JAVA 8 REVEALED

LAMBDAS, DEFAULT METHODS AND BULK DATA OPERATIONS

BY ANTON ARHIPOV

TABLE OF CONTENTS

INTRODUCTION
TO JAVA 8

> I proposed to delay the release of Java 8 in order to finish Project Lambda, which has been delayed due to Oracle's renewed focus on the security of the Java Platform. Thanks to everyone who responded to that proposal, in comments on my blog entry, on Twitter, and elsewhere. The feedback was generally in favor, though understandably tinged with disappointment. As of today we have a new plan of record: The target release date for Java 8 is now 2014/3/18.

Mark Reinhold
Chief Architect of the Java Platform Group at Oracle
http://mreinhold.org/blog/hold-the-train

I've been writing about this in bits and pieces for a few months now, and wanted to pull it all together into a single publication (you can download our pretty PDF version as well). I'm talking about Java 8, the long-awaited release. And even though some delays have occurred along the way, many believe it's going to be worth it because of three major additions to the language: Lambdas, default (or defender) methods, and bulk data operations.

In case you're hearing about it for the first time, Project Lambda is the major theme in the upcoming Java 8 and probably the most awaited feature for Java developers. Lambdas, as a language feature, do not come alone: to get most out of lambdas, you're gonna need interface improvements for the standard JDK libraries.

The main driver behind the new lambda feature et al is the hardware trend of going towards multi-core. Chip designers have nowhere to go but parallel, and software developers must find out a better way to utilize the features of the underlying hardware. For application developers, and in our case Java developers, need simple parallel libraries to make their work more efficient. The obvious place to start for such an evolution is the parallelization of collections. Lambdas, it turns out, are great for improving the readability of your code and the expressiveness of the language.

It would be fair to say that without more language support for parallel idioms, people will instinctively reach for serial idioms. Hence, the new parallel idioms should be simple, intuitive and easy to use.
The code that we write today is inherently serial. Consider the following example:

```
List persons = asList(new Person("Joe"), new Person("Jim"),
new Person("John"));
for(Person person : persons) {
  doSomething(person);
}
```

Historically, Java collections were not capable of expressing internal iterations, as the only way to describe an iteration flow was the **for** (or **while**) loop. The example above describes an iteration over the collection of elements and there's no good way to express that the elements of the collection could be processed concurrently. However, with the new additions to the Java language, we could describe the same operation in a slightly different way:

```
List persons = asList(new Person("Joe"), new Person("Jim"), new
Person("John"));
persons.forEach(this::doSomething);
```

Now, if the underlying library implementation actually supports parallel operations, the elements of the collection could be processed concurrently. We can just pass the operation into the **forEach** method and rely on it to perform computation for us. All the new language features that are being added to Java 8 are actually the tools to enable the path to multi-code and parallel libraries.

In this investigation, we provide the overview of the new language features in Java 8, such as lambda expressions and default methods. And as the goal of these language changes is to improve collections library, we will also cover some of new stuff in that area.

Here's what you will learn about by reading this Rebel Labs report:

Lambdas in Java 8: lambda syntax, SAM types, functional interfaces.

Default methods: the what, why, and how.

Bulk data operations for Java collections: the new way to express operations for data processing.

Although Java 8 is yet to be released, it is still possible to get a taste of what it will look like by downloading the early access binaries for all the target platforms from http://jdk8.java.net.

**Note: Feel free to access the same-titled HTML version of this document to extract code snippets more conveniently*

PART I
GETTING STARTED

In this part we are going to take a look at the syntax of the lambda expressions. We'll point the transition path from the traditional old-school Java syntax to the brand new lambda syntax.

We will also take a look under the covers to see how lambdas are represented by the runtime and what bytecode instructions are involved.

Getting started

If you are familiar with other languages that include lambda expressions, such as Groovy or Ruby, you might be surprised at first that it is not as simple in Java. In Java, lambda expression is **SAM type**, which is an interface with a single abstract method (yes, interfaces can include non-abstract methods now (default/defender methods which we'll go over later in the text).

