM doesn’t know any such thing, only the Java language does.

Today, everyone agrees that checked exceptions were a mistake. No other language after Java has engaged in using checked exceptions, and even Java 8 does no longer embrace them in the new Streams API ([which can actually be a bit of a pain, when your lambdas use IO or JDBC](http://blog.jooq.org/2014/05/23/java-8-friday-better-exceptions/)).

Do you want proof that the JVM doesn’t know such a thing? Try the following code:

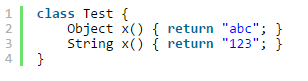


Not only does this compile, this also actually throws the SQLException, you don’t even need Lombok’s [@SneakyThrows](http://projectlombok.org/features/SneakyThrows.html) for that.

[More details about the above can be found in this article here](http://blog.jooq.org/2012/09/14/throw-checked-exceptions-like-runtime-exceptions-in-java/), or [here, on Stack Overflow](http://stackoverflow.com/q/12580598/521799).

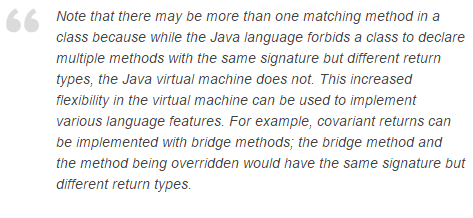
## 2. You can have method overloads differing only in return types

That doesn’t compile, right?

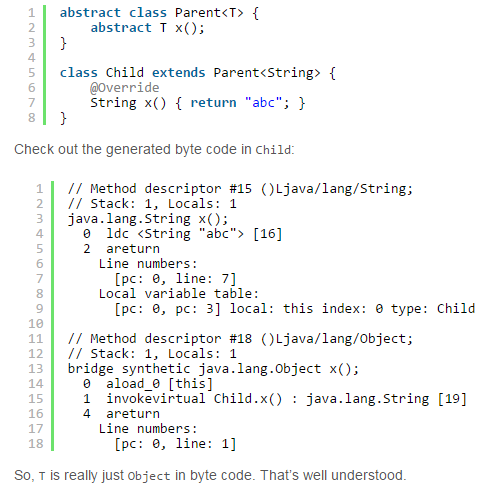


Right. The Java language doesn’t allow for two methods to be “override-equivalent” within the same class, regardless of their potentially differingthrows clauses or return types.

Check out the Javadoc of [Class.getMethod(String, Class...)](http://docs.oracle.com/javase/8/docs/api/java/lang/Class.html" \l "getMethod-java.lang.String-java.lang.Class...-). It reads:



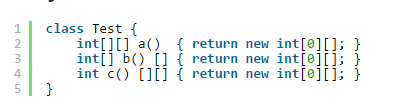
In fact, that’s pretty much what happens when you write the following:



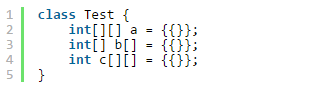
The synthetic bridge method is actually generated by the compiler because the return type of the Parent.x() signature may be expected to Object at certain call sites. Adding generics without such bridge methods would not have been possible in a binary compatible way. So, changing the JVM to allow for this feature was the lesser pain (which also allows covariant overriding as a side-effect…) Clever, huh?

Are you into language specifics and internals? [Then find some more very interesting details here](http://stackoverflow.com/q/442026/521799).

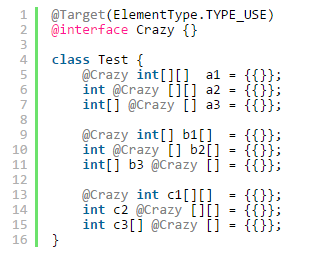
## 3. All of these are two-dimensional arrays!



Yes, it’s true. Even if your mental parser might not immediately understand the return type of the above methods, they are all the same! Similar to the following piece of code:

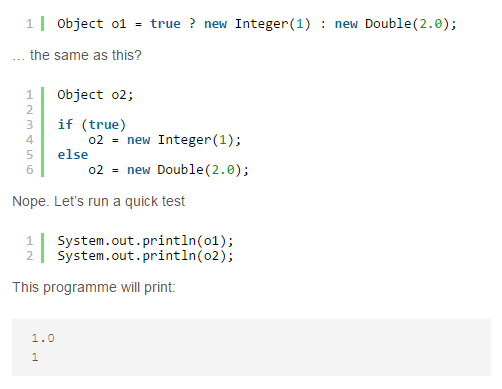


You think that’s crazy? Imagine using [JSR-308 / Java 8 type annotations](https://jcp.org/en/jsr/detail?id=308) on the above. The number of syntactic possibilities explodes!

