JAVA 8 FEATURES

THE ULTIMATE GUIDE

Java™ 8

ANDREY REDKO

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Java 8 Features

Contents

Preface

It's been a while since Java 8 is out in the public and everything points to the fact that this is a really major release.

We have provided an abundance of tutorials here at Java Code Geeks, like "Playing with Java 8 - Lambdas and Concurrency" ([1]), "Java 8 Date Time API Tutorial: LocalDateTime" ([2]) and "Abstract Class Versus Interface in the JDK 8 Era" ([3]).

We also referenced "15 Must Read Java 8 Tutorials" ([4]) from other sources. Of course, we examined some of the shortfalls also, like "The Dark Side of Java 8" ([5]).

Now, it is time to gather all the major Java 8 features under one reference guide for your reading pleasure. Enjoy!

References:

[1] http://www.javacodegeeks.com/2014/04/playing-with-java-8-lambdas-and-concurrency.html

[2] http://www.javacodegeeks.com/2014/04/java-8-date-time-api-tutorial-localdatetime.html

[3] http://www.javacodegeeks.com/2014/04/abstract-class-versus-interface-in-the-jdk-8-era.html

[4] http://www.javacodegeeks.com/2014/04/15-must-read-java-8-tutorials.html

[5] http://www.javacodegeeks.com/2014/04/java-8-friday-the-dark-side-of-java-8.html

About the Author

Andriy completed his Master Degree in Computer Science at Zhitomir Institute of Engineering and Technologies, Ukraine. For the last fifteen years he has been working as the Consultant/Software Developer/Senior Software Developer/Team Lead for a many successful projects including several huge software systems for customers from North America and Europe.

Through his career Andriy has gained a great experience in enterprise architecture, web development (ASP.NET, Java Server Faces, Play Framework), software development practices (test-driven development, continious integration) and software platforms (Sun JEE, Microsoft .NET), object-oriented analysis and design, development of the rich user interfaces (MFC, Swing, Windows Forms/WPF), relational database management systems (MySQL, SQL Server, PostgreSQL, Oracle), NoSQL solutions (MongoDB, Redis) and operating systems (Linux/Windows).

Andriy has a great experience in development of distributed (multi-tier) software systems, multi-threaded applications, desktop applications, service-oriented architecture and rich Internet applications. Since 2006 he is actively working primarily with JEE / JSE platforms.

As a professional he is always open to continuous learning and self-improvement to be more productive in the job he is really passioned about.

Chapter 1

Introduction

With no doubts, Java 8 release is the greatest thing in the Java world since Java 5 (released quite a while ago, back in 2004). It brings tons of new features to the Java as a language, its compiler, libraries, tools and the JVM (Java virtual machine) itself. In this tutorial we are going to take a look on all these changes and demonstrate the different usage scenarios on real examples.

The tutorial consists of several parts where each one touches the specific side of the platform:

- language
- compiler
- libraries
- tools
- runtime (JVM)

Chapter 2

New Features in Java language

Java 8 is by any means a major release. One might say it took so long to finalize in order to implement the features every Java developer was looking for. In this section we are going to cover most of them.

2.1 Lambdas and Functional Interfaces

Lambdas (also known as closures) are the biggest and most awaited language change in the whole Java 8 release. They allow us to treat functionality as a method argument (passing functions around), or treat a code as data: the concepts every functional developer is very familiar with. Many languages on JVM platform (Groovy, Scala, ...) have had lambdas since day one, but Java developers had no choice but hammer the lambdas with boilerplate anonymous classes.

Lambdas design discussions have taken a lot of time and community efforts. But finally, the trade-offs have been found, leading to new concise and compact language constructs. In its simplest form, a lambda could be represented as a comma-separated list of parameters, the → symbol and the body. For example:

```
Arrays.asList( "a", "b", "d" ).forEach( e -> System.out.println( e ) );
```

Please notice the type of argument **e** is being inferred by the compiler. Alternatively, you may explicitly provide the type of the parameter, wrapping the definition in brackets. For example:

```
Arrays.asList( "a", "b", "d" ).forEach( ( String e ) -> System.out.println( e ) );
```

In case lambda's body is more complex, it may be wrapped into square brackets, as the usual function definition in Java. For example:

```
Arrays.asList( "a", "b", "d" ).forEach( e -> {
    System.out.print( e );
    System.out.print( e );
} );
```

Lambdas may reference the class members and local variables (implicitly making them effectively **final** if they are not). For example, those two snippets are equivalent:

```
String separator = ",";
Arrays.asList( "a", "b", "d" ).forEach(
    ( String e ) -> System.out.print( e + separator ) );
```

And:

```
final String separator = ",";
Arrays.asList( "a", "b", "d" ).forEach(
    ( String e ) -> System.out.print( e + separator ) );
```

Lambdas may return a value. The type of the return value will be inferred by compiler. The **return** statement is not required if the lambda body is just a one-liner. The two code snippets below are equivalent:

```
Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) -> e1.compareTo( e2 ) );
```

