FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN ENTERPRISE SOFTWARE DEVELOPMENT

# Java 8

InfoQueue
eMag Issue 15 - July 2014

## Clarifying Lambdas in Java 8

Simon Ritter discusses the syntax and use of Lambda expressions, focusing on using Streams to greatly simplify the way bulk and aggregate operations are handled in Java.

# Contents

# A Letter from the Editor

**Victor Grazi** is the Java queue lead at InfoQ. Inducted as an Oracle Java Champion in 2012, Victor works at Nomura Securities on platform architecture, and as a technical consultant and Java evangelist. He is also a frequent presenter at technical conferences. Victor hosts the "Java Concurrent Animated" open source project on SourceForge.

After all the well deserved fanfare, Java 8 was released this year and is now planting deep roots in the development mainstream.

The syntax, API's and plumbing have changed substantially, perhaps more than in any version of Java since 1.0, topping even Java 5 in impact.

In this eMag, InfoQ takes practitioners on a visit with Java 8, exploring how we got here and how we should be moving forward with it.

Java 8 was not just a quantum release. There was some evolution in prior versions, especially Java 7. So our eMag begins with a piece on "Java 7 Features that Enable Java 8" by Ben Evans.

With the introduction of lambda expressions and the new Collection and Stream APIs, Java steps into the functional arena. Ben Evans explores this in a feature entitled "How Functional is Java 8".

Oracle evangelist Simon Ritter did a presentation at QCon San Francisco 2014 clarifying Lambdas and Streams. InfoQ distills the lessons learned into a visceral enlightenment on how to think in terms of these important new APIs.

Time and date handling in Java has until now been somewhat of a sore spot for Java developers. "Joda" Stephen Colebourn, who introduced the ever popular Joda Time framework and led the spec on the new Java Date Time APIs, does a deep dive into this revolutionary addition to the core Java 8 distribution.

With the advent of annotations in Java, and their fortification in Java 8, developers can annotate their code with intent, creating an opportunity for tooling to help eliminate bugs by verfiying usage against intent. Financial technology consultant Todd Schiller explores the usage patterns and existing tooling.

Next up, performance guru Monica Beckwith does a deep dive into the removal of Perm Gen from the Java runtime, and how to make sense of what replaces it.

Project Rhino brought first class JavaScript support to the Java language, compiling JavaScript into bytecode, and allowing teams to leverage their client side developers to build server side code. In his piece entitled "Nashorn: The Combined Power of Java and JavaScript in JDK 8" consultant Oliver Zeigermann explores ways to work with JavaScript in Java 8.

Finally Takipi's Tal Weiss has some fun exploring "8 Great Java 8 Features No One's Talking About.

This is an important eMag about one of the most important language releases in history. I hope you enjoy it.

# Java 7 Features That Enable Java 8

by *Ben Evans*

It's a truism of the tech industry that developers are never happier than when there's free beer or an opportunity to complain about something on offer.

So despite the efforts of Mark Reinhold and the Java team to involve the community in the roadmap after the Oracle acquisition (the Plan A/Plan B decision), many Java developers feel that Java 7 was not much of a release.

In this article, I'll try to refute this thesis, by exploring the features in Java 7 that lay the groundwork for the new features in Java 8.

## Diamond operator

Java has often been criticised for being overly verbose. One of the most common areas where this complaint is expressed is in assignment. In Java 6, we are forced to write assignment statements like this:

```
Map<String, String> m = new
HashMap<String, String>();
```

This statement contains a lot of redundant information. We should be able to somehow have the compiler figure out more of this by itself, and not require the programmer to be quite so explicit.

In fact, languages like Scala do a large amount of type inference from expressions, and in fact assignment statements can be written as simply as this:

```
val m = Map("x" -> 24, "y" -> 25, "z" ->
26);
```

The keyword val indicates that this variable may not be reassigned to (like the keyword final for Java variables). No type information is specified about the variable at all - instead the Scala compiler examines the right side of the assignment and determines the correct type for the variable by looking at which value is being assigned.

Java 7 introduced some limited type-inference capabilities, and assignment statements can now be written like this:

```
Map<String, String> m = new HashMap<>();
```

The key differences between this and the Scala form is that in Scala, values have explicit types, and it is the type of variables that is inferred. In Java 7, the type of variables is explicit, and type information about values is what is inferred.

Some developers have complained that they would have preferred the Scala solution, but it turns out to be less convenient in the context of a major feature for Java 8: lambda expressions.

In Java 8, we can write a function which adds 2 to an integer like this:

```
Function<Integer, Integer> fn = x -> x
+ 2;
```

The Function interface is new with Java 8. It resides in the java.util.function package along with specialized forms for primitive types. We've chosen this syntax as it resembles the Scala equivalent and allows the developer to see the similarities more easily.

By explicitly specifying the type of fn as a Function which takes one Integer argument and returns another Integer, the Java compiler is able to infer the type of the parameter x, which is Integer. This is the same pattern that we saw in Java 7 diamond syntax: we specify the types of variables and infer the types of values.

Let's look at the corresponding Scala lambda expression:

```
val fn = (x : Int) => x + 2;
```

Here, we have to explicitly specify the type of the parameter x, as we don't have the precise type of fn, and so we have nothing to infer from. The Scala form is not difficult to read, but the Java 8 form has a certain cleanliness of syntax which can be directly traced back to the diamond syntax of Java 7.

## Method handles

Method handles are simultaneously the most important new feature of Java 7 and the feature that is least likely to appear in the day-to-day life of most Java developers.

A method handle is a typed reference to a method for execution. It can be thought of as a typesafe function pointer (for developers familiar with C/C++) or as Core Reflection reimagined for the modern Java developer.

Method handles play a huge part in the implementation of lambda expressions. Early prototypes of Java 8 converted each lambda expression to an anonymous inner class at compile time.

More recent betas are more sophisticated. Let's start by recalling that a lambda expression (at least in Java) comprises a function signature (which in the method handles API will be represented by a MethodType object) and a body, but not necessarily a function name.

This suggests that we could convert the lambda expression into a synthetic method that has the correct signature and contains the body of the lambda. Look at our example:

```
Function<Integer, Integer> fn = x -> x
+ 2;
```

The Java 8 compiler turns that into a private method with this bytecode:

```
private static java.lang.Integer
lambda$0(java.lang.Integer);
   descriptor: (Ljava/lang/Integer;)
Ljava/lang/Integer;
   flags: ACC_PRIVATE, ACC_STATIC, ACC_
SYNTHETIC
   Code:
    stack=2, locals=1, args_size=1
      0: aload_0
      1: invokevirtual #13 // Method
java/lang/Integer.intValue:()I
      4: iconst_2
      5: iadd
      6: invokestatic #8 // Method java/
lang/Integer.valueOf:(I)Ljava/lang/
Integer;
      9: areturn
```

This has the correct signature (takes in an Integer and returns another one) and semantics. To use this lambda expression, we take a method handle that refers to it and use it to build an object of the appropriate type, as we'll see in the next feature we discuss.

## invokedynamic

The final feature of Java 7 that opens the door for Java 8 is even more esoteric than method handles. This is the new bytecode invokedynamic, the first new bytecode to be added to the platform since Java 1.0. This feature is almost impossible for Java developers to make use of in version 7, because version 7 javac will not under any circumstances emit a classfile that contains it.

Instead, the bytecode was designed for use by developers of non-Java languages such as JRuby, which require much more dynamic dispatch than Java. To see how invokedynamic works, let's discuss how Java's method calls are compiled into bytecode.

A standard Java method call will be turned into a piece of JVM bytecode that is often referred to as a call site. It comprises a dispatch opcode (such as invokevirtual for regular instance method calls) and a constant (an offset into the Constant Pool of the class) that indicates which method is to be called.

The different dispatch opcodes have different rules that govern how method lookup is done, but until Java 7, the constant was always a straightforward indication of which method was to be called.

invokedynamic is different. Instead of providing a constant that directly indicates which method is to be called, invokedynamic provides an indirection mechanism that allows user code to decide which method to call at run time.

When an invokedynamic site is first encountered, it does not yet have a known target. Instead, a method handle (called a bootstrap method) is invoked. This bootstrap method returns a CallSite object, which contains another method handle that is the actual target of the invokedynamic call:

1) invokedynamic site encountered in the execution stream (initially unlinked).

2) Call bootstrap method and return a CallSite object.

3) CallSite object contains a method handle (the target).

4) Invoke the target method handle.

The bootstrap method is the way in which user code chooses which method to call. For lambda expressions, the platform uses a library-supplied bootstrap method called a lambda meta-factory. This has static arguments that contain a method handle to the synthesized method (see last section) and the correct signature for the lambda.

The meta-factory returns a CallSite that contains a method handle, which will in turn return an instance of the correct type that the lambda expression has been converted to. So, a statement like:

```
Function<Integer, Integer> fn = x -> x
+ 2;
```

is converted to an invokedynamic call like this:

```
Code:
  stack=4, locals=2, args_size=1
     0: invokedynamic #2, 0 //
InvokeDynamic #0:apply:()Ljava/util/
function/Function;
     5: astore_1
```

The invokedynamic bootstrap method is the static method LambdaMetafactory.metafactory(), which returns a CallSite object that is linked to a target method handle, which will return an object that implements the Function interface.

When the invokedynamic instruction is complete, an object that implements Function and that has the lambda expression as the contents of its apply() method is seated on top of the stack, and the rest of the code can proceed normally.

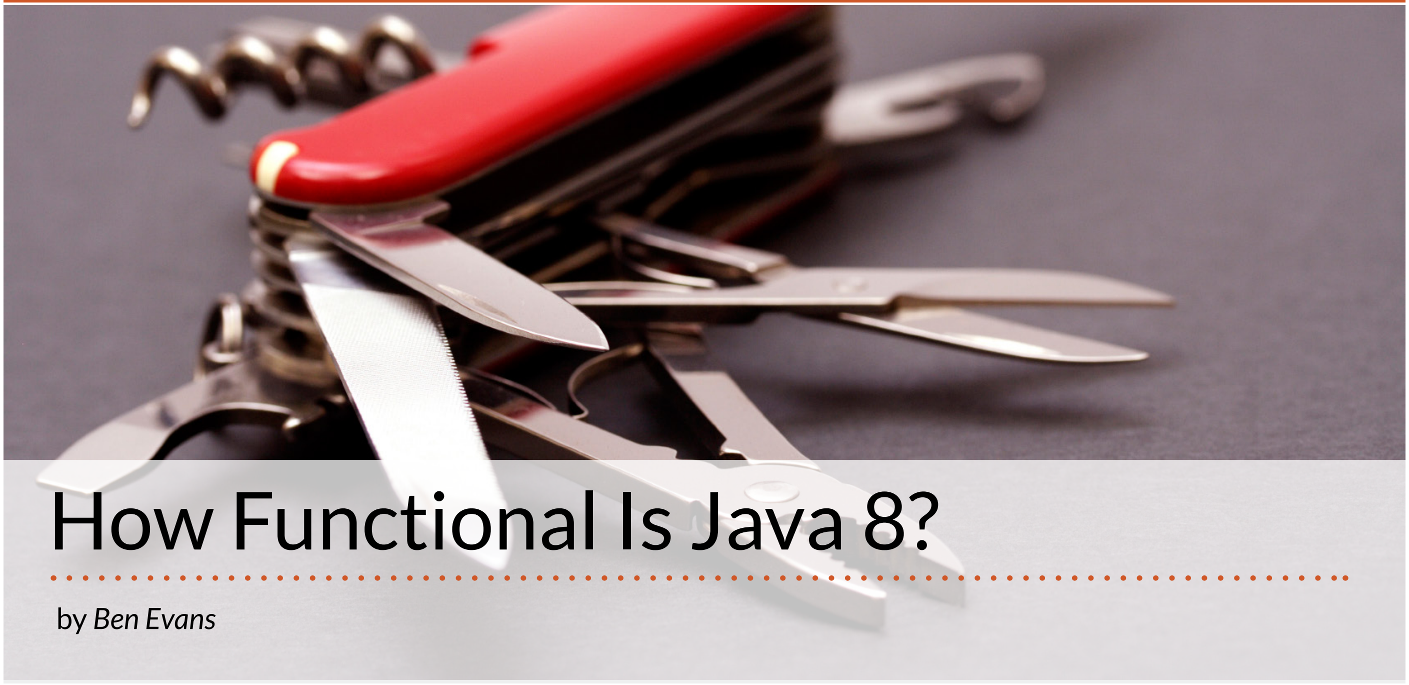
## Conclusion

Getting lambda expressions into the Java platform was always going to be challenging, but Java 7 eased that effort considerably by ensuring that the proper groundwork was in place,. Plan B not only provided developers with the early release of Java 7 but also allowed core technologies to be fully road-tested before their use in Java 8 and especially in lambda expressions.

**ABOUT THE AUTHOR**

**Ben Evans is co-founder of jClarity, a startup which delivers performance tools & services to help development & ops teams. He is an organizer for the LJC (London's JUG) and a member of the JCP Executive Committee, helping define standards for the Java ecosystem. He is a Java Champion; JavaOne Rockstar; co-author of "The Well-Grounded Java Developer" & the new edition "Java in a Nutshell" and a regular speaker on the Java platform, performance, concurrency, and related topics.**

READ THIS ARTICLE ONLINE ON InfoQ



# How Functional Is Java 8?

by *Ben Evans*

There's been a lot of talk about how Java 8 is bringing functional programming (FP) to Java, but what does that really mean?

In this article, I'll discuss what it means for a language, or a programming style, to be functional. By looking at the evolution of Java, particularly its type system, we can see how the new features of Java 8, especially lambda expressions, change the landscape and provide some key benefits of the functional style - before tackling the question "How functional is Java 8?"

## What is a functional programming language?

At its heart, a functional programming language is one that deals with code in the same way as data. This means that a function should be a first-class value and able to be assigned to variables, passed to functions, and so forth.

In fact, many functional languages go even further than this, and see computation and algorithms as more fundamental than the data they operate on. Some of these languages want to disentangle program state from functions (in a way that seems at odds with the desire of object-oriented languages that usually want to bring them closer together).

An example would be the Clojure programming language. Despite running on top of the class-based Java virtual machine, Clojure is fundamentally a functional language, and doesn't directly expose classes and objects in the high-level source language (although good interoperability with Java is provided).

A Clojure function, such as the log-processing function shown below, is a first-class citizen, and doesn't need to be bundled up in a class to exist.

```
(defn build-map-http-entries [log-file]
 (group-by :uri (scan-log-for-http-
entries log-file)))
```

Functional programming is most useful when programs are written in terms of functions that always return the same output for a given input (regardless of any other state present in the running program) and that do not cause any other effects or change any program state. Functions that obey this are sometimes called "pure" functions, and they behave in the same way that mathematical functions do.

The great advantage that pure functions have is that they are much easier to reason about because their operation does not depend on external state. Functions can easily be combined together - and this can be seen in developer workflow styles such as the REPL (read, execute, print, loop) style common to Lisp dialects and other languages with strong functional heritage.

## Functional programming in non-FP languages

Whether a language is functional or not is not a binary condition - instead, languages exist on a spectrum. At the extreme end are languages that basically enforce functional programming, often by prohibiting mutable data structures. Clojure is one example of a language that does not permit mutable data in the accepted sense.

However, there are other languages in which it is common to write programs in a functional style, despite the language not enforcing this. An example would be Scala, which is a blend of object-oriented and functional languages. It permits functions as values, such as:

```
val sqFn = (x: Int) => x * x
```

while retaining class and object syntax that is very close to that of Java.

At the other extreme, it is of course possible to write functional programs in completely non-functional languages, such as C, provided that suitable programmer discipline and conventions are maintained.

With this in mind, functional programming should be seen as a function of two factors - one of which is relevant to programming languages and one to programs written in that language:

1. To what extent does the underlying programming language support or enforce functional programming?

2. How does this particular program make use of the functional features provided by the language? Does it avoid non-functional features such as mutable state?

## Some Java history

Java is an opinionated language. it has been optimized for readability, for accessibility to junior programmers, and for long-term stability and supportability. These design decisions have come at a cost - in verbosity, and in a type system that can sometimes seem inflexible when compared to other languages.

However, Java's type system has evolved, albeit relatively slowly, over the history of the language. Let's take a look at some of the forms that it has assumed over the years.

## Java's original type system

Java's original type system is now well over 15 years old. It is simple and clear: types are either reference types or primitive types. Reference types are classes, interfaces, or arrays.

Classes are the heart of the Java platform. A class is the basic unit of functionality that the Java platform will load or link, and all code that is intended for execution must live inside a class.

Interfaces can't be instantiated directly and a class instead must be defined that implements the API defined by the interface.