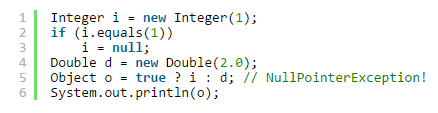


## 4. You don’t get the conditional expression

So, you thought you knew it all when it comes to using the conditional expression? Let me tell you, you didn’t. Most of you will think that the below two snippets are equivalent:



Yep! The conditional operator will implement numeric type promotion, if“needed”, with a very very very strong set of quotation marks on that“needed”. Because, would you expect this programme to throw aNullPointerException?



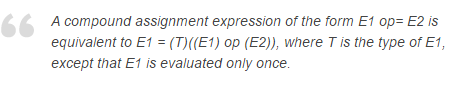
[More information about the above can be found here](http://blog.jooq.org/2013/10/08/java-auto-unboxing-gotcha-beware/).

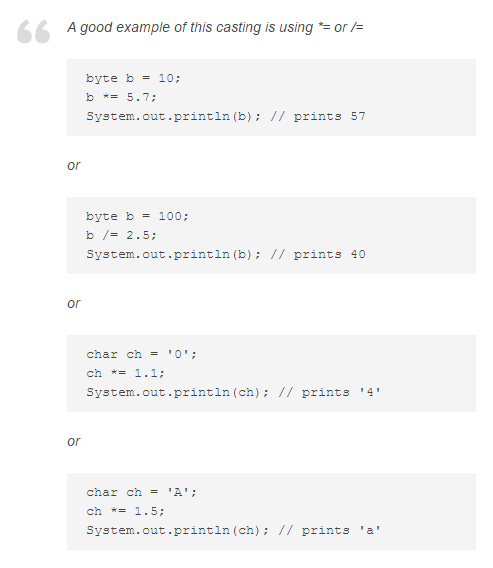
## 5. You also don’t get the compound assignment operator

Let’s consider the following two pieces of code:



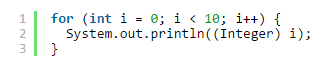
Intuitively, they should be equivalent, right? But guess what. They aren’t! The JLS specifies:



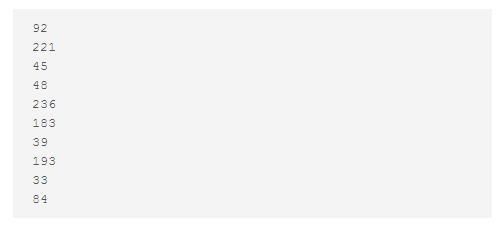


## 6. Random integers

Now, this is more of a puzzler. Don’t read the solution yet. See if you can find this one out yourself. When I run the following program:



then “sometimes”, I get the following output:

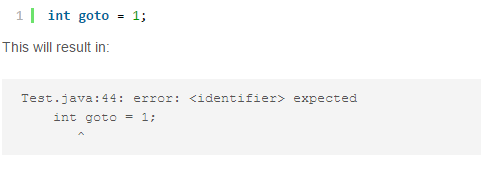


How is that even possible??

[OK, the solution is here (http://blog.jooq.org/2013/10/17/add-some-entropy-to-your-jvm/)](http://blog.jooq.org/2013/10/17/add-some-entropy-to-your-jvm/) and has to do with overriding the JDK’s Integer cache via reflection, and then using auto-boxing and auto-unboxing. Don’t do this at home! Or in other words, let’s think about it this way, once more

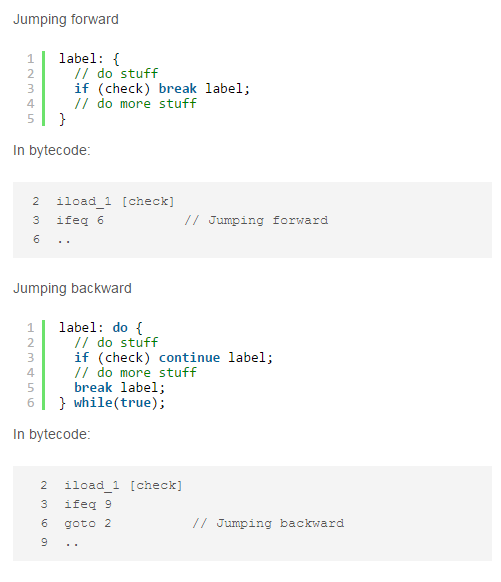
## 7. GOTO

Java has GOTO! Type it…



This is because goto is [an unused keyword](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html), just in case…

