(/)

New Features in Java 9

Last updated: January 16, 2024

Written by: baeldung (https://www.baeldung.com/author/baeldung)

Core Java (https://www.baeldung.com/category/java/core-java)

>= Java 8 (https://www.baeldung.com/tag/jdk8-and-later)

This article is part of a series:

1. Overview

Java 9 comes with a rich feature set. Although there are no new language concepts, new APIs and diagnostic commands will definitely be interesting to developers.

In this writeup we're going to have quick, high level look at some of the new features; a full list of new features is available here (http://openjdk.java.net/projects/jdk9/).

2. Modular System – Jigsaw Project

Let's start with the big one – bringing modularity into the Java platform.

A modular system provides capabilities similar to OSGi framework's system. Modules have a concept of dependencies, can export a public API and keep implementation details hidden/private.

One of the main motivations here is to provide modular JVM, which can run on devices with a lot less available memory. The JVM could run with only those modules and APIs which are required by the application. Check out this link (http://cr.openjdk.java.net/~mr/jigsaw/ea/module-summary.html) for a description of what these modules are.

Also, JVM internal (implementation) APIs like *com.sun.** are no longer accessible from application code.

Simply put, the modules are going to be described in a file called *module-info.java* located in the top of java code hierarchy:

```
module com.baeldung.java9.modules.car {
    requires com.baeldung.java9.modules.engines;
    exports com.baeldung.java9.modules.car.handling;
}
```

Our module *car* requires module *engine* to run and exports a package for *handling*.

For more in-depth example check OpenJDK Project Jigsaw: Module System Quick-Start Guide (http://openjdk.java.net/projects/jigsaw/quick-start).

3. A New HTTP Client

A long-awaited replacement of the old *HttpURLConnection*.

The new API is located under the *java.net.http* package.

It should support both HTTP/2 protocol (https://http2.github.io/) and WebSocket (https://en.wikipedia.org/wiki/WebSocket) handshake, with performance that should be comparable with the Apache HttpClient (https://hc.apache.org/httpcomponents-client-ga/), Netty (http://netty.io) and Jetty (http://eclipse.org/jetty).

Let have a look at this new functionality by creating and sending a simple HTTP request.

Update: The HTTP Client JEP (http://openjdk.java.net/jeps/110) is being moved to incubator module, so it is no longer available in the package *java.net.http* and instead is available under *jdk.incubator.http.*

3.1. Quick GET Request

The API uses the Builder pattern, which makes it really easy for quick use:

```
HttpRequest request = HttpRequest.newBuilder()
  .uri(new URI("https://postman-echo.com/get"))
  .GET()
  .build();

HttpResponse<String> response = HttpClient.newHttpClient()
  .send(request, HttpResponse.BodyHandler.asString());
```

4. Process API

The process API has been improved for controlling and managing operating-system processes.

4.1. Process Information

The class *java.lang.ProcessHandle* contains most of the new functionalities:

```
ProcessHandle self = ProcessHandle.current();
long PID = self.getPid();
ProcessHandle.Info procInfo = self.info();

Optional<String[]> args = procInfo.arguments();
Optional<String> cmd =  procInfo.commandLine();
Optional<Instant> startTime = procInfo.startInstant();
Optional<Duration> cpuUsage = procInfo.totalCpuDuration();
```

The *current* method returns an object representing a process of currently running JVM. The *Info* subclass provides details about the process.

4.2. Destroying Processes

Now – let's stop all the running child processes using *destroy()*:

```
childProc = ProcessHandle.current().children();
childProc.forEach(procHandle -> {
    assertTrue("Could not kill process " + procHandle.getPid(),
procHandle.destroy());
});
```

5. Small Language Modifications

5.1. Try-With-Resources

In Java 7, the *try-with-resources* syntax requires a fresh variable to be declared for each resource being managed by the statement.

In Java 9 there is an additional refinement: if the resource is referenced by a final or effectively final variable, a try-with-resources statement can manage a resource without a new variable being declared:

```
MyAutoCloseable mac = new MyAutoCloseable();
try (mac) {
    // do some stuff with mac
}

try (new MyAutoCloseable() { }.finalWrapper.finalCloseable) {
   // do some stuff with finalCloseable
} catch (Exception ex) { }
```

5.2. Diamond Operator Extension

Now we can use diamond operator in conjunction with anonymous inner classes:

```
FooClass<Integer> fc = new FooClass<>(1) { // anonymous inner class
};

FooClass<? extends Integer> fc0 = new FooClass<>(1) {
    // anonymous inner class
};

FooClass<?> fc1 = new FooClass<>(1) { // anonymous inner class
};
```

5.3. Interface Private Method

Interfaces in the upcoming JVM version can have *private* methods, which can be used to split lengthy default methods:

```
interface InterfaceWithPrivateMethods {

    private static String staticPrivate() {
        return "static private";
    }

    private String instancePrivate() {
        return "instance private";
    }

    default void check() {
        String result = staticPrivate();
        InterfaceWithPrivateMethods pvt = new
InterfaceWithPrivateMethods() {
            // anonymous class
        };
        result = pvt.instancePrivate();
    }
}}
```

6. JShell Command Line Tool

JShell is read–eval–print loop – REPL for short.

Simply put, it's an interactive tool to evaluate declarations, statements, and expressions of Java, together with an API. It is very convenient for testing small code snippets, which otherwise require creating a new class with the *main* method.