For instance, the well known Runnable interface is perfectly suitable for serving as a SAM type:

```
Runnable r = () -> System.out.println("hello lambda!");
```

```
Or the same could be applied to the Comparator interface:
```

```
Comparator cmp = (x, y) -> (x < y) ? -1 : ((x > y) ? 1 : 0);
```

```
The same can be written as follows:
```

```
Comparator cmp = (x, y) -> {
    return (x < y) ? -1 : ((x > y) ? 1 : 0);
};
```

So it seems like the one-liner lambda expressions have implicit **return** for the statement.Let me remind you how the same **Comparator** code is implemented in the pre-Java 8 syntax:

```
Comparator cmp = new Comparator() {
  @Override
  public int compare(Integer x, Integer y) {
    return (x < y) ? -1 : ((x > y) ? 1 : 0);
  }
};
```

As you can see, there's a good portion of the code that is identical in both examples, and that is the real code--the functional part of the comparator:

```
(x < y) ? -1 : ((x > y) ? 1 : 0)
```

Basically, what we care about when transforming the old-style Java syntax to the lambda syntax is the set of the parameter of the interface method, and the functional part itself.

Let's take a look at another example. What if I want to write a method that can accept a lambda expression as a parameter? Well, you have to declare the parameter as a functional interface, and then you can pass the lambda in:

```
interface Action {
  void run(String param);
}

public void execute(Action action){
  action.run("Hello!");
}
```

If we want to call the **execute(..)** method, we would normally pass it an anonymous implementation of the **Action** interface:

```
execute(new Action {
  public void run(String param){
    System.out.println(param);
  }
});
```

But as we now have a functional interfaces type as a parameter, we can invoke the **execute(..)** method as follows:

```
execute((String param) -> System.out.println(param));
```

Not that we can actually amend the lambda parameter type declaration:

```
execute(param -> System.out.println(param));
```

Generally, the rule is as follows: **You either declare the types of all the parameters for the lambda, or amend all of them.**

Effectively, the same expression can be replaced with a method reference since it is just a single method call with the same parameter as is:

```
execute(System.out::println);
```

However, if there's any transformations going on with the argument, we can't use method references and have to type the full lambda expression out:

```
execute(s -> System.out.println("*" + s + "*"));
```

The syntax is rather nice and we now have quite an elegant solution for lambdas in the Java language despite the fact Java doesn't have functional types *per se.*

Functional interfaces

As we learned, the runtime representation of a lambda is a functional interface (or a SAM type), an interface that defines only one abstract method. And although JDK already includes a number of interfaces, like Runnable and Comparator, that match the criteria, it is clearly not enough for API evolution. It just wouldn't be as logical if we started using Runnables all around the code.

There's a new package in JDK8, java.util.function, that includes a number of functional interfaces that are intended to be used by the new API. We won't list all of them here – just do yourself a favour and study the package yourself :)

Here's just a few interesting interfaces defined in the above mentioned package of the JDK library:

Consumer<T> - performs an action on type T without returning a result.
Supplier<T> - returns an instance of type T, no input required.
Predicate<T> - consumes an instance of type T as a parameter and produces boolean value.
Function<T,R> - takes instance of type T as a parameter and produces an instance of type R as a result.

There are over 40 new functional interfaces defined in the **java.util.function** package. Often the intention of the interface can be derived from the name. For instance, BiFunction is very similar to the Function interface mentioned above, with the only difference that it takes two input parameters instead of one.

An other common pattern that we could see in the new set of interfaces is when an interface extends the other interface in order to define the same type for multiple parameters. For instance, BinaryOperator extends BiFunction, and the purpose is just to ensure that the two input arguments would be of the same type.

```
@FunctionalInterface
public interface BinaryOperator extends BiFunction<T,T,T> {}
```

To emphasize the intention of an interface to be used as a functional one, the new @FunctionalInterface annotation can be applied to prevent your team mates from adding new method declarations into the interface. Aside of its runtime presence the annotation is used by **javac** to verify if the interface is really a functional interface and there's no more than one abstract method in it.
The following code will not compile:

```
@FunctionalInterface
interface Action {
  void run(String param);
  void stop(String param);
}
```

The compiler throw an error:

```
java: Unexpected @FunctionalInterface annotation
    Action is not a functional interface
    multiple non-overriding abstract methods found in interface
Action
```

But the following will compile just fine:

```
@FunctionalInterface
interface Action {
  void run(String param);
  default void stop(String param){}
}
```

Capturing variables

If a lambda expression accesses a non-static variable or an object that is defined outside of lambda body, then we have a situation when the lambda expression captures the outer scope, i.e. it is a capturing lambda.

