Java 8 New Features

<http://blog.takipi.com/5-features-in-java-8-that-will-change-how-you-code/>

1. **Lambda expressions**

Even if we really didn’t want to go mainstream here, there’s little doubt that from a developer’s perspective, the most dominant feature of Java 8 is the new support for Lambda expressions. This addition to the language brings Java to the forefront of functional programming, right there with other functional JVM-based languages such as Scala and Clojure.

At minimum, it’s recommended you become familiar with the Lambda syntax, especially as it relates to array and collection operations, where Lambdas have been tightly integrated into the core language libraries. It is highly likely that you’ll start seeing more and more code like the snippet below in both 3rd party and within your organization’s code.

|  |  |
| --- | --- |
| 1  2 | Map<Person.Sex, List<Person>> byGender =  roster.stream().collect(Collectors.groupingBy(Person::getGender)); |

#### 2. ****Parallel operations****

With the addition of Lambda expressions to arrays operations, Java introduced a key concept into the language of internal iteration. Essentially as developers we’re used to use loop operations as one of the most basic programming idioms, right up there with if and else.

The introduction of Lambda expressions turned that paradigm around, with the actual iteration over a collection on which a Lambda function is applied now carried out by the core library itself (i.e. internal iteration).

Here’s the same snippet as above, but using the [framework’s](http://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html) new parallel processing capabilities –

|  |  |
| --- | --- |
| 1  2  3 | ConcurrentMap<Person.Sex, List<Person>> byGender =  roster.parallelStream().collect(  Collectors.groupingByConcurrent(Person::getGender)); |

#### 3. ****Java + JavaScript =****

Java 8 is looking to right one of its biggest historical wrongs – the ever growing distance between Java and JavaScript, one that has only increased in the past few years. With this new release, Java 8 is introducing a completely new JVM JavaScript engine – Nashorn. This engine makes unique use of some of the [new features introduced in Java 7](http://blog.takipi.com/2014/02/10/java-8-compiling-lambda-expressions-in-the-new-nashorn-js-engine/) such as invokeDynamic to provide JVM-level speed to JavaScript execution right there with the likes of V8 and SpiderMonkey.

#### 4. ****New date / time APIs****

The complexity of the current native Java library API has been a cause of pain for Java developers for many years. Joda time has been filling this vacuum for years now, and with Java 8. An immediate [question that arose](http://blog.joda.org/2009/11/why-jsr-310-isn-joda-time_4941.html) early on was why didn’t Java 8 adopt Joda as its native time framework. Due to what was perceived as a design flaw in Joda, Java 8 implemented its own new date / time API from scratch. The good news is that unlike Calendar.getInstance(), the [new API](http://java.dzone.com/articles/introducing-new-date-and-time)s were designed with simplicity in mind, and clear operations to operate on manipulated values in both human readable and machine time formats.

#### 5. ****Concurrent accumulators****

One of the most common scenarios in concurrent programming is updating of numeric counters accessed by multiple threads. There have been many idioms to do this over the years, starting from synchronized blocks (which introduce a high level of contention), to read/write locks to AtomicInteger(s). While the last ones are more efficient, as they rely directly on processor CAS instructions, they require a higher degree of familiarity to implement the required semantics correctly.

<https://www.javacodegeeks.com/2014/03/8-new-features-for-java-8.html>

## **1.Lamda Expressions**

I started with lambda expressions as this is probably the most sought after feature in the language after probably Generics/Annotations in Java 5.

Here’s the syntax:

|  |  |
| --- | --- |
| 1 | (argtype arg...) -> { return some expression.. probably using these arguments } |

What it does is that it reduces the code where it is obvious, such as in an anonymous innerclass. (Swing action handlers just got sexy, yay!)

So, a thread can be changed as:

|  |  |
| --- | --- |
| 1 | Runnable oldRunner = new Runnable(){ |
| 2 | public void run(){ |

|  |  |
| --- | --- |
| 3 | System.out.println("I am running"); |
| 4 | } |

|  |  |
| --- | --- |
| 5 | }; |
| 6 | Runnable java8Runner = () ->{ |

|  |  |
| --- | --- |
| 7 | System.out.println("I am running"); |
| 8 | }; |

Similar to Scala, type inference is also possible in Lambdas. Consider the following available example:

|  |  |
| --- | --- |
| 1 | Comparator c = (a, b) -> Integer.compare(a.length(), b.length()); |

Here, the types of a,b (In this case String, from the Comparator interface) are inferred as the compare method is implemented.

