

[Home](#) » [Java](#) » [Core Java](#) » [Java 8 Features Tutorial – The ULTIMATE Guide \(PDF Download\)](#)

## ABOUT ANDREY REDKO



Andriy is a well-grounded software developer with more than 12 years of practical experience using Java/EE, C#/.NET, C++, Groovy, Ruby, functional programming (Scala), databases (MySQL, PostgreSQL, Oracle) and NoSQL solutions (MongoDB, Redis).



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

Posted by: [Andrey Redko](#) in [Core Java](#) on [May 2nd, 2014](#)

**Last updated Oct. 3, 2016**

**EDITORIAL NOTE:** *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!*

## Want to be a Java 8 Ninja ?

Subscribe to our newsletter and download the Java 8 Features **Ultimate Guide** [right now!](#)

In order to get you up to speed with the major Java 8 release, we have compiled a kick-ass guide with all the new features and goodies! Besides studying them online you may download the eBook in PDF format!

**Email address:**

[Sign up](#)

Thank you, your sign-up request was successful! Please check your e-mail inbox.

## 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)

## NEWSLETTER

**179,260** insiders are already receiving weekly updates and complimentary whitepapers!

**Join them now** to gain **access** to the latest news in as well as insights about Andriy Groovy and other related tech!

**Email address:**

[Sign up](#)

## RECENT JOBS

No job listings found.

## JOIN US



With **1,500** unique and valuable articles placed at your disposal, Constant lookout for new and interesting content to check out our **JCG** partners program!

- 4.7. Concurrency
- 5. New Java tools
- 5.1. Nashorn engine: jjs
- 5.2. Class dependency analyzer: jdeps
- 6. New Features in Java runtime (JVM)
- 7. Conclusions
- 8. Resources

# 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 -> 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 ) -> 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     System.out.print( e );
3     System.out.print( e );
4 } );
```

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 ) -> e1.compareTo( e2 ) );
```

And:

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

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
2 public 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
2 public 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     System.out.println( defaulable.notRequired() );
4
5     defaulable = DefaulableFactory.create( OverridableImpl::new );
6     System.out.println( defaulable.notRequired() );
7 }
```

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

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.

```

1 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**.

```

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

```

1 Collided com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d
2 Repaired com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d
3 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:

```

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

```

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:

```
1 filter1
2 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.

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

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

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

```
01 package com.javacodegeeks.java8.annotations;
02
03 import java.lang.annotation.ElementType;
04 import java.lang.annotation.Retention;
05 import java.lang.annotation.RetentionPolicy;
06 import java.lang.annotation.Target;
07 import java.util.ArrayList;
08 import java.util.Collection;
09
10 public class Annotations {
11     @Retention( RetentionPolicy.RUNTIME )
12     @Target( { ElementType.TYPE_USE, ElementType.TYPE_PARAMETER } )
13     public @interface NonEmpty {
14     }
15
16     public static class Holder< @NonEmpty T > extends @NonEmpty Object {
17         public void method() throws @NonEmpty Exception {
18         }
19     }
20
21     @SuppressWarnings( "unused" )
22     public static void main(String[] args) {
23         final Holder< String > holder = new @NonEmpty Holder< String >();
24         @NonEmpty Collection< @NonEmpty String > strings = new 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, Paraname 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;
```

```

06 public class ParameterNames {
07     public static void main(String[] args) throws Exception {
08         Method method = ParameterNames.class.getMethod( "main", String[].class );
09         for( final Parameter parameter: method.getParameters() ) {
10             System.out.println( "Parameter: " + parameter.getName() );
11         }
12     }
13 }

```

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

```
1 Parameter: arg0
```

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

```
1 Parameter: args
```

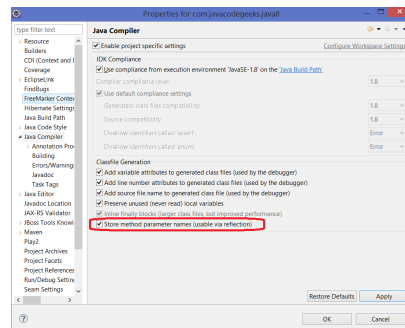
For experienced Maven users the **-parameters** argument could be added to the compiler using configuration section of the **maven-compiler-plugin**:

```

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 );
2 System.out.println( "Full Name is set? " + fullName.isPresent() );
3 System.out.println( "Full Name: " + fullName.orElseGet( () -> "[none]" ) );
4 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:

```

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

```

```

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

```

And here is the output:

```

1 First Name is set? true
2 First Name: Tom
3 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.