Consider the comparator example:

```
int minus_one = -1;
int one = 1;
int zero = 0;
Comparator cmp = (x, y) -> (x < y) ? minus_one : ((x > y) ? one : zero);
```

In order for this lambda expression to be valid, the variables minus_one, one and zero, it captures must be "effectively final". It means that the variables should either be declared final, or they should not be re-assigned.

Lambdas as return values

In the examples above, the interfaces were used as parameters of some other method. However, the use of the functional interface is not constrained to a parameter but it also can be a return type of a method. It means that we can actually return a lambda from a method:

```
public class ComparatorFactory {
  public Comparator makeComparator(){
    return Integer::compareUnsigned;
  }
}
```

The example above demonstrates a valid code of a method that returns a method reference. In fact, method reference cannot be returned from a method just like that. Instead, the compiler will generate code, using the **invokedynamic** bytecode instruction, to evaluate it to a method call that returns an instance of a **Comparator** interface. So the client code will just assume it works with an interface:

```
Comparator cmp =
    new ComparatorFactory().makeComparator();
    cmp.compare(10, -5); // -1
```

Functional interfaces

The code snippet used in the previous section, produces a Comporator instance that could be used by the client code. It all works beautifully. However, there's one serious drawback - if we try to serialize the comparator instance the code will throw NotSerializableException.

By default, lambdas could not be made serializable as it would be a security threat. To fix this issue, the so-called type intersection was introduced to Java 8:

```
public class ComparatorFactory {
  public Comparator makeComparator() {
    return (Comparator & Serializable) Integer::compareUnsigned;
  }
}
```

Serializable interface is generally know as a marker interface - it does not declare any methods. It can also be referred as **ZAM** (Zero Abstract Methods) type.

The general rule of application for the type intersection is as follows:
SAM & ZAM1 & ZAM2 & ZAM3

It means that if the result is of SAM type, then we can "intersect" it with one or more ZAM types. Effectively, we now say that the resulting instance of the **Comparator** interface is now also **Serializable**.
By casting the result as shown above, the compiler generates one more method into the compiled class:

```
private static java.lang.Object $deserializeLambda$(java.lang.
invoke.SerializedLambda);
```

Again, by applying the invokedynamic bytecode instruction trickery, the compiler will bind the invocation of **$deserializaLambda$(..)** method when an instance of the **Comparator** is created by **makeComparator()** method.

Decompiling lambdas

Let's take a look under the covers now. It would be interesting to see, how the code is actually compiled when we use lambda expressions in our code.

Currently (as of Java 7 and before), if you wanted to emulate lambdas in Java, you have to define an anonymous inner class. This results in a dedicated class file after compilation. And if you have multiple such classes defined in the code they just get a number suffix in the name of the class file. What about lambdas? Consider code like this:

```
public class Main {

  @FunctionalInterface
  interface Action {
    void run(String s);
  }

  public void action(Action action){
    action.run("Hello!");
  }

  public static void main(String[] args) {
    new Main().action((String s) -> System.out.print("*" + s + "*"));
  }

}
```

The compilation produces two class files: **Main.class** and **Main$Action**. class, and no numbered class which would usually appear for the anonymous class implementation. So there must be something in **Main.class** now that represents the implementation of the lambda expression that I've defined in main method.

```
$ javap -p Main

Warning: Binary file Main contains com.zt.Main
Compiled from "Main.java"
public class com.zt.Main {
  public com.zt.Main();
  public void action(com.zt.Main$Action);
  public static void main(java.lang.String[]);
  private static java.lang.Object lambda$0(java.lang.String);
}
```

Aha! There's a generated method **lambda$0** in the decompiled class! The -c -v switches will give us the real bytecode along with the constants pool definitions.