Arrays hold either primitive types, instances of classes, or other arrays.

The primitive types are all defined by the platform, and the programmer can't define new ones.

From the very earliest days, Java's type system has been insistent on a very important point: every type must have a name by which it can be referred. This is known as "nominative typing" - and Java is a strongly nominatively typed language.

Even the so-called "anonymous inner classes" still have a type by which the programmer must refer to them, being the type of the interface that they implement:

```
Runnable r = new Runnable() { public
void run() { System.out.println("Hello
World!"); } };
```

Another way of saying this is that every value in Java is either a primitive or an instance of some class.

## Alternatives to named types

Other languages do not have this fascination with named types. For example, Java has no equivalent to Scala's concept of a type that implements a specific method (of a specific signature). In Scala, this would be written:

```
x : {def bar : String}
```

Remember that Scala indicates the type of a variable on the right (after the : ) so this is read as something like "x is of a type that has a method called bar that returns String". We could use this to define a Scala method like this:

```
def showRefine(x : {def bar : String}) =
{ print(x.bar) }
```

Then, we can define a suitable Scala object like this:

```
object barBell { def bar = "Bell" }
```

and calling showRefine(barBell) does the expected thing:

```
showRefine(barBell) Bell
```

This is an example of refinement typing. Programmers coming from dynamic languages may be familiar with "duck typing". Structural refinement typing is similar, except that duck typing ("if it walks like a duck, and quacks like a duck, it's a duck") is about the runtime types, whereas these structural refinement types work at compile time.

In languages that fully support structural refinement typing, these refined types can be used anywhere the programmer might expect (such as the type of a parameter to a method). Java, by contrast, does not support this sort of typing (apart from a couple of slightly bizarre edge cases).

## The Java 5 type system

The release of Java 5 brought three major new features to the type system: enums, annotations, and generic types.

Enumerated types (enums) are similar to classes in some respects, but they have the property that only a specified number of instances may exist, and each instance is specified in the class description and distinct. Intended primarily as a typesafe constant rather than the then-common practice of using small integers for constants, the enum construction also permits additional patterns that are sometimes extremely useful.

Annotations are related to interfaces - the keyword to declare one is @interface - with the initial @ indicating that this is an annotation type. As the name suggests, they're used to annotate elements of Java code with additional information that doesn't affect behavior. Previously, Java had made use of so-called "marker interfaces" to provide a limited form of this metadata but annotations are considerably more flexible.

Java's generics provide parameterized types: the idea that one type can act as a container for objects of another type, without regard for the specifics of exactly which type is being contained. The type that fits into the container is often called the type parameter.

Of the features introduced by Java 5, enums and annotations provided new forms of reference type which require special treatment by the compiler and which are effectively disjoint from the existing type hierarchies.

Generics provide significant additional complexity to Java's type system - not least because they are purely a compile-time feature. This requires the Java developer to be mindful of both a compile-time and a run-time type system that are slightly different from each other.

Despite these changes, Java's insistence on nominative typing remained. Type names now include List<String> (read as: "List-of-String") and Map<Class<?>, CachedObject> ("Map-of-Class-of-Unknown-Type-to-CachedObject"), but these are still named types, and every non-primitive value is still an instance of a class.

## Features introduced in Java 6 and 7

Java 6 was essentially a performance and library-enhancement release. The only change to the type system was an expansion of the role of annotations, and the release of a capability for pluggable annotation processing. This did not impact most Java developers, and did not really provide for pluggable type systems in Java 6.

Java 7 did not materially change the type system. The only new features, all of them very minor, are:

Small improvements in type inference in the javac compiler.

Signature polymorphic dispatch, used as an implementation detail for method handles - which are in turn used to implement lambda expressions in Java 8.

Multi-catch provides some small traces of algebraic data types, but these are purely internal to javac and are not of any real consequence to the end-user programmer.

## The Java 8 type system

Throughout its history, Java has been essentially defined by its type system. It is central to the language and has maintained a strict adherence to nominative typing. From a practical point of view, the Java type system did not change much between Java 5 and Java 7.

At first sight, we might expect Java 8 to change that. After all, a simple lambda expression appears to remove us from nominative typing:

```
() -> { System.out.println("Hello
World!"); }
```

This is a method, without a name, that takes no parameters and returns void. It's still perfectly statically typed, but is now anonymous.

Have we escaped the Kingdom of the Nouns? Is this actually a new form of type for Java?

The answer is, perhaps unfortunately, no. The JVM, on which Java and other languages run, is very strictly tied to the concept of classes. Class loading is central to the Java platform's security and verification modes. Simply put, it would be very, very difficult to conceive of a type that was not in some way represented through a class.

Instead of creating a new kind of type, Java 8 lambda expressions are auto-converted by the compiler to be an instance of a class. The class of which they are an instance is determined by type inference. For example:

```
Runnable r = () -> { System.out.
println("Hello World!"); };
```

The lambda expression on the right side is a perfectly good Java 8 value - but its type is inferred from the value on the left, so it is actually a value of type Runnable. Note, however, that a lambda expression used in an incorrect way will result in a compiler error. Nominative typing is still the Java way, and even the introduction of lambdas has not changed that.

## How functional is Java 8?

Finally, let's turn to the question we posed at the start of the article: how functional is Java 8?

Before Java 8, a developer who wanted to write in a functional style would have to use nested types (usually anonymous inner classes) as a stand-in for function literals. The default collections libraries would not do the code any favors, and the curse of mutability would be ever-present.

Java 8's lambda expressions do not magically transform it into a functional language. Instead, their effect is to create a still imperative, still strongly nominative type language that has better syntax support for lambda expressions as function literals. Simultaneously, the enhancements to the collections libraries have allowed Java developers to start adopting simple functional idioms (such as filter and map) to tidy up otherwise unwieldy code.

Java 8 required the introduction of some new types to represent the basic building blocks of functional pipelines - interfaces such as Predicate, Function, and Consumer in java.util.function. These additions make Java 8 capable of slightly functional programming but Java's need to represent them as types (and their location in a utility package rather than the language core) speaks to the stranglehold that nominative typing has on the Java language, and how far the language is from the purity of Lisp dialects or other functional languages.

Despite all the above, this small subset of the power of functional languages may well be all that most developers actually need for their day-to-day development. For power users, other languages (on the JVM and elsewhere) still exist, and will doubtless continue to thrive.

### ABOUT THE AUTHOR

**Ben Evans** is co-founder of jClarity, a startup which delivers performance tools & services to help development & ops teams. He is an organizer for the LJC (London's JUG) and a member of the JCP Executive Committee, helping define standards for the Java ecosystem. He is a regular speaker on the Java platform, performance, concurrency, and related topics.

READ THIS ARTICLE ONLINE ON InfoQ

# Clarifying Lambdas in Java 8

presentation summary writen by *Victor Grazi*

If you ask average Java developers about Java 8, you will hear ebullient joy in their voices, especially on the prospect of using lambda expressions.

But after honest introspection, we might find the fervor tempered by some fear of the tangled web of new and mysterious APIs. Simon Ritter has unravelled some of the mystery in his talk on lambdas at the 2014 QCon London conference

Ritter begins with the question "Why do we want lambdas in Java?" Java as a language is Turing-complete, which means that any problem that can be solved with a computer program, can be expressed in Java. So, why do we need to invest in implementing and learning new features? Ritter answers that "lambdas can make life easier in terms of how you code and the way that we can take advantage of the platform underneath – multi-core, multi-processor systems."

Let's look at a sample piece of code that illustrates a common coding pattern in Java:

```
List<Student> students = ...
  double highestScore = 0.0;
  for (Student s : students) {
    if (s.gradYear == 2011) {
      if (s.score > highestScore) {
        highestScore = s.score;
      }
    }
  }
```

(The red parts represent the parts we are interested in; blue represents the boilerplate code.)

In this problem, we want to find the highest score in a Collection of students. We are using a common external-iteration idiom to scan and compare.

But there are some drawbacks to this. First, external iteration means that we, the developers, are responsible for the implementation, and because we are using a single loop, we are making it inherently serial. If we wanted to optimize this, we couldn't easily segment it into a parallel set of execution instructions.

Secondly, the highestScore variable is mutable and not thread-safe. So, even if we did want to break it up into multiple threads, we would have to introduce additional locking to prevent race conditions, which in turn can introduce performance issues.

Now, if we're clever, we can change the implementation a bit more towards the functional style by using an anonymous inner class.

```
    List<Student> students = ...
    double highestScore =
      students.filter(new
  Predicate<Student>() {
        public boolean op(Student s) {
          return s.getGradYear() == 2011;
        }
      }).map(new Mapper<Student, Double>()
  {
        public Double extract(Student s) {
          return s.getScore();
        }
      }).max();
```

In this implementation, we eliminate the mutable state, and hand the job of iteration to the library. We are chaining together a sequence of method calls in order to apply that operation in an expression of "Look at all of my students and filter out just the ones who graduated in 2011".

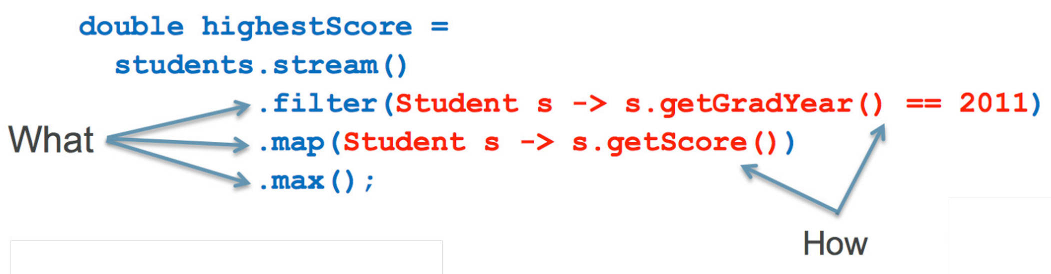
The anonymous inner class implements a Predicate interface (it contains one method, accepts one parameter, returns a boolean) with the method called "op", which simply compares the graduation year of that student to 2011 and returns the  result.

We pass the result (all students who graduated in 2011) to a map method, which is going to use another anonymous inner class to call the map interface-method with its single extract method, to extract the data we want (by calling getScore). We then pass that result, which is a set of scores of all the students who graduated in 2011, to a max method, delivering the largest number from that set of results.

Using this approach, we have handed all the iteration, filtering, and accumulation to the library code, freeing us from having to do that explicitly. Not only does that simplify our implementation, it also eliminates the shared state, making it easy to ask the library code to decompose it into a number of sub-tasks and allocate them to different threads to have them execute in parallel. In many cases, we could also perform a lazy evaluation, saving even more time.

So, an approach that uses the anonymous inner class is fast and thread-safe, but look at the color-coding. The ratio of blue to red is pretty high, indicating boilerplate bloat.

Enter lambda expressions!



```
double highestScore =
  students.stream()
        .filter(Student s -> s.getGradYear() == 2011)
        .map(Student s -> s.getScore())
        .max();
```

You can think of a lambda expression as a method in the sense that it accepts parameters, has a body, and returns a static type.

This example uses lambda expressions to accomplish the same high-score determination algorithm as above. Let's take a closer look.

First, we create a Stream from our Collection. The stream method is new in the COLLECTION interface, and works somewhat like a built-in Iterator (more on that later). The stream prepares the results from the collection, which we then pass to our filter method, describing the "how" part of the filter using a lambda expression that compares the graduation year of that student against 2011.

Notice that there is no explicit return statement. We simply say "Compare graduation year against 2011," and the compiler infers that the Predicate interface (which has a single method that needs a signature with a return type of boolean) is intended. The map method is processed similarly, using a lambda expression, by passing a parameter of Student S and mapping (a.k.a. translating) it to the return value, which is the score of that student. (The map method should not be confused with the very different java.util.Map of key-value pairs. Rather, the Stream class's map method returns a new Stream instance that consists of the results of the operation applied to all elements of the incoming Stream, producing, in this case, a Stream of all of the scores.)

Using lambdas, we have implemented the same algorithm with a lot less code. It is clearer, and therefore less error-prone, and, as we said, it can be swapped for a parallel algorithm since there is no shared state.

As Ritter says in his talk,

> Lambda expressions represent an anonymous function. So, I said they are like a method, but they are not really a method. They are an anonymous function in the sense that they do the same thing as a method, but they are not a method because they are not associated with a class. If you think about Java as you program now, you

> create a class and the class has methods. So, the method has a class associated with it. In the case of the lambda expression, there is no class associated with it!...
>
> It is like a method in its structure: it has a typed argument list; it has a return type that can be inferred; but you can also use an explicit return, if you want. It has potentially a set of thrown exceptions so you can throw exceptions from within lambda expression if you want to and it has a body, which defines what you are actually doing. You can do the same thing as with a method: you can group statements together; you can use braces and have multiple statements without any problem. The important thing about this is that it now enables us in a simple way to have parameterized behavior, not just parameterized values.

Ritter extends this concept by pointing out that since a lambda is a function without a class, the keyword "this" refers not to the lambda itself, but to the surrounding class. This is distinguished from an anonymous inner class, where "this" refers to the inner class itself.

It is helpful to look at the implementation choices made by the language designers to accomplish lambdas.

Looking at Java as a whole, there are many interfaces that have just one method.

Let's define a functional interface as an interface with exactly one abstract method, for example:

```
interface Comparator<T> { boolean
compare(T x, T y); }
interface FileFilter { boolean
accept(File x); }
interface Runnable { void run(); }
interface ActionListener { void
actionPerformed(…); }
interface Callable<T> { T call(); }
```

A lambda expression lets us define a functional interface (again, one abstract method), which the compiler identifies by structure. The compiler can determine the represented functional interface from its position. The type of a lambda expression is that of the associated functional interface.

Because the compiler understands where you are using a lambda expression, it can determine a lot about that expression. It knows the type of the functional interface, so it can infer other types for you.

But, Ritter observes:

> The thing to be very aware of here is that even though we are not explicitly putting the type information in there, this is not sneaking dynamic typing into Java. We would never do that. That is nasty, it is bad. So, what we are doing is to say that this is still very much statically typed, but it is more typing with less typing.

By the way, one thing that differentiates lambda expressions from closures, says Ritter, is that unlike closures, lambdas cannot access a variable from outside the lambda expression unless that variable is effectively final, meaning that although it does not require the final keyword (unlike an inner class), nonetheless the variable must not be reassigned. This convention renders it "effectively final.

## Method references

The method-reference syntax is another by-product of the new lambda-expression capability. This is a shortcut that allows you to reuse a method basically as a Lambda expression. You might have something like this:

```
FileFilter x = f -> f.canRead();
```

Such syntax tells the program to create a FileFilter that filters files based on a common property – in this case, if it can be read. Note in this example, we never mentioned that f is a file; the compiler inferred that from the signature of the lone method in FileFilter:

boolean accept(File pathname);

That can be simplified further using the new Java 8 "::" notation.

```
FileFilter x = File::canRead;
```

That is completely equivalent.

To call a constructor in a similar fashion, you can use the "::new" syntax. For example, if we have a functional interface:

```
interface Factory<T> {
 T make();
}
```

Then we can say:

```
Factory<List<String>> f =
ArrayList<String>::new;
```

This is equivalent to:

```
Factory<List<String>> f = () -> return
new ArrayList<String>();
```

And now, when f.make() is called, it will return a new ArrayList<String>.

By leveraging functional interfaces, the compiler can infer a lot about typing and intent, as we see from these examples.

## Library evolution

One advantage of lambdas and expressing code as data is that, as we have seen, existing libraries have been updated to accept lambdas as parameters. This introduced some complexity: how do you introduce new interface methods without breaking existing interface implementations?

To address this, Java introduced the concept of extension methods, a.k.a. defender methods.

Let's explain by way of example. The stream method was added to the Collection interface to provide basic lambda support. In order to add the stream method to that interface without breaking existing Collection implementations all over the globe, Java added stream as an extension method to the interface, providing a default implementation:

```
interface Collection<E> {
 default Stream<E> stream() {
 return StreamSupport.
stream(spliterator());
 }
}
```

So now you have the ability to implement the stream method, if you want to. And if you don't, Java provides a default implementation for you.

## Aggregate operations

Business operations frequently involve aggregation: find the sum, max, or average of a set of data, or group by something. Until now, such operations were typically performed by loops in an external iteration, which, as we said, restricts us from optimizing and introduces code bloat.

Java SE 8 Streams aim to solve that. In Ritter's words:

> A stream is a way of abstracting and specifying how we do aggregate computations. It is not a data structure. It is really a way of handling data, but it is not a data structure on its own and, interestingly, it can be finite but it can also be infinite. So, you can create a stream of, let's say, random numbers and there is not necessarily an end to that. This is where, sometimes, things get a little bit confusing. You think to yourself, "Well, if I got an infinite stream, I might just keep processing that data forever and ever. How do I stop what I am doing with that data?"