```

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
10         Task( final Status status, final Integer points ) {
11             this.status = status;
12             this.points = points;
13         }
14
15         public Integer getPoints() {
16             return points;
17         }
18
19         public Status getStatus() {
20             return status;
21         }
22
23         @Override
24         public String toString() {
25             return String.format( "[%s, %d]", status, points );
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 = 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: " + 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.

```

1 // Calculate total points of all tasks
2 final double totalPoints = tasks
3   .stream()
4   .parallel()
5   .map( task -> task.getPoints() ) // or map( Task::getPoints )
6   .reduce( 0, Integer::sum );
7
8 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:

```

1 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.

```

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

```

The console output of this example looks like that:

```

1 {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.

```

01 // Calculate the weight of each tasks (as percent of total points)
02 final 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( weighth -> ( long ) ( weighth * 100 ) ) // LongStream
09   .mapToObj( percentage -> percentage + "%" ) // Stream< String>
10   .collect( Collectors.toList() ); // List< String >
11
12 System.out.println( result );

```

The console output is just here:

```

1 [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.

```

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

```

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

```



```

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

```

The sample output on a console:

```

1 2014-04-12
2 2014-04-12
3 11:25:54.568
4 15:25:54.568

```

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
2 final LocalDateTime datetime = LocalDateTime.now();
3 final LocalDateTime datetimeFromClock = LocalDateTime.now( clock );
4
5 System.out.println( datetime );
6 System.out.println( datetimeFromClock );

```

The sample output on a console:

```

1 2014-04-12T11:37:52.309
2 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.

```

1 // Get the zoned date/time
2 final ZonedDateTime zonedDateTime = ZonedDateTime.now();
3 final ZonedDateTime zonedDateTimeFromClock = ZonedDateTime.now( clock );
4 final ZonedDateTime zonedDateTimeFromZone = ZonedDateTime.now( ZoneId.of( "America/Los_Angeles" ) );
5
6 System.out.println( zonedDateTime );
7 System.out.println( zonedDateTimeFromClock );
8 System.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
2 final LocalDateTime from = LocalDateTime.of( 2014, Month.APRIL, 16, 0, 0, 0 );
3 final LocalDateTime to = LocalDateTime.of( 2015, Month.APRIL, 16, 23, 59, 59 );
4
5 final Duration duration = Duration.between( from, to );
6 System.out.println( "Duration in days: " + duration.toDays() );
7 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:

```

1 Duration in days: 365
2 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.

```

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

```

The sample output on a console:

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

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

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

There are also URL-friendly encoder/decoder and MIME-friendly encoder/decoder provided by the Base64 class (**Base64.getUriEncoder()** / **Base64.getUriDecoder()**, **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 package com.javacodegeeks.java8.parallel.arrays;
02
03 import java.util.Arrays;
04 import java.util.concurrent.ThreadLocalRandom;
05
06 public class ParallelArrays {
07     public static void main( String[] args ) {
08         long[] arrayOfLong = new long [ 20000 ];
09
10         Arrays.parallelSetAll( arrayOfLong,
11             index -> ThreadLocalRandom.current().nextInt( 1000000 ) );
12         Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
13             i -> System.out.print( i + " " ) );
14         System.out.println();
15
16         Arrays.parallelSort( arrayOfLong );
17         Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
18             i -> System.out.print( i + " " ) );
19         System.out.println();
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 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**

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:

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

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

```
1 jjs func.js
```

The output on the console will be:

```
1 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**.

```
1 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**.

