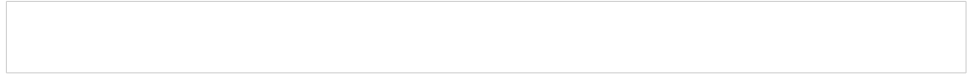


Java 8 Features Tutorial – The ULTIMATE Guide (PDF Download)

Posted by: Andrey Redko in Core Java May 2nd, 2014 52 Comments 9891 Views



EDITORIAL NOTE: In this post, we feature a comprehensive Java 8 Features Tutorial. 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](#), [Java 8 Date Time API Tutorial : LocalDateTime](#) and [Abstract Class Versus Interface in the JDK 8 Era](#).

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

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

Table Of Contents

- 1. Introduction
- 2. New Features in Java language
 - 2.1. Lambdas and Functional Interfaces
 - 2.2. Interface Default and Static Methods
 - 2.3. Method References
 - 2.4. Repeating annotations
 - 2.5. Better Type Inference
 - 2.6. Extended Annotations Support
- 3. New Features in Java compiler
 - 3.1. Parameter names
- 4. New Features in Java libraries
 - 4.1. Optional
 - 4.2. Streams
 - 4.3. Date/Time API (JSR 310)
 - 4.4. Nashorn JavaScript engine
 - 4.5. Base64
 - 4.6. Parallel Arrays
 - 4.7. Concurrency
- 5. New Java tools
 - 5.1. Nashorn engine: jjs

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)

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:

```
1 Arrays.asList( "a", "b", "d" ).forEach( e ->  
2   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:

```
1 Arrays.asList( "a", "b", "d" ).forEach( ( String e ) ->  
2   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:

```
1 Arrays.asList( "a", "b", "d" ).forEach( e ->  
2 {  
3     System.out.print( e );  
4     System.out.print( e );  
5 } );
```

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:

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

And:

```
1 final String separator = ",";  
2 Arrays.asList( "a", "b", "d" ).forEach(  
3     ( 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:

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

And:

```
1 Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) ->  
2 {  
3     int result = e1.compareTo( e2 );  
4     return result;  
5 } );
```

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:

```
1@FunctionalInterface
2public interface Functional {
3    void method();
4}
```

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

```
1@FunctionalInterface
2public interface FunctionalDefaultMethods {
3    void method();
4
5    default void defaultMethod() {
6    }
7}
```

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.

```

01 private interface Defaulable {
02     // Interfaces now allow default methods, the implementer may or
03     // may not implement (override) them.
04     default String notRequired() {
05         return "Default implementation";
06     }
07 }
08
09 private static class DefaultableImpl implements Defaulable {
10 }
11
12 private static class OverridableImpl implements Defaulable {
13     @Override
14     public String notRequired() {
15         return "Overridden implementation";
16     }
17 }

```

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.

```

1 private interface DefaulableFactory {
2     // Interfaces now allow static methods
3     static Defaulable create( Supplier< Defaulable > supplier ) {
4         return supplier.get();
5     }
6 }

```

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

```

1 public static void main( String[] args ) {
2     Defaulable defaulable = DefaulableFactory.create( DefaultableImpl::new
3 );
4     System.out.println( defaulable.notRequired() );
5
6     defaulable = DefaulableFactory.create( OverridableImpl::new );
7     System.out.println( defaulable.notRequired() );
8 }

```

The console output of this program looks like that:

```
1 Default implementation
```

```
2 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 existing 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.

```
01 public static class Car {
02     public static Car create( final Supplier< Car > supplier ) {
03         return supplier.get();
04     }
05
06     public static void collide( final Car car ) {
07         System.out.println( "Collided " + car.toString() );
08     }
09
10     public void follow( final Car another ) {
11         System.out.println( "Following the " + another.toString() );
12     }
13
14     public void repair() {
15         System.out.println( "Repaired " + this.toString() );
16     }
17 }
```

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.

```
1 final Car car = Car.create( Car::new );
2 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`.

```
1 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.

```
1cars.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`.

