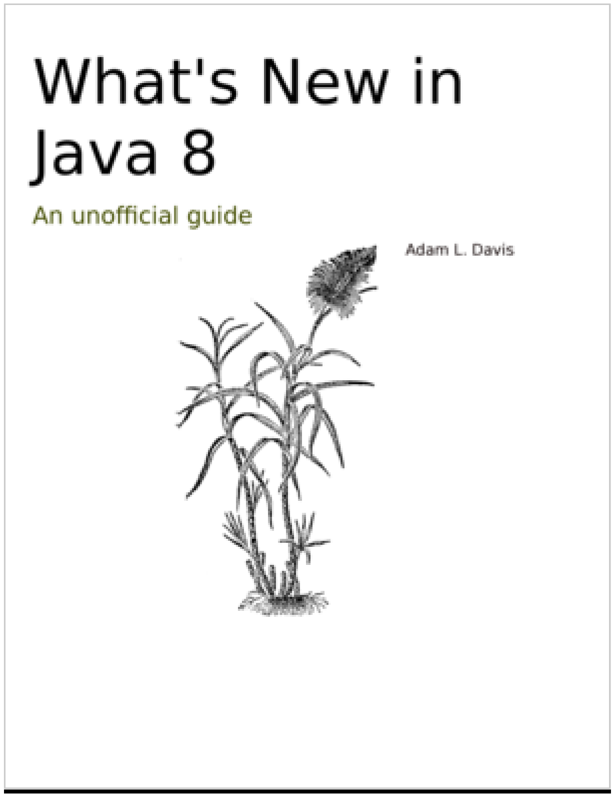
Sign In Sign Up

What's New in Java 8

An unofficial guide to Java and JDK 1.8

Adam L. Davis



$5.95 MINIMUM PRICE

$11.99 SUGGESTED PRICE

Add Ebook to Cart

What's New in Java 8

Table of Contents

Preface

Like many Java developers, the first time I heard about lambda expressions it piqued my interest. Also like many others, I was disappointed when it was set back. However, it is better late than never.

Java 8 is a giant step forward for the Java language. Writing this book has forced me to learn a lot more about it. In Project Lambda, Java gets a new closure syntax, method-references, and default methods on interfaces. It manages to add many of the features of functional languages without losing the clarity and simplicity Java developers have come to expect.

Aside from Project Lambda, Java 8 also gets a new Date and Time API (JSR 310), the Nashorn JavaScript engine, and removes the Permanent Generation from the HotSpot virtual machine, among other changes.

I would like to acknowledge the following people for providing valuable resources:

- Brian Goetz – "State of the Lambda"
- Aleksey Shipilev – jdk8-lambda-samples
- Richard Warburton – "Java 8 Lambdas"
- Julien Ponge – "Oracle Nashorn" in the Jan./Feb. 2014 issue of Java Magazine.
- Venkat Subramaniam – agiledeveloper.com
- All of the developers behind Java 8.
- The developers of Guava, joda-time, Groovy, and Scala.

1. Overview

This book is a short introduction to Java 8. After reading it, you should have a basic understanding of the new features and be ready to start using it.

This book assumes that you have a good understanding of Java the language and the JVM. If you're not familiar with the language, including features of Java 7, it might be hard to follow some of the examples.

Java 8 includes the following:

- Lambda expressions
- Method references
- Default Methods (Defender methods)
- A new Stream API.
- Optional
- A new Date/Time API.
- Nashorn, the new JavaScript engine
- Removal of the Permanent Generation
- and more…

The best way to read this book is with a Java 8 supporting IDE running so you can try out the new features.

Code examples can be found on github.

2. Lambda Expressions

The biggest new feature of Java 8 is language level support for lambda expressions (Project Lambda). A lambda expression is like syntactic sugar for an anonymous class[1] with one method whose type is inferred. However, it will have enormous implications for simplifying development.

2.1 Syntax

The main syntax of a lambda expression is "parameters -> body". The compiler can usually use the context of the lambda expression to determine the functional interface[2] being used and the types of the parameters. There are four important rules to the syntax:

- Declaring the types of the parameters is optional.
- Using parentheses around the parameter is optional if you have only one parameter.
- Using curly braces is optional (unless you need multiple statements).
- The “return” keyword is optional if you have a single expression that returns a value.

Here are some examples of the syntax:

```
() -> System.out.println(this)
(String str) -> System.out.println(str)
str -> System.out.println(str)
(String s1, String s2) -> { return s2.length() - s1.length(); }
(s1, s2) -> s2.length() - s1.length()
```

The last expression could be used to sort a list; for example:

```
Arrays.sort(strArray,
  (String s1, String s2) -> s2.length() - s1.length());
```

In this case the lambda expression implements the `Comparator` interface to sort strings by length.

2.2 Scope

Here’s a short example of using lambdas with the Runnable interface:

```
import static java.lang.System.out;

public class Hello {
     Runnable r1 = () -> out.println(this);
     Runnable r2 = () -> out.println(toString());

     public String toString() { return "Hello, world!"; }

     public static void main(String... args) {
             new Hello().r1.run(); //Hello, world!
             new Hello().r2.run(); //Hello, world!
     }
}
```

The important thing to note is both the r1 and r2 lambdas call the `toString()`

method of the Hello class. This demonstrates the scope available to the lambda.

You can also refer to final variables or effectively final variables. A variable is effectively final if it is only assigned once.

For example, using Spring's HibernateTemplate:

```
String sql = "delete * from User";
getHibernateTemplate().execute(session ->
    session.createSQLQuery(sql).uniqueResult());
```

In the above, you can refer to the variable sql because it is only assigned once. If you were to assign to it a second time, it would cause a compilation error.

2.3 Method references

Since a lambda expression is like an object-less method, wouldn't be nice if we could refer to existing methods instead of using a lamda expression? This is exactly what we can do with method references.

For example, imagine you frequently need to filter a list of Files based on file types. Assume you have the following set of methods for determining a file's type:

```
public class FileFilters {
      public static boolean fileIsPdf(File file) {/*code*/}
      public static boolean fileIsTxt(File file) {/*code*/}
      public static boolean fileIsRtf(File file) {/*code*/}
}
```

Whenever you want to filter a list of files, you can use a method reference as in the following example (assuming you already defined a method getFiles() that returns a Stream):

```
Stream<File> pdfs = getFiles().filter(FileFilters::fileIsPdf);
Stream<File> txts = getFiles().filter(FileFilters::fileIsTxt);
Stream<File> rtfs = getFiles().filter(FileFilters::fileIsRtf);
```

Method references can point to:

- Static methods.
- Instance methods.
- Methods on particular instances.
- Constructors (ie. TreeSet::new)

For example, using the new java.nio.file.Files.lines method:

```
Files.lines(Paths.get("Nio.java"))
            .map(String::trim)
            .forEach(System.out::println);
```

The above reads the file “Nio.java”, calls trim() on every line, and then prints out the lines.