The main method reveals that invokedynamic is now issued to dispatch the call:

```
public static void main(java.lang.String[]);
   Code:
     0: new           #4       // class com/zt/Main
     3: dup
     4: invokespecial  #5       // Method "<init>":()V
     7: invokedynamic  #6,  0   // InvokeDynamic #0:lambda:()Lcom/
zt/Main$Action;
   12: invokevirtual  #7       // Method action:(Lcom/zt/
Main$Action;)V
   15: return
```

And in the constant pool it is possible to find the bootstrap method that links it all at runtime:

BootstrapMethods:

```
  0: #40 invokestatic java/lang/invoke/LambdaMetafactory.metaFactory:(      \
      Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;             \
      Ljava/lang/invoke/MethodType;Ljava/lang/invoke/MethodHandle;          \
      Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodType;)         \
      Ljava/lang/invoke/CallSite;
  Method arguments:
    #41 invokeinterface com/zt/Main$Action.run:(Ljava/lang/String;)Ljava/lang/Object;
    #42 invokestatic com/zt/Main.lambda$0:(Ljava/lang/String;)Ljava/lang/Object;
    #43 (Ljava/lang/String;)Ljava/lang/Object;
```

You can see that MethodHandle API is used all around but we won't dive in this right now. For now we can just confirm that the definition refers to the generated method **lambda$0**.

What if I define my own static method with the same name? **lambda$0** is a valid identifier after all! So I defined my own **lambda$0** method:

```
public static void lambda$0(String s){
  return null;
}
```

With this compilation failed, not allowing me to have this method in the code:

```
java: the symbol lambda$0(java.lang.String) conflicts with a
        compiler-synthesized symbol in com.zt.Main
```

It actually tells us that lambdas are captured before the other structures in the class during the compilation.

SUMMARY

Drawing a parallel with Part 1 of this report,, we can definitely say that lambdas will have a great impact on Java very soon. The syntax is quite nice and once developers realize that these features provide value to their productivity, we will see a lot of code that leverages these features.

TRY JREBEL & TAKE YOUR PICK.

THE DEVELOPERS BUCKET (WISH) LIST

PART II
DEFAULT METHODS

Once published, it is impossible to add methods to an interface without breaking the existing implementations. The purpose of default methods, introduced in Java 8, is to enable interfaces to evolve without introducing incompatibility with existing implementations.

Why default methods?

Suppose Java 8 is out and has lambdas. Now you would like to start using lambdas and the most obvious use case for that is to apply a lambda to every element of a collection.

```
List<?> list = ...
list.forEach(...); // lambda code goes here
```

The **forEach** isn't declared by **java.util.List** nor the j**ava.util.Collection** interface yet. One obvious solution would be to just add the new method to the existing interface and provide the implementation where required in the JDK. However, once published, it is impossible to add methods to an interface without breaking the existing implementation.

So it'd be really frustrating if we had lambdas in Java 8 but couldn't use those with the standard collections library since backwards compatibility can't be sacrificed.

Due to the problem described above a new concept was introduced. **Virtual extension methods**, or, as they are often called, **defender methods,** can now be added to interfaces providing a default implementation of the declared behavior.

Simply speaking, interfaces in Java can now implement methods. The benefit that default methods bring is that now it's possible to add a new default method to the interface and it doesn't break the implementations.

The default methods isn't the language feature that would be appropriate to use every day, but it is an essential feature for Java Collections API update to be able to use lambdas naturally.

Defenders 101

Let's start with the simplest example possible: an **interface A,** and a **class Clazz** that implements **interface A**.

```
public interface A {
  default void foo(){
     System.out.println("Calling A.foo()");
  }
}

public class Clazz implements A {
}
```

The code compiles even though Clazz does not implement method **foo().** Method **foo()** default implementation is now provided by **interface A.**

And the client code that uses the example:

```
Clazz clazz = new Clazz();
clazz.foo(); // Calling A.foo()
```

There is one common question that people ask about default methods when they hear about the new feature for the first time: "*What if the class implements two interfaces and both those interfaces define a default method with the same signature?*"

Let's use the previous example to illustrate this situation:

```
public interface A {
  default void foo(){
    System.out.println("Calling A.foo()");
  }
}

public interface B {
  default void foo(){
    System.out.println("Calling B.foo()");
  }
}

public class Clazz implements A, B {
}
```

This code fails to compile with the following result:

```
java: class Clazz inherits unrelated defaults for foo() from
types A and B
```

To fix that, in Clazz, we have to resolve it manually by overriding the conflicting method:

```
public class Clazz implements A, B {
    public void foo(){}
}
```

