What's New in Java 11

Introduction

For decades, Java has been one of the most commonly used programming languages in enterprise software, and its release train is moving faster than ever. Starting with Java 9, released in 2017, the release cadence has been changed to every six months. Java 10 added new features to the platform. For many users, however, the changes were not substantial enough to justify the migration effort to Java 10. But Java 11 goes beyond the typical new features and bug fixes.

As a Java developer, it's difficult to keep up with the latest and greatest developments included in every single release. This report explains the benefits of upgrading to Java 11 and gives you a swift but condensed and insightful overview of its compelling features.

Migrating from an earlier version of Java might not be as straightforward as you would expect. Java 11 removes a variety of technologies and APIs from the platform. In this report, you learn about migration strategies that help you to get up to speed as quickly as possible. But, despite potential obstacles, the new features in Java 11 make upgrading worth it.

Convergence of Oracle JDK and OpenJDK

OpenJDK, the open source counterpart of the Oracle JDK, has been around since Java 7. For the most part, language features, core Java libraries, and the tooling of both distributions are very similar, if not exactly the same. In addition to those aspects, Oracle JDK versions before Java 11 include commercial features like Flight Recorder, Java Mission Control, and the Z Garbage Collector that could be activated by passing in the `-XX:+UnlockCommercialFeatures` option when starting up a new JVM. With Java 11, commercial features become free of charge and have been added to OpenJDK. We discuss some of those features in a later section.

Oracle's long-term goal was to converge both distributions by making the commercial tooling fully open source. With Java 11, this milestone has been reached along with some minor packaging and cosmetic differences (http://bit.ly/2zV5i8P). So, which distribution should you pick for your project, you might ask? Let's have a look at the license implications for each option.

Licensing of the Oracle JDK and OpenJDK

There's a single, distinguishing factor between the Oracle JDK and OpenJDK: the license agreement. The OpenJDK continues to be licensed with the GNU General Public License, version 2 (GPL 2) with a classpath exception. This license allows you to use Java for open source projects as well as commercial products without any restrictions or attribution for development and production purposes. The classpath exception decouples the licenses terms of the OpenJDK from the licenses terms applicable to external libraries bundled with the executable program. Regarding the Oracle JDK, the license has changed from the Binary Code License (BCL) for Oracle Java SE technologies to a commercial license. As a result, you can no longer use the Oracle JDK for production without buying into the Java SE subscription model (http://bit.ly/2C3CaxJ).

> **NOTE**
>
> The subscription model also applies to any Java 8 patch release published after January 2019 if it is used for commercial products.

It's important that you understand the license implications to make a choice that's right for your organization. If you are working on mission-critical software and you'd like to rely on Oracle's support for long-term patches, the commercial Oracle JDK license is likely the best option. For open source software and products that require only short-term patch releases for the next six months, the OpenJDK should fully suffice. Either way, you will need to decide which Java distribution to use as soon as you upgrade to Java 11. Be aware that there are alternative, free distributions of the OpenJDK from other vendors. For more information, see the Java Champions' blog post "Java Is Still Free" (http://bit.ly/2LhgG3v).

Six-Month Release Cadence: What It Means to You

In the early days of Java, new releases were a big deal. They used to happen only every two to three years and contained a lot of significant, sometimes breaking, changes like Generics in Java 5, or syntax changes in Java 8, or the Java 9 module system. Migrating a software project to a new Java version often required a lot of planning and effort. Some projects even had to postpone the software production process until the migration had been fully completed.

Many industry leaders moved to a Continuous Delivery model to drive innovation and mitigate risk: fewer changes, more frequent releases. Oracle is committed to following this example by releasing a new major version every six months, and a minor release every two months. As a result, upgrading to a new version of Java will have less of an impact on your project. Figure 1-1 shows the expected dates of the upcoming major and minor Java releases.

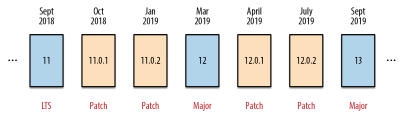


Figure 1-1. Java's projected release cycle

With more frequent releases comes the responsibility of ensuring platform stability. For that very reason, Oracle provides a long-term support release every three years. Next, we have a look at what "long-term support" means in practice.

Long-Term Support

Oracle offers lifetime Premier Support for all major Java versions that have been designated a long-term support (LTS) release. The benefit of an LTS release is that paying customers will receive periodic patch updates even after the next major version has been released. For example, the next LTS release after Java 11 will be Java 17, which is planned for September 2021. In the meantime, you will continue to receive new releases, as explained in the previous section. For any major Java version, OpenJDK users will receive only two more patch releases. After that, it is assumed that you will need to upgrade to the next major version.

NOTE

On October 30, 2018, Amazon announced free long-term support for OpenJDK 8 and OpenJDK 11 Java runtimes in Amazon Linux 2, a Linux server operating system from Amazon Web Services (AWS). On November 14, James Gosling announced Corretto, a production-ready, long-term support distribution of Java.

LTS releases are especially attractive for more conservative enterprise customers with a slower upgrade rate. You might be unsure whether you need a commercial or free license for your software projects. A safe bet is to start with one of the free OpenJDK-based distributions. If you later realize that you need commercial support, you can still switch to the Oracle JDK and pay for the license.