Notice that System.out::println refers to the println method on an instance of PrintStream.

2.4 Functional Interfaces

In Java 8 a functional interface is defined as an interface with exactly one abstract method. This even applies to interfaces that were created with previous versions of Java.

Java 8 comes with several new functional interfaces in the package, java.util.function.

- Function<T,R> - takes an object of type T and returns R.
- Supplier<T> - just returns an object of type T.
- Predicate<T> - returns a boolean value based on input of type T.
- Consumer<T> - performs an action with given object of type T.
- BiFunction - like Function but with two parameters.
- BiConsumer - like Consumer but with two parameters.

It also comes with several corresponding interfaces for primitive types, such as:

- IntConsumer
- IntFunction<R>

- IntPredicate
- IntSupplier

See the java.util.function Javadocs for more information.

The coolest thing about functional interfaces is that they can be assigned to anything that would fulfill their contract. Take the following code for example:

```
Function<String, String> atr = (name) -> {return "@" + name;};
Function<String, Integer> leng = (name) -> name.length();
Function<String, Integer> leng2 = String::length;
```

This code is perfectly valid Java 8. The first line defines a function that prepends “@” to a String. The last two lines define functions that do the same thing: get the length of a String.

The Java compiler is smart enough to convert the method reference to String’s length() method into a Function (a functional interface) whose apply method takes a String and returns an Integer. For example:

```
for (String s : args) out.println(leng2.apply(s));
```

This would print out the lengths of the given strings.

Any interface can be functional interface, not merely those that come with Java. To declare your intention that an interface is functional, use the @FunctionalInterface annotation. Although not necessary, it will cause a compilation error if your interface does not satisfy the requirements (ie. one abstract method).

Github

See jdk8-lambda-samples for more examples.

2.5 Comparisons to Java 7

To better illustrate the benefit of Lambda-expressions, here are some examples of how code from Java 7 can be shortened in Java 8.

Creating an ActionListener

```java
// Java 7
ActionListener al = new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println(e.getActionCommand());
    }
};
// Java 8
ActionListener al8 = e -> System.out.println(e.getActionCommand());
```

Printing out a list of Strings

```java
// Java 7
for (String s : list) {
    System.out.println(s);
}
//Java 8
list.forEach(System.out::println);
```

Sorting a list of Strings

```java
// Java 7
Collections.sort(list, new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
});
//Java 8
Collections.sort(list, (s1, s2) -> s1.length() - s2.length());
// or
list.sort(Comparator.comparingInt(String::length));
```

Sorting

For the sorting examples, assume you have the following `Person` class:

```java
public static class Person {

    String firstName;
    String lastName;

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }
}
```

Here’s how you might sort this list in Java 7 by last-name and then first-name:

```java
Collections.sort(list, new Comparator<Person>() {
    @Override
    public int compare(Person p1, Person p2) {
        int n = p1.getLastName().compareTo(p2.getLastName());
        if (n == 0) {
            return p1.getFirstName().compareTo(p2.getFirstName());
        }
        return n;
    }
});
```

In Java 8, this can be shortened to the following:

```java
list.sort(Comparator.comparing(Person::getLastName)
        .thenComparing(Person::getFirstName));
```

This example uses a static method on an interface (`comparing`) and a default method (`thenComparing`) which are discussed in the next chapter.

---

1. A lambda expression is not an anonymous class; it actually uses `invokedynamic` in the byte-code.↩
2. We will explain what “functional interface” means in a later section.↩

3. Default Methods

In order to add the `stream` method (or any others) to the core Collections API, Java needed another new feature, Default methods (also known as Defender Methods or Virtual Extension methods). This way they could add new methods to the `List` interface for example without breaking all the existing implementations (backwards compatibility).

Default methods can be added to any interface. Like the name implies, any class that implements the interface but does not override the method will get the default implementation.

For example, the stream method in the Collection interface is defined something like the following:

```java
default public Stream stream() {
	return StreamSupport.stream(spliterator());
}
```

See the Java docs for more on Spliterators.

You can always override a default method if you need different behavior.

3.1 Default and Functional

An interface can have one or more default methods and still be functional.

For example, take a look at the Iterable interface:

```java
@FunctionalInterface
public interface Iterable {
	Iterator iterator();
	default void forEach(Consumer<? super T> action) {
		Objects.requireNonNull(action);
		for (T t : this) {
			action.accept(t);
		}
	}
}
```

It has both the iterator() method and the forEach method.

3.2 Multiple Defaults

In the unlikely case that your class implements two or more interfaces that define the same default method, Java will throw a compilation error. You will need to override the method and choose from one of the methods. For example:

```
interface Foo {
     default void talk() {
             out.println("Foo!");
     }
}
interface Bar {
     default void talk() {
             out.println("Bar!");
     }
}
class FooBar implements Foo, Bar {
     @Override
     void talk() { Foo.super.talk(); }
}
```

In the above code, talk is overridden and calls Foo's talk method. This is similar to the way you refer to a super class in pre-Java-8.

3.3 Static Methods on Interface

Although not strictly related to default methods, the ability to add static methods to interfaces is a similar change to the Java language.

For example, there are many static methods on the new Stream interface. This makes "helper" methods easier to find since they can be located directly on the interface, instead of a different class such as StreamUtil or Streams.

Here's an example in the new Stream interface:

```
public static<T> Stream<T> of(T... values) {
    return Arrays.stream(values);
}
```

The above method creates a new stream based on the given values.

4. Streams

The Stream interface is such a fundamental part of Java 8 it deserves its own chapter.

4.1 What is a Stream?

The Stream interface is located in the java.util.stream package. It represents a sequence of objects somewhat like the Iterator interface. However, unlike the Iterator, it supports parallel execution.

The Stream interface supports the map/filter/reduce pattern and executes lazily, forming the basis (along with lambdas) for functional-style programming in Java 8.

There are also corresponding primitive streams (IntStream, DoubleStream, and LongStream) for performance reasons.

4.2 Generating Streams

There are many ways to create a Stream in Java 8. Many of the existing Java core library classes have Stream returning methods in Java 8.

Streaming Collections

The most obvious way to create a stream is from a `Collection`.