But what if we would like to call the default implementation of method **foo()** from **interface A** instead of implementing our own. It is possible to refer to **A#foo()** as follows:

```
public class Clazz implements A, B {
  public void foo(){
    A.super.foo();
  }
}
```

The real examples of the the default method implementations can be found in JDK8. Going back to the example of **forEach** method for collections, we can find its default implementation in **java.lang.Iterable interface**:

```
@FunctionalInterface
public interface Iterable {
  Iterator iterator();

  default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
      action.accept(t);
    }
  }
}
```

The **forEach** method takes java.util.function. Consumer functional interface type as a parameter which enables us to pass in a lambda or a method reference as follows:

```
List<?> list = ...
list.forEach(System.out::println);
```

Defenders 101

Let's take a look on how the default methods are actually invoked.

From the client code perspective, default methods are just ordinary virtual methods. Hence the name - virtual extension methods. So in case of the simple example with one class that implements an interface with a default method, the client code that invokes the default method will generate **invokeinterface** at the call site.

```
A clazz = new Clazz();
clazz.foo(); // invokeinterface foo()

Clazz clazz = new Clazz();
clazz.foo(); // invokevirtual foo()
```

In case of the default methods conflict resolution, when we override the default method and would like to delegate the invocation to one of the interfaces, the **invokespecial** is inferred as we would call the implementation specifically:

```
public class Clazz implements A, B {
  public void foo(){
    A.super.foo(); // invokespecial foo()
  }
}
```

Here's the javap output:

```
public void foo();
 Code:
  0: aload_0
  1: invokespecial #2    // InterfaceMethod A.foo:()V
  4: return
```

As you can see, invokespecial instruction is used to invoke the **interface method foo()**. This is also something new from the bytecode point of view as previously you would only invoke methods via **super** that points to a class (parent class), and not to an interface.

SUMMARY

Default methods are an interesting addition to the Java language. You can think of them as a bridge between lambdas and JDK libraries. The primary goal of default methods is to enable an evolution of standard JDK interfaces and provide a smooth experience when we finally start using lambdas in Java 8.

PART III
BULK DATA OPERATIONS FOR JAVA COLLECTIONS

The goal of bulk data operations is to provide new features for data processing utilizing lambda functions including parallel operations. The parallel implementation is the central element of this feature. It builds upon the java.util.concurrency Fork/Join implementation introduced in Java 7.

Bulk operations - what's in it?

As the original change spec says, the purpose of bulk operations is to:

Add functionality to the Java Collections Framework for bulk operations upon data. This is commonly referenced as "filter/map/reduce for Java." The bulk data operations include both serial (on the calling thread) and parallel (using many threads) versions of the operations. Operations upon data are generally expressed as lambda functions.

With the addition of lambdas to Java language and the new API for collections, we will be able to leverage parallel features of the underlying platform in a much more efficient way.

Stream API

The new java.util.stream package has been added to JDK which allows us to perform filter/map/reduce-like operations with the collections in Java 8.

The Stream API would allow us to declare either sequential or parallel operations over the stream of data:

```
List persons = ..

// sequential version
Stream stream = persons.stream();

//parallel version
Stream parallelStream = persons.parallelStream();
```

A stream is something like an iterator. However, a stream can only be traversed once, then it's used up. Streams may also be infinite, which basically means that streams are "lazy" - we never know in advance, how many elements we will have to process.

The java.util.stream.Stream interface serves as a gateway to the bulk data operations. After the reference to a stream instance is acquired, we can perform the interesting tasks with the collections.

One important thing to notice about Stream API is that the source data is not mutated during the operations. This is due to the fact that the source of the data might not exist as such, or the initial data might be required somewhere else in the application code.

Stream sources

Streams can use different sources to consume data and the standard JDK API is extended with the new methods to make the experience more pleasant.