```
1final Car police = Car.create( Car::new );
```

```
2cars.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
```

```
1Repaired
```

```
2com.javacodegeeks.java8.method.references.MethodReferences$Car@7a811397d
```

```
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:

```

01package com.javacodegeeks.java8.repeatable.annotations;
02import java.lang.annotation.ElementType;
03import java.lang.annotation.Repeatable;
04import java.lang.annotation.Retention;
05import java.lang.annotation.RetentionPolicy;
06import java.lang.annotation.Target;
07public class RepeatingAnnotations {
08    @Target( ElementType.TYPE )
09    @Retention( RetentionPolicy.RUNTIME )
10    public @interface Filters {
11        Filter[] value();
12    }
13
14    @Target( ElementType.TYPE )
15    @Retention( RetentionPolicy.RUNTIME )
16    @Repeatable( Filters.class )
17    public @interface Filter {
18        String value();
19    };
20
21    @Filter( "filter1" )
22    @Filter( "filter2" )
23    public interface Filterable {
24    }
25
26    public static void main(String[] args) {
27        for( Filter filter: Filterable.class.getAnnotationsByType( Filter.class
28        29) ) {
29            System.out.println( filter.value() );
30        }
31    }
32}
33}

```

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:

```

1filter1
2filter2

```

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.

```
01 package com.javacodegeeks.java8.type.inference;
02 public class Value< T > {
03     public static < T > T defaultValue() {
04         return null;
05     }
06
07     public T getOrDefault( T value, T defaultValue ) {
08         return ( value != null ) ? value : defaultValue;
09     }
10 }
11 }
```

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

```
1 package com.javacodegeeks.java8.type.inference;
2 public class TypeInference {
3     public static void main(String[] args) {
4         final Value< String > value = new Value<>();
5         value.getOrDefault( "22", Value.defaultValue() );
6     }
7 }
8 }
```

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;
01 import java.lang.annotation.ElementType;
02 import java.lang.annotation.Retention;
03 import java.lang.annotation.RetentionPolicy;
04 import java.lang.annotation.Target;
05 import java.util.ArrayList;
06 import java.util.Collection;
07 public class Annotations {
08     @Retention( RetentionPolicy.RUNTIME )
09     @Target( { ElementType.TYPE_USE, ElementType.TYPE_PARAMETER } )
10     public @interface NonEmpty {
11     }
12
13     public static class Holder< @NonEmpty T > extends @NonEmpty
14 Object {
15         public void method() throws @NonEmpty Exception {
16         }
17     }
18
19     @SuppressWarnings( "unused" )
20     public static void main(String[] args) {
21         final Holder< String > holder = new @NonEmpty Holder< String
22 >();
23         @NonEmpty Collection< @NonEmpty String > strings = new
24 ArrayList<>();
25     }
26 }

```

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.

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`).

```

01 package com.javacodegeeks.java8.parameter.names;
02 import java.lang.reflect.Method;
03 import java.lang.reflect.Parameter;
04 public class ParameterNames {
05     public static void main(String[] args) throws Exception {
06         Method method = ParameterNames.class.getMethod( "main",
07 String[].class );
08         for( final Parameter parameter: method.getParameters() ) {
09             System.out.println( "Parameter: " + parameter.getName() );
10         }
11     }
12 }
13 }

```

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

```
1Parameter: arg0
```

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

```
1Parameter: args
```

For [experienced Maven users](#) the `-parameters` argument could be added to the compiler using configuration section of the maven-compiler-plugin:

```

01 <plugin>
02     <groupId>org.apache.maven.plugins</groupId>
03     <artifactId>maven-compiler-plugin</artifactId>
04     <version>3.1</version>
05     <configuration>
06         <compilerArgument>-parameters</compilerArgument>
07         <source>1.8</source>
08         <target>1.8</target>
09     </configuration>
10 </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.

Picture 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.

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 Optionals as a solution to NullPointerExceptions, 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 nulls.

```
1 Optional< String > fullName = Optional.ofNullable( null );  
1 System.out.println( "Full Name is set? " + fullName.isPresent() );  
2 System.out.println( "Full Name: " + fullName.orElseGet( () -> "[none]" ) );  
3 System.out.println( fullName.map( s -> "Hey " + s + "!" ).orElse( "Hey  
4 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:

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

Let us briefly look on another example:

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

And here is the output:

1First Name is set? **true**

2First Name: Tom

3Hey 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.