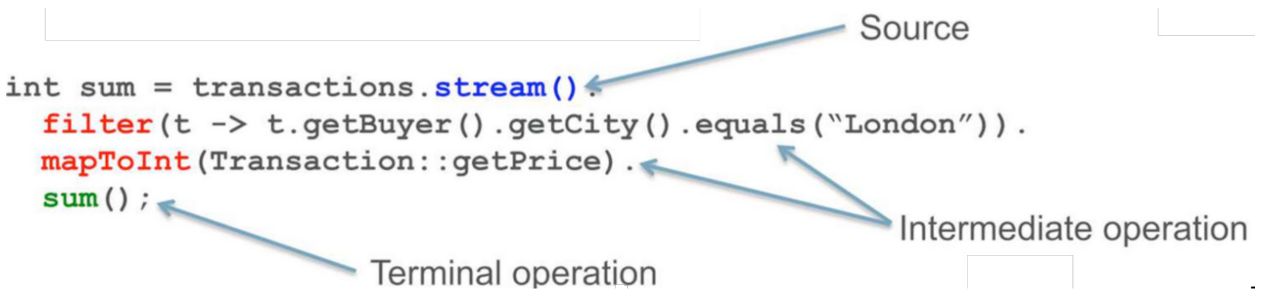
The answer is that potentially you will not. You can actually quite easily write a piece of code using streams that will continue forever, in the same way that "while(true);" loop will continue forever. It's the same with a Stream: if you use an infinite Stream, it could never end. But you can also allow that Stream to stop – say, to provide an infinite stream of random numbers but with some particular point at which to finish, That Stream will stop and you can carry on with your code.

Streams provide a pipeline of data with three important components:

1. A source of backing data

2. Zero or more intermediate operations, producing a pipeline of filtered streams

3. A terminal operation. That does one of two things: it either creates a result or it creates a side effect. (A side effect means you might not actually return a result but instead, for example, get a printout.)



```
int sum = transactions.stream().
  filter(t -> t.getBuyer().getCity().equals("London")).
  mapToInt(Transaction::getPrice).
  sum();
```

Let's look at an example, in which we start with a Collection of "transactions" and want to determine the total price of all transactions that take place with buyers from London.

In this example, we apply a stream to our Collection of transactions.

Then, we apply the filter operation to produce a new Stream of London buyers.

Next, we apply the intermediate operation mapToInt to extract the prices.

Finally, we apply the terminal sum operation to compute the result.

From the point of view of the execution, what happens here is that the filter and the map methods (our intermediate operations) do not really do any computational work. They are just responsible for setting up the pipeline of operations, and the actual computation is performed lazily, deferred until you call the terminal operation – the sum in this case – which is when all the work happens.

## Stream sources

There are a number of ways to obtain a Stream. Many methods have been added to the Collection API (using the extension methods in interfaces mechanism we discussed.)

If you have, say, a List, Set, or Map.Entry, you can call a Stream method on that and get a Stream of the contents of that collection.

An example is the stream() method, or parallelStream(), which tells the underlying library to use the fork/join framework to decompose the actions into a number of subtasks.

There are other ways of getting a stream:

- Pass an array to the stream() method in the Arrays class.
- Pass in something to Stream.of(), a static method on the Stream class.
- Call one of the new static methods to return particular streams, for example:
  - IntStream.range(), supplying a start and end index. For example, IntStream.range(1, 10) would generate a stream from 1 to 9 with an incremental step of 1. (IntRange. rangeClosed(1,10) will generate a stream from 1 to 10).
  - Files.walk() passing in a path and some optional control parameters returns a stream of individual files and sub-directories.
  - Implement the java.util.Spliterator interface to define your own way of creating a Stream. For more information on Spliterator, see Oracle's SE 8 Javadocs.

## Stream terminal operations

After we have piped together all of these streams, we must specify a terminal operation to execute the pipeline and all operations (either sequentially or in parallel) and produce the final results (or side effects).

```
int sum = transactions.stream().
  filter(t -> t.getBuyer().getCity().equals("London")).  // Lazy
  mapToInt(Transaction::getPrice).                       // Lazy
  sum();                                   // Execute the pipeline
```

## Iterable interface

This is an old friend we've known since Java 1.5 days, except that it now has a forEach() method that accepts a Consumer, the prototype class that accepts a single argument, returns no value, and produces a side effect. But this is still an external iteration, and the better approach is to supply a lambda to the map() method.

```
List<T> l = ...
s.map(λ).forEach(e -> l.add(e));
```

Replace with

```
List<T> l = s.map(λ).collect(Collectors.toList());
```

## Examples

Ritter concluded his presentation with a number of useful examples, which we list here, with explanations in the comments. (Lines in blue indicate the special usage demonstrated by each example.)

Example 1. Convert words to upper case:

```
List<String> output = wordList.
    stream().
    map(String::toUpperCase). // Map the
   entire list to an upper-case stream.
    collect(Collectors.toList()); //
   Convert the stream to a list.
```

Example 2. Find even-length words in a list:

```
List<String> output = wordList.
    stream().
    filter(w -> (w.length() & 1 == 0). //
   Select only words of even-numbered
   length.
    collect(Collectors.toList());
```

Example 3. Count lines in a file:

```
long count = bufferedReader.
lines().  // Get a stream of individual
lines. This is a new method on

// bufferedReader that returns a
stream<string>.

count();  // Our terminal operator
counts the elements of the incoming
stream.
```

Example 4. Join lines 3 and 4 into a single string:

```
String output = bufferedReader.
lines().
   skip(2).
   // Skip the first two lines.
   limit(2).
   // Get a stream consisting of the
   next two lines only.
   collect(Collectors.joining());
   // Join the lines.
```

Example 5. Find the length of the longest line in a file:

```
int longest = reader.
   lines().
   mapToInt(String::length).
   // Create a new Stream with the
   String lengths replacing

   // the actual Strings with their
   corresponding lengths.

   max().
   // Get the max element of the stream
   of
   // lengths (as an OptionalInt).

   getAsInt();
   // Replace the OptionalInt with an
   int.
```

Example 6. Collect all words in a file into a list:

```
List<String> output = reader.
   lines().
   flatMap(line -> Stream.of(line.
   split(REGEXP))).   // Get a stream of
   the words from

   // all of the lines.

   filter(word -> word.length() > 0).
   // Filter out the empty strings.

   collect(Collectors.toList());
   // Create the return list.
```

Example 7. Return the list of lower-case words in alphabetical order:

```
List<String> output = reader.
   lines().
   flatMap(line -> Stream.of(line.
   split(REGEXP))).
   filter(word -> word.length() > 0).

   map(String::toLowerCase).
   // Replace the source Stream with a
   Stream of
   // lower-case words.

   sorted().                          //
   Replace the stream with the sorted
   version.
   collect(Collectors.toList());
   // Create the return List.
```

## Conclusion

Simon Ritter concludes the presentation by declaring:

> Java needs lambda expressions to make life easier. It needs lambda expressions so that we can implement Streams, we can implement the idea of passing behavior as well as passing values. We also need to be able to extend existing interfaces, which is why Java SE 8 has extension methods, and that solves the problem of backward compatibility. What this allows us to do is to provide this idea of bulk operations on Collections and be able to do things that are far more simple, (in a) far more readable kind of way. Java SE 8 is basically evolving the language; it is evolving the class libraries and also the virtual machines at the same time.

Java 8 is available for download, and there is good lambda support in all of the major IDE's. I encourage all Java developers to download it and give Project Lambda a spin.

### ABOUT THE SPEAKER

**Simon Ritter** is Head of Java Technology Evangelism at Oracle Corporation. Simon has been in the IT business since 1984 and holds a Bachelor of Science degree in Physics from Brunel University in the U.K.

WATCH THE FULL PRESENTATION ON InfoQ

# Intuitive, Robust Date and Time Handling Finally Comes to Java

by *Stephen Colebourne*

The concepts of date and time are fundamental to many applications. Things as diverse as birth dates, rental periods, event timestamps, and shop opening hours are all based on date and time, yet Java SE had no good API to handle them. With Java SE 8, a new set of packages called java.time provides a well-structured API to cover date and time.

## History

When Java was first launched, in version 1.0, the only support for dates and times was the **java.util. Date** class. The first thing most developers noted about the class was that it didn't represent a date. What it did represent was actually quite simple: an instantaneous point in time based on millisecond precision, measured from the epoch of 1970-01-01Z. However, because the standard **toString()** form printed the date and time in the JVM's default time zone, some developers wrongly assumed that the class was time-zone aware.

The Date class was deemed impossible to fix when the time came for improvements in version 1.1. As a result a new **java.util.Calendar API** was added. Sadly, the Calendar class isn't really much better than java. util.Date. Some of the issues of these classes are:

Mutability. Classes like dates and time simply should be immutable.

Offset. Years in **Date** start from 1900. Months in both classes start from zero.

Naming. **Date** is not a "date". **Calendar** is not a "calendar".

Formatting. The formatter only works with **Date**, not **Calendar**, and is not thread-safe

Around 2001, the Joda-Time project started. Its purpose was simple: to provide a quality date and time library for Java. It took a while, but eventually version 1.0 of Joda-Time was launched and it became a popular, widely used library. Over time, demand grew to provide a library like Joda-Time in JDK. With the help of Michael Nascimento Santos from Brazil, the official process to create and integrate a new date and time API for the JDK, JSR-310, started.

## Overview

The new **java.time** API consists of five packages:

java.time - the base package containing the value objects

java.time.chrono - provides access to different calendar systems

java.time.format - allows date and time to be formatted and parsed

java.time.temporal - the low-level framework and extended features

java.time.zone - support classes for time zones

Most developers will primarily use the base and format packages, and perhaps the temporal package. Thus, although there are 68 new public types, most developers will only actively use around a third of them.

## Dates

The **LocalDate** class is one of the most important in the new API. It is an immutable value type that represents a date. There is no representation of time of day or time zone.

The "local" terminology is familiar from Joda-Time and comes originally from the ISO-8601 date and time standard. It relates specifically to the absence of a time zone. In effect, a local date is a description of a date, such as "5 April 2014". That particular local date will start at different points on the timeline depending on your location on Earth. Thus, the local date will start in Australia 10 hours before it starts in London and 18 hours before it starts in San Francisco.

The **LocalDate** class is designed to have all the methods that are commonly needed:

```
LocalDate date = LocalDate.of(2014,
Month.JUNE, 10);
int year = date.getYear(); // 2014
Month month = date.getMonth(); // JUNE
int dom = date.getDayOfMonth(); // 10
DayOfWeek dow = date.getDayOfWeek(); //
Tuesday
int len = date.lengthOfMonth(); // 30
(days in June)
boolean leap = date.isLeapYear(); //
false (not a leap year)
```

In the example, we see a date being created using a factory method (all constructors are private). It is then queried for some essential information. Note the **Month** and **DayOfWeek** enums designed to make code more readable and reliable.

In the next example, we see how an instance is manipulated. As the class is immutable, each manipulation results in a new instance, with the original unaffected.

```
LocalDate date = LocalDate.of(2014,
Month.JUNE, 10);
date = date.withYear(2015); // 2015-06-
10
date = date.plusMonths(2); // 2015-08-10
date = date.minusDays(1); // 2015-08-09
```

These changes are relatively simple, but often there is a need to make more complex alterations to a date. The **java.time** API includes the **TemporalAdjuster** mechanism to handle this. TemporalAdjuster is a pre-packaged utility capable of manipulating a date, such as obtaining the instance corresponding to the last day of the month. Common ones are supplied in the API but you can add your own. Using an adjuster is easy but does benefit from static imports:

```
import static java.time.DayOfWeek.*

import static java.time.temporal.
TemporalAdjusters.*

LocalDate date = LocalDate.of(2014,
Month.JUNE, 10);
date = date.with(lastDayOfMonth());
date = date.with(nextOrSame(WEDNESDAY));
```

The immediate reaction to seeing an adjuster in use is typically an appreciation of how close the code is to the intended business logic. Achieving that is important with date and time business logic. The last thing that we want to see is large numbers of manual manipulation of dates. If you have a common manipulation that you are going to perform many times in your codebase, consider writing your own adjuster once and getting your team to pull it in as a pre-written, pre-tested component.

## Dates and times as values

It is worth spending a brief moment considering what turns the **LocalDate** class into a value. Values are simple data types where two instances that are equal are entirely substitutable – in other words, object identity has no real meaning. The **String** class is the canonical example of a value: we care whether two strings are true by **equals()**; we don't care if they are identical by **==**.

Most date and time classes should also be values, and the java.time API fulfills this expectation. Thus, there is never a good reason to compare two LocalDate instances using ==, and in fact the Javadoc warns against it.

For those wanting to know more, see my recent definition of VALJOs, which defines a strict set of rules for value objects in Java, including immutability, factory methods, and good definitions of **equals, hashCode, toString,** and **compareTo**.

## Alternate calendar systems

The **LocalDate** class, like all main date and time classes in **java.time**, is fixed to a single calendar system: the calendar system defined in the ISO-8601 standard.

The ISO-8601 calendar system is the world's de facto civil calendar, also called the proleptic Gregorian calendar. Standard years are 365 days long, and leap years are 366 days long. Leap years occur every four years, but not every 100, unless divisible by 400. The year before the year 1 is considered to be year 0 for consistent calculations.

The first impact of using this calendar system as the default is that dates are not necessarily compatible with the results from **GregorianCalendar**. In **GregorianCalendar**, there is a switch from the Julian calendar system to the Gregorian one that occurs by default on 15 October, 1582. Before that date, it uses the Julian calendar, which has a leap year every four years without fail. After that date, it switches to the Gregorian calendar and the more complicated leap-year system we use today.

Given that this change in calendar system is a historical fact, why does the new **java.time** API not model it? The reason is that most Java applications that make use of such historic dates are incorrect today, and it would be a mistake to continue that. While the Vatican City in Rome changed calendar systems on 15 October, 1582, most of the world did not. In particular, the British Empire, including the American colonies, did not change until nearly 200 years later on 14 September, 1752. Russia didn't change until 14 February, 1918, and Sweden's change was particularly messy. Thus, the meaning of a date prior to 1918 is in fact quite open to interpretation, and faith in the single cutover found in **GregorianCalendar** is misplaced. The choice to have no cutover in **LocalDate** is a rational one. An application requires additional contextual information to accurately interpret a specific historical date that may differ between the Julian and Gregorian calendars.

The second impact of using the ISO-8601 calendar system in all the main classes is a need for an additional set of classes to handle other calendar systems. The **Chronology** interface is the main entry point to alternate calendar systems, allowing them to be looked up by locale. Four other calendar systems are provided in Java SE 8: the Thai Buddhist, the Minguo, the Japanese, and the Hirah. Other calendar systems can be supplied by applications.

Each calendar system has a dedicated date class, thus there is a **ThaiBuddhistDate**, **MinguoDate, JapaneseDate,** and **HijrahDate**. These are used if building a highly localized application, such as one for the Japanese government. An additional interface, **ChronoLocalDate**, is used as the base abstraction of these four, which, with LocalDate, allows the writing of code without knowing what calendar system it is operating on. Despite the existence of this abstraction, the intention is that it is rarely used.

Understanding why the abstraction is to be rarely used is critical to correct use of the whole**java. time** API. The truth is that the code of applications that tries to operate in a calendar-system-neutral manner is broken. For example, you cannot assume that there are 12 months in a year, yet developers do and add 12 months in the assumption that they have added a whole year. You cannot assume that all months are roughly the same length; for example, the Coptic calendar has 12 months of 30 days and one month of five or six days. Nor can you assume that the next year has a number one larger than the current year, as calendars like the Japanese restart year numbering with every new Emperor, and that change typically happens somewhere other than the start of a year – meaning that you can't even assume that two days in the same month have the same year!

The only way to write code across a large application in a calendar-system-neutral way is to have a heavy regime of code review where every line of date and time code is double-checked for bias towards the ISO calendar system. That's why the recommended use of **java.time** is to use **LocalDate** throughout your application, including all storage, manipulation, and interpretation of business rules. The only time that **ChronoLocalDate** should be used is when localizing for input or output, typically achieved by storing the user's preferred calendar system in their user profile, and even then most applications do not really need that level of localization.

For the full rationale in this area, see the Javadoc of **ChronoLocalDate**.

## Time of day

Moving beyond dates, the next concept to consider is local time of day, represented by **LocalTime**. Classic examples of this might be to represent the time that a convenience store opens, say from 07:00 to 23:00 (7 a.m. to 11p.m.). Such stores might open at those hours across the whole of the USA, but the times are local to each time zone.