First source of data for streams is, of course, collections:

```
List list;
Stream stream = list.stream();
```

An other interesting source of data are so-called generators:

```
Random random = new Random();
Stream randomNumbers = Stream.generate(random::nextInt);
```

There's a number of utility methods to help defining the ranges of data:

```
IntStream range = IntStream.range(0, 50, 10);
range.forEach(System.out::println); // 0, 10, 20, 30, 40
```

Also, some of the existing classes in the standard library. For instance, Random class has been extended with some useful methods:

```
new Random()
  .ints()              // generate a stream of random integers
  .limit(10)           // we only need 10 random numbers
  .forEach(System.out::println);
```

Intermediate operations

Intermediate operations are used to described the transformations that should be performed over the stream of data. The filter(..) and map(..) methods in the good examples of intermediate operations. The return type of these methods is Stream, so that it would allow chaining of more operations. Here's a list of some useful intermediate operations:

filter excludes all elements that don't match a Predicate.

map perform transformation of elements using a Function.

flatMap transforms each element into zero or more elements by way of another Stream.

peek perform some action on each element as it is encountered.

distinct excludes all duplicate elements according to their **equals(..)behavior.**

sorted ensures that stream elements in subsequent operations are \ encountered according to the order imposed by a Comparator.

limit ensures that subsequent operations only see up to a maximum number of elements.
substream ensure that subsequent operations only see a range (by index) of elements.

Some of the operations, like sorted, distinct and limit are stateful, meaning the resulting stream of these operations depend on the values that the operation processed previously. As the Javadoc says, all intermediate operations are lazy. Let's take a look at some of the operations in more details.

FILTER

Filtering a stream of data is the first natural operation that we would need. Stream interface exposes a filter(..) method that takes in a Predicate SAM that allows us to use lambda expression to define the filtering criteria:

```
List persons = ...
Stream personsOver18 = persons.stream().filter(p -> p.getAge() > 18);
```

MAP

Assume we now have a filtered data that we can use for the real operations, say transforming the objects. The map operations allows us to apply a function, that takes in a parameter of one type, and returns something else. First, let's see how it would have been described in the good 'ol way, using an anonymous inner class:

```
Stream students = persons.stream()
    .filter(p -> p.getAge() > 18)
    .map(new Function<Person, Student>() {
          @Override
          public Student apply(Person person) {
            return new Student(person);
          }
         });
```

Now, converting this example into a lambda syntax we get the following:

```
Stream map = persons.stream()
    .filter(p -> p.getAge() > 18)
    .map(person -> new Student(person));
```

And since the lambda that is passed to the map(..) method just consumes the parameter without doing anything else with it, then we can transform it further to a method reference:

Terminating operations

Usually, dealing with a stream will involve these steps:

1. Obtain a stream from some source.
2. Perform one or more intermediate operations, like filter, map, etc.
3. Perform one terminal operation.

A terminal operation must be the final operation invoked on a stream. Once a terminal operation is invoked, the stream is "consumed" and is no longer usable.

There are several types of terminal operations available:

reducers like reduce(..), count(..), findAny(..), findFirst(..) terminate stream processing. Depending on the intention, the terminal operation can be a short-circuiting one. For instance, findFirst(..) will terminate the stream processing as soon as it encounters a matching element.

collectors, as the name implies, are for collecting the processed elements into a resulting collection.

forEach performs some action for each element in the stream.

iterators are the good 'ol way to work with collections if none of the options above satisfies our needs.

The most interesting terminal operation type are the so-called "collectors".

COLLECTORS

While stream abstraction is continuous by its nature, we can describe the operations on streams but to acquire the final results we have to collect the data somehow. The Stream API provides a number of so-called "terminal" operations. The collect() method is one of the terminal operations that allow us to collect the results:

```
List students = persons.stream()
    .filter(p -> p.getAge() > 18)
    .map(Student::new)
    .collect(new Collector<Student, List>() { ... });
```

Fortunately, in most cases you wouldn't need to implement the Collector interfaces yourself. Instead, there's a Collectors utility class for convenience:

```
List students = persons.stream()
    .filter(p -> p.getAge() > 18)
    .map(Student::new)
    .collect(Collectors.toList());
```

Or in case if we would like to use a specific collection implementation for collecting the results:

```
List students = persons.stream()
    .filter(p -> p.getAge() > 18)
    .map(Student::new)
    .collect(Collectors.toCollection(ArrayList::new));
```

Parallel & sequential

One interesting feature of the new Stream API is that it doesn't require to operations to be either parallel or sequential from beginning till the end. It is possible to start consuming the data concurrently, then switch to sequential processing and back at any point in the flow:

```
List students = persons.stream()
    .parallel()
    .filter(p -> p.getAge() > 18)  // filtering will be performed concurrently
    .sequential()
    .map(Student::new)
    .collect(Collectors.toCollection(ArrayList::new));
```

The cool part here is that the concurrent part of data processing flow will manage itself automatically, without requiring us to deal with the concurrency issues.