With the basic knowledge of Java 11 under your belt, let's dive into the new features this release has to offer. They might be small compared to other releases. Nevertheless, they can improve how you work with Java on a day-to-day basis.

New Features

For many developers, the main motivation for upgrading to a new Java version is to increase productivity by applying the latest and greatest features. In this section, you will learn about the language and platform improvements introduced with Java 11. We'll begin by looking at the feature with the biggest impact: the new HTTP client.

Built-in and Improved HTTP Communication with HttpClient

In a world of microservices and service APIs, HTTP communication is inevitable. It's common to need to write code that makes a call to an endpoint to retrieve or modify data. `HttpURLConnection`, an API for HTTP communication, has been around for ages but couldn't keep up with the requirements of modern-day applications. As a result, in the past, Java developers had to resort to more advanced, feature-rich libraries like Apache HttpComponents or Square's OKHttp, which already supported HTTP/2 and WebSocket communication.

Oracle recognized this shortcoming in Java's feature set and introduced an HTTP client implementation as an experimental feature with Java 9. `HttpClient` (http://bit.ly/2Pwl6DL) has grown up and is now a final feature in Java 11. If there's one reason to upgrade to Java 11, this is it! No more external dependencies that you need to maintain as part of your build process.

In a nutshell, Java's HTTP API can handle synchronous and asynchronous communication and supports HTTP 1.1 and 2 as well as all common HTTP methods like GET, POST, PUT, and DELETE. A typical HTTP interaction with the `java.net.http` module looks like this:

1. Create an instance of `HttpClient` and configure it as needed.
2. Create an instance of `HttpRequest` and populate the information.
3. Pass the request to the client, perform the request, and retrieve an instance of `HttpResponse`.
4. Process the information contained in the `HttpResponse`.

Sounds abstract? Let's have a look at concrete examples for synchronous and asynchronous HTTP communication.

SYNCHRONOUS COMMUNICATION

First, let's write a program that makes a synchronous GET call. Figure 1-2 shows that a synchronous call blocks the client until the HTTP request has been handled by the server and the response has been sent back.

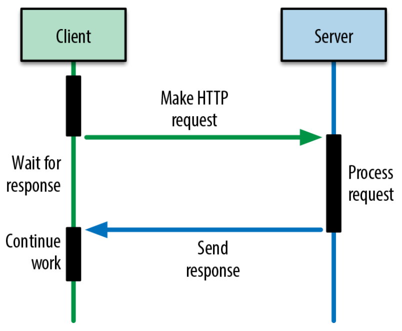


Figure 1-2. Synchronous HTTP communication

For demonstration purposes, we use Postman Echo, a RESTful service for testing HTTP clients. Example 1-1 creates an HTTP client instance, performs a GET call, and renders some of the response information on the console.

Example 1-1. Making a synchronous GET call with HttpClient API

```
import java.io.IOException;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.time.Duration;

HttpClient httpClient = HttpClient.newBuilder()
        .connectTimeout(Duration.ofSeconds(10))
        .build();                                          ❶

try {
    String urlEndpoint = "https://postman-echo.com/get";
    URI uri = URI.create(urlEndpoint + "?foo1=bar1&foo2=bar2");
    HttpRequest request = HttpRequest.newBuilder()
            .uri(uri)
            .build();                                      ❷
    HttpResponse<String> response = httpClient.send(request,
            HttpResponse.BodyHandlers.ofString());         ❸
} catch (IOException | InterruptedException e) {
    throw new RuntimeException(e);
}

System.out.println("Status code: " + response.statusCode());
System.out.println("Headers: " + response.headers()
        .allValues("content-type"));                       ❹
System.out.println("Body: " + response.body());
❹
```

❶ Creating and configuring the HTTP client instance. The client uses a connection timeout of 10 seconds.

❷ Creating and configuring the HTTP request. By default, an HTTP request uses the GET method.

❸ Sending the HTTP request and retrieving the response.

❹ Processing of the data returned by the HTTP response; for example, the status code, headers, and the body.

> **NOTE**
>
> You might have noticed that the `HttpClient` API relies heavily on the builder pattern. The builder pattern is especially useful when creating more complex objects that can be fed with optional parameters. Instead of exposing constructors with a number of parameter overloads, the builder pattern provides expressive methods for configuring the object, leading to a fluent API that's easier to understand and use.

The `HttpClient` API isn't limited to just making a synchronous call. In the following section, we explore an example for asynchronous communication.

ASYNCHRONOUS COMMUNICATION

Asynchronous communication is useful if you don't want to wait on the response and deal with the response later or if you want to process a list of calls in parallel. Figure 1-3 shows what this looks like.

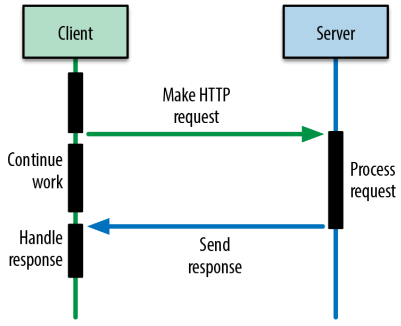


Figure 1-3. Asynchronous HTTP communication

Let's assume that you want to implement a tool for parsing links used on a web page and verifying them by running an, HTTP GET request against them. Example 1-2 defines a list of URIs and checks them by emitting an asynchronous call and collecting the results.