**LocalTime** is a value type with no associated date or time zone. When adding or subtracting an amount of time, it will wrap around midnight. Thus, 20:00 plus 6 hours results in 02:00.

Using a **LocalTime** is similar to using **LocalDate**:

```
LocalTime time = LocalTime.of(20, 30);
int hour = date.getHour(); // 20
int minute = date.getMinute(); // 30
time = time.withSecond(6); // 20:30:06
time = time.plusMinutes(3); // 20:33:06
```

The adjuster mechanism also works with LocalTime, however there are fewer complicated manipulations of times that call for it.

## Combined date and time

The next class to consider is **LocalDateTime**. This value type is a simple combination of **LocalDate** and **LocalTime**. It represents both a date and a time without a time zone.

A **LocalDateTime** is created either directly or by combining a date and time:

```
LocalDateTime dt1 = LocalDateTime.
of(2014, Month.JUNE, 10, 20, 30);
LocalDateTime dt2 = LocalDateTime.
of(date, time);
LocalDateTime dt3 = date.atTime(20, 30);
LocalDateTime dt4 = date.atTime(time);
```

The third and fourth options use **atTime()**, which provides a fluent way to build up a date-time. Most of the supplied date and time classes have "at" methods that can be used in this way to combine the object you have with another object to form a more complex one.

The other methods on **LocalDateTime** are similar to those of **LocalDate** and **LocalTime**. This familiar pattern of methods is very useful to help learn the API. This table summarises the method prefixes used:

| Prefix | Description |
|---|---|
| of | Static factory methods that create an instance from constituent parts. |
| from | Static factory methods that try to extract an instance from a similar object. A from() method is less type-safe than an of() method. |
| now | Static factory method that obtains an instance at the current time. |
| parse | Static factory method that allows a string to be parsed into an instance of the object. |
| get | Gets part of the state of the date-time object. |
| is | Checks if something is true or false about the date-time object. |
| with | Returns a copy of the date-time object with part of the state changed. |
| plus | Returns a copy of the date-time object with an amount of time added. |
| minus | Returns a copy of the date-time object with an amount of time subtracted. |
| to | Converts this date-time object to another, which may represent part or all of the state of the original object. |
| at | Combines this date-time object with additional data to create a larger or more complex date-time object. |
| format | Provides the ability to format this date-time object. |

## Instant

When dealing with dates and times, we usually think in terms of years, months, days, hours, minutes, and seconds. However, this is only one model of time, one I refer to as "human". The second common model is "machine" or "continuous" time. In this model, a single large number represents any point in time. This approach is easy for computers to deal with, and is seen in the UNIX count of seconds from 1970, matched in Java by the millisecond count from 1970.

The **java.time** API provides a machine view of time via the **Instant** value type. It provides the ability to represent a point on the timeline without any other contextual information, such as a time zone. Conceptually, it simply represents the number of seconds since the epoch of 1970 (midnight at the start of the 1 January, 1970 UTC). Since the API is based on nanoseconds, the Instant class provides the ability to store the instant to nanosecond precision.

```
Instant start = Instant.now();
// perform some calculation
Instant end = Instant.now();
assert end.isAfter(start);
```

The **Instant** class will typically be used for storing and comparing timestamps, where you need to record when an event occurred but do not need any information about the time zone it occurred in.

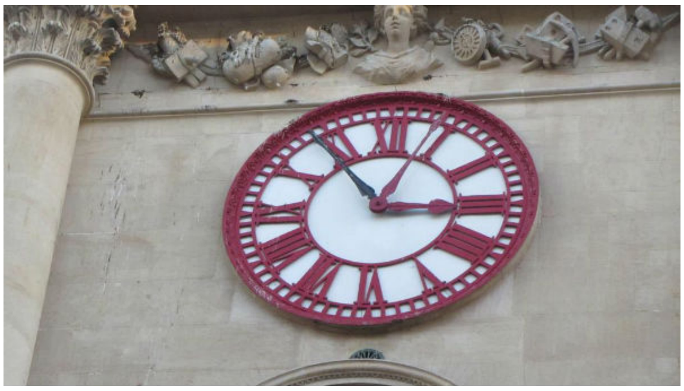
In many ways, the interesting aspect of Instant is what you cannot do with it, rather than what you can. For example, these lines of code will throw exceptions:

```
instant.get(ChronoField.MONTH_OF_YEAR);
instant.plus(6, ChronoUnit.YEARS);
```

They throw exceptions because Instant only consists of a number of seconds and nanoseconds and provides no ability to handle units meaningful to humans. If you need that ability, you need to provide time-zone information.

## Time zones

The concept of time zones was introduced by the UK, where the invention of the railway and other improvements in communication suddenly meant that people could cover distances where the change in solar time was important. Up to that point, every village and town had its own definition of time based on the sun and typically reckoned by sundial.



An example of the confusion this brought initially is shown in this photo of the clock on the old Exchange building in Bristol, UK. The red hands show Greenwich Mean Time while the black hand shows Bristol Time, 10 minutes different.

A standard system of time zones evolved, driven by technology, to replace the older local solar time. But the key fact is that time zones are political creations. They are often used to demonstrate political control over an area, such as the recent change in Crimea to Moscow time. As with anything political, the associated rules frequently defy logic. The rules can and do change with very little notice.

The rules of time zones are collected and gathered by an international group who publish the IANA Time Zone Database. This set of data contains an identifier for each region on the Earth and a history of time zone changes in each. The identifiers are of the form Europe/London or America/New_York.

Before the **java.time** API, you used the **TimeZone** class to represent time zones. Now you use the **ZoneId** class. There are two key differences. Firstly, **ZoneId** is immutable, which provides that ability to store instances in static variables amongst other things. Secondly, the actual rules themselves are held in **ZoneRules**, not in **ZoneId** itself; simply call **getRules()** on **ZoneId** to obtain the rules.

One common case of time zone is a fixed offset from UTC/Greenwich. You commonly encounter this when you talk about time differences, such as New York being five hours behind London. The class **ZoneOffset**, a subclass of **ZoneId**, represents the offset of a time from the zero meridian of Greenwich in London.

As a developer, it would be nice to not have to deal with time zones and their complexities. The **java.time** API allows you to do that so far as it is possible. Wherever you can, use the **LocalDate**, **LocalTime**, **LocalDateTime**, and **Instant** classes. When you cannot avoid time zones, the **ZonedDateTime** class handles the requirement.

The **ZonedDateTime** class manages the conversion from the human timeline, seen on desktop calendars and wall clocks, to the machine timeline measured with the ever-incrementing count of seconds. As such, you can create a ZonedDateTime from either a local class or an instant:

```
ZoneId zone = ZoneId.of("Europe/Paris");

LocalDate date = LocalDate.of(2014,
Month.JUNE, 10);
ZonedDateTime zdt1 = date.
atStartOfDay(zone);

Instant instant = Instant.now();
ZonedDateTime zdt2 = instant.
atZone(zone);
```

One of the most annoying issues of time zones is Daylight Saving Time (DST) or Summer Time. With DST, the offset from Greenwich is changed two (or more) times a year, typically moving forward in spring and back in autumn. When these adjustments happen, we all have to change the wall clocks dotted around the house. These changes are referred to by **java.time** as offset transitions. In spring, there is a "gap" in the local timeline when some local times do not occur. By contrast, in autumn, there is an "overlap" when some local times occur twice.

The **ZonedDateTime** class handles this in its factory methods and manipulation methods. For example, adding one day will add a logical day, which may be more or less than 24 hours if the DST boundary is crossed. Similarly, the method **atStartOfDay()** is so named because you cannot assume that the resulting time will be midnight - there might be a DST gap from midnight to 1 a.m.

Here's one final tip on DST. If you want to demonstrate that you have thought about what should happen in a DST overlap (where the same local time occurs twice), you can use one of the two special methods dedicated to handling overlaps:

```
zdt = zdt.withEarlierOffsetAtOverlap();
zdt = zdt.withLaterOffsetAtOverlap();
```

Use of one of these two methods will select the earlier or later time if the object is in a DST overlap. In all other circumstances, the methods will have no effect.

## Amounts of time

The date and time classes discussed so far represent points in time in various ways. Two additional value types are provided for amounts of time.

The **Duration** class represents an amount of time measured in seconds and nanoseconds – for example, 23.6 seconds.

The Period class represents an amount of time measured in years, months, and days – for example, three years, two months, and six days.

These can be added to, and subtracted from, the main date and time classes:

```
Period sixMonths = Period.ofMonths(6);
LocalDate date = LocalDate.now();
LocalDate future = date.plus(sixMonths);
```

## Formatting and parsing

An entire package is devoted to formatting and printing dates and times: **java.time.format**. The package revolves around **DateTimeFormatter** and its associated builder **DateTimeFormatterBuilder**.

The most common ways to create a formatter are static methods and constants on DateTimeFormatter. These include:

Constants for commons ISO formats, such as **ISO_LOCAL_DATE**.

Pattern letters, such as **ofPattern("dd/MM/uuuu")**.

Localized styles, such as **ofLocalizedDate(FormatStyle.MEDIUM)**.

Once you have a formatter, you typically use it by passing it to the relevant method on the main date and time classes:

```
DateTimeFormatter f = DateTimeFormatter.
ofPattern("dd/MM/uuuu");
LocalDate date = LocalDate.
parse("24/06/2014", f);
String str = date.format(f);
```

In this way, you are insulated from the format and parse methods on the formatter itself.

If you need to control the locale of formatting, use the withLocale(Locale) method on the formatter. Similar methods allow control of the calendar system, time zone, decimal numbering system, and resolution of parsing.

If you need even more control, see the **DateTimeFormatterBuilder** class, which allows complex formats to be built up step by step. It also provides abilities such as case-insensitive parsing, lenient parsing, padding, and optional sections of the formatter.

## Summary

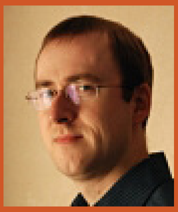
The **java.time** API is a new, comprehensive model for date and time in Java SE 8. It takes the ideas and implementation started in Joda-Time to the next level and finally allows developers to leave **java.util.Date** and **Calendar** behind. It's definitely time to enjoy date and time programming again!

Official tutorial at Oracle

Unofficial project home page

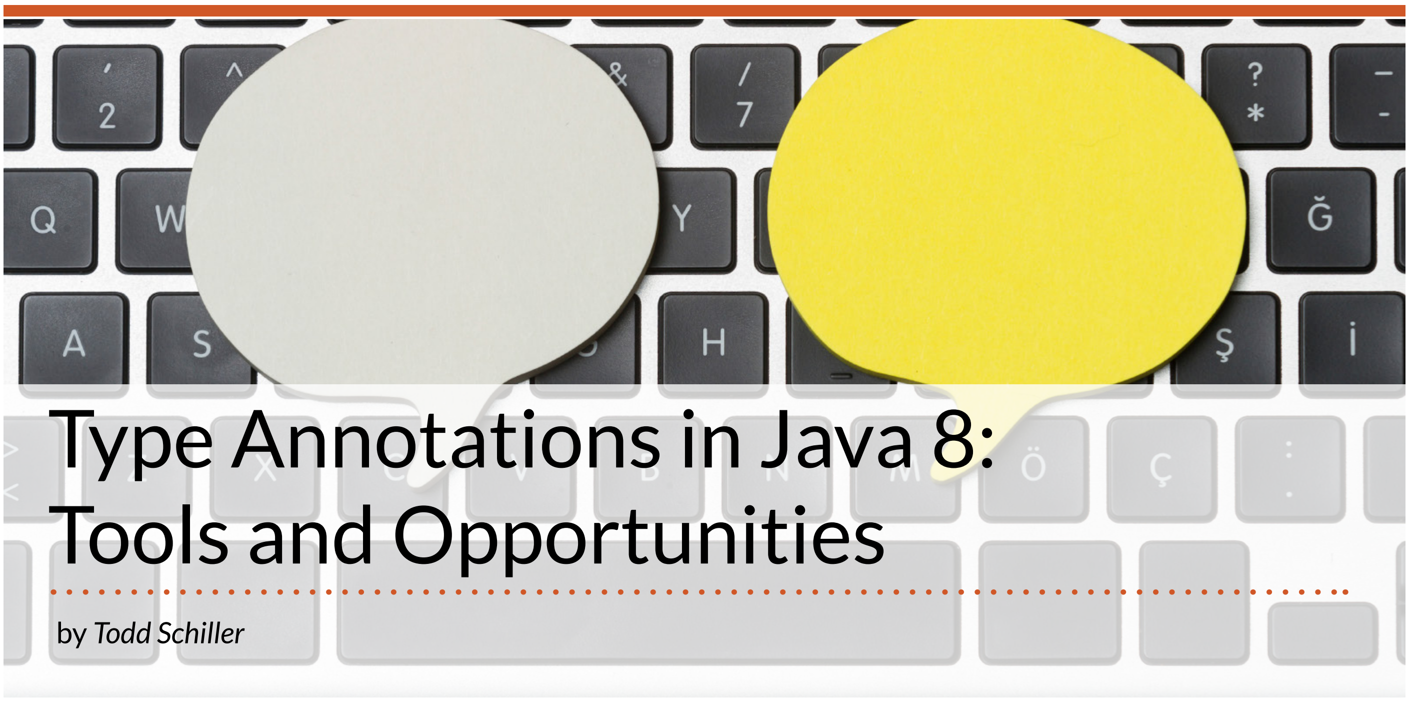
ThreeTen-Extra project, with additional classes that supplement core Java SE

### ABOUT THE AUTHOR



**Stephen Colebourne** is a Java Champion and JavaOne Rock Star speaker. He has been working with Java since version 1.0 and contributing to open-source software since 2000. He has made major contributions to Apache Commons and created the Joda open-source projects including Joda-Time. He blogs on Java and is a frequent conference speaker and contributor to discussions on language change such as Project Lambda. He was co-spec lead on JSR-310, which created the new java.time API in Java SE 8.

READ THIS ARTICLE ONLINE ON InfoQ



# Type Annotations in Java 8: Tools and Opportunities

by *Todd Schiller*

In previous versions of Java, developers could write annotations only on declarations. With Java 8, you can now write annotations on any use of a type such as types in declarations, generics, and casts:

```
@Encrypted String data;
List<@NonNull String> strings;
myGraph = (@Immutable Graph) tmpGraph;
```

At first glance, type annotations aren't the sexiest feature of the new Java release. Indeed, annotations are just syntax! Tools are what give annotations their semantics (i.e. their meaning and behavior). This article introduces the new type-annotation syntax and practical tools to boost productivity and build higher-quality software.

In the financial industry, our fluctuating market and regulatory environments mean that time to market is more important than ever. Sacrificing security or quality, however, is not an option: simply confusing percentage points and basis points can have serious consequences. The same story is playing out in every other industry.

As a Java programmer, you're probably already using annotations to improve the quality of your software. Consider the **@Override** annotation, which was introduced back in Java 1.5. In large projects with non-trivial inheritance hierarchies, it's hard to keep track of which implementation of a method will execute at run time. If you're not careful, when modifying a method declaration you might cause a subclass method to not be called. Eliminating a method call in this manner can introduce a defect or security vulnerability. In response, the **@Override** annotation was introduced so that developers could document methods as overriding a superclass method. The Java compiler then uses the annotations to warn the developer if the program doesn't match their intentions. Used this way, annotations act as a form of machine-checked documentation.

Annotations have also played a central role in making developers more productive through techniques such as metaprogramming. The idea is that annotations can tell tools how to generate new code, transform code, or behave at run time. For example, the Java Persistence API (JPA), also introduced in Java 1.5, allows developers to declaratively specify the correspondence between Java objects and database entities using annotations on declarations such as **@Entity**. Tools such as Hibernate use these annotations to generate mapping files and SQL queries at run time.

In the case of JPA and Hibernate, annotations are used to support the DRY (don't repeat yourself) principle. Interestingly, wherever you look for tools for supporting development best practices, annotations are not hard to find! Some notable examples are reducing coupling with dependency injection and separating concerns with aspect-oriented programming.

This raises the question: if annotations are already being used to improve quality and boost productivity, why do we need type annotations?

The short answer is that type annotations enable more: they allow more kinds of defects to be detected automatically and give you more control of your productivity tools.

## Type-annotation syntax

In Java 8, type annotations can be written on any use of a type, such as the following:

```
@Encrypted String data
List<@NonNull String> strings
MyGraph = (@Immutable Graph) tmpGraph;
```

Introducing a new type annotation is as simple as defining an annotation with the **ElementType.TYPE_PARAMETER** target, **ElementType.TYPE_USE** target, or both targets:

```
@Target({ElementType.TYPE_PARAMETER,
ElementType.TYPE_USE})
public @interface Encrypted { }
```

The **ElementType.TYPE_PARAMETER** target indicates that the annotation can be written on the declaration of a type variable (e.g. **class MyClass<T> {...}**). The **ElementType.TYPE_USE** target indicates that the annotation can be written on any use of a type (e.g. types appearing in declarations, generics, and casts).