```
01 org.springframework.core-3.0.5.RELEASE.jar -> C:\Program Files\Java\jdk1.8.0\jre\lib\rt.jar
02 org.springframework.core (org.springframework.core-3.0.5.RELEASE.jar)
03     -> java.io
04     -> java.lang
05     -> java.lang.annotation
06     -> java.lang.ref
07     -> java.lang.reflect
08     -> java.util
09     -> java.util.concurrent
10     -> org.apache.commons.logging not found
11     -> org.springframework.asm not found
12     -> org.springframework.asm.commons not found
13 org.springframework.core.annotation (org.springframework.core-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>

- Why is Java 8, new features of Java 8? : <http://blog.danigupda.me/2014/03/why-is-java-8-new-features-javac7-excellent-computer-science-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/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>

Tagged with: JAVA 8 LAMBDA ULTIMATE

## Do you want to know how to develop your skillset to become a **Java Rockstar**?

Subscribe to our newsletter to start Rocking **right now!**

To get you started we give you our best selling eBooks for **FREE!**

1. JPA Mini Book
2. JVM Troubleshooting Guide
3. JUnit Tutorial for Unit Testing
4. Java Annotations Tutorial
5. Java Interview Questions
6. Spring Interview Questions
7. Android UI Design

and many more ....

**Email address:**

[Sign up](#)

## 44 COMMENTS



*covuslic*

May 11th, 2014 at 10:38 pm

Chap. 2.1) final String separator  
Aren't strings by definition final anymore in Java 8?

[Reply](#)



*Jesper*

May 12th, 2014 at 6:22 pm

They are immutable. The final keyword applies to the variable containing the reference to the string object.

[Reply](#)



*Andrey Redko*

May 14th, 2014 at 1:19 am

Exactly, thanks a lot. In the context of declaration, the 'final' keyword has no relation to a type (String, int,...) but to a variable declaration only: the variable cannot be reassigned or point to another value.

[Reply](#)

Great job Andrey!

Reply



*Andrey Redko*  
May 14th, 2014 at 1:17 am

Thanks a lot!

Reply



*Andrey Redko*  
May 14th, 2014 at 1:30 am

Thank a lot! I hope you found something useful for you!

Reply



*Andrey Redko*  
May 13th, 2014 at 2:05 am

Hi covuslic,

In the context of declaration, the 'final' keyword has no relation to a type (String, int,...) but to a variable declaration only: the variable cannot be reassigned or point to another value.

Thank you.  
Andrey Redko

Reply



*Enrique*  
May 17th, 2014 at 12:23 am

Thank you very useful article!

Reply



*Doogie78*  
May 20th, 2014 at 11:51 am

Thank you for all the information on Java 8.

Reply



*Tirthal*  
July 29th, 2014 at 1:42 pm

This is indeed the ultimate guide on Java 8. Thank you Andrey.

Reply



*yumi*  
August 7th, 2014 at 5:47 pm

Hi, I really like your article. If you don't mind, I'd like to translate your article into Korean. Please let me know that. Thank you.

Reply



*Andrey Redko*  
August 10th, 2014 at 12:13 am

Hi Yumi,

Thank you for this great chance for the article to be translated into Korean. Although I am the author, the article has been published on behalf of JCG. I have contacted the JCG and they are willing to grant permission for such a translation. If you don't mind, please contact Ilias Tsagklis at [ilias.tsagklis@javacodegeeks.com](mailto:ilias.tsagklis@javacodegeeks.com) to received the official rights.

Thank you very much.

Best Regards,  
Andrey Redko

Reply



*Yumi*  
September 14th, 2014 at 8:59 am

Hi Andrey Redko,

I just published this article that I translated into Korean on my blog.

I sent an email as you mentioned before and got an permisson translation into Korean.

Best Regards,  
Yumi

Reply



*Andrey Redko*  
September 14th, 2014 at 3:57 pm

Hi Yumi,  
  
Thank you very much!  
I have browsed through the article, excellent job, thank you!  
I really hope it will be an interesting reading for the Korean developers!  
Thank you!  
  
Best Regards,  
Andrey Redko

Reply



*Andrey Redko*  
August 10th, 2014 at 12:12 am

Hi Yumi,  
  
Thank you for this great chance for the article to be translated into Korean. Although I am the author, the article has been published on behalf of JCG. I have contacted the JCG and they are willing to grant the permission for such a translation. If you don't mind, please contact Ilias Tsagklis at [ilias.tsagklis@javacodegeeks.com](mailto:ilias.tsagklis@javacodegeeks.com) to receive the official rights.  
  
Thank you very much.  
  