The Collection interface has two default methods on it for creating streams:

- stream(): Returns a sequential Stream with the collection as its source.
- parallelStream(): Returns a possibly parallel Stream with the collection as its source.

The ordering of the Stream relies on the underlying collection just like an Iterator.

Streaming Files

The `BufferedReader` now has the `lines()` method which returns a Stream; for example[1]:

```
try (FileReader fr = new FileReader("file");
    BufferedReader br = new BufferedReader(fr)) {
    br.lines().forEach(System.out::println);
}
```

You can also read a file as a Stream using `Files.lines(Path filePath)`; for example:

```
try (Stream st = Files.lines(Paths.get("file"))) {
    st.forEach(System.out::println);
}
```

Note this populates lazily; it does not read the entire file when you call it.

⚠ Files.lines(Path): Any IOException that is thrown while processing the file (after the file is opened) will get wrapped in an UncheckedIOException and thrown.

Streaming File Trees

There are several static methods on the Files class for navigating file trees using a Stream.

- list(Path dir) – Stream of files in the given directory.
- walk(Path dir)[2] – Stream that traverses the file tree depth-first starting at the given directory.
- walk(Path dir, int maxDepth) – Same as walk(dir) but with a maximum depth.

Streaming Text Patterns

The Pattern class now has a method, splitAsStream(CharSequence), which creates a Stream.

For example:

```
import java.util.regex.Pattern;
// later on...
Pattern patt = Pattern.compile(",");
patt.splitAsStream("a,b,c")
    .forEach(System.out::println);
```

The above uses a very simple pattern, a comma, and splits the text into a stream and prints it out. This would produce the following output:

```
a
b
c
```

Infinite Streams

Using the generate or iterate static methods on Stream, you can create a Stream of values including never ending streams. For example, you could call generate in the following way to create an infinite supply of objects:

```
Stream.generate(() -> new Dragon());
```

For example, you could use this technique to produce a stream of CPU load or memory usage. However, you should use this with caution. It is similar to an infinite loop.

You could also use generate to create an infinite random number supply; for example:

```
Stream.generate(() -> Math.random());
```

However, the java.util.Random class does this for you with the following new methods: ints(), longs(), and doubles(). Each of those methods is overloaded with definitions similar to the following:

- ints(): An infinite Stream of random integers.
- ints(int n, int m): An infinite Stream of random integers from n (inclusive) to m (exclusive).
- ints(long size): A Stream of given size of random integers.
- ints(long size, int n, int m): A Stream of given size of random integers with given bounds.

The iterate method is similar to generate except it takes an initial value and a Function that modifies that value. For example, you can iterate over the Integers using the following code:

```
Stream.iterate(1, i -> i+1)
    .forEach(System.out::print);
```

This would print out "1234…" continuously until you stop the program.

There are ways to limit an infinite stream which we will cover later (filter and limit).

Ranges

There are also new methods for creating ranges of numbers as Streams.

For example, the static method, range, on the IntStream interface:

```
IntStream.range(1, 11)
    .forEach(System.out::println);
```

The above would print out the numbers one through ten.

Each primitive Stream (IntStream, DoubleStream, and LongStream) has a corresponding range method.

Streaming Anything

You can create a Stream from any number of elements or an array using the two following methods:

```
Stream<Integer> s = Stream.of(1, 2, 3);
Stream<Object> s2 = Arrays.stream(array);
```

Stream.of can take any number of parameters of any type.

4.3 For Each

The most basic thing you can do with a Stream is loop through it using the forEach method.

For example, to print out all of the files in the current directory, you could do the following:

```
Files.list(Paths.get("."))
```

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

For the most part, this replaces the “for loop”. It is more concise, and more object-oriented since you are delegating the implementation of the actual loop.

4.4 Map/Filter/Reduce

Lambda expressions and default methods allow us to implement map/filter/reduce in Java 8. Actually it is already implemented for us in the standard library.

For example, imagine you want to get the current point scores from a list of player-names and find the player with the most points. You have a simple class, PlayerPoints, and a getPoints method defined as the following:

```
public static class PlayerPoints {
 public final String name;
 public final long points;

 public PlayerPoints(String name, long points) {
   this.name = name;
   this.points = points;
 }

 public String toString() {
   return name + ":" + points;
 }
}

public static long getPoints(final String name) {
     // gets the Points for the Player
}
```

Finding the highest player could be done very simply in Java 8 as shown in the following code:

```
PlayerPoints highestPlayer =
  names.stream().map(name -> new PlayerPoints(name, getPoints(name)))
      .reduce(new PlayerPoints("", 0.0),
                      (s1, s2) -> (s1.points > s2.points) ? s1 : s2)
```

This could also be done in Java 7 with the dollar library (or similarly with Guava or Functional-Java), but it would be much more verbose as shown in the following:

```
PlayerPoints highestPlayer =
  $(names).map(new Function<String, PlayerPoints>() {
             public PlayerPoints call(String name) {
                     return new PlayerPoints(name, getPoints(name))
             }
     })
     .reduce(new PlayerPoints("", 0.0),
     new BiFunction<PlayerPoints, PlayerPoints, PlayerPoints>() {
             public PlayerPoints call(PlayerPoints s1, PlayerPoints
                     return (s1.points > s2.points) ? s1 : s2;
             }
     });
```

The major benefit to coding this way (apart from the reduction in lines of code) is the ability to hide the underlying implementation of map/reduce. For example, it's possible that map and reduce are implemented concurrently, allowing you to easily take advantage of multiple processors. We'll describe one way to do this (ParallelArray) in the following section.

4.5 Parallel Array

The ParallelArray was part of JSR-166, but ended up being excluded from the standard Java lib. It does exist and was released to the public domain (you can download it from the JSR website).

Although it was already out there, it really wasn't easy to use until closures were included in the Java language. In Java 7 using the ParallelArray looks like the following:

```
// with this class
public class Student {
    String name;
    int graduationYear;
    double gpa;
}
// this predicate
final Ops.Predicate<Student> isSenior =
     new Ops.Predicate<>() {
             public boolean op(Student s) {
                     return s.graduationYear == Student.THIS_YEAR;
             }
     };
// and this conversion operation
final Ops.ObjectToDouble<Student> selectGpa =
     new Ops.ObjectToDouble<>() {
             public double op(Student student) {
                     return student.gpa;
```

```
                }
    };
// create a fork-join-pool
ForkJoinPool fjPool = new ForkJoinPool();
ParallelArray<Student> students = new ParallelArray<>(fjPool, data)
// find the best GPA:
double bestGpa = students.withFilter(isSenior)
                         .withMapping(selectGpa)
                         .max();
```

In Java 8, you can do the following:

```
// create a fork-join-pool
ForkJoinPool pool = new ForkJoinPool();
ParallelArray<Student> students = new ParallelArray<>(pool,data);
// find the best GPA:
double bestGpa = students
    .withFilter((Student s) -> (s.graduationYear == THIS_YEAR))
    .withMapping((Student s) -> s.gpa)
    .max();
```

However, Java 8's addition of stream() and parallelStream() make this even easier:

```
double bestGpa = students
    .parallelStream()
    .filter(s -> (s.graduationYear == THIS_YEAR))
    .mapToDouble(s -> s.gpa)
    .max().getAsDouble();
```

This makes it extremely simple to switch between a sequential implementation and a concurrent one.