The symbol used to separate the block from arguments, -> is quite similar to => already used in Scala and if you are good at it, there is not much reason to switch as you will feel the way lambdas are implemented in java is inadequate(and verbose), but for a good ‘ol java programmer, this is the way to go.

## **2.Generic Type changes and improvements**

Taking clues from Lambdas, generic collections can also infer the data types to be used to an extent. The methods for instance using a generic collection need not specify genric types. Hence, the following method

|  |  |
| --- | --- |
| 1 | SomeClass.method(); |

Can be called simply ignoring the type information:

|  |  |
| --- | --- |
| 1 | SomeClass.method(); |

The type can be inferred by the method signature, which is helpful in nested calls like

|  |  |
| --- | --- |
| 1 | myCollection.sort().removeUseless().beautify(); |

## **3. Stream Collection Types (**[**java.util.stream**](http://download.java.net/jdk8/docs/api/java/util/stream/package-summary.html)**)**

A stream is a iterator that allows a single run over the collection it is called on. Along with Lambdas, this is another noteworthy feature to watch out for. You can use streams to perform functional operations like filer or map/reduce over collections which can be streamed as individual elements using Stream objects. Streams can run sequentially or parallely as desired. The parallel mode makes use of fork/join framework and can leverage power of multiple cores.

Example:

|  |  |
| --- | --- |
| 1 | List guys = list.getStream.collect(Collectors.toList()) |

can also be implemented parallely as

|  |  |
| --- | --- |
| 1 | List guys = list.getStream.parallel().collect(Collectors.toList() |

Another nice example that reduces the collection to a single item is by calling reduce algorithem.

|  |  |
| --- | --- |
| 1 | int sum = numberList.stream().reduce(0, (x, y) -> x+y); |

or,

|  |  |
| --- | --- |
| 1 | int sum = numberList.stream().reduce(0, Integer::sum); |

## **4. Functional Interfaces (**[**java.util.function**](http://download.java.net/jdk8/docs/api/java/util/function/package-summary.html)**)**

These interfaces contain some default methods which need not be implemented and can run directly from the interface. This helps with existing code – changing interfaces need not make all the classes implementing it implement new methods. This is similar to Traits in Scala and functional interfaces will be compatible with lambdas.

## **5. Nashorn –**[**The Node.js on JVM**](http://strongloop.com/strongblog/how-to-run-node-js-on-the-jvm-with-avatar-js-and-loopback)

This is the javascript engine that enables us to run javascript to run on a  jvm. It is similar to the V8 engine provided by chrome over which Node.js runs. It is compatible with Node.js applications while also allowing actual Java libraries to be called by the javascript code running on server. This is exciting to say at the least as it marries scalability and asynchronous nature of Node.js with safe and widespread server side Java middleware directly.

## **6. Date/Time changes (**[**java.time**](http://download.java.net/jdk8/docs/api/java/time/package-summary.html)**)**

http://download.java.net/jdk8/docs/api/java/time/package-summary.html

The Date/Time API is moved to java.time package and Joda time format is followed. Another goodie is that most classes are Threadsafe and immutable.

## **7. Type Annotations**

Now annotations can be used to decorate generic types itself.

Eg:

|  |  |
| --- | --- |
| 1 | List<@Nullable String> |

which is not desired always, but can prove to be useful in certain circumstances. Apart from decorating Generic types, it can also be used in constructors and casting.

|  |  |
| --- | --- |
| 1 | new @NonEmpty @Readonly List(myNonEmptyStringSet) |
| 2 | new @Interned MyObject() |

|  |  |
| --- | --- |
| 3 |  |
| 4 | myString = (@NonNull String) myObject; |

Even the array objects can be annoted:

|  |  |
| --- | --- |
| 1 | @NotNull String[] arr; |

The inclusion of RuntimeVisibleTypeAnnotations and RuntimeInvisibleTypeAnnotations attributes which cause the .class file to save the annotation information.

## **8.Other – (nice to have) Changes**

Reflection api is slightly increased with the support of TypeName, GenericString, etc.

String.join() method is a welcome addition as a lot of self created utility classes are created instead. So, the following example

|  |  |
| --- | --- |
| 1 | String abc= String.join(" ", "Java", "8"); |

Will get evaluated as “Java 8”.

In the Collections package, the Comparator interface is revamped and methods like reversed, comparing and thenCOmparing have been added which allow easy customization of comparison over multiple fields. Other libraries like the Concurrency and NIO have also been updated but is nothing noteworthy for following up and is keeping with the changes in the api.