Best Regards,  
Andrey Redko

Reply



*Tim*  
August 21st, 2014 at 3:07 pm

Hi,  
  
Calculating the percentages in the "Streams" section: you forgot to cast "points" to double, otherwise you divide a long by a long and end up with a long, which is 0 for each task.  
  
Grtz

Reply



*Andriy Redko*  
August 23rd, 2014 at 2:10 am

Hi Tim,  
  
Thank you very much for your comment. I would like to argue that casting is not necessary as 'point' (long) is divided by 'totalPoints' (double), which by Java type conversion rules should pick 'double' as the resulting type. Would you mind please to give me a bit more details with code snippet and the issue you are having, I will be happy to address it.  
  
Thank you a lot!  
  
Best Regards,  
Andriy Redko

Reply



*JK*  
October 9th, 2014 at 7:18 am

Lambda expression examples –  
<http://jeet-software.blogspot.in/2014/10/java-8-features.html>

Reply



*KC*  
January 20th, 2015 at 7:09 am

Nice guide.  
  
Section 2.2 has "Defaulable" as typo repeated several times. I believe it should be "Defaultable"

Reply



*Kiran*  
February 4th, 2015 at 5:37 pm

@Andriy Redko In 2.2 section: is private modifier allowed for interface and also for static class ?

```

default String notRequired(){
    return "Default implementation";
}

private static class DefaultableImpl implements Defaultable{
}
private static class OverridableImpl implements Defaultable{

@Override
public String notRequired(){
    return "Overridden implementation";
}
}
}

```

[Reply](#)


*Andrey Redko*  
February 28th, 2015 at 10:07 pm

Hi Kiran,

Sorry for late response. My apologies for leaving off this detail but private modifier is allowed for interfaces and static classes when they are defined inside another class (inner interfaces / classes). For example:

```

public class DefaultMethods {
    private interface Defaultable {
        default String notRequired(){
            return "Default implementation";
        }
    }
}

```

Aside from that, private modifier is not allowed for interfaces (and for static classes as well as those could be defined only as inner classes).

Thank you.

Best Regards,  
Andriy Redko

[Reply](#)


*J.W*  
March 9th, 2015 at 6:00 pm

Excellent article!

[Reply](#)


*Ajaya*  
March 18th, 2015 at 1:31 am

Need the link for other java e-books e.g. "Java Annotations Tutorial"

[Reply](#)


*Aslam*  
March 18th, 2015 at 3:26 pm

How can I download this tutorial in pdf?  
I have already signed up.

[Reply](#)


*Rahul S*  
April 3rd, 2015 at 12:00 pm

Please let me know how to download this e-book?

[Reply](#)

( 0 )



*rocks*  
April 20th, 2015 at 1:33 am

Thank a ton for putting all these wonderful feature together, I am using java from last 9 years and last year I got attracted toward Python for its concise code no boilerplate code, but looks like JDK8 Got nice features to keep its developers sticks to java...

[Reply](#)


*Velu*  
April 21st, 2015 at 5:48 pm

Hi Andrey

Great work friend. all your examples are straight forward and simple enough to understand. Also you manage to cover lot of features.

[Reply](#)

*Paiker*  
May 26th, 2015 at 1:16 pm

Hi,  
i am trying to download this complete article as PDF but unable to see the link to download. Can anyone help me please.  
Thanks,  
Paiker

[Reply](#)

*Sandip Mahanta*  
June 2nd, 2015 at 6:48 am

Well written. Very easy to understand. Great job!

[Reply](#)

*Mats*  
June 21st, 2015 at 3:35 pm

Just subscribe to the newsletter and you will be notified via your email address. After you confirm your address you will receive another email with the download links.

[Reply](#)

*Pawel*  
June 25th, 2015 at 3:56 pm

Hi,  
The article is nice but there is one feature that I don't like. Illustrating the java 8 new possibilities with examples an author uses another features that haven't been yet even mentioned and it causes confusion.  
Best regards,  
Pawel

[Reply](#)

*Andriy Redko*  
June 25th, 2015 at 11:44 pm

H Pawel,  
Thank you very much for the feedback. Would you mind please to give the examples where confusion happens so the mistakes like that won't appear anymore?  
Thank you a lot.  
Best Regards,  
Andriy Redko

[Reply](#)

*Selvan Chinnathambi*  
September 4th, 2015 at 2:22 am

great summary

[Reply](#)

*harry*  
October 27th, 2015 at 12:24 pm

thanks for the summary. But where is the PDF?