```
01 public class Streams {
02     private enum Status {
03         OPEN, CLOSED
04     };
05
06     private static final class Task {
07         private final Status status;
08         private final Integer points;
09         Task( final Status status, final Integer points ) {
10             this.status = status;
11             this.points = points;
12         }
13
14         public Integer getPoints() {
15             return points;
16         }
17
18         public Status getStatus() {
19             return status;
20         }
21
22         @Override
23         public String toString() {
24             return String.format( "[%s, %d]", status, points );
25         }
26     }
27 }
28 }
```

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.

```
1 final Collection< Task > tasks =
```

```

1 Arrays.asList(
2     new Task( Status.OPEN, 5 ),
3     new Task( Status.OPEN, 13 ),
4     new Task( Status.CLOSED, 8 )
5 );

```

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 answer is streams: a sequence of elements supporting sequential and parallel aggregate operations.

```

1 // Calculate total points of all active tasks using sum()
2 final long totalPointsOfOpenTasks = tasks
3     .stream()
4     .filter( task -> task.getStatus() == Status.OPEN )
5     .mapToInt( Task::getPoints )
6     .sum();
7
8 System.out.println( "Total points: " +
9     totalPointsOfOpenTasks );

```

And the output on the console looks like that:

```

1 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.

```

1// Calculate total points of all tasks
2final double totalPoints = tasks
3    .stream()
4    .parallel()
5    .map( task -> task.getPoints() ) // or map( Task::getPoints )
6    .reduce( 0, Integer::sum );
7

```

```

8System.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:

```

1Total points (all tasks):
226.0

```

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.

```

1// Group tasks by their status
2final Map< Status, List< Task > > map = tasks
3    .stream()
4    .collect( Collectors.groupingBy( Task::getStatus ) );
5System.out.println( map );

```

The console output of this example looks like that:

```

1{CLOSED=[[CLOSED, 8]], OPEN=[[OPEN, 5], [OPEN,
213]]}

```

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.

```

01// Calculate the weight of each tasks (as percent of total points)
02final Collection< String > result = tasks
03    .stream() // Stream< String >
04    .mapToInt( Task::getPoints ) // IntStream
05    .asLongStream() // LongStream
06    .mapToDouble( points -> points / totalPoints ) // DoubleStream
07    .boxed() // Stream< Double >
08    .mapToLong( weigth -> ( long )( weigth * 100 ) ) // LongStream
09    .mapToObj( percentage -> percentage + "%" ) // Stream< String>
10    .collect( Collectors.toList() ); // List< String >
11
12System.out.println( result );

```

The console output is just here:

```

1[19%, 50%,
230%]

```

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.

```

1 final Path path = new File( filename ).toPath();
2 try( Stream< String > lines = Files.lines( path, StandardCharsets.UTF_8 ) ) {
3     lines.onClose( () ->
4         System.out.println("Done!") ).forEach( System.out::println );
5 }

```

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()`.

```

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

```

The sample output on a console:

```

1 2014-04-12T15:19:29.282Z
2 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, `LocalTime` holds only the time

part without time-zone in the ISO-8601 calendar system.

Both `LocalDate` and `LocalTime` could be created from `Clock`.

```
01// Get the local date and local time
02final LocalDate date = LocalDate.now();
03final LocalDate dateFromClock = LocalDate.now( clock );
04
05System.out.println( date );
06System.out.println( dateFromClock );
07
08// Get the local date and local time
09final LocalTime time = LocalTime.now();
10final LocalTime timeFromClock = LocalTime.now( clock );
11
12System.out.println( time );
13System.out.println( timeFromClock );
```

The sample output on a console:

```
2014-04-12
12014-04-12
211:25:54.56
38
415:25:54.56
8
```

The `LocalDateTime` combines together `LocalDate` 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.

```
1// Get the local date/time
2final LocalDateTime datetime = LocalDateTime.now();
3final LocalDateTime datetimeFromClock = LocalDateTime.now( clock );
4
5System.out.println( datetime );
6System.out.println( datetimeFromClock );
```

The sample output on a console:

```
12014-04-12T11:37:52.309
22014-04-12T15:37:52.309
```