But that’s not the exciting part. The exciting part is that you can actually implement goto with break, continue and labelled blocks:

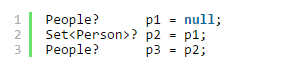


## 8. Java has type aliases

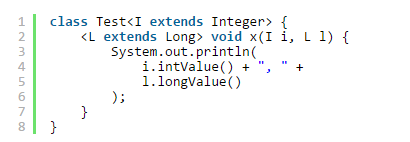
In other languages ([e.g. Ceylon](http://blog.jooq.org/2013/12/03/top-10-ceylon-language-features-i-wish-we-had-in-java/)), we can define type aliases very easily:



A People type constructed in such a way can then be used interchangably with Set<Person>:



In Java, we can’t define type aliases at a top level. But we can do so for the scope of a class, or a method. Let’s consider that we’re unhappy with the namings of Integer, Long etc, we want shorter names: I and L. Easy:



In the above programme, Integer is “aliased” to I for the scope of the Testclass, whereas Long is “aliased” to L for the scope of the x() method. We can then call the above method like this:

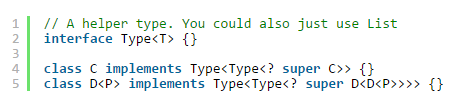


This technique is of course not to be taken seriously. In this case, Integerand Long are both final types, which means that the types I and L areeffectively aliases (almost. assignment-compatibility only goes one way). If we had used non-final types (e.g. Object), then we’d be really using ordinary generics.

Enough of these silly tricks. Now for something truly remarkable!

## 9. Some type relationships are undecidable!

OK, this will now get really funky, so take a cup of coffee and concentrate. Consider the following two types:

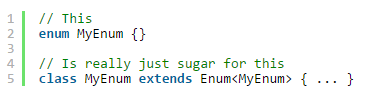


Now, what do the types C and D even mean?

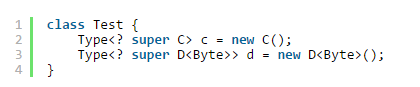
They are somewhat recursive, in a similar (yet subtly different) way that[java.lang.Enum](http://docs.oracle.com/javase/8/docs/api/java/lang/Enum.html) is recursive. Consider:



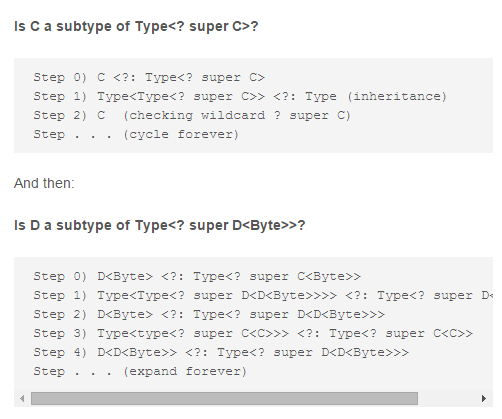
With the above specification, an actual enum implementation is just mere syntactic sugar:



With this in mind, let’s get back to our two types. Does the following compile?



Hard question, and [Ross Tate](http://www.cs.cornell.edu/~ross/) has an answer to it. The question is in fact undecidable:



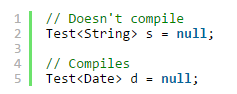
Try compiling the above in your Eclipse, it’ll crash!

## 10. Type intersections

Java has a very peculiar feature called type intersections. You can declare a (generic) type that is in fact the intersection of two types. For instance:



The generic type parameter T that you’re binding to instances of the classTest must implement both Serializable and Cloneable. For instance,String is not a possible bound, but Date is:



This feature has seen reuse in Java 8, where you can now cast types to ad-hoc type intersections. How is this useful? Almost not at all, but if you want to coerce a lambda expression into such a type, there’s no other way. Let’s assume you have this crazy type constraint on your method:



You want a Runnable that is also Serializable just in case you’d like to execute it somewhere else and send it over the wire. Lambdas and serialisation are a bit of a quirk.

[Lambdas can be serialised](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#serialization):



But even if that’s true, they do not automatically implement theSerializable marker interface. To coerce them to that type, you must cast. But when you cast only to Serializable…



… then the lambda will no longer be Runnable.

Cast it to both types:

|  |  |
| --- | --- |
| 1 | execute((Runnable & Serializable) (() -> {})); |