Once annotations on types are in the source code, like annotations on declarations, they can both be persisted in the class file and made available at run time via reflection (using the **RetentionPolicy.CLASS** or **RetentionPolicy.RUNTIME** policy on the annotation definition). There are two primary differences between type annotations and their predecessors. First, unlike declaration annotations, type annotations on the types of local variable declarations can also be retained in class files. Second, the full generic type is retained and is accessible at run time.

Although the annotations can be stored in the class file, annotations don't affect the regular execution of the program. For example, a developer might declare two **File** variables and a **Connection** variable in the body of a method:

```
File file = ...;
@Encrypted File encryptedFile = ...;
@Open Connection connection = ...;
```

When executing the program, passing either of these files to the connection's **send(...)** method will result in same method implementation being called:

```
// These lines call the same method
connection.send(file);
connection.send(encryptedFile);
```

As you'd expect, the lack of run-time effect implies that while the types of parameters can be annotated, methods cannot be overloaded based on the annotated types:

```
public class Connection{
     void send(@Encrypted File file) {
... }
     // Impossible:
     // void send( File file) { ... }
     . . .
}
```

The intuition behind this limitation is that the compiler doesn't have any way of knowing the relationship between annotated and un-annotated types, or between types with different annotations.

But wait! There's an **@Encrypted** annotation on the variable **encryptedFile** that corresponds to the parameter file in the method signature – but where is the annotation in the method signature corresponding to the **@Open** annotation on the **connection** variable? In the call connection.send(...), the connection variable is referred to as the method›s «receiver». (The «receiver» terminology is from the classic object-oriented analogy of passing messages between objects). Java 8 introduces a new syntax for method declarations so that type annotations can be written on a method's receiver:

```
void send(@Open Connection this, @
Encrypted File file)
```

Again, since annotations don't affect execution, a method declared with the new receiver-parameter syntax has the same behavior as one using the traditional syntax. In fact, currently the only use of the new syntax is so that a type annotation can be written on the type of the receiver.

A full explanation of the type-annotation syntax, including syntax for multidimensional arrays, can be found on the [JSR (Java Specification Request) 308 website](#).

## Detecting defects with annotations

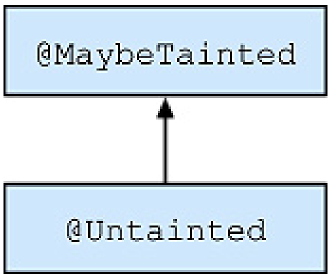
Writing annotations in the code serves to emphasize the errors in buggy code:

```
@Closed Connection connection = ...;
File file = ...;
…
connection.send(file); // Bad!: closed
and unencrypted!
```

However, the above code will still compile, run, and crash – Java's compiler does not check user-defined annotations. Instead, the Java platform exposes two APIs, the Java Compiler plugin and Pluggable Annotations Processing APIs, so that third parties can develop their own analyses.

In the previous examples, the annotations were, in effect, qualifying what values variables could contain. We could imagine other ways of qualifying the **File** type: **@Open File, @Localized File, @ NonNull File**. We could imagine these annotations qualifying other types, too, such as **@Encrypted String**. Because type annotations are independent of the Java type system, concepts expressed as annotations can be reused for many types.

But how might these annotations be checked automatically? Intuitively, some annotations are subtypes of other annotations, and their usage can be type checked. Consider the problem of preventing SQL-injection attacks caused by the database executing user-provided (tainted) input. We might think of data as being either **@Untainted** or **@ MaybeTainted**, corresponding to whether the data is guaranteed to be free from user input:



```
@MaybeTainted String userInput;
@Untainted String dbQuery;
```

The **`@MaybeTainted`** annotation can be thought of as a supertype of the **`@Untainted`** annotation. There are a couple ways of thinking about this relation. First, the set of values that might be tainted must be a superset of the values that we know aren't tainted (a value that is certainly untainted can be an element of the set of values that may be tainted). Conversely, the **`@ Untainted`** annotation provides a strictly stronger guarantee than the **`@MaybeTainted`** annotation. Let's see if our subtyping intuition works in practice:

```
userInput = dbQuery; // OK
dbQuery = "SELECT FROM * WHERE " +
userInput; // Type error!
```

The first line checks out – we can't get in trouble if we make the assumption that an untainted value is actually tainted. Our subtyping rule reveals a bug in the second line: we're trying to assign the supertype to the more restrictive subtype.

## The Checker Framework

The [Checker Framework](#) is a framework for checking Java annotations. First released in 2007, the framework is an active open-source project led by the co-lead of the JSR 308 specification, Michael Ernst. The Checker Framework comes prepackaged with a broad array of annotations and checkers for detecting defects such as null-pointer dereferences, unit-of-measure mismatches, security vulnerabilities, and threading/concurrency errors. Because the checkers use type checking under the hood, their results are sound – a checker won't miss any potential errors, whereas a tool using just heuristics might. The framework uses the compiler API to run these checks during compilation. As a framework, you can also quickly create your own annotation checkers to detect application-specific defects.

The goal of the framework is to detect defects without forcing you to write a lot of annotations. It does this primarily through two features: smart defaults and control-flow sensitivity. For example, when detecting null-pointer defects, the checker assumes that parameters are non-null by default. The checker can also make use of conditionals to determine that dereferencing an expression is safe.

```
void nullSafe(Object nonNullByDefault, @
Nullable Object mightBeNull){
     nonNullByDefault.hashCode(); // OK
due to default
     mightBeNull.hashCode(); // Warning!
```

```
        if (mightBeNull != null){
            mightBeBull.hashCode(); // OK
    due to check
        }
    }
```

In practice, defaults and control-flow sensitivity mean that you rarely have to write annotations in the body of methods; the checker can infer and check the annotations automatically. By keeping the semantics for annotations out of the official Java compiler, the Java team has ensured that third-party tool designers and users can make their own design decisions. This allows customized error checking to meet a project's individual needs.

The ability to define your own annotations also enables what you might consider domain-specific type checking. For example, in finance, interest rates are often quoted using percentages while the difference between rates is often described using basis points (1/100th of 1%). Using the Checker Framework's Units Checker, you could define the two annotations **@Percent** and **@BasisPoints** to make sure you don't mix them up:

```
BigDecimal pct = returnsPct(...); //
annotated to return @Percent
requiresBps(pct); // error: @BasisPoints
is required
```

Here, because the Checker Framework is control-flow sensitive, it knows that **pct** is a **@Percent BigDecimal** at the time of the call to **requiresBps(pct)** based on two facts: first, **returnsPct(...)** is annotated to return a **@Percent BigDecimal**; second, **pct** hasn't been reassigned before the call to **requiresBps(...)**. Often developers use naming conventions to try to prevent these kinds of defects. What the Checker Framework gives you is a guarantee that these defects don't exist, even as the code changes and grows.

The Checker Framework has already been run on millions of lines of code, exposing hundreds of defects in even well-tested software. Perhaps my favorite example is when the framework was run on the popular Google Collections library (now Google Guava). It revealed null-pointer defects that even extensive testing and heuristic-based static analysis tools had not.

These kinds of results are achievable without cluttering the code. In practice, verifying a property with the Checker Framework requires only two or three annotations per thousand lines of code!

Those of you using Java 6 or Java 7 can still use the Checker Framework to improve the quality of your code. The framework supports type annotations written as comments (e.g. /*@NonNull*/ String). Historically, the reason for this is that the Checker Framework was co-developed with JSR 308 (the type-annotation specification) beginning back in 2006.

While the Checker Framework is the best framework for taking advantage of the new syntax for error checking, it's not the only one right now. Both Eclipse and IntelliJ are type-annotation-aware:

| Support | |
|---|---|
| Checker Framework | Full support, including annotations in comments |
| Eclipse | Null error analysis support |
| IntelliJ IDEA | Can write custom inspectors, no null error analysis support |

| No Support | |
|---|---|
| PMD | |
| Coverity | |
| Find Bugs | |
| Check Style | No support for Java 8 |

## Boosting productivity with type annotations

The main driver behind the new type-annotation feature was error checking. Perhaps not surprisingly, error-checking tools have the best current and planned support for annotations. However, there are compelling applications for type annotations in productivity tools as well. To get a feeling for why, consider these examples of how annotations are used:

| | |
|---|---|
| Aspect-oriented programming | @Aspect, @Pointcut, etc. |
| Dependency injection | @Autowired, @Inject, etc. |
| Persistence | @Entity, @Id, etc. |

The annotations are declarative specifications of (1) how tools should generate code or auxiliary files, and (2) how the tools should impact the run-time behavior of the program. Using annotations in these ways can be considered metaprogramming. Some frameworks, such as Lombok, take metaprogramming with annotations to the extreme, resulting in code that barely looks like Java anymore.

Let's first consider aspect-oriented programming (AOP). AOP aims to separate concerns such as logging and authentication from the main business logic of the program. With AOP, you run a tool at compile time that adds additional code to your program based on a set of rules. For example, we could define a rule that automatically adds authentication based on type annotations:

```
void showSecrets(@Authenticated User
user){
     // Automatically inserted using
AOP:
     if (!AuthHelper.ensureAuth(user))
throw . . .;
}
```

As before, the annotation is qualifying the type. Instead of checking the annotations at compile time, however, the AOP framework is being used to automatically perform verification at run time. This example shows type annotations being used to give you more control over how and when the AOP framework modifies the program.

Java 8 also supports type annotations on local declarations that are persisted in the class file. This opens up new opportunities for performing fine-grained AOP. For example, adding tracing code in a disciplined way:

```
// Trace all calls made to the ar object
@Trace AuthorizationRequest ar = . . .;
```

Again, type annotations give more control when metaprogramming with AOP. Dependency injection is a similar story. With Spring 4, you could finally use generics as a form of qualifier:

```
@Autowired private Store<Product> s1;
@Autowired private Store<Service> s2;
```

Using generics eliminated the need to introduce classes such as **ProductStore** and **ServiceStore** or to use fragile name-based injection rules.

With type annotations, it's not hard to imagine (read: this is not implemented in Spring yet) using annotations to further control injection:

```
@Autowired private Store<@Grocery
Product> s3;
```

This example demonstrates type annotations serving as a tool for separating concerns, keeping the project's type hierarchy clean. This separation is possible because type annotations are independent of the Java type system.

## The road ahead

We've seen how the new type annotations can be used to both detect and prevent program errors and boost productivity. However, the real potential for type annotations is in combining error checking and metaprogramming to enable new development paradigms.

The basic idea is to build run times and libraries that leverage annotations to automatically make programs more efficient, parallel, or secure, and to automatically enforce that developers use those annotations correctly.

A great example of this approach is Adrian Sampson's EnerJ framework for energy-efficient computing via approximate computing. EnerJ is based on the observation that sometimes, such as when processing images on mobile devices, it makes sense to trade accuracy for energy savings. A developer using EnerJ annotates data that is non-critical using the @Approx type annotation. Based on these annotations, the EnerJ run time takes various short cuts when working with that data. For example, it might store and perform calculations on the data using low-energy approximate hardware. However, having approximate data moving through the program is dangerous; as a developer, you don't want control flow to be affected by approximate data. Therefore, EnerJ uses the Checker Framework to make sure that no approximate data can flow into data used in control flow (e.g. in an if-statement).

The applications of this approach aren't limited to mobile devices. In finance, we often face a trade-off between accuracy and speed. In these cases, the run time can be left to control the number of Monte Carlo paths or convergence criteria, or even run computation on specialized hardware, based on the current demands and available resources.

The beauty of this approach is that the concern of how the execution is performed is kept separate from the core business logic describing what computation to perform.

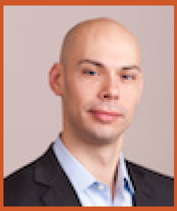
## Conclusion

In Java 8, you can write annotations on any use of a type in addition to being able to write annotations on declarations. By themselves, annotations don't affect program behavior. However, by using tools such as the Checker Framework, you can use type annotations to automatically check and verify the absence of software defects and boost your productivity with metaprogramming. While it will take some time for existing tools to take full advantage of type annotations, the time is now to start exploring how type annotations can improve both your software quality and your productivity.

## Acknowledgements

I thank Michael Ernst, Werner Dietl, and the New York City Java Meetup Group for providing feedback on the presentation on which this article is based. I thank Scott Bressler, Yaroslav Dvinov, and Will Leslie for reviewing a draft of this article.

## ABOUT THE AUTHOR



**Todd Schiller** is the head of FinLingua, a financial-software-development and consulting company. FinLingua's consulting practice helps development teams adopt domain-specific language, metaprogramming, and program-analysis techniques. Todd is an active member of the software-engineering research community; his research on specification and verification has been presented at premier international conferences including ICSE and OOPSLA.

READ THIS ARTICLE ONLINE ON InfoQ

# Where Has the Java PermGen Gone?

by *Monica Beckwith*

The Java Virtual Machine (JVM) uses an internal representation of its classes containing per-class metadata such as class hierarchy information, method data and information (such as bytecodes, stack and variable sizes), the runtime constant pool and resolved symbolic reference and Vtables.

In the past (when custom class loaders weren't that common), the classes were mostly "static" and were infrequently unloaded or collected, and hence were labeled "permanent". Also, since the classes are a part of the JVM implementation and not created by the application they are considered "non-heap" memory.

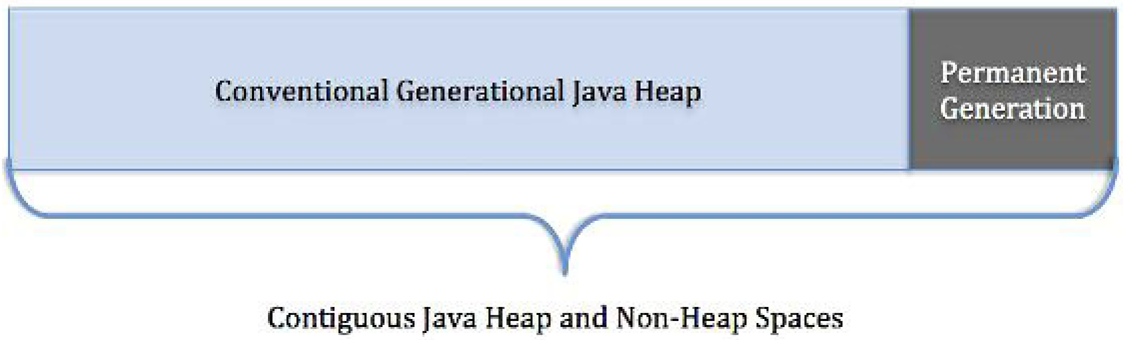
For HotSpot JVM prior to JDK8, these "permanent" representations would live in an area called the "permanent generation". This permanent generation was contiguous with the Java heap and was limited to -XX:MaxPermSize that had to be set on the command line before starting the JVM or would default to 64M (85M for 64bit scaled pointers). The collection of the permanent generation would be tied to the collection of the old generation, so whenever either gets full, both the permanent generation and the old generation would be collected. One of the obvious problems that you may be able to call out right away is the dependency on the XX:MaxPermSize. If the classes metadata size is beyond the bounds of XX:MaxPermSize, your application will run out of memory and you will encounter an OOM (Out of Memory) error.