Groovy GPars

You can do something similar to this right now if you use Groovy with the GPars library in the following way:

```
GParsPool.withPool {
   // a map-reduce functional style (students is a Collect
   def bestGpa = students.parallel
       .filter{ s -> s.graduationYear == Student.THIS_YEAR
       .map{ s -> s.gpa }
       .max()
}
```

The static method GParsPool.withPool takes in a closure and augments any Collection with several methods (using Groovy's Category mechanism). The parallel method actually creates a ParallelArray (JSR-166) from the given Collection and uses it with a thin wrapper around it.

4.6 Peek

You can peek into a stream to do some action without interrupting the stream.

For example you could print out elements to debug code:

```
Files.list(Paths.get("."))
    .map(Path::getFileName)
    .peek(System.out::println)
    .forEach(p -> doSomething(p));
```

You can use any action you want, but you should not try to modify elements; you should use map instead.

4.7 Limit

The limit(int n) method can be used to limit a stream to the given number of elements. For example:

```
Random rnd = new Random();
rnd.ints().limit(10)
    .forEach(System.out::println);
```

The above would print out ten random integers.

4.8 Sort

Stream also has the sorted() method for sorting a stream. Like all intermediate methods on Stream (such as map, filter, and peek), the sorted() method executes lazily. Nothing happens until a terminating operation (such as reduce or forEach) is called. However, you should call a limiting operation like limit before calling sorted() on an infinite stream.

For example, the following would throw a runtime exception (using build 1.8.0-b132):

```
rnd.ints().sorted().limit(10)
    .forEach(System.out::println);
```

However, the following code works just fine:

```
rnd.ints().limit(10).sorted()
    .forEach(System.out::println);
```

Also, you should call `sorted()` after any calls to `filter`. For example, this code prints out the first five Java file-names in the current directory:

```
Files.list(Paths.get("."))
    .map(Path::getFileName) // still a path
    .map(Path::toString) // convert to Strings
    .filter(name -> name.endsWith(".java"))
    .sorted() // sort them alphabetically
    .limit(5) // first 5
    .forEach(System.out::println);
```

The code above does the following:

- Lists the files in the current directory.
- Maps those files to file names.
- Finds names that end with ".java".
- Takes only the first five (sorted alphabetically).
- Prints them out.

4.9 Collectors and Statistics

Since Streams are lazily evaluated and support parallel execution, you need a special way to combine results; this is called a Collector.

A Collector represents a way to combine the elements of a Stream into one result. It consists of three things:

- A supplier of an initial value.
- An accumulator which adds to the initial value.
- A combiner which combines two results into one.

There are two ways to do this: collect(supplier,accumulator,combiner), or collect(Collector) (types left off for brevity).

Luckily, Java 8 comes with several Collectors built in. Import them the following way:

```java
import static java.util.stream.Collectors.*;
```

Simple Collectors

The simplest collectors are things like toList() and toCollection():

```java
// Accumulate names into a List
List<String> list = dragons.stream()
        .map(Dragon::getName)
        .collect(toList());

// Accumulate names into a TreeSet
Set<String> set = dragons.stream()
        .map(Dragon::getName)
        .collect(toCollection(TreeSet::new));
```

Joining

If you're familiar with Apache Commons' StringUtil.join, the joining collector is similar to it. It combines the stream using a given delimiter. For example:

```java
String names = dragons.stream()
        .map(Dragon::getName)
        .collect(joining(","));
```

This would combine all of the names into one String separated by commas.

Statistics

More complex collectors resolve to a single value. For example, you can use an "averaging" Collector to get the average; for example:

```java
System.out.println("\n----->Average line length:");
System.out.println(
    Files.lines(Paths.get("Nio.java"))
        .map(String::trim)
        .filter(s -> !s.isEmpty())
        .collect(averagingInt(String::length))
        );
```

The above code calculates the average length of non-empty lines in the file “Nio.java”.

Sometimes you want to collect multiple statistics about a collection. Because Streams are consumed when you call collect, you need to calculate all of your statistics at once. This is where SummaryStatistics comes in. First import the one you want to use:

```
import java.util.IntSummaryStatistics;
```

Then use the summarizingInt collector; for example:

```
IntSummaryStatistics stats = Files.lines(Paths.get("Nio.java"))
        .map(String::trim)
        .filter(s -> !s.isEmpty())
        .collect(summarizingInt(String::length));

System.out.println(stats.getAverage());
System.out.println("count=" + stats.getCount());
System.out.println("max=" + stats.getMax());
System.out.println("min=" + stats.getMin());
```

The above code performs the same average as before, but also computes the maximum, minimum, and count of the elements.

There’s also summarizingLong and summarizingDouble.

Equivalently, you can map your stream to a primitive type and then call summaryStatistics(). For example:

```
IntSummaryStatistics stats = Files.lines(Paths.get("Nio.java"))
    .map(String::trim)
    .filter(s -> !s.isEmpty())
    .mapToInt(String::length)
    .summaryStatistics();
```

4.10 Grouping and Partitioning

The groupingBy collector groups elements based on a function you provide.

For example:

```
// Group by first letter of name
List<Dragon> dragons = getDragons();
Map<Character,List<Dragon>> map = dragons.stream()
        .collect(groupingBy(dragon -> dragon.getName().charAt(0)));
```

Similarly, the partitioningBy method creates a map with a boolean key. For example:

```
// Group by whether or not the dragon is green
Map<Boolean,List<Dragon>> map = dragons.stream()
        .collect(partitioningBy(Dragon::isGreen));
```

Parallel Grouping

To execute grouping in parallel (if you don't care about ordering) you should use the groupingByConcurrent method. The underlying stream should be unordered to allow grouping to occur in parallel; for example:

dragons.parallelStream().unordered().collect(groupingByConcurrent(Dragon::getColor));.