And:

```
Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) -> {
    int result = e1.compareTo( e2 );
    return result;
} );
```

Language designers put a lot of thought on how to make already existing functionality lambda-friendly. As a result, the concept of functional interfaces has emerged. The function interface is an interface with just one single method. As such, it may be implicitly converted to a lambda expression. The **java.lang.Runnable** and **java.util.concurrent.Callable** are two great examples of functional interfaces. In practice, the functional interfaces are fragile: if someone adds just one another method to the interface definition, it will not be functional anymore and compilation process will fail. To overcome this fragility and explicitly declare the intent of the interface as being functional, Java 8 adds special annotation @FunctionalInterface (all existing interfaces in Java library have been annotated with @FunctionalInterface as well). Let us take a look on this simple functional interface definition:

```
@FunctionalInterface
public interface Functional {
    void method();
}
```

One thing to keep in mind: default and static methods do not break the functional interface contract and may be declared:

```
@FunctionalInterface
public interface FunctionalDefaultMethods {
    void method();

    default void defaultMethod() {
    }
}
```

Lambdas are the largest selling point of Java 8. It has all the potential to attract more and more developers to this great platform and provide state of the art support for functional programming concepts in pure Java. For more details please refer to official documentation.

2.2 Interface's Default and Static Methods

Java 8 extends interface declarations with two new concepts: default and static methods. Default methods make interfaces somewhat similar to traits but serve a bit different goal. They allow adding new methods to existing interfaces without breaking the binary compatibility with the code written for older versions of those interfaces.

The difference between default methods and abstract methods is that abstract methods are required to be implemented. But default methods are not. Instead, each interface must provide so called default implementation and all the implementers will inherit it by default (with a possibility to override this default implementation if needed). Let us take a look on example below.

```
private interface Defaulable {
    // Interfaces now allow default methods, the implementer may or
    // may not implement (override) them.
    default String notRequired() {
        return "Default implementation";
    }
}

private static class DefaultableImpl implements Defaulable {
}
```

```
private static class OverridableImpl implements Defaulable {
    @Override
    public String notRequired() {
        return "Overridden implementation";
    }
}
```

The interface **Defaulable** declares a default method **notRequired()** using keyword **default** as part of the method definition. One of the classes, **DefaultableImpl**, implements this interface leaving the default method implementation as-is. Another one, **OverridableImpl** , overrides the default implementation and provides its own.

Another interesting feature delivered by Java 8 is that interfaces can declare (and provide implementation) of static methods. Here is an example.

```
private interface DefaulableFactory {
    // Interfaces now allow static methods
    static Defaulable create( Supplier< Defaulable > supplier ) {
        return supplier.get();
    }
}
```

The small code snippet below glues together the default methods and static methods from the examples above.

```
public static void main( String[] args ) {
    Defaulable defaulable = DefaulableFactory.create( DefaultableImpl::new );
    System.out.println( defaulable.notRequired() );

    defaulable = DefaulableFactory.create( OverridableImpl::new );
    System.out.println( defaulable.notRequired() );
}
```

The console output of this program looks like that:

```
Default implementation
Overridden implementation
```

Default methods implementation on JVM is very efficient and is supported by the byte code instructions for method invocation. Default methods allowed existing Java interfaces to evolve without breaking the compilation process. The good examples are the plethora of methods added to **java.util.Collection** interface: **stream()**, **parallelStream()**, **forEach()**, **removeIf()**, ...

Though being powerful, default methods should be used with a caution: before declaring method as default it is better to think twice if it is really needed as it may cause ambiguity and compilation errors in complex hierarchies. For more details please refer to official documentation.

2.3 Method References

Method references provide the useful syntax to refer directly to exiting methods or constructors of Java classes or objects (instances). With conjunction of Lambdas expressions, method references make the language constructs look compact and concise, leaving off boilerplate.

Below, considering the class **Car** as an example of different method definitions, let us distinguish four supported types of method references.

```
public static class Car {
    public static Car create( final Supplier< Car > supplier ) {
        return supplier.get();
    }

    public static void collide( final Car car ) {
```

```
        System.out.println( "Collided " + car.toString() );
    }

    public void follow( final Car another ) {
        System.out.println( "Following the " + another.toString() );
    }

    public void repair() {
        System.out.println( "Repaired " + this.toString() );
    }
}
```

The first type of method references is constructor reference with the syntax **Class::new** or alternatively, for generics, **Class< T >::new**. Please notice that the constructor has no arguments.

```
final Car car = Car.create( Car::new );
final List< Car > cars = Arrays.asList( car );
```

The second type is reference to static method with the syntax **Class::static_method**. Please notice that the method accepts exactly one parameter of type **Car**.

```
cars.forEach( Car::collide );
```

The third type is reference to instance method of arbitrary object of specific type with the syntax **Class::method**. Please notice, no arguments are accepted by the method.

```
cars.forEach( Car::repair );
```

And the last, fourth type is reference to instance method of particular class instance the syntax **instance::method**. Please notice that method accepts exactly one parameter of type **Car**.

```
final Car police = Car.create( Car::new );
cars.forEach( police::follow );
```

Running all those examples as a Java program produces following output on a console (the actual **Car** instances might be different):

```
Collided com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d
Repaired com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d
Following the com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d
```

For more examples and details on method references, please refer to official documentation.