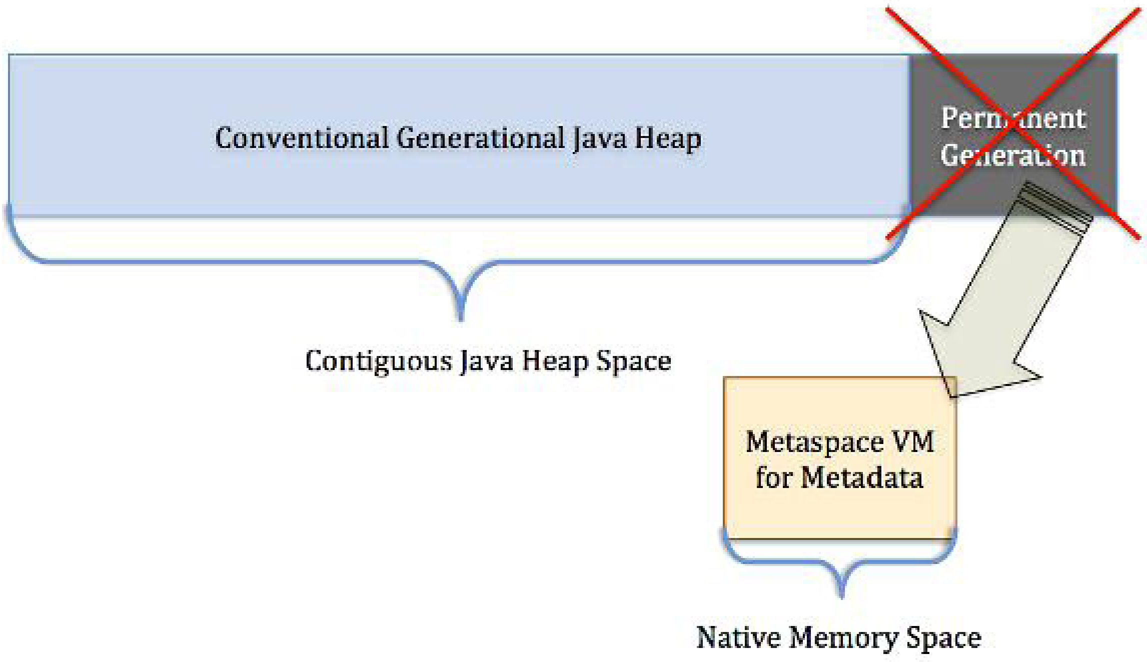
## Bye, Bye PermGen, Hello Metaspace!

With the advent of JDK8, we no longer have the PermGen. No, the metadata information is not gone, just that the space where it was held is no longer contiguous to the Java heap. The metadata has now moved to native memory to an area known as the "Metaspace".

The move to Metaspace was necessary since the PermGen was really hard to tune. There was a possibility that the metadata could move with every full garbage collection. Also, it was difficult to size the PermGen since the size depended on a lot of factors such as the total number of classes, the size of the constant pools, size of methods, etc.

Additionally, each garbage collector in HotSpot needed specialized code for dealing with metadata in the PermGen. Detaching metadata from PermGen not only allows the seamless management of Metaspace, but also allows for improvements such as simplification of full garbage collections and future concurrent de-allocation of class metadata.


Conventional Generational Java Heap
Permanent Generation
Contiguous Java Heap and Non-Heap Spaces


Conventional Generational Java Heap
Permanent Generation
Contiguous Java Heap Space
Metaspace VM for Metadata
Native Memory Space

## What Does The Removal Of Permanent Space Mean To The End Users?

Since the class metadata is allocated out of native memory, the max available space is the total available system memory. Thus, you will no longer encounter OOM errors and could end up spilling into the swap space. The end user can either choose to cap the max available native space for class metadata, or the user can let the JVM grow the native memory in-order to accommodate the class metadata.

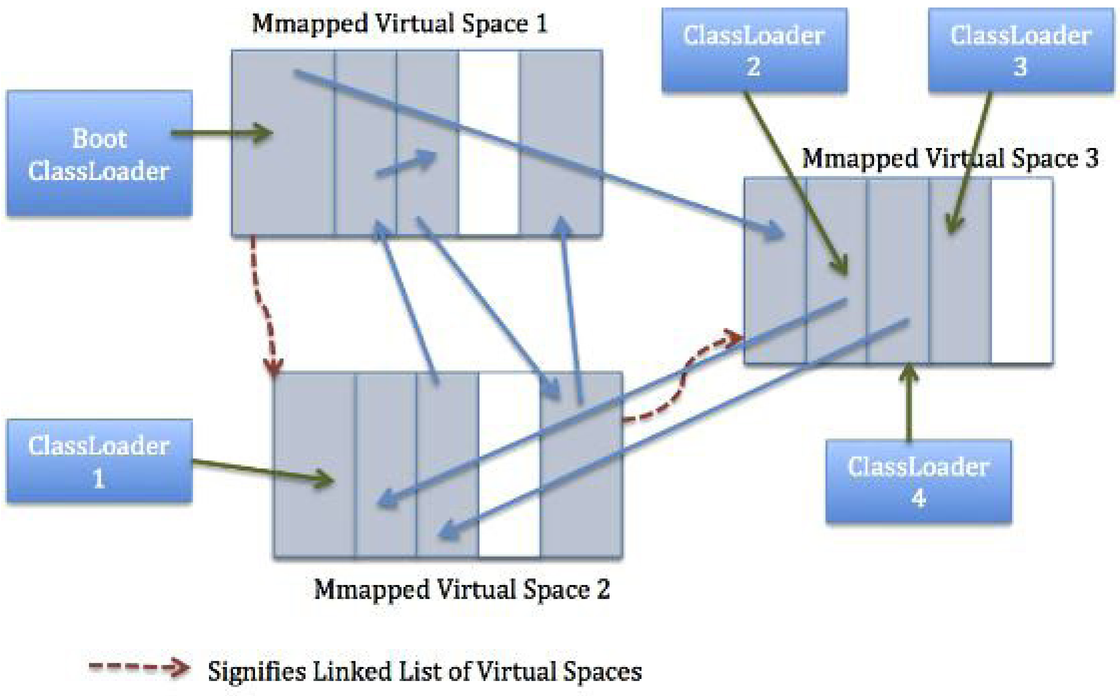
Note: The removal of PermGen doesn't mean that your class loader leak issues are gone. So, yes, you will still have to monitor your consumption and plan accordingly, since a leak would end up consuming your entire native memory causing swapping that would only get worse.

## Moving On To Metaspace And Its Allocation:

The Metaspace VM now employs memory management techniques to manage Metaspace. Thus moving the work from the different garbage collectors to just the one Metaspace VM that performs all of its work in C++ in the Metaspace.
A theme behind the Metaspace is simply that the lifetime of classes and their metadata matches the lifetime of the class loaders'. That is, as long as the classloader is alive, the metadata remains alive in the Metaspace and can't be freed.

We have been using the term "Metaspace" loosely. More formally, per classloader storage area is called "a metaspace". And these metaspaces are collectively called "the Metaspace". The reclamation of metaspace per classloader can happen only after its classloader is no longer alive and is reported dead by the garbage collector. There is no relocation or compaction in these metaspaces. But metadata is scanned for Java references.

The Metaspace VM manages the Metaspace allocation by employing a chunking allocator. The chunking size depends on the type of the classloader. There is a global free list of chunks. Whenever a classloader needs a chunk, it gets it out of this global list and maintains its own chunk list. When any classloader dies, its chunks are freed, and returned back to the global free list. The chunks are further divided into blocks and each block holds a unit of metadata. The allocation of blocks from chunks is linear (pointer bump). The chunks are allocated out of memory mapped (mmapped) spaces. There is a linked list of such global virtual mmapped spaces and whenever any virtual space is emptied, its returned back to the operating system.



The figure above shows Metaspace allocation with metachunks in mmapped virtual spaces. Classloaders 1 and 3 depict reflection or anonymous classloaders and they employ "specialized" chunk size. Classloaders 2 and 4 can employ small or medium chunk size based on the number of item in those loaders.

## Tuning And Tools For Metaspace

As previously mentioned, a Metaspace VM will manage the growth of the Metaspace. But there may be scenarios where you may want to limit the growth by explicitly setting the -XX:MaxMetaspaceSize on the command line. By default, the –XX:MaxMetaspaceSize doesn't have a limit, so technically the Metaspace size could grow into swap space and you would start getting native allocation failures.

For a 64-bit server class JVM, the default/initial value of –XX:MetaspaceSize is 21MB. This is the initial high watermark. Once this watermark is hit, a full garbage collection is triggered to unload classes (when their classloaders are no longer alive) and the high watermark is reset. The new value of the high watermark depends on the amount of freed Metaspace. If insufficient space is freed up, the high watermark goes up; if too much space is freed, the high watermark goes down. This will be repeated multiple times if the initial watermark is too low. And you will be able to visualize the repeated full garbage collections in your garbage collector logs. In such a scenario, you are advised to set the –XX:MetaspaceSize to a higher value on the command line in order to avoid the initial garbage collections.

After subsequent collections, the Metaspace VM will automatically adjust your high watermark, so as to push the next Metaspace garbage collection further out.

There are also two options: XX:MinMetaspaceFreeRatio and XX:MaxMetaspaceFreeRatio. These are analogous to GC FreeRatio parameters and they can be set on the command line as well.

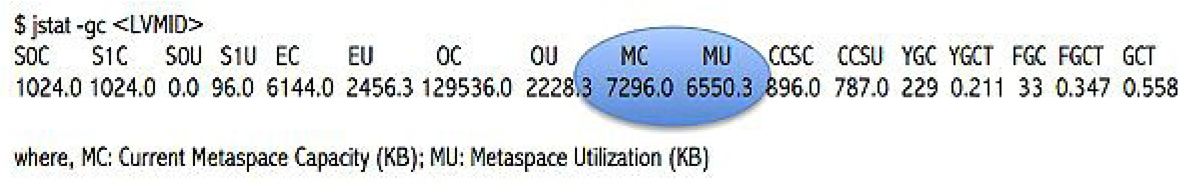
A few tools have been modified to help get more information regarding the Metaspace and they are listed here:

**jmap –clstats <PID>**: prints class loader statistics. (This now supersedes –permstat that used to print class loader stats for JVMs prior to JDK8). An example output while running DaCapo's Avrora benchmark:

```
$ jmap -clstats <PID>
Attaching to process ID 6476, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.5-b02
finding class loader instances ..done.
computing per loader stat ..done.
please wait.. computing liveness.liveness analysis may be inaccurate ...
class_loader classes      bytes parent_loader     alive? type

<bootstrap>     655  1222734     null      live   <internal>
0x000000074004a6c0    0    0    0x000000074004a708    dead      java/util/
ResourceBundle$RBClassLoader@0x00000007c0053e20
0x000000074004a760    0    0      null      dead      sun/misc/
Launcher$ExtClassLoader@0x00000007c002d248
0x00000007401189c8     1      1471 0x00000007400752f8    dead      sun/reflect/
DelegatingClassLoader@0x00000007c0009870
0x000000074004a708    116   316053    0x000000074004a760   dead      sun/misc/
Launcher$AppClassLoader@0x00000007c0038190
0x00000007400752f8    538  773854    0x000000074004a708   dead      org/dacapo/
harness/DacapoClassLoader@0x00000007c00638b0
total = 6      1310   2314112           N/A       alive=1, dead=5     N/A
```

**jstat –gc <LVMID>**: now prints Metaspace information as shown in the following example:



```
$ jstat -gc <LVMID>
S0C    S1C    S0U  S1U  EC     EU     OC       OU     MC     MU     CCSC  CCSU  YGC YGCT  FGC FGCT  GCT
1024.0 1024.0 0.0  96.0 6144.0 2456.3 129536.0 2228.3 7296.0 6550.3 896.0 787.0 229 0.211 33  0.347 0.558

where, MC: Current Metaspace Capacity (KB); MU: Metaspace Utilization (KB)
```

jcmd <PID> GC.class_stats: This is a new diagnostic command that enables the end user to connect to a live JVM and dump a detailed histogram of Java class metadata.

**Note**: With JDK8 build 13, you have to start Java with **XX:+UnlockDiagnosticVMOptions**.

```
$ jcmd <PID> help GC.class_stats
9522:
GC.class_stats
Provide statistics about Java class meta data. Requires
-XX:+UnlockDiagnosticVMOptions.
```

```
Impact: High: Depends on Java heap size and content.

Syntax : GC.class_stats [options] [<columns>]

Arguments:
     columns : [optional] Comma-separated list of all the columns
to show. If not specified, the following columns are shown:
InstBytes,KlassBytes,CpAll,annotations,MethodCount,Bytecodes,MethodAll,ROAll,RWAll,Total
(STRING, no default value)

Options: (options must be specified using the <key> or <key>=<value> syntax)
     -all : [optional] Show all columns (BOOLEAN, false)
     -csv : [optional] Print in CSV (comma-separated values) format for spreadsheets
(BOOLEAN, false)
     -help : [optional] Show meaning of all the columns (BOOLEAN, false)
```

**Note**: For more information on the columns, please see here.

An example output:

```
 $ jcmd <PID> GC.class_stats


7140:

Index Super InstBytes KlassBytes annotations   CpAll MethodCount Bytecodes MethodAll   ROAll   RWAll   Total ClassName

    1    -1    426416        480           0       0           0         0         0      24     576     600 [C

    2    -1    290136        480           0       0           0         0         0      40     576     616 [Lavrora.arch.legacy.LegacyInstr;

    3    -1    269840        480           0       0           0         0         0      24     576     600 [B

    4    43    137856        648           0   19248         129      4886     25288   16368   30568   46936 java.lang.Class

    5    43    136968        624           0    8760          94      4570     33616   12072   32000   44072 java.lang.String

    6    43     75872        560           0    1296           7       149      1400     880    2680    3560 java.util.HashMap$Node

    7   836     57408        608           0     720           3        69      1480     528    2488    3016 avrora.sim.util.MulticastFSMProbe

    8    43     55488        504           0     680           1        31       440     280    1536    1816 avrora.sim.FiniteStateMachine$State

    9    -1     53712        480           0       0           0         0         0      24     576     600 [Ljava.lang.Object;

   10    -1     49424        480           0       0           0         0         0      24     576     600 [I

   11    -1     49248        480           0       0           0         0         0      24     576     600 [Lavrora.sim.platform.ExternalFlash$Page;

   12    -1     24400        480           0       0           0         0         0      32     576     608 [Ljava.util.HashMap$Node;

   13   394     21408        520           0     600           3        33      1216     432    2080    2512 avrora.sim.AtmelInterpreter$IORegBehavior

   14   727     19800        672           0     968           4        71      1240     664    2472    3136 avrora.arch.legacy.LegacyInstr$MOVW

…<snipped>

…<snipped>

1299  1300         0        608          0     256           1         5       152     104    1024    1128 sun.util.resources.LocaleNamesBundle

 1300  1098         0        608           0    1744          10       290      1808    1176    3208    4384 sun.util.resources.OpenListResourceBundle

 1301  1098         0        616           0    2184          12       395      2200    1480    3800    5280 sun.util.resources.ParallelListResourceBundle

              2244312     794288        2024 2260976       12801    561882   3135144 1906688 4684704 6591392 Total

                34.0%      12.1%        0.0%   34.3%           -      8.5%     47.6%   28.9%   71.1%  100.0%

Index Super InstBytes KlassBytes annotations   CpAll MethodCount Bytecodes MethodAll   ROAll   RWAll   Total ClassName
```

## Current Issues

As mentioned earlier, the Metaspace VM employs a chunking allocator. There are multiple chunk sizes depending on the type of classloader. Also, the class items themselves are not of a fixed size, thus there are chances that free chunks may not be of the same size as the chunk needed for a class item. All this could lead to fragmentation. The Metaspace VM doesn't (yet) employ compaction hence fragmentation is a major concern at this moment.