Example 1-2. Performing an asynchronous HTTP call and reacting to the response later

```
import java.io.IOException;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.time.Duration;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.stream.Stream;

import static java.util.stream.Collectors.toList;

final List<URI> uris = Stream.of(
        "https://www.google.com/",
        "https://www.github.com/",
        "https://www.ebay.com/"
        ).map(URI::create).collect(toList());          ❶

HttpClient httpClient = HttpClient.newBuilder()
        .connectTimeout(Duration.ofSeconds(10))
        .followRedirects(HttpClient.Redirect.ALWAYS)
        .build();

CompletableFuture[] futures = uris.stream()
        .map(uri -> verifyUri(httpClient, uri))
        .toArray(CompletableFuture[]::new);            ❷
CompletableFuture.allOf(futures).join();               ❷

private CompletableFuture<Void> verifyUri(HttpClient httpClient,
                                           URI uri) {
    HttpRequest request = HttpRequest.newBuilder()
            .timeout(Duration.ofSeconds(5))
            .uri(uri)
            .build();

    return httpClient.sendAsync(request,
            HttpResponse.BodyHandlers.ofString())
            .thenApply(HttpResponse::statusCode)
            .thenApply(statusCode -> statusCode == 200)
            .exceptionally(__ -> false)
            .thenAccept(valid -> {
                if (valid) {
                    System.out.println("[SUCCESS] Verified "
                            + uri);
                } else {
                    System.out.println("[FAILURE] Could not "
                            + "verify " + uri);
                }
            });                                        ❸
}
```

❶ Defines a list of URIs that should be verified.

❷ Verifies all URIs asynchronously and evaluates the future result.

❸ Sends an asynchronous HTTP request, checks whether the status code of the response is OK, and reacts to the outcome by printing a message.

As Examples 1-1 and 1-2 demonstrate, the new HttpClient functionality is very powerful and expressive. HttpClient has even more features in store than those mentioned here. It can also handle reactive streams, authentication, and cookies, to name a few. For more information, refer to the Oracle learning material (http://openjdk.java.net/groups/net/httpclient/) and Javadocs (http://bit.ly/2Pwl6DL).

Another interesting feature in Java 11 is the ability to launch a Java program without the need for compilation. Let's explore how this works.

Launching Single-File Programs Without Compilation

Java is not a scripting language. For every program that you'd like to execute, you need to first compile it by explicitly running the `javac` command. This roundtrip makes it less convenient to try out little snippets of code for testing purposes. Java 11 changes that. You can now execute Java source code contained in a single file without the need to compile it first, as shown in Figure 1-4.

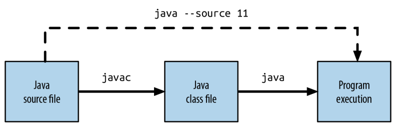


Figure 1-4. Executing a Java program with and without compilation

The new functionality is a great launching pad for beginners to the Java language and enables users to try out logic in an ad hoc fashion.

It's easy to get started. Let's pick a simple example. The file *HelloWorld.java* renders the message "Hello World" on the console via `System.out.println` as part of a `main` method:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

To execute, run the `java` command with Java 11. Java will compile the source file on the fly and run the program with the class file on the classpath. Any compilation error that might occur will be rendered on the console:

```
$ java -version
openjdk version "11" 2018-09-25
OpenJDK Runtime Environment 18.9 (build 11+28)
OpenJDK 64-Bit Server VM 18.9 (build 11+28, mixed mode)

$ java HelloWorld.java
Hello World!
```

Launching a single-file program without compilation is very convenient, but there are a couple limitations that you need to be aware of. With Java 11, the program cannot use any external dependencies other than the `java.base` module. Furthermore, you can launch only a single-file program. Calling methods from other Java source files is currently not supported, but you can define multiple classes in the same file if needed. I personally see this feature as a helpful tool for rapid prototyping.

In the next section, you learn about the new methods that have been introduced to existing library classes.

New Library Methods for Strings, Collections, and Files

All too often, Java developers must rely on external libraries like Google Guava or Apache Commons for convenience methods that they otherwise would need to copy and paste from one project to another. The implementations might be small, but they do create additional API surface that needs to be maintained and tested. Java 11 adds convenience methods to the `String` (http://bit.ly/2QILrn4), `Collections` (http://bit.ly/2SLY9ij), and `Files` (http://bit.ly/2rwSd12) APIs.

Let's explore the new functionality by example. For the fun of it, let's write some JUnit tests that apply the new APIs and verify their correct behavior for different use cases.

STRING API ENHANCEMENTS

The `String` API adds four new convenience methods. Let's have a look at each one of them by example.

Repeating a string

The first method we look at is `String.repeat(Integer)` (http://bit.ly/2C3FuJ0). Judging by the name, you might have already guessed the purpose of the functionality. It simply repeats a string *n* times.