2.4 Repeating annotations

Since Java 5 introduced the annotations support, this feature became very popular and is very widely used. However, one of the limitations of annotation usage was the fact that the same annotation cannot be declared more than once at the same location. Java 8 breaks this rule and introduced the repeating annotations. It allows the same annotation to be repeated several times in place it is declared.

The repeating annotations should be themselves annotated with @Repeatable annotation. In fact, it is not a language change but more a compiler trick as underneath the technique stays the same. Let us take a look on quick example:

```
package com.javacodegeeks.java8.repeatable.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
```

```
import java.lang.annotation.Target;

public class RepeatingAnnotations {
    @Target( ElementType.TYPE )
    @Retention( RetentionPolicy.RUNTIME )
    public @interface Filters {
        Filter[] value();
    }

    @Target( ElementType.TYPE )
    @Retention( RetentionPolicy.RUNTIME )
    @Repeatable( Filters.class )
    public @interface Filter {
        String value();
    };

    @Filter( "filter1" )
    @Filter( "filter2" )
    public interface Filterable {
    }

    public static void main(String[] args) {
        for( Filter filter: Filterable.class.getAnnotationsByType( Filter.class ) ) {
            System.out.println( filter.value() );
        }
    }
}
```

As we can see, there is an annotation class **Filter** annotated with @Repeatable(**Filters.class**). The **Filters** is just a holder of **Filter** annotations but Java compiler tries hard to hide its presence from the developers. As such, the interface **Filterable** has **Filter** annotation defined twice (with no mentions of **Filters**).

Also, the Reflection API provides new method **getAnnotationsByType()** to return repeating annotations of some type (please notice that Filterable.**class**.getAnnotation(Filters.**class**) will return the instance of **Filters** injected by the compiler).

The program output looks like that:

```
filter1
filter2
```

For more details please refer to official documentation.

2.5 Better Type Inference

Java 8 compiler has improved a lot on type inference. In many cases the explicit type parameters could be inferred by compiler keeping the code cleaner. Let us take a look on one of the examples.

```
package com.javacodegeeks.java8.type.inference;

public class Value< T > {
    public static< T > T defaultValue() {
        return null;
    }

    public T getOrDefault( T value, T defaultValue ) {
        return ( value != null ) ? value : defaultValue;
    }
}
```

And here is the usage of **Value< String >** type.

```
package com.javacodegeeks.java8.type.inference;

public class TypeInference {
    public static void main(String[] args) {
        final Value< String > value = new Value<>();
        value.getOrDefault( "22", Value.defaultValue() );
    }
}
```

The type parameter of **Value.defaultValue()** is inferred and is not required to be provided. In Java 7, the same example will not compile and should be rewritten to **Value.< String >defaultValue()**.

2.6 Extended Annotations Support

Java 8 extends the context where annotation might be used. Now, it is possible to annotate mostly everything: local variables, generic types, super-classes and implementing interfaces, even the method's exceptions declaration. Couple of examples are show below.

```
package com.javacodegeeks.java8.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.util.ArrayList;
import java.util.Collection;

public class Annotations {
    @Retention( RetentionPolicy.RUNTIME )
    @Target( { ElementType.TYPE_USE, ElementType.TYPE_PARAMETER } )
    public @interface NonEmpty {
    }

    public static class Holder< @NonEmpty T > extends @NonEmpty Object {
        public void method() throws @NonEmpty Exception {
        }
    }

    @SuppressWarnings( "unused" )
    public static void main(String[] args) {
        final Holder< String > holder = new @NonEmpty Holder< String >();
        @NonEmpty Collection< @NonEmpty String > strings = new ArrayList<>();
    }
}
```

The **ElementType.TYPE_USE** and **ElementType.TYPE_PARAMETER** are two new element types to describe the applicable annotation context. The **Annotation Processing API** also underwent some minor changes to recognize those new type annotations in the Java programming language.