4.11 Comparisons to Java 7

To better illustrate the benefit of Streams in Java 8, here are some examples of code from Java 7 compared to their new versions.

Finding a maximum

```
// Java 7
double max = 0;

for (Double d : list) {
    if (d > max) {
        max = d;
    }
}
//Java 8
max = list.stream().reduce(0.0, Math::max);
// or
max = list.stream().mapToDouble(Number::doubleValue).max().getAsDoul
```

Calculating an average

```
double total = 0;
double ave = 0;
// Java 7
for (Double d : list) {
    total += d;
}
ave = total / ((double) list.size());
//Java 8
ave = list.stream().mapToDouble(Number::doubleValue).average().getAsD
```

Printing the numbers one through ten

```
// Java 7
for (int i = 1; i < 11; i++) {
    System.out.println(i);
}
// Java 8
IntStream.range(1, 11)
    .forEach(System.out::println);
//or
Stream.iterate(1, i -> i+1).limit(10)
    .forEach(System.out::println);
```

Joining Strings

```
// Java 7 using commons-util
List<String> names = new LinkedList<>();
for (Dragon dragon : dragons)
    names.add(dragon.getName());
String names = StringUtils.join(names, ",");
// Java 8
String names = dragons.stream()
    .map(Dragon::getName)
    .collect(Collectors.joining(","));
```

1. Of course you should add a catch statement to this for error handling.↩
2. The actual method signature is walk(Path start, FileVisitOption... options) but you will probably just use walk(Path).↩

5. Optional

Java 8 comes with the Optional class in the java.util package for avoiding null return values (and thus NullPointerException). It is very similar to Google Guava's Optional, which is similar to Nat Pryce's Maybe class and Scala's Option class.

The Billion Dollar Mistake

Tony Hoare, the inventor of null, has gone on record calling it his “billion-dollar mistake”. Despite your opinion of null, many efforts have been made to make null-checks part of the compilation or automated-code-check process; for example, the @Nonnull annotation of JSR-305. `Optional` makes it very simple for API designers to avoid null.

You can use `Optional.of(x)` to wrap a non-null value, `Optional.empty()` to represent a missing value, or `Optional.ofNullable(x)` to create an Optional from a reference that may or may not be null.

After creating an instance of Optional, you then use `isPresent()` to determine if the there is a value and `get()` to get the value. Optional provides a few other helpful methods for dealing with missing values:

- `orElse(T)` – Returns the given default value if the Optional is empty.
- `orElseGet(Supplier<T>)` – Calls on the given Supplier to provide a value if the Optional is empty.
- `orElseThrow(Supplier<X extends Throwable>)` – Calls on the given Supplier for an exception to throw if the Optional is empty.

It also includes functional style (lambda friendly) methods, like the following:

- `filter(Predicate<? super T> predicate)` – Filters the value and returns a new Optional.
- `flatMap(Function<? super T,Optional<U>> mapper)` – Performs a mapping operation which returns an Optional.
- `ifPresent(Consumer<? super T> consumer)` – Executes the given Consumer only if there is a value present (no return value).
- `map(Function<? super T,? extends U> mapper)` – Uses the given mapping Function and returns a new Optional.

Stream Optional

The new `Stream` interface has multiple methods which return Optional (in case there are no values in the Stream):

- `reduce(BinaryOperator<T> accumulator)` – Reduces the stream to a single value.
- `max(Comparator<? super T> comparator)` – Finds the maximum value.
- `min(Comparator<? super T> comparator)` – Finds the minimum value.

6. Nashorn

Nashorn replaces Rhino as the default JavaScript engine for the Oracle JVM. Nashorn is much faster since it uses the `invokedynamic` feature of the JVM. It also includes a command line tool (`jjs`).

6.1 jjs

JDK 8 includes the command line tool `jjs` for running JavaScript.

You can run JavaScript files from the command line (assuming you have Java 8's bin in your `PATH`):

```
$ jjs script.js
```

This can be useful for running scripts; for example, let's say you wanted to quickly find the sum of some numbers:

```
var data = [1, 3, 5, 7, 11]
var sum = data.reduce(function(x, y) {return x + y}, 0)
print(sum)
```

Running the above code should print out `27`.

6.2 Scripting

Running jjs with the `-scripting` option starts up an interactive shell where you can type and evaluate JavaScript.

You can also embed variables into strings and have them evaluate; for example:

```
jjs> var date = new Date()
jjs> print("${date}")
```

This would print out the current date and time.

6.3 ScriptEngine

You can also run JavaScript dynamically from Java.

First, you need to import the ScriptEngine:

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
```

Second, you use the `ScriptEngineManager` to get the Nashorn engine:

```
ScriptEngineManager engineManager = new ScriptEngineManager();
ScriptEngine engine = engineManager.getEngineByName("nashorn");
```

Now you can evaluate javascript at any point:

```
engine.eval("function p(s) { print(s) }");
engine.eval("p('Hello Nashorn');");
```

The `eval` method can also take a `FileReader` as input:

```
engine.eval(new FileReader('library.js'));
```

This way you can include and run any JavaScript. However, keep in mind that the typical variables available to you in the browser (window, document, etc.) are not available.

6.4 Importing

You can import and use Java classes and packages using the JavaImporter.

For example, import java.util, the IO, and NIO file packages:

```
var imports = new JavaImporter(java.util, java.io, java.nio.file);
with (imports) {
        var paths = new LinkedList();
        print(paths instanceof LinkedList); //true
        paths.add(Paths.get("file1"));
        paths.add(Paths.get("file2"));
        paths.add(Paths.get("file3"));
        print(paths) // [file1, file2, file3]
}
```

The above demonstrates that paths is an instance of LinkedList and prints out the list.

Later on you could add the following code to write text into the files:

```
for (var i=0; i < paths.size(); i++)
      Files.newOutputStream(paths.get(i))
              .write("test\n".getBytes());
```

We can use existing Java classes, but we can also create new ones.

6.5 Extending

You can extend Java classes and interfaces using the Java.type and Java.extend functions. For example, you can extend the Callable interface and implement the call method:

```
var concurrent = new JavaImporter(java.util, java.util.concurrent);
var Callable = Java.type("java.util.concurrent.Callable");
with (concurrent) {
  var executor = Executors.newCachedThreadPool();
  var tasks = new LinkedHashSet();
  for (var i=0; i < 200; i++) {
    var MyTask = Java.extend(Callable, {call: function() {print("tas
    var task = new MyTask();
    tasks.add(task);
    executor.submit(task);
  }
}
```

6.6 Invocable

You can also invoke JavaScript functions directly from Java.

Firstly, you need to cast the engine to the Invocable interface:

```
Invocable inv = (Invocable) engine;
```

Then, to invoke any function, simple use the `invokeFunction` method, for example:

```
engine.eval("function p(s) { print(s) }");
inv.invokeFunction("p", "hello");
```

Lastly, you can use the `getInterface` method to implement any interface in JavaScript.

For example, if you have the following `JPrinter` interface, you can use it like so:

```
public static interface JPrinter {
    void p(String s);
}
// later on...
JPrinter printer = inv.getInterface(JPrinter.class);
printer.p("Hello again!");
```

7. New Date and Time API

Java 8 introduces a new Date/Time API that is thread-safe, easier to read, and more comprehensive than the previous API. Java's Calendar implementation has not changed much since it was first introduced and Joda-Time is widely regarded as a better replacement. Java 8's new Date/Time API is very similar to Joda-Time.

7.1 New Classes

The main difference you will notice is that there are several different classes to represent time, date, time period, and timezone specific data. Also there are transformers for dates and times.

For dates and times without a timezone, use the following:

- `LocalDate` – Day, month, year.

- LocalTime – Time of day only.
- LocalDateTime – Both date and time.

For timezone specific times you use ZonedDateTime.

Previous to Java 8, to calculate the time eight hours in the future you would need to write something like the following:

```
Calendar cal = Calendar.getInstance();
cal.add(Calendar.HOUR, 8);
cal.getTime(); // actually returns a Date
```

In Java 8, you can more simply write the following:

```
LocalTime now = LocalTime.now();
LocalTime later = now.plus(8, HOURS);
```

There are also well-named methods such as plusDays, plusMonths, minusDays, and minusMonths. For example:

```
LocalDate today = LocalDate.now();
LocalDate thirtyDaysFromNow = today.plusDays(30);
LocalDate nextMonth = today.plusMonths(1);
LocalDate aMonthAgo = today.minusMonths(1);
```

Note that each method returns a different instance of LocalDate. The original LocalDate, today, remains unchanged. This is because the new Date-Time types are immutable. This allows them to be thread-safe and cacheable.

7.2 Creation

Creating new date and time objects is much easier and less error-prone in Java 8. Every type is immutable and has static factory methods.

For example, creating a new LocalDate for March 15, 2014 is as simple as:

```
LocalDate date = LocalDate.of(2014, 3, 15);
```

For more type-safety, you can use the new Month enum:

```java
date = LocalDate.of(2014, Month.MARCH, 15);
```

You can also easily create a LocalDateTime by combining an instance of LocalDate with a LocalTime:

```java
LocalTime time = LocalTime.of(12, 15, 0);
LocalDateTime datetime = date.atTime(time);
```

You could also use any of the following methods (on LocalDate):

- atTime(int hour, int minute)
- atTime(int hour, int minute, int second)
- atTime(int hour, int minute, int second, int nanoOfSecond)

Every class also has the now() method, which corresponds to the instant (or date) it is called.

7.3 Enums

Java 8 adds several enums, such as java.time.temporal.ChronoUnit for expressing things like “days” and “hours” instead of the integer constants used in the Calendar API. For example:

```java
LocalDate today = LocalDate.now();
LocalDate nextWeek = today.plus(1, ChronoUnit.WEEKS);
LocalDate nextMonth = today.plus(1, ChronoUnit.MONTHS);
LocalDate nextYear = today.plus(1, ChronoUnit.YEARS);
LocalDate nextDecade = today.plus(1, ChronoUnit.DECADES);
```

There’s also the java.time.DayOfWeek and java.time.Month enums.

The month enum can be used to create LocalDates and is returned by LocalDate::getMonth. For example here is how you might create a LocalDate and print out the month.

```java
// import java.time.Month;
LocalDate date = LocalDate.of(2014, Month.MARCH, 27);
System.out.println(date.getMonth());
```

This would print out “MARCH”.

7.4 Clock

The `Clock` can be used in conjunction with dates and times to help build your tests. During production a normal Clock can be used, and a different one during tests.

To get the default clock, use the following:

```
Clock.systemDefaultZone();
```

The Clock can then be passed into factory methods; for example:

```
LocalTime time = LocalTime.now(clock);
```

7.5 Period and Duration

Java 8 has two types for representing time differences as humans understand them, Period and Duration.

Duration is a time-based amount of time, such as '34.5 seconds'. Period is a date-based amount of time, such as '2 years, 3 months and 4 days'.

Periods and Durations can be determined using the `between` method:

```
Period p = Period.between(date1, date2);
Duration d = Duration.between(time1, time2);
```

They can also be created using static methods. For example, Durations can be created for any amount of seconds, minutes, hours, or days:

```
Duration twoHours = Duration.ofHours(2);
Duration tenMinutes = Duration.ofMinutes(10);
Duration thirtySecs = Duration.ofSeconds(30);
```

Periods and Durations can be added or subtracted from Java 8 date types. For example:

```
LocalTime t2 = time.plus(twoHours);
```

7.6 Temporal Adjusters

A TemporalAdjuster can be used to do tricky date "math" that is popular in business applications. For example they can be used to find the "first Monday of the month" or "next Tuesday".

The java.time.temporal.TemporalAdjusters class contains a bunch of useful methods for creating TemporalAdjusters. Here are a few of them:

- firstDayOfMonth()
- firstDayOfNextMonth()
- firstInMonth(DayOfWeek)
- lastDayOfMont()
- next(DayOfWeek)
- nextOrSame(DayOfWeek)
- previous(DayOfWeek)
- previousOrSame(DayOfWeek)

To use a TemporalAdjuster use the with method. This method returns an adjusted copy of the date-time or date object. For example:

```
import static java.time.temporal.TemporalAdjusters.*;
//...
LocalDate nextTuesday = LocalDate.now().with(next(DayOfWeek.TUESDAY))
```

7.7 Instant

The Instant class represents a point in time measured to the nanosecond. It forms the basis of time measurements in the Java 8 date-time API.

Much like the old Date class, Instant measures time starting from the "epoch" (Jan. 1, 1970) and is time-zone ignorant.

7.8 Time Zones

Time-Zones are represented by the java.time.ZoneId class. There are two types of ZoneIds, fixed offsets and geographical regions. This is to compensate for things like "daylight saving time" which can be very complex.

You can get an instance of a ZoneId in many ways including the following two:

```
ZoneId mountainTime = ZoneId.of("America/Denver");
ZoneId myZone = ZoneId.systemDefault();
```

To print out all available IDs, use getAvailableZoneIds():