I can think of more than one situation in which the `repeat` method could come in handy. Imagine that you need to build a data table with padded whitespaces between key/value pairs to improve its readability. The following test case demonstrates the use of the method for that very scenario:

```
@Test
void canRepeatString() {
    assertEquals("CPU Usage:                  5%",
            renderInfo("CPU Usage:", "5%"));
    assertEquals("Memory Usage:          9.14 MB",
            renderInfo("Memory Usage:", "9.14 MB"));
    assertEquals("Free Disk:             96.5 GB",
            renderInfo("Free Disk:", "96.5 GB"));
}

private String renderedInfo(String title, String value) {
    return title + " ".repeat(30 - title.length()
            - value.length()) + value;
}
```

Checking for empty or whitespaces

I am sure you have needed to implement data validation at some point of your career. Suppose that you want to verify the validity of a value entered into a web-based input field. Of course, we do not want to allow blank values.

Java 11 introduces the method `String.isBlank()` (http://bit.ly/2Bf524e), which indicates whether a string is empty or contains only whitespaces. Now that this functionality has become a Java core feature, there's no more need to pull in Apache StringUtils as external dependencies. You can see its use in the following test case:

```
@Test
void canCheckIfStringContainsWhitespaces() {
    String nameFormFieldWithoutWhitespace = "Duke";
    String nameFormFieldWithWhitespace = " ";
    assertFalse(nameFormFieldWithoutWhitespace.isBlank());
    assertTrue(nameFormFieldWithWhitespace.isBlank());
}
```

Removing leading and trailing whitespaces

Validating strings goes hand in hand with sanitizing strings. Developers often encounter data that needs to be cleaned to extract the actual information. This might involve filtering characters that are hidden, erroneous, or not useful.

The newly introduced method `String.strip()` (http://bit.ly/2RRmfYC) takes care of removing leading and trailing whitespaces. You can be even more specific by removing just the leading characters by using `String.stripLeading()`

(http://bit.ly/2BcuMhA) or just the trailing characters by using `String.stripTrailing()` (http://bit.ly/2QIM7sC). The following test case demonstrates the use of all three methods:

```
@Test
void canStripStringOfLeadingAndTrailingWhitespaces() {
    String nameFormField = "      Java Duke     ";
    assertEquals("Java Duke", nameFormField.strip());
    assertEquals("Java Duke     ", nameFormField.stripLeading());
    assertEquals("      Java Duke", nameFormField.stripTrailing()
}
```

Processing multiline strings

Processing multiline strings line by line can be tedious and memory intensive. First, you must split the string based on line terminators. Next, you need to iterate over each line. For software systems that handle the operation for large texts, memory consumption can be a big concern.

The method `String.lines()` (http://bit.ly/2SEGo47) introduced in Java 11 lets you process multiline texts as a `Stream`. A `Stream` represents a sequence of elements that can be processed sequentially without having to load all of the elements into memory at once. The following shows a test case that uses the method in practice:

```
@Test
void canStreamLines() {
    String testString = "This\nis\na\ntest";
    List<String> lines = new ArrayList<>();
    testString.lines().forEach(line -> lines.add(line));
    assertEquals(List.of("This", "is", "a", "test"), lines);
}
```

COLLECTIONS API ENHANCEMENTS

In the past, turning a collection into an array was cumbersome. You were required to provide a new instance of the array, its generic type, and the final size of the array.

Java 11 makes the conversion more convenient. Instead of passing an array instance, you now can provide a function by using the method `Collection.toArray(IntFunction<T[]>)` (http://bit.ly/2zSm7Bs). The following code example compares the old and the new way for converting a collection to an array:

```
@Test
void canTurnListIntoArray() {
    List<String> months = new ArrayList<>();
    months.add("January");
    months.add("February");
    months.add("March");
    assertArrayEquals(new String[]
            { "January", "February", "March" },
            months.toArray(new String[months.size()]));    ❶
    assertArrayEquals(new String[]
            { "January", "February", "March" },
            months.toArray(String[]::new));                ❷
}
```

❶ Converting a collection to an array pre–Java 11.

❷ Converting a collection to an array using the new method in Java 11.

FILES API ENHANCEMENTS

For developers, reading and writing files is a very common operation. Pre–Java 11 code requires you to write unnecessary boilerplate code. To work around this issue, projects will usually introduce a reusable utility method or resort to one of the many open source libraries.

Java 11 adds the methods `Files.readString(Path)` (http://bit.ly/2SFuKWN) and `Files.writeString(Path, CharSequence, OpenOption)` (http://bit.ly/2EdehoN) with various overloads, which make it much easier to read and write files. The following code snippet shows both methods in action:

```
@Test
void canReadString() throws URISyntaxException, IOException {
    URI txtFileUri = getClass().getClassLoader()
            .getResource("helloworld.txt").toURI();
    String content = Files.readString(Path.of(txtFileUri),
            Charset.defaultCharset());
    assertEquals("Hello World!", content);
}

@Test
void canWriteString() throws IOException {
    Path tmpFilePath = Path.of(
            File.createTempFile("tempFile", ".tmp").toURI());
    Path returnedFilePath = Files.writeString(tmpFilePath,
            "Hello World!", Charset.defaultCharset(),
            StandardOpenOption.WRITE);
    assertEquals(tmpFilePath, returnedFilePath);
}
```

Enhancements to Optional and Predicate

Java 8 added the `Optional` (http://bit.ly/2zS3qh4) type. This type is a single-value container that contains an optional value. Its main purpose is to avoid cluttering the code with unnecessary null checks. You can check whether a value is present within the container without actually retrieving it by calling the `isPresent()` method.