Chapter 3

New Features in Java compiler

3.1 Parameter names

Literally for ages Java developers are inventing different ways to preserve method parameter names in Java byte-code and make them available at runtime (for example, Paranamer library). And finally, Java 8 bakes this demanding feature into the language (using Reflection API and **Parameter.getName()** method) and the byte-code (using new **javac** compiler argument **-parameters**).

```
package com.javacodegeeks.java8.parameter.names;

import java.lang.reflect.Method;
import java.lang.reflect.Parameter;

public class ParameterNames {
    public static void main(String[] args) throws Exception {
        Method method = ParameterNames.class.getMethod( "main", String[].class );
        for( final Parameter parameter: method.getParameters() ) {
            System.out.println( "Parameter: " + parameter.getName() );
        }
    }
}
```

If you compile this class without using **-parameters** argument and then run this program, you will see something like that:

```
Parameter: arg0
```

With **-parameters** argument passed to the compiler the program output will be different (the actual name of the parameter will be shown):

```
Parameter: args
```

For experienced Maven users the **-parameters** argument could be added to the compiler using configuration section of the **maven-compiler-plugin**:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
        <compilerArgument>-parameters</compilerArgument>
        <source>1.8</source>
        <target>1.8</target>
    </configuration>
</plugin>
```

Latest **Eclipse Kepler SR2** release with Java 8 (please check out this download instructions) support provides useful configuration option to control this compiler setting as the picture below shows.

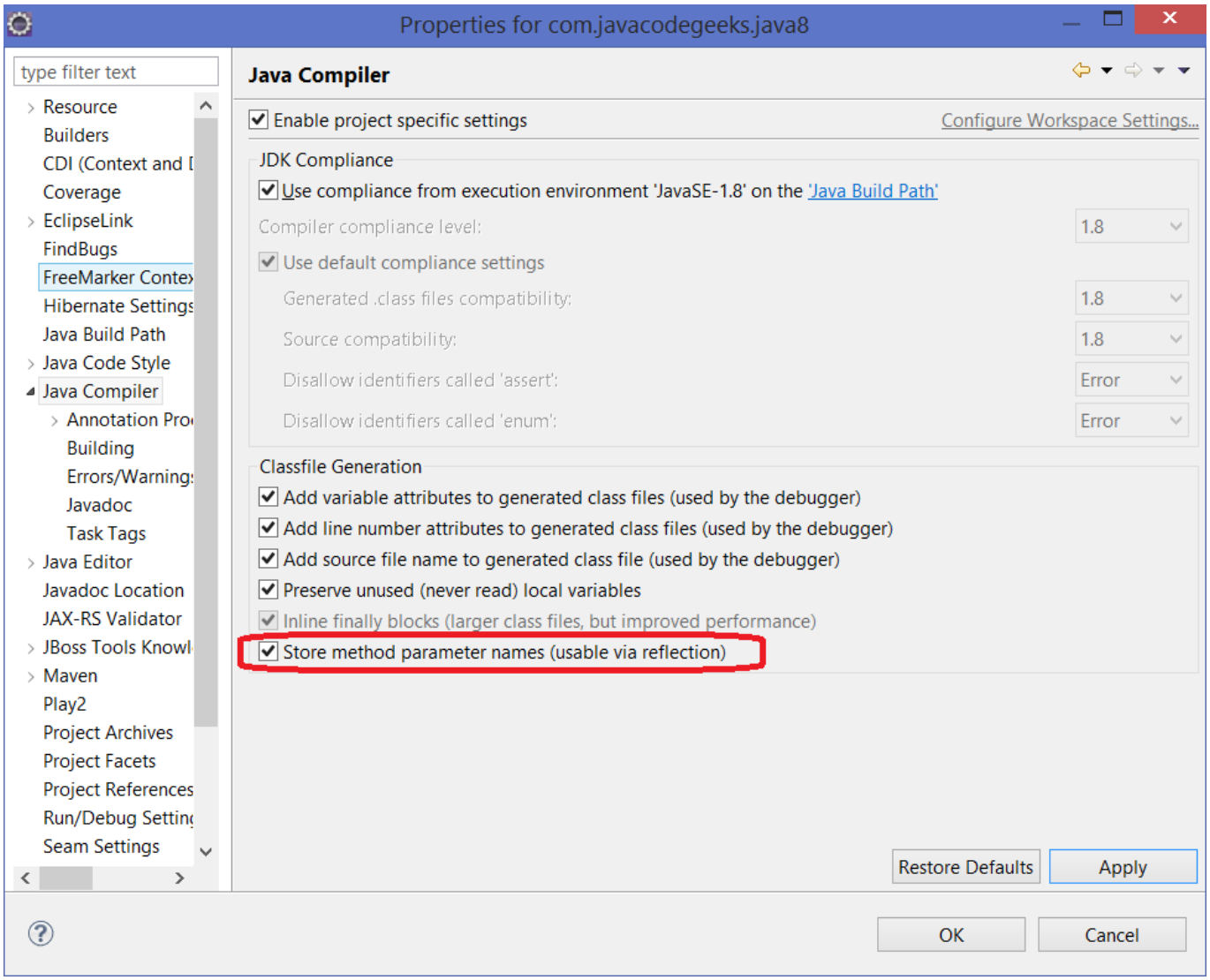

Figure 3.1: Configuring Eclipse projects to support new Java 8 compiler parameters argument

Additionally, to verify the availability of parameter names, there is a handy method **isNamePresent()** provided by **Parameter** class.

Chapter 4

New Features in Java libraries

Java 8 adds a lot of new classes and extends existing ones in order to provide better support of modern concurrency, functional programming, date/time, and many more.

4.1 Optional

The famous **NullPointerException** is by far the most popular cause of Java application failures. Long time ago the great Google Guava project introduced the **Optional** as a solution to **NullPointerException**, discouraging codebase pollution with **null** checks and encouraging developers to write cleaner code. Inspired by Google Guava, the **Optional** is now a part of Java 8 library.

Optional is just a container: it can hold a **value** of some type **T** or just be **null**. It provides a lot of useful methods so the explicit **null** checks have no excuse anymore. Please refer to official Java 8 documentation for more details.

We are going to take a look on two small examples of **Optional** usages: with the **nullable** value and with the value which does not allow **null**.