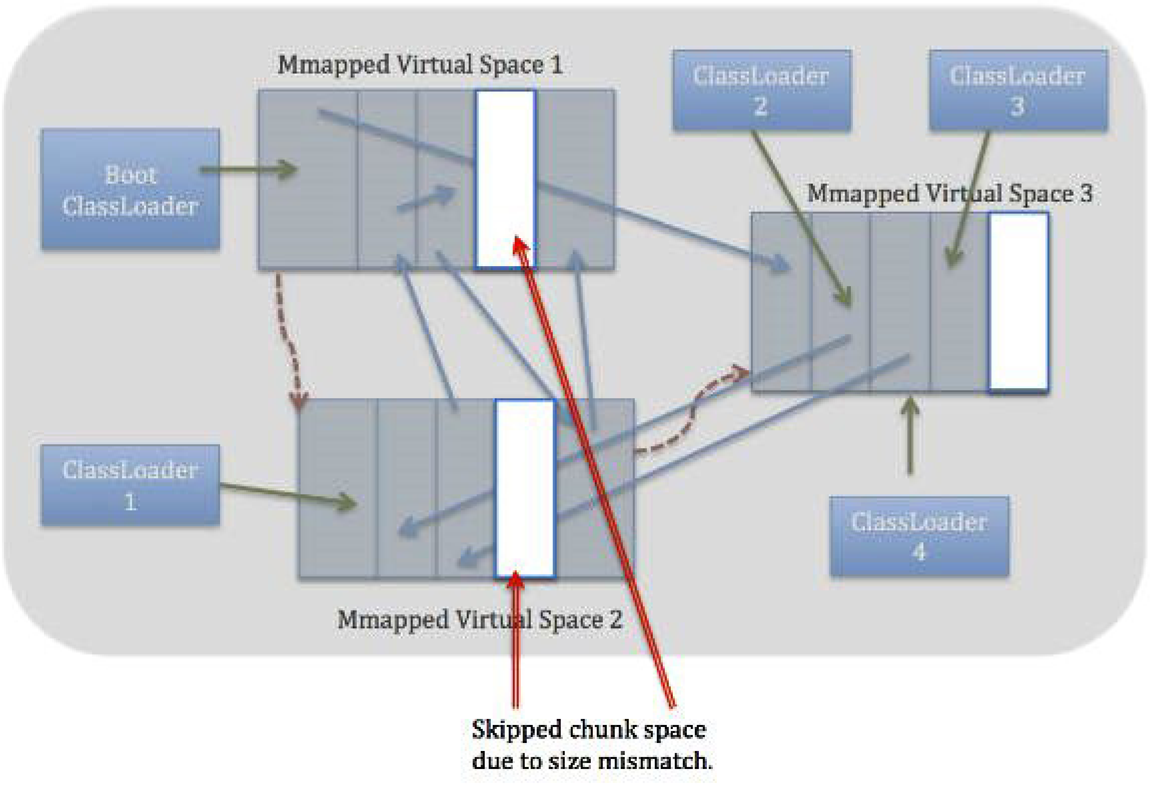
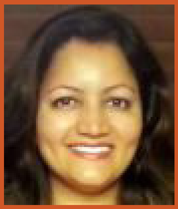


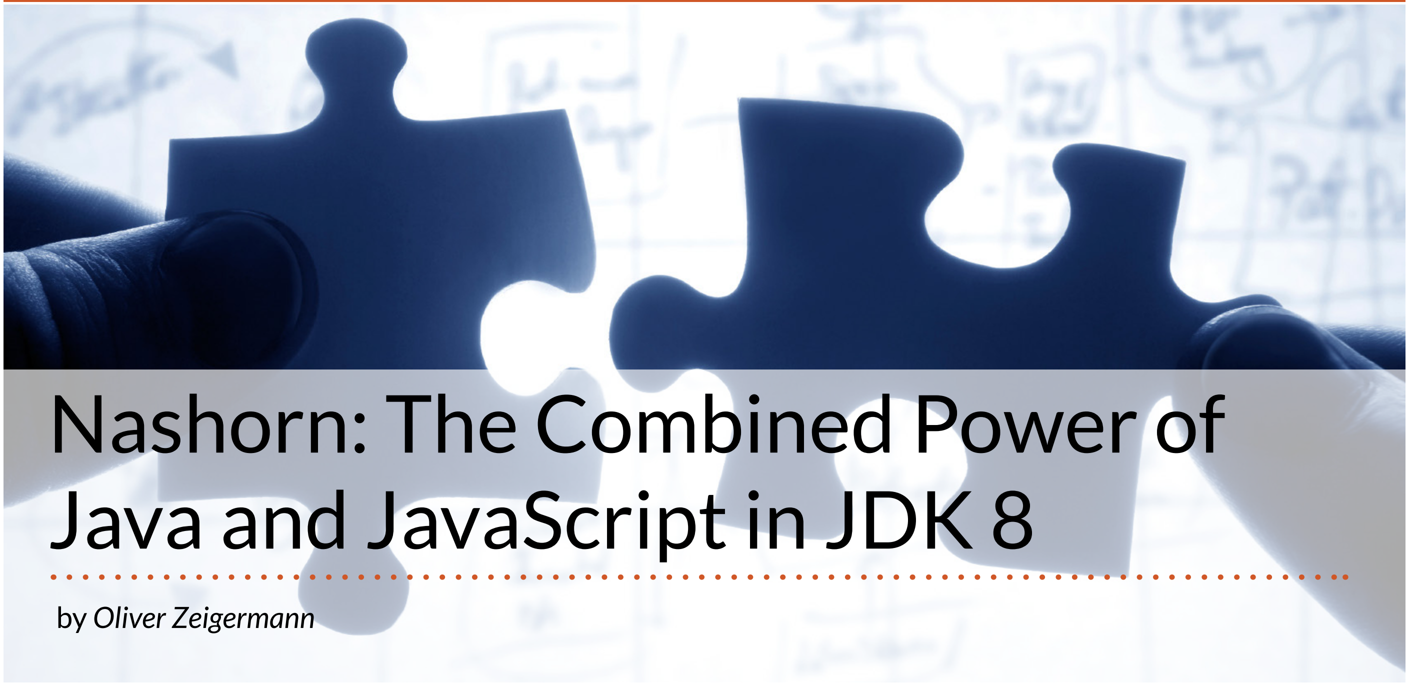
Figure highlights fragmentation due to size mismatch.

### ABOUT THE AUTHOR



**Monica Beckwith** is a performance engineer working in the hardware industry for well over a decade. Her most current title is Performance Architect at Servergy - a startup that offers a new class of hyper-efficient Cleantech Servers ®. Prior to Servergy, Monica has worked at Oracle/Sun and AMD optimizing the JVM for server class systems. Monica was voted a Rock Star speaker @JavaOne 2013. You can follow Monica on twitter @mon_beck.

READ THIS ARTICLE ONLINE ON InfoQ



# Nashorn: The Combined Power of Java and JavaScript in JDK 8

by *Oliver Zeigermann*

Ever since JDK 6, Java has shipped with a [bundled JavaScript engine] based on [Mozilla's Rhino]. This feature allowed you to embed JavaScript code into Java and even call into Java from the embedded JavaScript. Additionally, it provided the capability to run JavaScript from the command line using [jrunscript]. That was pretty good provided you didn't require great performance and you could live with the limited ECMAScript 3 feature set.

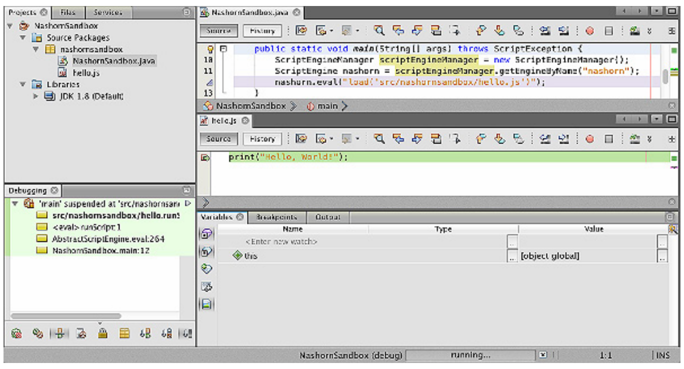
Starting with the JDK 8, [Nashorn] replaces Rhino as Java's embedded JavaScript engine. Nashorn supports the full [ECMAScript 5.1 specification] plus some [extensions]. It compiles JavaScript to Java bytecode using new language features based on [JSR 292], including invokedynamic, that were introduced in [JDK 7].

This brings a two to 10-times performance boost over the former Rhino implementation, although it is still somewhat [short of V8], the engine inside Chrome and Node.js. If you are interested in details of the implementation, you can have a look at these slides from the [2013 JVM Language Summit].

As Nashorn comes with JDK 8, it also adds very neat support for [functional interfaces], as we shall see in more detail shortly.

Let's kick it off with a very small example. First, you might want to install JDK 8 and NetBeans, IntelliJ IDEA, or Eclipse. All of them provide at least basic support for integrated JavaScript development.

Let's create a simple Java project consisting of the following two example files and let the program run:



NashornSandbox.java

```
   public static void main(String[] args) throws ScriptException {
10     ScriptEngineManager scriptEngineManager = new ScriptEngineManager();
11     ScriptEngine nashorn = scriptEngineManager.getEngineByName("nashorn");
       nashorn.eval("load('src/nashornsandbox/hello.js')");
13 }
```

hello.js

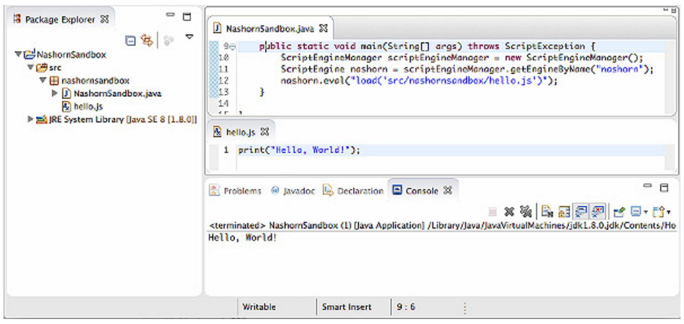
```
print("Hello, World!");
```

Debugging: 'main' suspended at 'src/nashornsan…
src/nashornsandbox/hello.run:5
<eval>.runScript:1
AbstractScriptEngine.eval:264
NashornSandbox.main:12

| Name | Type | Value |
|---|---|---|
| <Enter new watch> | | |
| this | | [object global] |

NashornSandbox (debug) running... 1:1 INS

In line 12, we use the engine's eval method to evaluate any JavaScript code. In this case, we just load the top JavaScript file and evaluate it. You might find "print" to be unfamiliar. It is not a built-in function for JavaScript, but [Nashorn provides this and other convenience functions] that come in handy in a scripting environment. You could have also embedded the printing of "Hello, World!" directly into the string passed into the eval method, but having your JavaScript in a file of its own opens up a whole world of tooling for it.



NashornSandbox.java

```
9      public static void main(String[] args) throws ScriptException {
10         ScriptEngineManager scriptEngineManager = new ScriptEngineManager();
11         ScriptEngine nashorn = scriptEngineManager.getEngineByName("nashorn");
12         nashorn.eval("load('src/nashornsandbox/hello.js')");
13     }
14
```

hello.js

```
1 print("Hello, World!");
```

Console

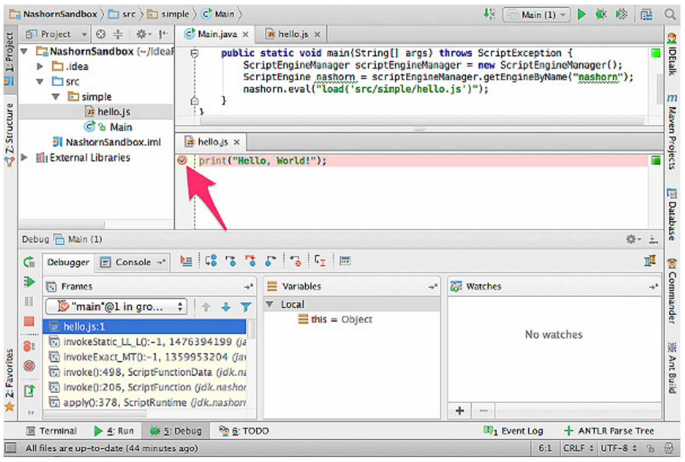
```
<terminated> NashornSandbox (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/Contents/Ho
Hello, World!
```

Writable | Smart Insert | 9 : 6

Eclipse currently has no dedicated Nashorn support via its [JavaScript Development Tools](#) (JSDT) project but it does support basic tooling and editing for JavaScript.

[IntelliJ IDEA](#) 13.1 (Community and Ultimate Editions) provides outstanding JavaScript and [Nashorn](#) support. It has a fully featured debugger and it even allows refactoring to be synchronized between Java and JavaScript; if, for example, you rename a Java class that is referenced from JavaScript or if you rename a JavaScript file sourced in from Java, the IDE will modify the corresponding references across languages.

Here is an example of how you can debug the JavaScript called from Java (note that NetBeans also provides JavaScript debugging as shown in the screenshot below):



```
public static void main(String[] args) throws ScriptException {
    ScriptEngineManager scriptEngineManager = new ScriptEngineManager();
    ScriptEngine nashorn = scriptEngineManager.getEngineByName("nashorn");
    nashorn.eval("load('src/simple/hello.js')");
}
```

```
print("Hello, World!");
```

You may say the tooling looks nice and the new implementation fixes the performance as well as the compliance problems, but ask why you should it. One reason would be general scripting. Sometimes it comes in handy to be able to throw in any kind of string and just let it be interpreted. Sometimes it might just be nice to not have a compiler in the way, or not to worry about static typing. Or maybe you are interested in the Node.js programming model, which can be used with Java, as we will see at the end of this article. There is also a [case to be made](#) that developing JavaFX will be much faster using JavaScript as opposed to Java.

## Shell scripts

The Nashorn engine can be called from the command line using the [jjs](#) command. You can call it without any argument, which will bring you into an interactive mode, you can pass the name of a JavaScript file you want executed, or you can use it as a replacement for a shell script, like this:

```
#!/usr/bin/env jjs

var name = $ARG[0];
print(name ? “Hello, ${name}!” : “Hello,
world!”);
```

To pass in program arguments to jjs, prefix them with "--". So for example you might call:

```
./hello-script.js -- Joe
```

Without the "--" prefix, the parameter would be interpreted as a file name.

## Passing data to and from Java

As indicated above, you can call JavaScript directly from your Java code; just obtain an engine and call its eval method. You can pass in data explicitly as strings:

```
ScriptEngineManager scriptEngineManager
=       new ScriptEngineManager();
ScriptEngine nashorn =
      scriptEngineManager.
getEngineByName(“nashorn”);
String name = “Olli”;
nashorn.eval(“print(‘” + name + “’)”);
```

You can also pass bindings from Java that can be accessed as global variables from inside the JavaScript engine:

```
int valueIn = 10;
SimpleBindings simpleBindings = new
SimpleBindings();
simpleBindings.put(“globalValue”,
valueIn);
nashorn.eval(“print (globalValue)”,
simpleBindings);
```

The results of a JavaScript eval computation will be returned from the engine's eval method:

```
Integer result = (Integer) nashorn.
eval(“1 + 2”);
assert(result == 3);
```

## Using Java classes in Nashorn

As mentioned before, one of the most powerful features of Nashorn comes from calling Java classes from inside JavaScript. You can not only access classes and create instances, you can also subclass them, call their static members, and do virtually anything you could do from Java.

As an example, let's take a look at threads. JavaScript does not have any language features for concurrency and all common runtimes are single-threaded or at least without any shared state. It is interesting to see that in the Nashorn environment, JavaScript could in fact run concurrently and with shared state, just like in Java:

```
// this is how we get access to Java
class Thread
var Thread = Java.type("java.lang.
Thread");

// subclass with our run method
var MyThread = Java.extend(Thread, {
    run: function() {
        print("Run in separate thread");
    }
});
var th = new MyThread();
th.start();
th.join();
```

Note that the canonical way to access a class from Nashorn is to use Java.type and you can extend a class using Java.extend.

## Functional delight

By all counts, with the release of JDK 8, Java has, at least to a certain extent, become a [functional language](). You can now use higher-order functions on collections, for example to iterate over their elements. A higher-order function is a function that takes another function as a parameter and does something meaningful with it. Have a look at this example in Java:

```
List<Integer> list = Arrays.asList(3, 4,
1, 2);
list.forEach(new Consumer() {

    @Override
    public void accept(Object o) {
        System.out.println(o);
    }
});
```

In this example, instead of iterating over the elements using an external loop as we would traditionally have done, we now pass a Consumer function to the forEach operation, a higher-order internal-looping operation that executes the consumer's accept method by passing in each element of the collection one by one.

As already mentioned, the functional-language approach for such a higher-order function would rather accept a function parameter than an object. While passing around references to functions per se is not traditionally Java's province, JDK 8 now has some syntactic sugar for expressing just that using lambda expressions (a.k.a. "closures"). For example:

```
List<Integer> list = Arrays.asList(3, 4,
1, 2);
list.forEach(el -> System.out.
println(el));
```

In this case, the parameter to forEach has the form of such a function reference. This is possible because Consumer is a functional interface, sometimes called a single abstract method (SAM) type.

So why are we talking about lambdas in a discussion of Nashorn? Because in JavaScript you can write code like this as well and Nashorn is especially well-prepared to bridge the gap between Java and JavaScript in this case. In particular, it allows you to even pass plain JavaScript functions as implementations of functional interfaces (SAM types).

Let us have a look at some plain JavaScript code that does the same thing as our Java code above. Note that there is no built-in list type in JavaScript, only arrays – but those arrays are dynamically sized and have methods comparable to the ones of a Java list. So, in this example we are calling the forEach method of a JavaScript array:

```
var jsArray = [4,1,3,2];
jsArray.forEach(function(el) { print(el)
} );
```

The similarity is obvious, but that isn't all. You can also convert such a JavaScript array to a Java list:

```
var list = java.util.Arrays.
asList(jsArray);
```

See? And yes, this is JavaScript running inside Nashorn. As this is now a Java list, you can call its forEach method. Note that this is not the same forEach method that we called on the JavaScript array, but rather Java's forEach method defined on collections. Still, we are passing in a plain JavaScript function here:

```
list.forEach(function(el) { print(el) }
);
```

Nashorn allows us to provide plain JavaScript function references where a functional interface (SAM type) is expected. This is thus not only possible from Java, but also from JavaScript.