In the latest version of Java, you can also ask for the inverse: "Is the value empty?" Instead of using the negation of the `isPresent()` method, you can now just use the `isEmpty()` (http://bit.ly/2EfFNCg) method to make the code easier to understand. The following test case compares the use of both methods:

```
@Test
void canCheckOptionalForEmpty() {
    String payDay = null;
    assertTrue(!Optional.ofNullable(payDay).isPresent());   ❶
    assertTrue(Optional.ofNullable(payDay).isEmpty());      ❷
    payDay = "Monday";
    assertFalse(!Optional.ofNullable(payDay).isPresent());  ❶
    assertFalse(Optional.ofNullable(payDay).isEmpty());     ❷
}
```

❶ Checking whether a value in `Optional` is present pre–Java 11.

❷ Checking whether a value in `Optional` is present in Java 11.

A `Predicate` is often used to filter elements in a collection of objects. Java 11 adds the static method `Predicate.not(Predicate)` (http://bit.ly/2EgyZnX), which returns a predefined `Predicate` instance that negates the given `Predicate`.

Suppose that you want to filter a stream of strings, such as certain months. The new `Predicate` method can be enormously helpful with filtering all months that do not fulfill a certain condition. In the code that follows, you'll find an example that filters out all months that do not begin with the letter "M." That's far more expressive than having to pass `(Predicate<String>) month -> month.startsWith("M")).negate()` to the `filter` method.

```
@Test
void canUsePredicateNotAsFilter() {
    List<String> months = List.of("January", "February", "March")
    List<String> filteredMonths = months
            .stream()
            .filter(Predicate.not(month -> month.startsWith("M"))
            .collect(Collectors.toList());
    assertEquals(List.of("January", "February"), filteredMonths);
}
```

That's it for the core library API extensions. Java 11 also improved the handling of the `var` keyword. Next, we take a look at those changes.

USING THE VAR KEYWORD IN LAMBDAS

Starting with Java 10, you can declare local variables without having to assign the type. Instead of declaring the type, the variable uses the `var` keyword. At runtime, the assignment for the variable automatically determines the type; it is inferred. Even when using the `var` keyword, the variable is still statically typed.

With Java 11, you can now also use the `var` keyword for parameters of a Lambda. Using `var` for Lambda parameters comes with a major benefit: you can annotate the variable. For example, you could indicate that the value of the variable cannot be null by using JSR-303's `@NotNull` annotation.

Let's go back to the `Predicate.not(Predicate)` method used in an earlier example. In the following example, we enhanced the Lambda definition by providing the `var` keyword plus an annotation:

```
import javax.validation.constraints.NotNull;

@Test
void canUseVarForLambdaParameters() {
    List<String> months = List.of("January", "February", "March")
    List<String> filteredMonths = months
            .stream()
            .filter(Predicate.not((@NotNull var month) ->
                    month.startsWith("M")))
            .collect(Collectors.toList());
    assertEquals(List.of("January", "February"), filteredMonths);
}
```

Next up, let's have a look at the switch from Unicode 8 to Unicode 10.

ADOPTION OF UNICODE 10

Java 10 uses the Unicode 8 standard for localization. The Unicode standard evolves continuously, so it was about time that Java 11 follow the latest version, Unicode 10. The upgrade from Unicode 8 to 10 includes 16,018 new characters and 10 new scripts. A script is a collection of letters and other written signs used to represent textual information. For example, Unicode 10 adds Masaram Gondi (http://bit.ly/2SHI0Kn), a South-Central Dravidian language.

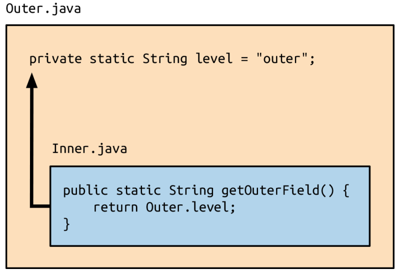
One of the long-awaited additions to Unicode 10 is the "Colbert emoji" (http://bit.ly/2GeVtYM), a face with a neutral mouth and single eyebrow raised made popular by comedian Stephen Colbert (http://bit.ly/2zZtRBr). The following example makes use of the Colbert emoji:

```
@Test
void canUseColbertEmoji() {
    String unicodeCharacter = "\uD83E\uDD28";
    assertEquals("🤨", unicodeCharacter);
}
```

Next up, we look at a lower-level change in Java 11: nest-based access control. The impact to the end user is less visible, but it addresses a long-standing technical debt that was finally refactored.

NEST-BASED ACCESS CONTROL

To understand the next change to Java 11, we need to take a look at a nested class. Let's assume that you have an outer class called `Outer.java` and an inner class called `Inner.Java` within it. `Inner.java` can access all private fields and methods of `Outer.java` because they logically belong together. Figure 1-5 visualizes the data structure.



```
private static String level = "outer";
```

```
public static String getOuterField() {
    return Outer.level;
}
```

Figure 1-5. Accessing a parent field from a nested class

As you might know, the Java compiler creates a new class file for inner classes, in this case `Outer$Inner.class`. Under the hood, the compilation process also creates so-called accessibility-broadening bridge methods between the classes.

Java 11 pays down the technical debt of having to generate those bridge methods with the help of so-called nestmates. Inner classes can now access fields and methods from outer classes without additional work from the compiler.