```
System.out.println(ZoneId.getAvailableZoneIds());
```

7.9 Backwards Compatibility

The original Date and Calendar objects have the toInstant() method to convert them to the new Date-Time API. You can then use an ofInstant(Insant,ZoneId) method to get a LocalDateTime or ZonedDateTime object; for example:

```
Date date = new Date();
Instant now = date.toInstant();
LocalDateTime dateTime = LocalDateTime.ofInstant(now, myZone);
ZonedDateTime zdt = ZonedDateTime.ofInstant(now, myZone);
```

8. No More Permanent Generation

> The proposed implementation will allocate class meta-data in native memory and move interned Strings and class statics to the Java heap. [http://openjdk.java.net/jeps/122]

Most allocations for the class metadata are now allocated out of native memory. This means that you won’t have to set the “XX:PermSize” options anymore (they don’t exist).

This also means that you will get a “java.lang.OutOfMemoryError: Metadata space” error message instead of “java.lang.OutOfMemoryError: Permgen space” when you run out of memory.

This is part of the convergence of the Oracle JRockit and HotSpot JVMs.

9. Miscellaneous

Java 8 has tons of new features that you might miss with all of the focus on lambdas. Here are some of them:

- java.util.Base64
- Cryptography upgrades (lots)
- JDBC 4.2
- Repeatable Annotations
- Annotations on types

For a more complete list, please see the official list.

9.1 Base64

Until now, Java developers have had to rely on third-party libraries for encoding and decoding Base-64. Since it is such a frequent operation, a large project will typically contain several different implementations of Base64. For example: Apache commons-codec, Spring, and Guava all have separate implementations.

For this reason, Java 8 has `java.util.Base64`. It acts like a factory for Base64 encoders and decoders and has the following methods:

- getEncoder()
- getDecoder()
- getUrlEncoder()
- getUrlDecoder()

Each factory method returns either an Encoder or Decoder.

The URL Base-64 Encoder provides an encoding that is URL and Filename safe (62 is - and 63 is _).

9.2 Annotations on Java Types

Prior to Java 8, annotations could be used on any declaration. In Java 8, annotations can also be applied to the use of types. Here are some examples:

```
// Class instance creation:
new @Interned RocketShip();

// Type cast:
notNullString = (@NonNull String) str;

// implements clause:
class ImmutableSet<T> implements
        @Readonly Set<@Readonly T> { ... }

// Thrown exception declaration:
void launchRocket() throws
     @Critical FireException { ... }
```

This new ability is primarily aimed at supporting type-checking frameworks, such as Checker. These frameworks help find errors in your code at compile time.

9.3 Repeating Annotations

Java 8 will allow annotations annotated with @Repeatable to be repeated.

For example, let's say you're coding a game and want to use annotations to schedule when methods should be called. You can declare multiple schedules using multiple annotations:

```
// the first of the month and every monday at 7am
```

For this to be possible, you need to have the following:

- The Schedule annotation needs to use the meta-annotation @Repeatable.
- There needs t as declared by the @Repeatable annotation.

Due to Java's emphasis on backwards-compatibility, repeating annotations are actually stored within another annotation (that you provide). The @Repeatable annotation takes in a value that is the class of the containing annotation. For example:

```
// Schedule.java
@Repeatable(Schedules.class)
public @interface Schedule {...}
```

```
// Schedules.java
public @interface Schedules {
    Schedule[] value;
}
```

Schedule is now a repeatable annotation.

You can use reflection to access repeatable annotations at runtime. To do this there is a new method called `getAnnotationsByType(Class annotationClass)` on `Class`, `Constructor`, `Method`, etc. It returns an array of all such annotations (or an empty array if there are none).

10. Functional Programming in Java 8

Java 8 manages to add many of the features of functional languages without significantly changing the Java language.

When lambda expressions, method-references, the Stream interface, and immutable data-structures are combined, Java enables what could be called “functional programming” (FP).

For the purposes of this book, the three pillars of FP are as follows:

- Functions

10.1 Functions

Of course, as the name implies, functional programming is based on functions as a firs [illegible] guably elevates functions to a first-class feature with the Lambda Project and functional interfaces.

The `Function` interface (and related interfaces IntFunction, DoubleFunction, LongFunction, BiFunction, etc.) represents the compromise made by Java 8 in elevating functions to objects. This interface allows functions to be passed as arguments, stored as variables, and be returned by methods.

The `Function` interface has the following default methods:

- andThen(Function): Returns a composed function that first applies this function to its input, and then applies the given function to the result.
- compose(Function): Similar to andThen but in reversed order (first applies the given function to its input, and then this function).
- identity(): Returns a function that always returns its input argument.

You can use these methods to create a chain for creating a function; for example:

```java
Function<Integer,String> f = Function.<Integer>identity()
        .andThen(i -> 2*i).andThen(i -> "str" + i);
```

The resulting function would take an Integer, multiply it by two, and then prepend “str” to it.

You can use andThen any number of times to create a single function. Also, remember that functions can be passed and returned from methods. Here’s an example involving the new Date-Time API:

```java
public Function<LocalDate,LocalDateTime> dateTimeFunction(
    final Function<LocalDate,LocalDate> f) {

    return f.andThen(d -> d.atTime(2, 2));
}
```

This method would take in a function that operates on a LocalDate and convert it into a function that returns a LocalDateTime (with a time of 2:02am).

Tuples

If you need a functional interface for a method with more than two parameters (eg. “TriFunction”) you need to make it yourself or use a library. Another way to handle this issue is to use a data structure called a Tuple.

A Tuple is a typed data structure for holding a number of elements. Some languages, such as Scala, have built-in support for Tuples. Tuples are useful whenever you are handling multiple related values, but don’t want all of the overhead of creating a new class.

Here’s a very simple example of implementing a Tuple with two elements:

```
public class Tuple2<A, B> {
    public final A _1;
    public final B _2;

    public Tuple2(A a, B b) {
            this._1 = a;
            this._2 = b;
    }

    @Override
    public A get_1() {
            return _1;
    }

    @Override
    public B get_2() {
            return _2;
    }
}
```

Tuples also allow you to approximate returning multiple values.

There are several implementations of Tuples available in Java, such as javatuples and totallylazy.

10.2 Immutability

In functional programming, state is considered harmful and avoided whenever possible. Instead, immutable (unchangeable) data structures are preferred. For example, `String` is an immutable type in Java.

As you may have learned, Java 8’s new Date-Time classes are immutable. What you may not have realized is that almost everything added in Java 8 is immutable (Optional and Streams for example).

However, you need to be careful when using Java 8’s new functional patterns to not accidentally fall back into the mutable mind-set. For example, the following type of code should be avoided:

```
int[] myCount = new int[1];
list.forEach(dragon -> {
    if (dragon.isGreen()) myCount[0]++;
}
```

You may think you are being clever, but this kind of thing can cause problems. Instead, you should do something like the following:

```
list.stream().filter(Dragon::isGreen).count();
```

If you ever find yourself resorting to mutability, consider if you could use some combination of “filter”, “map”, “reduce” or “collect” instead.

10.3 Concurrency

With the increasing popularity of multi-core processors, concurrent programming has become more important. Functional programming forms a solid basis for concurrent programming and Java 8 supports concurrency in many different ways.

One of those ways is the parallelStream() method on Collection. It provides a very quick way to use a Stream concurrently. However, like all optimizations, you should test to make sure that your code is actually faster, and it should be used sparingly. Too much concurrency could actually cause your application to slow down.

Another way Java 8 supports concurrency is with the new CompletableFuture class. It has the supplyAsync static method that takes in the functional interface Supplier. It also has the method thenAccept which takes in a Consumer that handles completion of the task. The CompletableFuture calls on the given supplier in a different thread and executes the consumer when complete.

When used in conjunction with things like the CountDownLatch, AtomicInteger, AtomicLong, AtomicReference, … you can implement thread-safe, concurrent FP-like code; for example:

```
public Dragon closestDragon(Location location) {
    AtomicReference<DragonDistance> closest =
        new AtomicReference<>(DragonDistance.worstMatch());
    CountDownLatch latch = new CountDownLatch(dragons.size());
    dragons.forEach(dragon -> {
        CompletableFuture.supplyAsync(() -> dragon.distance(locatio
          .thenAccept(result -> {
            closest.accumulateAndGet(result, DragonDistance::closest
            latch.countDown();
            });
        });
    try {
        latch.await();
    } catch (InterruptedException e) {
        throw new RuntimeException("Interrupted during calculations'
    }
    return closest.get().getDragon();
}
```

This example finds the closest dragon to a certain Location (assume that Dragon's distance method involves a time-consuming calculation).

However, this could be simplified using the parallelStream() default method (since only one type of calculation is going on) in the following way:

```
public Dragon closestDragon(Location location) {
    return dragons.parallelStream()
      .map(dragon -> dragon.distance(location))
      .reduce(DistancePair.worstMatch(), DragonDistance::closest)
      .getDragon();
}
```

This performs essentially the same task as the previous example but in a more concise (and functional) way.

10.4 Tail-Call Optimization

One of the hallmarks of functional programming is tail-call recursion[1]. It solves the same problem as iteration (which does not exist in FP). Unfortunately, it can cause stack-overflows if not properly optimized by the compiler.

Tail-Call optimization refers to when a compiler converts a recursive function call into a loop to avoid using the call stack. For example, a function that uses tail-call recursion in Lisp will be automatically optimized this way.

Java 8 does not support tail-call optimization like some other languages (yet). However, it is possible to approximate it using something like the following interface:

```
@FunctionalInterface
public interface Tail<T> {

    Tail<T> apply();

    default boolean isDone() {
        return false;
    }

    default T result() {
        throw new UnsupportedOperationException("Not done yet.");
    }

    default T invoke() {
        return Stream.iterate(this, Tail::apply)
                .filter(Tail::isDone)
                .findFirst()
                .get()
                .result();
    }
```

The Tail interface has three default methods and one abstract-method (apply). The invoke() method contains the meat of the “tail-call optimization”:

- It takes advantage of Stream’s iterate method to create an infinite Stream which will continuously call Tail’s apply method.
- Then it uses filter and findFirst to stop the Stream when isDone() returns true.
- Finally, it returns the result.

To implement the “done” condition, there is the following additional static method on Tail:

```
static <T> Tail<T> done(final T value) {
    return new Tail<T>() {
        @Override
        public T result() {
          return value;
        }
        @Override
        public boolean isDone() {
          return true;
        }
        @Override
        public Tail<T> apply() {
          throw new UnsupportedOperationException("Not supported.")
        }
    };
}
```

With the Tail interface you can mimic tail-call recursion quite easily in Java 8. Here's an example of calculating factorial using this interface (using BigInteger so that very large factorials can be computed):

```
public static BigInteger factorial(int n) {
    return streamFactorial(BigInteger.ONE, n).invoke();
}
private static Tail<BigInteger> streamFactorial(BigInteger x, int n
    return () -> {
      switch (n) {
        case 1:
          return Tail.done(x);
        default:
          return streamFactorial(x.multiply(BigInteger.valueOf(n)),
      }
    };
}
```

Using this method, you can make extremely fast programs while still maintaining the functional style.

Of course the JVM does a lot optimization by itself, so this may not always be the best course. However, it is something to keep in mind.

> In this particular case, a simple recursive factorial is faster than the code above, however it causes a StackOverflowError for sufficiently large numbers whereas streamFactorial does not.

1. Tail call recursion is when a function call happens inside a function as its final action.↩

11. Conclusion

Thank you for reading this short introduction to Java 8. Hopefully you learned a lot and are ready to starting using it yourself.

To recap, Java 8 includes the following:

- Lambda expressions

- Method references
- Default Methods (Defender methods)
- A new Stream API.
- Optional
- A new Date/Time API.
- Nashorn, the new JavaScript engine
- Removal of the Permanent Generation

To keep track of possible future features of Java, you might want to look at JEPS.

Backports

If for some reason you can’t immediately upgrade to Java 8, there are some ways to backport some Java 8 features to previous versions.

For each of the following features, here is the backport or similar library:

- Lambdas – Retrolambda
- Lazily Evaluated Sequences – totallylazy
- Optional – guava
- Date/Time – ThreeTen
- Nashorn – nashorn-backport

Use the backports with caution.

Podcast

Press

Contact Us

Readers

In-Progress & Serial Publishing

Ebook Formats With No DRM

Variable Pricing

100% Happiness Guarantee

Interacting With Authors

The Leanpub App

Kindle Support

Reader FAQ

Reader Help

Authors

Leanpub For Authors

'How To' Introduction

Writing With Leanpub

Writing In Word

Publishing With Leanpub

Royalties and Payments

Leanpub Book Pricing

Uploading A Book

Packaging Books With Videos

The Lean Publishing Manifesto

Author Manual

Author FAQ

Author Help

Books

Agile

Data Science

Computer Programming

Fiction

Non-Fiction

More...

More

Leanpub for Causes

Publishers

Friends of Leanpub

Terms of Service

Copyright Take Down Policy

Privacy Policy