```
Optional< String > fullName = Optional.ofNullable( null );
System.out.println( "Full Name is set? " + fullName.isPresent() );
System.out.println( "Full Name: " + fullName.orElseGet( () -> "[none]" ) );
System.out.println( fullName.map( s -> "Hey " + s + "!" ).orElse( "Hey Stranger!" ) );
```

The **isPresent()** method returns **true** if this instance of **Optional** has non-null value and **false** otherwise. The **orElseGet()** method provides the fallback mechanism in case **Optional** has **null** value by accepting the function to generate the default one. The **map()** method transforms the current **Optional**'s value and returns the new **Optional** instance. The **orElse()** method is similar to **orElseGet()** but instead of function it accepts the default value. Here is the output of this program:

```
Full Name is set? false
Full Name: [none]
Hey Stranger!
```

Let us briefly look on another example:

```
Optional< String > firstName = Optional.of( "Tom" );
System.out.println( "First Name is set? " + firstName.isPresent() );
System.out.println( "First Name: " + firstName.orElseGet( () -> "[none]" ) );
System.out.println( firstName.map( s -> "Hey " + s + "!" ).orElse( "Hey Stranger!" ) );
System.out.println();
```

And here is the output:

```
First Name is set? true
First Name: Tom
Hey Tom!
```

For more details please refer to official documentation.

4.2 Streams

The newly added Stream API (**java.util.stream**) introduces real-world functional-style programming into the Java. This is by far the most comprehensive addition to Java library intended to make Java developers significantly more productive by allowing them to write effective, clean, and concise code.

Stream API makes collections processing greatly simplified (but it is not limited to Java collections only as we will see later). Let us take start off with simple class called Task.

```
public class Streams  {
    private enum Status {
        OPEN, CLOSED
    };

    private static final class Task {
        private final Status status;
        private final Integer points;

        Task( final Status status, final Integer points ) {
            this.status = status;
            this.points = points;
        }

        public Integer getPoints() {
            return points;
        }

        public Status getStatus() {
            return status;
        }

        @Override
        public String toString() {
            return String.format( "[%s, %d]", status, points );
        }
    }
}
```

Task has some notion of points (or pseudo-complexity) and can be either **OPEN** or **CLOSED**. And then let us introduce a small collection of tasks to play with.

```
final Collection< Task > tasks = Arrays.asList(
    new Task( Status.OPEN, 5 ),
    new Task( Status.OPEN, 13 ),
    new Task( Status.CLOSED, 8 )
);
```

The first question we are going to address is how many points in total all **OPEN** tasks have? Up to Java 8, the usual solution for it would be some sort of **foreach** iteration. But in Java 8 the answers is streams: a sequence of elements supporting sequential and parallel aggregate operations.

```
// Calculate total points of all active tasks using sum()
final long totalPointsOfOpenTasks = tasks
    .stream()
    .filter( task -> task.getStatus() == Status.OPEN )
    .mapToInt( Task::getPoints )
    .sum();

System.out.println( "Total points: " + totalPointsOfOpenTasks );
```

And the output on the console looks like that:

```
Total points: 18
```

There are a couple of things going on here. Firstly, the tasks collection is converted to its stream representation. Then, the **filter** operation on stream filters out all **CLOSED** tasks. On next step, the **mapToInt** operation converts the stream of **Tasks** to the stream of **Integers** using **Task::getPoints** method of the each task instance. And lastly, all points are summed up using **sum** method, producing the final result.

Before moving on to the next examples, there are some notes to keep in mind about streams (more details here). Stream operations are divided into intermediate and terminal operations.

Intermediate operations return a new stream. They are always lazy, executing an intermediate operation such as **filter** does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate

Terminal operations, such as **forEach** or **sum**, may traverse the stream to produce a result or a side-effect. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used. In almost all cases, terminal operations are eager, completing their traversal of the underlying data source.

Yet another value proposition of the streams is out-of-the box support of parallel processing. Let us take a look on this example, which does sums the points of all the tasks.