You might ask yourself, "Compiler optimizations are great, but how does it affect me as a developer"? Well, prior to Java 11, it was not possible to access a field from the outer class via reflection without setting the accessibility control to `true`. With the change of nest-based access control, this control is no longer necessary. The following class hierarchy demonstrates the changed behavior:

```
import java.lang.reflect.Field;

public class Outer {
    private static String level = "outer";

    public static class Inner {
        public static String getOuterViaRegularFieldAccess() {
            return Outer.level;
        }

        public static String getOuterViaReflection() {
            try {
                Field levelField = Outer.class
                        .getDeclaredField("level");
                // levelField.setAccessible(true);        ❶
                return levelField.get(null).toString();
            } catch (NoSuchFieldException
                    | IllegalAccessException e) {
                throw new RuntimeException(e);
            }
        }
    }

    public static void main(String[] args) {
        Inner.getOuterViaRegularFieldAccess();             ❷
        Inner.getOuterViaReflection();
    }
}
```

❶ Required for Java 10 or earlier.

❷ Already valid in Java 10.

As with the previous code examples in this section, we also verify the correct behavior with the help of a test, as shown in the following (for more detailed information on this change, take a look at JEP 181):

```
private final Outer.Inner inner = new Outer.Inner();

@Test
void canAccessPrivateFieldFromOuterClass() {
    assertEquals("outer", inner.getOuterViaRegularFieldAccess());
}

@Test
void canAccessPrivateFieldFromOuterClassViaReflection() {
    assertEquals("outer", inner.getOuterViaReflection());
}
```

The last developer-visible feature we explore is Java Flight Recorder (http://bit.ly/2B9sOyH), a feature that used to be available only for commercial use. Keep this tool in mind if you ever need to profile a Java application.

RECORDING OS AND JVM EVENTS WITH JAVA FLIGHT RECORDER

Java Flight Recorder (JFR) is a tool for profiling a running Java application and capturing diagnostics for further analysis. This tool is not new to Java. It was rolled out as a part of Java 7, but it was considered a commercial feature that you needed to explicitly enable by using the JVM options `-XX:+UnlockCommercialFeatures -XX:+FlightRecorder`.

In Java 11, JFR went open source and is included in the OpenJDK. To give you an impression on how to use JFR, let's have a look at the steps required to capture metrics. First, you need to run a Java program and provide the necessary VM options to enable the recording. In the following example, we use the time-based recording option set to 60 seconds:

```
$ java -XX:StartFlightRecording=duration=60s,
filename=recording.jfr,settings=profile,name=SampleRecording MyMa
Started recording 1. The result will be written to:
/Users/bmuschko/dev/projects/whats-new-in-java-11/recording.jfr
```

As you might have noticed from the console output, the profiling data has been captured in the file *recording.jfr*. It's time to have a look at it visually so that we can analyze the metrics. Meet Java Mission Control (JMC)!

JMC can inspect the data produced by JFR and visualize it in a well-arranged user interface. Be aware that JMC is not part of the JDK; you must download and install (http://bit.ly/2C5QQwd) it independently.

NOTE

JFR recordings are not backward compatible. You will need an early access version of JMC 7 to inspect a recording produced with Java 11.

To begin, simply point JMC to the recording file produced by JFR. Figure 1-6 depicts JMC rendering the memory consumption of a sample application. I am not going to go further into detail on the functionality of JFR and JMC. There's plenty of information available online if you plan to use the tools for your own projects.

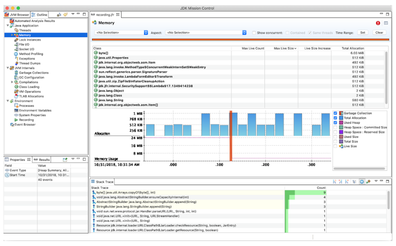


Figure 1-6. JMC rendering recorded memory consumption

That's it for the new features. I am sure you will find that the new functionalities that we've discussed here will help you to become more productive in the future. Next, we discuss the features and APIs that have been removed with Java 11.

Removed Features and APIs

Upgrading a project to a newer version of Java sounds easy in theory. But in practice, there are various factors that complicate the upgrade process. One example is the continued use of deprecated features and APIs. A good practice is to periodically review your code, identify the usage of deprecated functionality, and find a replacement as early as possible.

Java 11 removes features and APIs that have been marked deprecated in earlier versions. In this section, we review each of them and propose alternative options where applicable. This section also serves to provide information that can help you assess how a removal might affect your project.

Removed Java EE Modules

Java 9 started the process of modularizing the JDK. While modularizing Java, it became abundantly clear that some technologies and APIs belonged to Java EE instead of Java SE. The Java team deprecated those modules and remove them with the next LTS version. Thus, with the release of Java 11, these technologies are no longer a part of the JDK. Table 1-1 lists the removed modules.

Table 1-1. Modules removed from Java 11

Module	**Packages**
JavaBeans Activation Framework (JAF)	`javax.activation`
Java Transaction API (JTA)	`java.transaction`
Commons Annotation API	`java.xml.ws.annotation`
Java Architecture for XML Binding (JAXB)	`jdk.xml.bind`
Java API for XML-Based Web Services (JAX-WS)	`jdk.xml.ws`
CORBA	`javax.activity, javax.rmi, org.omg`

So, what will happen if you use one of these modules in your code and directly upgrade to Java 11? Your code will no longer compile, or you could run into runtime errors. Although prior versions allowed for a workaround by manually adding the module using `--add-modules`, this no longer works for Java 11.