TOO LONG, DIDN'T READ (TL;DR)

SUMMARY, CONCLUSION, FAREWELL AND A COMIC

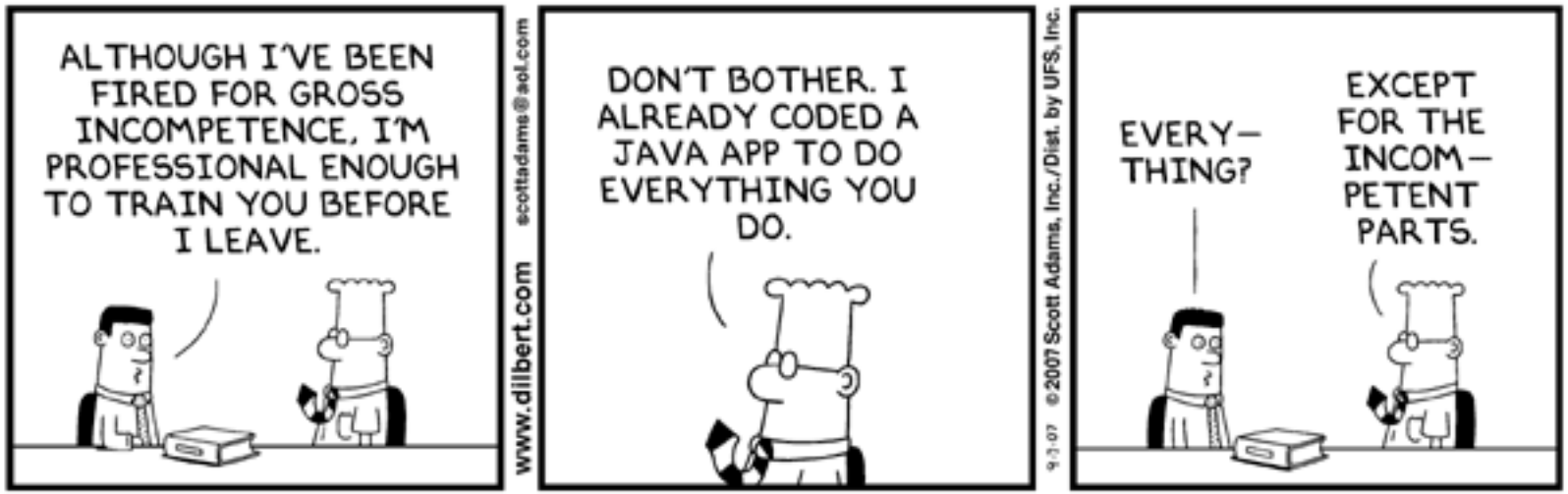
Summary of Findings and a Goodbye Comic ;-)

Here are the areas that we have covered that you can expect to see in Java 8:

1. Lambda expressions
2. Default methods
3. Bulk data operations for Java collections

As we have seen, **lambdas** are greatly improve readability of the code and making Java the language more expressive, especially when we move on to new Stream API. **Default methods,** in turn, are essential for API evolution, connecting the Collections API with lambdas for our convenience. Nevertheless, the ultimate goal of all this new features is to introduce parallel libraries and to seamlessly take advantage of the multi-core hardware.

I'm gonna make this a really short conclusion here as well, which basically comes down to this: the JVM itself is a great piece of engineering, whether you want to admit it or not, the Java platform is still alive and kicking. These new changes will allow us to leverage the platform and the language in a much more efficient way, and will ideally give the critics of Java a bit more substance to chew on.



© Scott Adams, Inc./Dist. by UFS, Inc.

Rebel Labs is the research & content division of ZeroTurnaround

Contact Us

Twitter: @Rebel_Labs
Web: http://zeroturnaround.com/rebellabs
Email: labs@zeroturnaround.com

Estonia
Ülikooli 2, 5th floor
Tartu, Estonia, 51003
Phone: +372 740 4533

USA
545 Boylston St., 4th flr.
Boston, MA, USA, 02116
Phone: 1(857)277-1199

Czech Republic
Osadní 35 - Building B
Prague, Czech Republic 170 00
Phone: +372 740 4533