```
// Calculate total points of all tasks
final double totalPoints = tasks
   .stream()
   .parallel()
   .map( task -> task.getPoints() ) // or map( Task::getPoints )
   .reduce( 0, Integer::sum );

System.out.println( "Total points (all tasks): " + totalPoints );
```

It is very similar to the first example except the fact that we try to process all the tasks in **parallel** and calculate the final result using **reduce** method.

Here is the console output:

```
Total points (all tasks): 26.0
```

Often, there is a need to performing a grouping of the collection elements by some criteria. Streams can help with that as well as an example below demonstrates.

```
// Group tasks by their status
final Map< Status, List< Task > > map = tasks
    .stream()
    .collect( Collectors.groupingBy( Task::getStatus ) );
System.out.println( map );
```

The console output of this example looks like that:

```
{CLOSED=[[CLOSED, 8]], OPEN=[[OPEN, 5], [OPEN, 13]]}
```

To finish up with the tasks example, let us calculate the overall percentage (or weight) of each task across the whole collection, based on its points.

```
// Calculate the weight of each tasks (as percent of total points)
final Collection< String > result = tasks
    .stream()                                       // Stream< String >
    .mapToInt( Task::getPoints )                    // IntStream
    .asLongStream()                                 // LongStream
    .mapToDouble( points -> points / totalPoints )  // DoubleStream
    .boxed()                                        // Stream< Double >
    .mapToLong( weigth -> ( long )( weigth * 100 ) ) // LongStream
    .mapToObj( percentage -> percentage + "%" )      // Stream< String>
```

```
    .collect( Collectors.toList() );                    // List< String >

System.out.println( result );
```

The console output is just here:

```
[19%, 50%, 30%]
```

And lastly, as we mentioned before, the Stream API is not only about Java collections. The typical I/O operations like reading the text file line by line is a very good candidate to benefit from stream processing. Here is a small example to confirm that.

```
final Path path = new File( filename ).toPath();
try( Stream< String > lines = Files.lines( path, StandardCharsets.UTF_8 ) ) {
    lines.onClose( () -> System.out.println("Done!") ).forEach( System.out::println );
}
```

The **onClose** method called on the stream returns an equivalent stream with an additional close handler. Close handlers are run when the **close()** method is called on the stream.

Stream API together with Lambdas and Method References baked by Interface's Default and Static Methods is the Java 8 response to the modern paradigms in software development. For more details, please refer to official documentation.

4.3 Date/Time API (JSR 310)

Java 8 makes one more take on date and time management by delivering New Date-Time API (JSR 310). Date and time manipulation is being one of the worst pain points for Java developers. The standard **java.util.Date** followed by **java.util.Calendar** hasn't improved the situation at all (arguably, made it even more confusing).

That is how Joda-Time was born: the great alternative date/time API for Java. The Java 8's New Date-Time API (JSR 310) was heavily influenced by Joda-Time and took the best of it. The new **java.time** package contains all the classes for date, time, date/time, time zones, instants, duration, and clocks manipulation. In the design of the API the immutability has been taken into account very seriously: no change allowed (the tough lesson learnt from * java.util.Calendar*). If the modification is required, the new instance of respective class will be returned.

Let us take a look on key classes and examples of their usages. The first class is **Clock** which provides access to the current instant, date and time using a time-zone. **Clock** can be used instead of **System.currentTimeMillis()** and **TimeZone.getDefault()**.

```
// Get the system clock as UTC offset
final Clock clock = Clock.systemUTC();
System.out.println( clock.instant() );
System.out.println( clock.millis() );
```

The sample output on a console:

```
2014-04-12T15:19:29.282Z
1397315969360
```

Other new classes we are going to look at are **LocaleDate** and **LocalTime**. **LocaleDate** holds only the date part without a time-zone in the ISO-8601 calendar system. Respectively, **LocaleTime** holds only the time part without time-zone in the ISO-8601 calendar system. Both **LocaleDate** and **LocaleTime** could be created from **Clock**.

```
// Get the local date and local time
final LocalDate date = LocalDate.now();
final LocalDate dateFromClock = LocalDate.now( clock );

System.out.println( date );
System.out.println( dateFromClock );

// Get the local date and local time
final LocalTime time = LocalTime.now();
```

```
final LocalTime timeFromClock = LocalTime.now( clock );

System.out.println( time );
System.out.println( timeFromClock );
```

The sample output on a console:

```
2014-04-12
2014-04-12
11:25:54.568
15:25:54.568
```

The **LocalDateTime** combines together **LocaleDate** and **LocalTime** and holds a date with time but without a time-zone in the ISO-8601 calendar system. A quick example is shown below.

```
// Get the local date/time
final LocalDateTime datetime = LocalDateTime.now();
final LocalDateTime datetimeFromClock = LocalDateTime.now( clock );

System.out.println( datetime );
System.out.println( datetimeFromClock );
```

The sample output on a console:

```
2014-04-12T11:37:52.309
2014-04-12T15:37:52.309
```

If case you need a date/time for particular timezone, the **ZonedDateTime** is here to help. It holds a date with time and with a time-zone in the ISO-8601 calendar system. Here are a couple of examples for different timezones.