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
1final ZonedDateTime zonedDatetime = ZonedDateTime.now();
2final ZonedDateTime zonedDatetimeFromClock = ZonedDateTime.now( clock
3);
4final ZonedDateTime zonedDatetimeFromZone =
5ZonedDateTime.now( ZoneId.of( "America/Los_Angeles" ) );
6
7System.out.println( zonedDatetime );
8System.out.println( zonedDatetimeFromClock );
9System.out.println( zonedDatetimeFromZone );
```

The sample output on a console:

```
1 2014-04-12T11:47:01.017-04:00[America/New_York]
2 2014-04-12T15:47:01.017Z
3 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.

```
1// Get duration between two dates
2final LocalDateTime from = LocalDateTime.of( 2014, Month.APRIL, 16, 0, 0, 0
3);
4final LocalDateTime to = LocalDateTime.of( 2015, Month.APRIL, 16, 23, 59,
559 );
6final Duration duration = Duration.between( from, to );
7System.out.println( "Duration in days: " + duration.toDays() );
8System.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:

```
1Duration in days: 365
2Duration in hours:
38783
```

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.

```
1ScriptEngineManager manager = new ScriptEngineManager();
2ScriptEngine engine = manager.getEngineByName( "JavaScript" );
3System.out.println( engine.getClass().getName() );
4System.out.println( "Result:" + engine.eval( "function f() { return 1; }; f() +
51;" ) );
```

The sample output on a console:

```
1jdk.nashorn.api.scripting.NashornScriptEngine
2Result: 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.

```
01
02 package com.javacodegeeks.java8.base64;
03 import java.nio.charset.StandardCharsets;
04 import java.util.Base64;
05 public class Base64s {
06     public static void main(String[] args) {
07         final String text = "Base64 finally in Java 8!";
08
09         final String encoded = Base64
10             .getEncoder()
11             .encodeToString( text.getBytes( StandardCharsets.UTF_8 ) );
12         System.out.println( encoded );
13
14         final String decoded = new String(
15             Base64.getDecoder().decode( encoded ),
16             StandardCharsets.UTF_8 );
17         System.out.println( decoded );
18     }
19 }
20
```

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

```
1QmFzZTY0IGZpbmFsbHkgaW4gSmF2YSA4IQ==
2Base64 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.

```

01
02 package com.javacodegeeks.java8.parallel.arrays;
03 import java.util.Arrays;
04 import java.util.concurrent.ThreadLocalRandom;
05 public class ParallelArrays {
06     public static void main( String[] args ) {
07         long[] arrayOfLong = new long [ 20000 ];
08
09         Arrays.parallelSetAll( arrayOfLong,
10             index -> ThreadLocalRandom.current().nextInt( 1000000 ) );
11         Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
12             i -> System.out.print( i + " " ) );
13         System.out.println();
14
15         Arrays.parallelSort( arrayOfLong );
16         Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
17             i -> System.out.print( i + " " ) );
18         System.out.println();
19     }
20 }
21

```

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):

```

1 Unsorted: 591217 891976 443951 424479 766825 351964 242997 642839
2 119108 552378
2 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

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:

```
1function f() {  
2    return 1;  
3}  
4};  
5print( f() + 1 );
```

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

```
1jjs func.js
```

The output on the console will be:

```
12
```

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.

```
1jdeps 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
01 org.springframework.core (org.springframework.core-3.0.5.RELEASE.jar)
02   -> java.io
03   -> java.lang
04   -> java.lang.annotation
05   -> java.lang.ref
06   -> java.lang.reflect
07   -> java.util
08   -> java.util.concurrent
09   -> org.apache.commons.logging not found
10   -> org.springframework.asm not found
11   -> org.springframework.asm.commons not found
12   -> org.springframework.core.annotation (org.springframework.core-
13 3.0.5.RELEASE.jar)
14   -> java.lang
15   -> java.lang.annotation
16   -> java.lang.reflect
17   -> java.util

```

For more details please refer to [official documentation](#).

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.

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 our [courses](#) for more advanced training!

You are welcome to contribute with your comments about the exciting new Java 8 features!

8. Resources

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

- Java 8 Tutorials on JCG Examples: <https://examples.javacodegeeks.com/?s=java+8>
- 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>

Java 8 Features Tutorial was last updated on Oct. 3, 2016