The *jshell* executable itself can be found in *<JAVA_HOME>/bin* folder:

```
jdk-9\bin>jshell.exe
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro
jshell> "This is my long string. I want a part of it".substring(8,19);
$5 ==> "my long string"
```

The interactive shell comes with history and auto-completion; it also provides functionality like saving to and loading from files, all or some of the written statements:

```
jshell> /save c:\develop\JShell_hello_world.txt
jshell> /open c:\develop\JShell_hello_world.txt
Hello JShell!
```

Code snippets are executed upon file loading.

7. JCMD Sub-Commands

Let's explore some of the new subcommands in *jcmd* command line utility. We will get a list of all classes loaded in the JVM and their inheritance structure.

In the example below we can see the hierarchy of *java.lang.Socket* loaded in JVM running Eclipse Neon:

```
jdk-9\bin jcmd 14056 VM.class_hierarchy -i -s java.net.Socket
14056:
java.lang.Object/null
|--java.net.Socket/null
|  implements java.io.Closeable/null (declared intf)
|  implements java.lang.AutoCloseable/null (inherited intf)
|  |--
org.eclipse.ecf.internal.provider.filetransfer.httpclient4.CloseMonitorin
gSocket
|  |  implements java.lang.AutoCloseable/null (inherited intf)
|  |  implements java.io.Closeable/null (inherited intf)
|  |--javax.net.ssl.SSLSocket/null
|  |  implements java.lang.AutoCloseable/null (inherited intf)
|  |  implements java.io.Closeable/null (inherited intf)
```

The first parameter of *jcmd* command is the process id (PID) of the JVM on which we want to run the command.

Another interesting subcommand is *set_vmflag*. We can modify some JVM parameters online, without the need of restarting the JVM process and modifying its startup parameters.

You can find out all the available VM flags with subcommand *jcmd 14056 VM.flags -all*

8. Multi-Resolution Image API

The interface *java.awt.image.MultiResolutionImage* encapsulates a set of images with different resolutions into a single object. We can retrieve a resolution-specific image variant based on a given DPI metric and set of image transformations or retrieve all of the variants in the image.

The *java.awt.Graphics* class gets variant from a multi-resolution image based on the current display DPI metric and any applied transformations.

The class *java.awt.image.BaseMultiResolutionImage* provides basic implementation:

```
BufferedImage[] resolutionVariants = ....
MultiResolutionImage bmrImage
  = new BaseMultiResolutionImage(baseIndex, resolutionVariants);
Image testRVImage = bmrImage.getResolutionVariant(16, 16);
assertSame("Images should be the same", testRVImage,
resolutionVariants[3]);
```

9. Variable Handles

The API resides under *java.lang.invoke* and consists of *VarHandle* and *MethodHandles*. It provides equivalents of *java.util.concurrent.atomic* and *sun.misc.Unsafe* operations upon object fields and array elements with similar performance.

With Java 9 Modular system access to *sun.misc.Unsafe* will not be possible from application code.

10. Publish-Subscribe Framework

The class *java.util.concurrent.Flow* provides interfaces that support the Reactive Streams (http://www.reactive-streams.org/) publish-subscribe framework. These interfaces support interoperability across a number of asynchronous systems running on JVMs.

We can use utility class *SubmissionPublisher* to create custom components.

11. Unified JVM Logging

This feature introduces a common logging system for all components of the JVM. It provides the infrastructure to do the logging, but it does not add the actual logging calls from all JVM components. It also does not add logging to Java code in the JDK.

The logging framework defines a set of *tags* – for example, *gc*, *compiler*, *threads*, etc. We can use the command line parameter *-Xlog* to turn on logging during startup.

Let's log messages tagged with 'gc' tag using 'debug' level to a file called 'gc.txt' with no decorations:

```
java -Xlog:gc=debug:file=gc.txt:none ...
```

-Xlog:help will output possible options and examples. Logging configuration can be modified runtime using *jcmd* command. We are going to set GC logs to info and redirect them to a file – gc_logs:

```
jcmd 9615 VM.log output=gc_logs what=gc
```

12. New APIs

12.1. Immutable Set

java.util.Set.of() – creates an immutable set of a given elements. In Java 8 creating a Set of several elements would require several lines of code. Now we can do it as simple as:

```
Set<String> strKeySet = Set.of("key1", "key2", "key3");
```

The *Set* returned by this method is JVM internal class: *java.util.ImmutableCollections.SetN*, which extends public *java.util.AbstractSet*. It is immutable – if we try to add or remove elements, an *UnsupportedOperationException* will be thrown.

You can also convert an entire array into a *Set* with the same method.

12.2. Optional to Stream

java.util.Optional.stream() gives us an easy way to you use the power of Streams on Optional elements:

```
List<String> filteredList = listOfOptionals.stream()
  .flatMap(Optional::stream)
  .collect(Collectors.toList());
```

13. Conclusion

Java 9 will come with a modular JVM and lots of other new and diverse improvements and features.

You can find the source code for the examples over on GitHub (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-9-new-features).

Next »
New Features in Java 10 (/java-10-overview)

« Previous
New Features in Java 8 (/java-8-new-features)

(/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON SERIES (/JACKSON)
APACHE HTTPCLIENT SERIES (/HTTPCLIENT-SERIES)
REST WITH SPRING SERIES (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE SERIES (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE SERIES (/SPRING-REACTIVE-SERIES)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS/)
PARTNER WITH BAELDUNG (/PARTNERS/WORK-WITH-US)
EBOOKS (/LIBRARY/)
FAQ (HTTPS://WWW.BAELDUNG.COM/LIBRARY/FAQ)
BAELDUNG PRO (/MEMBERS/)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)