```
// Get the zoned date/time
final ZonedDateTime zonedDatetime = ZonedDateTime.now();
final ZonedDateTime zonedDatetimeFromClock = ZonedDateTime.now( clock );
final ZonedDateTime zonedDatetimeFromZone = ZonedDateTime.now( ZoneId.of( "America/ ↩
    Los_Angeles" ) );

System.out.println( zonedDatetime );
System.out.println( zonedDatetimeFromClock );
System.out.println( zonedDatetimeFromZone );
```

The sample output on a console:

```
2014-04-12T11:47:01.017-04:00[America/New_York]
2014-04-12T15:47:01.017Z
2014-04-12T08:47:01.017-07:00[America/Los_Angeles]
```

And finally, let us take a look on **Duration** class: an amount of time in terms of seconds and nanoseconds. It makes very easy to compute the different between two dates. Let us take a look on that.

```
// Get duration between two dates
final LocalDateTime from = LocalDateTime.of( 2014, Month.APRIL, 16, 0, 0, 0 );
final LocalDateTime to = LocalDateTime.of( 2015, Month.APRIL, 16, 23, 59, 59 );

final Duration duration = Duration.between( from, to );
System.out.println( "Duration in days: " + duration.toDays() );
System.out.println( "Duration in hours: " + duration.toHours() );
```

The example above computes the duration (in days and hours) between two dates, **16 April 2014** and **16 April 2015**. Here is the sample output on a console:

```
Duration in days: 365
Duration in hours: 8783
```

The overall impression about Java 8's new date/time API is very, very positive. Partially, because of the battle-proved foundation it is built upon (Joda-Time), partially because this time it was finally tackled seriously and developer voices have been heard. For more details please refer to official documentation.

4.4 Nashorn JavaScript engine

Java 8 comes with new Nashorn JavaScript engine which allows developing and running certain kinds of JavaScript applications on JVM. Nashorn JavaScript engine is just another implementation of javax.script.ScriptEngine and follows the same set of rules, permitting Java and JavaScript interoperability. Here is a small example.

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName( "JavaScript" );

System.out.println( engine.getClass().getName() );
System.out.println( "Result:" + engine.eval( "function f() { return 1; }; f() + 1;" ) );
```

The sample output on a console:

```
jdk.nashorn.api.scripting.NashornScriptEngine
Result: 2
```

We will get back to the Nashorn later in the section dedicated to new Java tools.

4.5 Base64

Finally, the support of Base64 encoding has made its way into Java standard library with Java 8 release. It is very easy to use as following example shows off.

```
package com.javacodegeeks.java8.base64;

import java.nio.charset.StandardCharsets;
import java.util.Base64;

public class Base64s {
    public static void main(String[] args) {
        final String text = "Base64 finally in Java 8!";

        final String encoded = Base64
            .getEncoder()
            .encodeToString( text.getBytes( StandardCharsets.UTF_8 ) );
        System.out.println( encoded );

        final String decoded = new String(
            Base64.getDecoder().decode( encoded ),
            StandardCharsets.UTF_8 );
        System.out.println( decoded );
    }
}
```

The console output from program run shows both encoded and decoded text:

```
QmFzZTY0IGZpbmFsbHkgaW4gSmF2YSA4IQ==
Base64 finally in Java 8!
```

There are also URL-friendly encoder/decoder and MIME-friendly encoder/decoder provided by the Base64 class:

Base64.getUrlEncoder() / Base64.getUrlDecoder(), **Base64.getMimeEncoder() / Base64.getMimeDecoder()**.

4.6 Parallel Arrays

Java 8 release adds a lot of new methods to allow parallel arrays processing. Arguably, the most important one is **parallelSort()** which may significantly speedup the sorting on multicore machines. The following small example demonstrates this new method family (**parallelXxx**) in action.

```
package com.javacodegeeks.java8.parallel.arrays;

import java.util.Arrays;
import java.util.concurrent.ThreadLocalRandom;

public class ParallelArrays {
    public static void main( String[] args ) {
        long[] arrayOfLong = new long [ 20000 ];

        Arrays.parallelSetAll( arrayOfLong,
            index -> ThreadLocalRandom.current().nextInt( 1000000 ) );
        Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
            i -> System.out.print( i + " " ) );
        System.out.println();

        Arrays.parallelSort( arrayOfLong );
        Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
            i -> System.out.print( i + " " ) );
        System.out.println();
    }
}
```

This small code snippet uses method **parallelSetAll()** to fill up arrays with 20000 random values. After that, the **parallelSort()** is being applied. The program outputs first 10 elements before and after sorting so to ensure the array is really ordered. The sample program output may look like that (please notice that array elements are randomly generated):

```
Unsorted: 591217 891976 443951 424479 766825 351964 242997 642839 119108 552378
Sorted: 39 220 263 268 325 607 655 678 723 793
```

4.7 Concurrency

New methods have been added to the **java.util.concurrent.ConcurrentHashMap** class to support aggregate operations based on the newly added streams facility and lambda expressions.