[Reply](#)

*Praful Prajapati*  
November 18th, 2015 at 2:22 pm

New Features in Java 8  
<http://javasoftwaretchnology.blogspot.in/2015/11/whats-new-in-java-8.html>

[Reply](#)

*Praful Prajapati*  
December 8th, 2015 at 2:07 pm





*Ivan*  
February 18th, 2016 at 3:21 pm

Hello.  
Not able to figure it out how to use supplier for generics.  
Class::new when creating generic type.  
Is any example or specification?  
THX

Reply



*Andriy Redko*  
February 19th, 2016 at 11:33 pm

Hi Ivan,  
Thank you for the comment. As always, there are a limitations with genetics in Java.  
You cannot use T::new or something like that to create a new instance of the generic class.  
Thanks.  
Best Regards,  
Andriy Redko

Reply



*Sandor Csete*  
September 21st, 2016 at 4:05 pm

Where is the PDF Guys?

Reply



*Eleftheria Drosopoulou*  
September 24th, 2016 at 11:33 pm

You need to join our newsletter here: <http://eepurl.com/xRlkj>  
After you enter a VALID email address, you will need to verify your subscription.  
A verification email will be sent to you and after you successfully verify your email,  
you will get a final welcome email which will contain the links to download the ebooks!

Reply



*Yasin*  
October 28th, 2016 at 12:35 pm

Hi Andriy,  
Thanks for all your efforts in creating this guide. Really enjoyed reading this post. It's really helpful .  
Regards,  
Yasin

Reply



*boriifan*  
April 27th, 2017 at 11:14 pm

Thank you alot

Reply



*Hector Fontanez*  
July 27th, 2017 at 7:52 pm

Section 2.1 reads: "In case lambda's body is more complex, it may be wrapped into square brackets". It should read CURLY (not square) brackets.

Reply



*Andriy Redko*  
July 30th, 2017 at 5:14 am

Thank you very much. That is absolutely correct. My apologies for the confusion.

Reply

## LEAVE A REPLY

Your email address will not be published. Required fields are marked \*

<https://www.javacodegeeks.com/2014/05/java-8-features-tutorial.html>

Name \*

Email \*

Website



Sign me up for the newsletter!

 Receive Email Notifications?

Or, you can subscribe without commenting.

Post Comment

## KNOWLEDGE BASE

[Courses](#)
[Examples](#)
[Resources](#)
[Tutorials](#)
[Whitepapers](#)

## PARTNERS

[Mkyong](#)

## THE CODE GEEKS NETWORK

[.NET Code Geeks](#)
[Java Code Geeks](#)
[System Code Geeks](#)
[Web Code Geeks](#)

## HALL OF FAME

["Android Full Application Tutorial" series](#)
[11 Online Learning websites that you should check out](#)
[Advantages and Disadvantages of Cloud Computing – Cloud computing pros and cons](#)
[Android Google Maps Tutorial](#)
[Android JSON Parsing with Gson Tutorial](#)
[Android Location Based Services Application – GPS location](#)
[Android Quick Preferences Tutorial](#)
[Difference between Comparator and Comparable in Java](#)
[GWT 2 Spring 3 JPA 2 Hibernate 3.5 Tutorial](#)
[Java Best Practices – Vector vs ArrayList vs HashSet](#)

## ABOUT JAVA CODE GEEKS

JCGs (Java Code Geeks) is an independent online community focused on ultimate Java to Java developers resource center; targeted at the technical team lead (senior developer), project manager and junior developers. JCGs serve the Java, SOA, Agile and Telecom communities with daily new domain experts, articles, tutorials, reviews, announcements, code snippets, source projects.

## DISCLAIMER

All trademarks and registered trademarks appearing on Java Code Geeks are the property of their respective owners. Java is a trademark or registered trademark of Oracle Corporation in the United States and other countries. Examples JCGs is not connected to Oracle Corporation and is not sponsored by Oracle Corporation.