<https://www.infoq.com/articles/Java-8-Quiet-Features>

# **8 Great Java 8 Features No One's Talking about**

## **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](https://www.goodreads.com/author/quotes/73409.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](http://www.takipiblog.com/2014/04/16/java-8-longadders-the-fastest-way-to-add-numbers-concurrently/) (up to 10x), which kind of defeats its purpose. Java 8 introduces a new ReadWrite lock – called[StampedLock](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html). 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](http://www.docjar.com/html/api/sun/misc/Unsafe.java.html) 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[LongAdder](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html)s. 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 its[add()](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html#add-long-) and[intValue()](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html#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%](http://www.takipiblog.com/2014/04/03/new-parallelism-apis-in-java-8-behind-the-glitz-and-glamour/)) due to the cost of increased CPU context switches.

## **4. Switching to the new Date API**

Java 8 introduces a complete new [date-time](http://www.oracle.com/technetwork/articles/java/jf14-date-time-2125367.html) 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()](http://docs.oracle.com/javase/8/docs/api/java/util/Date.html#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.

Posted by [**Tal Weiss**](https://www.infoq.com/author/Tal-Weiss) on May 30, 2014 | [6**Discuss**](https://www.infoq.com/articles/Java-8-Quiet-Features#theCommentsSection)

* Share
* **[Share](https://www.addthis.com/bookmark.php?v=250&username=infoq)**
* |
* [**Read later**](https://www.infoq.com/articles/Java-8-Quiet-Features)
* [**My Reading List**](https://www.infoq.com/showbookmarks.action)

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](https://www.goodreads.com/author/quotes/73409.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).

**Related Vendor Content**

### [Node.js Performance Cheat Sheet](https://www.infoq.com/vendorcontent/show.action?vcr=3932&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

### [The Ultimate DevOps Toolkit](https://www.infoq.com/vendorcontent/show.action?vcr=3934&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

### [Top 10 Java Performance Problems](https://www.infoq.com/vendorcontent/show.action?vcr=3933&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

### [GitHub Integration with Upsource: Why and How](https://www.infoq.com/vendorcontent/show.action?vcr=3954&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

### [Start your FREE TRIAL of AppDynamics Pro](https://www.infoq.com/infoq/url.action?i=12135&t=f)

**Related Sponsor**

[https://cdn.infoq.com/statics_s1_20160712-0252u2/resource/sponsorship/featuredcategory/7703/Assets-Logo-RSB-07012016.png](https://www.infoq.com/infoq/url.action?i=12137&t=f)

[**How to Build (and Scale) with Microservices**](https://www.infoq.com/infoq/url.action?i=12136&t=f)

On paper this sounds great. The problem is that the ReadWriteLock can be [super slow](http://www.takipiblog.com/2014/04/16/java-8-longadders-the-fastest-way-to-add-numbers-concurrently/) (up to 10x), which kind of defeats its purpose. Java 8 introduces a new ReadWrite lock – called[StampedLock](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html). 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](http://www.docjar.com/html/api/sun/misc/Unsafe.java.html) 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[LongAdder](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html)s. 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 its[add()](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html#add-long-) and[intValue()](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html#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%](http://www.takipiblog.com/2014/04/03/new-parallelism-apis-in-java-8-behind-the-glitz-and-glamour/)) due to the cost of increased CPU context switches.

## **4. Switching to the new Date API**

Java 8 introduces a complete new [date-time](http://www.oracle.com/technetwork/articles/java/jf14-date-time-2125367.html) 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()](http://docs.oracle.com/javase/8/docs/api/java/util/Date.html#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](http://docs.oracle.com/javase/8/docs/api/java/lang/Process.html) class -

1. [destroyForcibly](http://docs.oracle.com/javase/8/docs/api/java/lang/Process.html#destroyForcibly--) - terminates a process with a much higher degree of success than before.
2. [isAlive](http://docs.oracle.com/javase/8/docs/api/java/lang/Process.html#destroyForcibly--) tells if a process launched by your code is still alive.
3. 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](http://docs.oracle.com/javase/8/docs/api/java/lang/Math.html#addExact-int-int-) 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](http://docs.oracle.com/javase/8/docs/technotes/guides/security/enhancements-8.html) 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()](http://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html#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](https://i.chzbgr.com/maxW500/6044325376/hCEF652C1/) - 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>](http://docs.oracle.com/javase/8/docs/api/java/util/Optional.html).

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 as[isPresent()](http://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#isPresent--)to check if an non-null value is available, or  [ifPresent()](http://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#ifPresent-java.util.function.Consumer-) 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);