Also, new methods have been added to the **java.util.concurrent.ForkJoinPool** class to support a common pool (check also our free course on Java concurrency).

The new **java.util.concurrent.locks.StampedLock** class has been added to provide a capability-based lock with three modes for controlling read/write access (it might be considered as better alternative for infamous **java.util.concurrent.locks.ReadWriteLock**).

New classes have been added to the **java.util.concurrent.atomic** package:

- **DoubleAccumulator**
- **DoubleAdder**
- **LongAccumulator**
- **LongAdder**

Chapter 5

New Java tools

Java 8 comes with new set of command line tools. In this section we are going to look over most interesting of them.

5.1 Nashorn engine: jjs

jjs is a command line based standalone Nashorn engine. It accepts a list of JavaScript source code files as arguments and runs them. For example, let us create a file **func.js** with following content:

```
function f() {
     return 1;
};

print( f() + 1 );
```

To execute this fie from command, let us pass it as an argument to **jjs**:

```
jjs func.js
```

The output on the console will be:

```
2
```

For more details please refer to official documentation.

5.2 Class dependency analyzer: jdeps

jdeps is a really great command line tool. It shows the package-level or class-level dependencies of Java class files. It accepts **.class** file, **a directory**, or **JAR file** as an input. By default, **jdeps** outputs the dependencies to the system output (console).

As an example, let us take a look on dependencies report for the popular Spring Framework library. To make example short, let us analyze only one JAR file: **org.springframework.core-3.0.5.RELEASE.jar**.

```
jdeps org.springframework.core-3.0.5.RELEASE.jar
```

This command outputs quite a lot so we are going to look on the part of it. The dependencies are grouped by packages. If dependency is not available on a classpath, it is shown as **not found**.

```
org.springframework.core-3.0.5.RELEASE.jar -> C:\Program Files\Java\jdk1.8.0\jre\lib\rt.jar
   org.springframework.core (org.springframework.core-3.0.5.RELEASE.jar)
      -> java.io
      -> java.lang
```

```
      -> java.lang.annotation
      -> java.lang.ref
      -> java.lang.reflect
      -> java.util
      -> java.util.concurrent
      -> org.apache.commons.logging                          not found
      -> org.springframework.asm                             not found
      -> org.springframework.asm.commons                     not found
   org.springframework.core.annotation (org.springframework.core-3.0.5.RELEASE.jar)
      -> java.lang
      -> java.lang.annotation
      -> java.lang.reflect
      -> java.util
```

For more details please refer to official documentation.

Chapter 6

New Features in Java runtime (JVM)

The **PermGen** space is gone and has been replaced with Metaspace (JEP 122). The JVM options **-XX:PermSize** and **-XX:MaxPermSize** have been replaced by **-XX:MetaSpaceSize** and **-XX:MaxMetaspaceSize** respectively.

Chapter 7

Conclusions

The future is here: Java 8 moves this great platform forward by delivering the features to make developers much more productive. It is too early to move the production systems to Java 8 but in the next couples of months its adoption should slowly start growing. Nevertheless the time is right to start preparing your code bases to be compatible with Java 8 and to be ready to turn the switch once Java 8 proves to be safe and stable enough.

As a confirmation of community Java 8 acceptance, recently Pivotal released Spring Framework 4.0.3 with production-ready Java 8 support.

If you enjoyed this, then **subscribe to our newsletter** to enjoy weekly updates and complimentary whitepapers! Also, check out **JCG Academy** for more advanced training!

Chapter 8

Resources

Some additional resources which discuss in depth different aspects of Java 8 features:

- What's New in JDK 8: http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html
- The Java Tutorials: http://docs.oracle.com/javase/tutorial/
- WildFly 8, JDK 8, NetBeans 8, Java EE 7: http://blog.arungupta.me/2014/03/wildfly8-jdk8-netbeans8-javaee7-excellent-combo-enterprise-java/
- Java 8 Tutorial: http://winterbe.com/posts/2014/03/16/java-8-tutorial/
- JDK 8 Command-line Static Dependency Checker: http://marxsoftware.blogspot.ca/2014/03/jdeps.html
- The Illuminating Javadoc of JDK 8: http://marxsoftware.blogspot.ca/2014/03/illuminating-javadoc-of-jdk-8.html
- The Dark Side of Java 8: http://blog.jooq.org/2014/04/04/java-8-friday-the-dark-side-of-java-8/
- Installing Java 8 Support in Eclipse Kepler SR2: http://www.eclipse.org/downloads/java8/
- Java 8: http://www.baeldung.com/java8
- Oracle Nashorn. A Next-Generation JavaScript Engine for the JVM: http://www.oracle.com/technetwork/articles/java/jf14-nashorn-2126515.html