The next version of ECMAScript (which is expected to become final in 2014) will include a short syntax for functions that will allow them to be written nearly as Java lambdas, except that it uses a fat arrow, "=>". This will drive the alignment even further.

## Special Nashorn JavaScript dialect

As I mentioned in the introduction, Nashorn supports JavaScript in the ECMAScript 5.1 version plus some extensions. I do not necessarily recommend using those extensions because, being neither Java nor JavaScript, they can feel unnatural to developers. On the other hand, Oracle uses two extensions throughout its documentation, so we should become familiar with them.

First, let us set the stage for the first extension. As you have seen before, you can extend a Java class from JavaScript using Java.extend. If you want to subclass an abstract Java class or implement an interface, you can use a more convenient syntax. In this case, you can virtually call the constructor of the abstract class or the interface and pass in a JavaScript object literal that describes the implemented methods. JavaScript object literals are just name/value pairs, similar to what you may know from the JSON format. This allows us to implement the Runnable interface like this:

```
var r = new java.lang.Runnable({
    run: function() {
        print("running...\n");
    }
});
```

In this example, we virtually call the constructor of Runnable with an object literal that specifies the implementation of the run method. Note that this is something the Nashorn implementation is giving us; it would otherwise not be possible in JavaScript.

The code of this example already looks similar to how we would implement an interface as an anonymous inner class in Java, but it's not quite the same. This brings us to the first extension, which lets you pass the last parameter after the closing ")" when you make a constructor call. Doing this, our code looks like this:

```
var r = new java.lang.Runnable() {
    run: function() {
       print("running...\n");
    }
};
```

That does exactly the same thing, but has an even greater resemblance to Java.

The second frequently used extension is a shortcut for functions that allows you to omit both the curly braces as well as the return statement for the method body in a single line function. Take our example from the previous section:

```
list.forEach(function(el) { print(el) }
);
```

That could be expressed as the slightly more terse:

```
list.forEach(function(el) print(el));
```

## Avatar.js

We have seen that with Nashorn we have a premium JavaScript engine embedded into Java. We have also seen that from Nashorn we can access any Java class. Avatar.js goes one step further and brings, as the web site says, "the Node programming model, APIs, and module ecosystem to the Java platform". To understand what this means and why it is exciting, we first have to understand what Node is. Node basically extracts Chrome's V8 JavaScript engine to make it work from the command line without the need for a browser. It thus makes JavaScript executable not only in the browser but also on the server side. To execute JavaScript on a server in any meaningful way, you will at least need to access the file system and the network. To achieve this, Node embeds a library called libuv that does this in an asynchronous way. Practically, this means that your calls to the operating system never block even if they take a while to return. Instead of blocking, you provide a callback function that will be triggered once the call is done, delivering the results if there are any.

There are several companies using Node for serious applications, among them Walmart and PayPal.

Let's take a look at a small JavaScript example that I have adapted from Node's web site:

```
// load module 'http' (this is blocking)
to handle http requests
var http = require('http');

// when there is a request we return
'Hello, World\n'
function handleRequest(req, res) {
  res.writeHead(200, {'Content-Type':
'text/plain'});
  res.end('Hello, World\n');
}

// we listen on localhost, port 1337
// and give handleRequest as call back
// you see the non-blocking /
asynchronous nature here
http.createServer(handleRequest).
listen(1337, '127.0.0.1');
// logs to the console to reassure that
we are on our way
console.log('Get your hello at
http://127.0.0.1:1337/');
```

To run this code, you would need to install Node, save the above JavaScript code into a file, and, finally, call Node with this file as a parameter.

The goal of Avatar.js is to provide the same core API as Node by binding libuv to Java classes and then making them accessible to JavaScript. Even though this may sound cumbersome, it works surprisingly well. Avatar.js supports a large number of Node modules and its support of Express, the mainstream web framework for Node, indicates that this could indeed work with a large number of existing projects.

Unfortunately, at the time of this writing, there is no binary distribution for Avatar.js. There is a readme that explains how to build it from source, but if you are not so much into building from scratch, you can get the binaries without building them. Both approaches work, but I recommend the second one for quicker results.

Once you have set up your binaries and put them into a lib folder, you would then call the Avatar.js framework using something like:

```
java -Djava.library.path=lib -jar lib/
avatar-js.jar helloWorld.js
```

We assume that the demo server (the code above) is saved in a file called "helloWorld.js".

Again, let us ask, why is this useful? The good people at Oracle (see slide 10 of this JavaOne San Francisco 2013 presentation) see a couple of use cases for such a library. I mainly concur with two of them, namely:

You have a Node application and want to use certain Java libraries to complement the Node API.

You want to switch to JavaScript and the Node API, but need to embed legacy Java code either partially or completely.

Both use cases work by using Avatar.js and calling any required Java classes from the JavaScript code, which is supported by Nashorn, as we have seen.

Let me give you an example of the first use case. JavaScript currently has just a single type for expressing numbers called "number". This would be equivalent to the Java "double", with the same limitations: JavaScript's number, like Java's double, is not able to express arbitrary range and precision, for example when dealing with money.

In Java, you could use BigDecimal, which supports exactly that. But JavaScript has no built-in equivalent, so you could just access the BigDecimal class from your JavaScript code to safely handle monetary values.

Let us look at an example web service that calculates the percentage of some amount. First, we need a function that does the actual calculation:

```
var BigDecimal = Java.type('java.math.
BigDecimal');

function calculatePercentage(amount,
percentage) {
    var result = new BigDecimal(amount).
multiply(
     new BigDecimal(percentage)).divide(
          new BigDecimal("100"), 2,
BigDecimal.ROUND_HALF_EVEN);
    return result.toPlainString();
}
```

In JavaScript, there are no declared types, but apart from that the code looks pretty similar to the Java code I have written for this task:

```
public static String calculate(String
amount, String percentage) {
    BigDecimal result = new
BigDecimal(amount).multiply(
     new BigDecimal(percentage)).divide(
          new BigDecimal("100"), 2,
BigDecimal.ROUND_HALF_EVEN);
    return result.toPlainString();
}
```

We just need to replace the handleRequest function of the Node example above to complete our code. It goes like this:

```
// load utility module 'url' to parse
url
var url = require('url');

function handleRequest(req, res) {
    // '/calculate' is the path of our
web service
    if (url.parse(req.url).pathname ===
'/calculate') {
        var query = url.parse(req.url,
true).query;
        // amount and percentage are
passed in as query parameters
        var result =
calculatePercentage(query.amount,

query.percentage);
        res.writeHead(200, {'Content-
Type': 'text/plain'});
        res.end(result + '\n');
    }
}
```

We use a second core module of Node to process the URL of the request to parse out the query parameters for amount and percentage.

I start the server (as shown above) and issue a request like this using the web browser:

```
http://localhost:1337/calculate?
amount=99700000000000000086958613&
percentage=7.59
```

I get the correct answer of "7567230000000000006600158.73", which would have been impossible using JavaScript's plain "number" type.

The second use case would make sense when you decide to migrate your existing JEE application to JavaScript and Node. In this case, you can easily access all your existing services from within JavaScript. Another related use case would be to have a new piece of server functionality built using JavaScript and Node that still can benefit from existing JEE services.

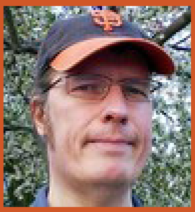
In the same direction, there is also Project Avatar, which is based on Avatar.js. Details are beyond the scope of this article but have a look at this Oracle announcement for a quick overview. The basic idea is to write your application in JavaScript and access JEE services. Project Avatar comes with a combined binary distribution for Avatar.js, but requires GlassFish for installation and development.

## Wrap-up

Project Nashorn has enhanced the original JDK 6 Rhino implementation by greatly improving performance for longer running applications, for example when used inside a web server. Nashorn integrates Java with JavaScript and even takes the new lambdas of JDK 8 into account. A real innovation comes with Avatar.js, which builds on those features and provides for integration of enterprise Java and JavaScript code while being largely compatible with the de facto standard for JavaScript server programming.

Complete examples including Avatar.js binaries for Mac OS X can be found on GitHub.

**ABOUT THE AUTHOR**



**Oliver Zeigermann** is a self-employed software architect/developer, consultant, and coach from Hamburg, Germany. He is currently focused on using JavaScript in enterprise applications.

READ THIS ARTICLE ONLINE ON InfoQ

# 8 Great Java 8 Features No One's Talking About

by *Tal Weiss*

If you haven't seen some of the videos or tutorials around Java 8, you've probably been super-busy or have a more interesting social life than I do (which isn't saying much). With new features like lambda expressions and Project Nashorn taking so much of the spotlight, I wanted to focus on some new APIs that have been a bit under the radar, but make Java 8 better in so many ways.

## 1. Stamped Locks

Multi-threaded code has long been the bane of server developers (just ask Oracle Java Language Architect and concurrency guru Brian Goetz). Over time complex idioms were added to the core Java libraries to help minimize thread waits when accessing shared resources. One of these is the classic ReadWriteLock that lets you divide code into sections that need to be mutually exclusive (writers), and sections that don't (readers).

On paper this sounds great. The problem is that the ReadWriteLock can be super slow(up to 10x), which kind of defeats its purpose. Java 8 introduces a new ReadWrite lock – called StampedLock. The good news here is that this guy is seriously fast. The bad news is that it's more complicated to use and lugs around more state. It's also not reentrant, which means a thread can have the dubious pleasure of deadlocking against itself.

StampedLock has an "optimistic" mode that issues a stamp that is returned by each locking operation to serve as a sort of admission ticket; each unlock operation needs to be passed its correlating stamp. Any thread that happens to acquire a write lock while a reader was holding an optimistic lock, will cause the optimistic unlock to be invalidated (the stamp is no longer valid). At that point the application can start all over, perhaps with a pessimistic lock (also implemented in StampedLock.) Managing that is up to you, and one stamp cannot be used to unlock another – so be super careful.

Let's see this lock in action-

```
long stamp = lock.tryOptimisticRead();
// non blocking path – super fast
work(); // we're hoping no writing will
go on in the meanwhile
if (lock.validate(stamp)){
    //success! no contention with a
writer thread
}
else {
    //another thread must have acquired
a write lock in the meanwhile, changing
the stamp.
    //bummer – let's downgrade to a
heavier read lock

    stamp = lock.readLock(); //this is a
traditional blocking read lock
        try {
    //no writing happening now
           work();
        }
        finally {
              lock.unlock(stamp); //
release using the correlating stamp
        }
}
```

## 2. Concurrent Adders

Another beautiful addition to Java 8, meant specifically for code running at scale, is the concurrent "Adders". One of the most basic concurrency patterns is reading and writing the value of a numeric counter. As such, there are many ways in which you can do this today, but none so efficient or elegant as what Java 8 has to offer.

Up until now this was done using Atomics, which used a direct CPU compare and swap (CAS) instruction (via the sun.misc.Unsafe class) to try and set the value of a counter. The problem was that when a CAS failed due to contention, the AtomicInteger would spin, continually retrying the CAS in an infinite loop until it succeeded. At high levels of contention this could prove to be pretty slow.

Enter Java 8's LongAdders. This set of classes provides a convenient way to concurrently read and write numeric values at scale. Usage is super simple. Just instantiate a new LongAdder and use itsadd() and intValue() methods to increase and sample the counter.

The difference between this and the old Atomics is that here, when a CAS fails due to contention, instead of spinning the CPU, the Adder will store the delta in an internal cell object allocated for that thread. It will then add this value along with any other pending cells to the result of intValue(). This reduces the need to go back and CAS or block other threads.

If you're asking yourself when should I prefer to use concurrent Adders over Atomics to manage counters? The simple answer is – always.

## 3. Parallel Sorting

Just as concurrent Adders speed up counting, Java 8 delivers a concise way to speed up sorting. The recipe is pretty simple. Instead of -

```
Array.sort(myArray);
```

You can now use –

```
Arrays.parallelSort(myArray);
```

This will automatically break up the target collection into several parts, which will be sorted independently across a number of cores and then grouped back together. The only caveat here is that when called in highly multi-threaded environments, such as a busy web container, the benefits of this approach will begin to diminish (by more than 90%) due to the cost of increased CPU context switches.

## 4. Switching to the new Date API

Java 8 introduces a complete new date-time API. You kind of know it's about time when most of the methods of the current one are marked as deprecated... The new API brings ease-of-use and accuracy long provided by the popular Joda time API into the core Java library.

As with any new API the good news is that it's more elegant and functional. Unfortunately there are still vast amounts of code out there using the old API, and that won't change any time soon.

To help bridge the gap between the old and new API's, the venerable Date class now has a new method called toInstant() which converts the Date into the new representation. This can be especially effective in those cases where you're working on an API that expects the classic form, but would like to enjoy everything the new API has to offer.

## 5. Controlling OS Processes

Launching an OS process from within your code is right there with JNI calls – it's something you do half-knowing there's a good chance you're going to get some unexpected results and some really bad exceptions down the line.

Even so, it's a necessary evil. But processes have another nasty angle to them - they have a tendency to dangle. The problem with launching process from within Java code so far has been that is was hard to control a process once it was launched.

To help us with this Java 8 introduces three new methods in the Process class -

destroyForcibly - terminates a process with a much higher degree of success than before.

isAlive tells if a process launched by your code is still alive.

A new overload for waitFor() lets you specify the amount of time you want to wait for the process to finish. This returns whether the process exited successfully or timed-out in which case you might terminate it.

Two good use-cases for these new methods are -

If the process did not finish in time, terminate and move forward:

```
if (process.wait(MY_TIMEOUT, TimeUnit.
MILLISECONDS)){
       //success! }
else {
    process.destroyForcibly();
}
```

Make sure that before your code is done, you're not leaving any processes behind. Dangling processes can slowly but surely deplete your OS.

```
for (Process p : processes) {
       if (p.isAlive()) {
             p.destroyForcibly();
       }
}
```

## 6. Exact Numeric Operations

Numeric overflows can cause some of the nastiest bugs due to their implicit nature. This is especially true in systems where int values (such as counters) grow over time. In those cases things that work well in staging, and even during long periods in production, can start breaking in the weirdest of ways, when operations begin to overflow and produce completely unexpected values.

To help with this Java 8 has added several new "exact" methods to the Math class geared towards protecting sensitive code from implicit overflows, by throwing an unchecked ArithmeticException when the value of an operation overflows its precision.

```
int safeC = Math.multiplyExact(bigA,
bigB); // will throw ArithmeticException
if result exceeds +-2^31
```

The only downside is that it's up to you to find those places in your code where overflows can happen. Not an automagical solution by any stretch, but I guess it's better than nothing.

## 7. Secure Random Generation

Java has been under fire for several years for having security holes. Justified or not, a lot of work has been done to fortify the JVM and frameworks from possible attacks. Random numbers with a low-level of entropy make systems that use random number generators to create encryption keys or hash sensitive information more susceptible to hacking.

So far selection of the Random Number Generation algorithms has been left to the developer. The problem is that where implementations depend on specific hardware / OS / JVM, the desired algorithm may not be available. In such cases applications have a tendency to default to weaker generators, which can put them at greater risk of attack.

Java 8 has added a new method called SecureRandom.getInstanceStrong() whose aim is to have the JVM choose a secure provider for you. If you're writing code without complete control of the OS / hardware / JVM on which it would run (which is very common when deploying to the cloud or PaaS), my suggestion is to give this approach some serious consideration.

## 8. Optional References

NulPointers are like stubbing your toes - you've been doing it since you could stand up, and no matter how smart you are today - chances are you still do. To help with this age-old problem Java 8 is introducing a new template called Optional<T>.

Borrowing from Scala and Haskell, this template is meant to explicitly state when a reference passed to or returned by a function can be null. This is meant to reduce the guessing game of whether a reference can be null, through over-reliance on documentation which may be out-of-date, or reading code which may change over time.

```
Optional<User> tryFindUser(int userID) {
```

or -

```
void processUser(User user,
Optional<Cart> shoppingCart) {
```

The Optional template has a set of functions that make sampling it more convenient, such asisPresent() to check if an non-null value is available, or  ifPresent() to which you can pass a Lambda function that will be executed if isPresent is true. The downside is that much like with Java 8's new date-time APIs, it will take time and work till this pattern takes hold and is absorbed into the libraries we use and design everyday.

New Lambda syntax for printing an optional value -

```
value.ifPresent(System.out::print);
```

## ABOUT THE AUTHOR

**Tal Weiss** is the CEO of Takipi. Tal has been designing scalable, real-time Java and C++ applications for the past 15 years. He still enjoys analyzing a good bug though, and instrumenting Java code. In his free time Tal plays Jazz drums.

READ THIS ARTICLE ONLINE ON InfoQ