Your best bet is to find a replacement for the functionality available as an external dependency and add it to your compilation classpath. You will find that many of the removed modules have a representation on Maven Central. Table 1-2 lists some possible replacements. For additional references, refer to JEP 320.

Table 1-2. Replacement libraries for removed modules in Java 11

Module	Replacement on Maven Central
JavaBeans Activation Framework (JAF)	`javax.activation:javax.activation-api` (http://bit.ly/2QqxOt5)
Java Transaction API (JTA)	`javax.transaction:javax.transaction-api` (http://bit.ly/2Lc2U1l)
Commons Annotation API	`javax.annotation:javax.annotation-api` (http://bit.ly/2B99CBc)
Java Architecture for XML Binding (JAXB)	`org.glassfish.jaxb:jaxb-runtime` (http://bit.ly/2Pu5CQJ)
Java API for XML-Based Web Services (JAX-WS)	`com.sun.xml.ws:jaxws-ri` (http://bit.ly/2B7Uowg)
CORBA	`org.glassfish.corba:glassfish-corba` (http://bit.ly/2UuupIl)

To illustrate a migration scenario, let's assume that you are working on a project that needs to process XML. A portion of the code base uses JAXB to translate a POJO and its field values to XML. In the following listing, you can find such a POJO including JAXB-typical annotations:

```
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "person")
public class Person {
    private String name;

    @XmlElement(name = "fullName")
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Compiling the class with Java 11 would result in a similar error, shown in the following snippet. The JAXB classes have been removed from the JDK:

```
/Users/bmuschko/dev/projects/whats-new-in-java-11/src/main/java/
com/bmuschko/java11/removals/jaxb/Person.java
Error:(3, 33) java: package javax.xml.bind.annotation
does not exist
Error:(4, 33) java: package javax.xml.bind.annotation
does not exist
Error:(6, 2) java: cannot find symbol
  symbol: class XmlRootElement
```

To fix the compilation issue, you need to add an external dependency to your build file. The following XML snippet demonstrates the use of the Glassfish JAXB runtime dependency in a Maven *pom.xml* file:

```
<dependencies>
    <dependency>
        <groupId>org.glassfish.jaxb</groupId>
        <artifactId>jaxb-runtime</artifactId>
        <version>2.3.1</version>
    </dependency>
</dependencies>
```

Of course, you can add the same dependency to a Gradle build script. The following sample build script declares the dependency in the context of a Java library project:

```
plugins {
    id 'java-library'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.glassfish.jaxb:jaxb-runtime:2.3.1'
}
```

In addition to the removed Java EE modules, Java 11 also removed several APIs. Because the impact of these removals on the end user is relatively small, we don't go into further detail about these here. For more information, check out the list of removed APIs (http://bit.ly/2SLYXnl).

Applets and Java Web Start

Remember those days when we used to think of Java primarily in the context of Applets? Well, that time has long passed. Today, Java is used in many other contexts, such as server applications, mobile, and Internet of Things (IoT) devices.

Fast-forward to 2018, and browser providers have either begun to remove or have already removed support for plug-in–based technologies like Flash and Java because of performance and security concerns. Java 9 followed the trend and deprecated support for Applets in 2017. Java 11 closes the circle and removes Applets completely without a replacement.

Java Web Start provides a way to download a Java desktop application from a browser instead of running it *in* the browser. To many users, it was somewhat of a surprise that Java 11 also removes support for Java Web Start. For that reason, converting an Applet to a Java Web Start application is out of the question.

If you are still maintaining applications based on Applets or Java Web Start, you will need to look into a replacement if you are planning to upgrade to Java 11. The next section touches on JavaFX, a valid and powerful option for desktop application development.

JavaFX Migrates to OpenJFX

As of Java 11, JavaFX—the technology for building Java client applications—is no longer distributed with the Oracle JDK. But this doesn't mean that the project is dead. It was simply reduced to OpenJFX, the open source counterpart of JavaFX. Consequently, OpenJFX can release new versions independent of the JDK release cycle. Especially with the involvement of the wider Java community, this is a big win. We can likely expect more frequent releases.

OpenJFX provides a very approachable and detailed documentation page, as shown in Figure 1-7. You can integrate the technology with Maven and Gradle build processes. Getting started with OpenFX couldn't be easier.

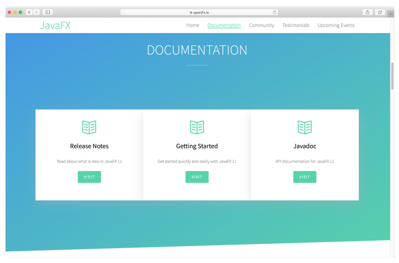


Figure 1-7. The community-driven home of OpenJFX

As with previous Java versions, features and APIs go through a deprecation cycle before they are actually removed, which we discuss in the next section.

Deprecated Features and APIs

Java 11 deprecates the Nashhorn JavaScript engine, some VM options, and support for the compression scheme Pack200. I strongly advise incorporating the recommended replacements as soon as possible to ensure a smooth upgrade procedure to a post–Java 11 version. Let's go through them one by one.

NOTE

It's difficult to keep track of all of the Java versions, especially if you are making the jump across multiple versions. The tool `jdeps` (http://bit.ly/2QnU6Ma) helps with detecting the use of deprecated APIs in your project code and its dependencies. Integrating `jdeps` as part of your Continuous Integration process is a great idea. You will receive fast feedback and gain deep insight into the use of deprecated APIs.

Nashorn JavaScript Engine

The Java team is on a mission to slim down the JDK. The next candidate for removal is Nashorn, the JavaScript engine. One of the main motivations for deprecating Nashorn is the rapid pace at which ECMAScript language constructs and their APIs change. Consequently, it's challenging to maintain compatibility while at the same time ensuring that new features are added.

Instantiating the Nashorn engine still works just fine, as shown in the following code snippet:

```
import javax.script.ScriptEngineManager;
new ScriptEngineManager().getEngineByName("nashorn");
```

The only thing you'd see is a warning message indicating that the feature will be removed in a future version of Java:

```
Warning: Nashorn engine is planned to be removed from
a future JDK release
```

You might have noticed that the message is not very specific on which version will make the removal effective. As mentioned by Thomas Wuerthinger (http://bit.ly/2A2NzfV), senior research director at Oracle Labs, this will probably happen as soon as GraalVM is considered production ready.

Although the deprecation of the Nashorn JavaScript engine probably has the greatest impact on user projects, there are some JVM options and other tools that have been marked for later removal that are worth mentioning.

VM Options and Other Tools

There are two VM options worth mentioning that have been deprecated with Java 11, one of which we already touched on in "Recording OS and JVM events with Java Flight Recorder". Before Java 11, commercial features were available only for paying customers. All of those tools have been open sourced, so there is no longer any need for the flags `-XX:+UnlockCommercialFeatures` and `-XX:+LogCommercialFeatures`. If you use them with Java 11, both flags will generate a warning message when running a Java program.

Another flag that became obsolete is `-XX:+AggressiveOpts`. This flag was originally intended to activate experimental features of the C2 compiler. Now, those features have been either fully integrated or removed.

One more tool and API that I would like to mention for the sake of completeness is Pack200 (http://bit.ly/2UCad7a). *Pack200* is a compression scheme for JAR files that was introduced in Java 5. The compression mechanism was especially popular in the context of Applets. As mentioned earlier, support for Applets has been dropped, thus making this tool less useful. For that very reason, Java marked it as deprecated. For more information, refer to JEP 336 (http://openjdk.java.net/jeps/336).

The last but very important section in this report covers performance and security.

Performance and Security

Developers usually look for the shiny, convenient features in a new Java release that directly affect and improve their daily work routines. Nonfunctional aspects like performance and security, on the other hand, become extremely important after you release your project to production. A slow-running application won't win over customers. Security holes in your runtime environment lead to diminished trust with your users. Java 11 can deliver on this front, as well. First, we take a look at performance improvements.

Garbage Collection

Garbage collection is a built-in feature of Java that's responsible for the creation and destruction of objects. Let's begin by reviewing the improvements that have been made to the default garbage collector. Later, we analyze the new, experimental garbage collectors that were added in this release.

IMPROVEMENTS TO THE DEFAULT G1 COLLECTOR

Java 7 introduced the Garbage-First (G1) Collector. It later became the default garbage collector with Java 9. G1 is optimized for high probability and throughput. As mentioned by Thomas Schatzl (http://bit.ly/2RSr047), Java 11 improves pause times by up to 60% on x64 processors at a highly reduced memory footprint. That's a big gain for anyone ready to upgrade to Java 11 in production systems.

Moreover, this release of Java introduces two experimental garbage collectors. Let's see what they can bring to the table. We begin with ZGC.

ZGC: A SCALABLE, LOW-LATENCY GARBAGE COLLECTOR

Garbage collection is one of Java's compelling features. As a developer, you don't need to take care of marking objects for removal after you are done using them. The garbage collector takes care of it behind the scenes. One issue that you might face when garbage collection kicks in is pause times. This results in a poor customer experience because users must wait for the application to respond. Wait times are even more damaging to mission-critical applications committed to responsiveness, high throughput, and availability with agreed-upon Service-Level Agreements (SLAs).

The Z Garbage Collector (ZGC) is a scalable, low-latency garbage collector. What does that mean in practice? Although ZGC does not eliminate pause times completely, it guarantees a maximum pause time of 10 ms. Moreover, it can handle heaps from small to large (multiple terabytes). One other optimization that is important to mention is that with ZGC pause times do not increase the heap size. These conditions make ZGC a good for fit for memory-hungry applications that deal with a lot of data.

Do you want to try running a Java program with ZGC? You can enable the garbage collector by providing the VM options `-XX:+UnlockExperimentalVMOptions -XX:+UseZGC`. ZGC has been marked as experimental, so you will need to be cautious when using it for production systems. For more information, see the ZGC documentation page.

Java 11 adds another experimental garbage collector: Epsilon. In the next section, we take a brief look at its use cases and performance criteria.

THE NO-OP GARBAGE COLLECTOR: EPSILON

The second, experimental garbage collector available in Java 11 is called the Epsilon Garbage Collector. Epsilon is a low-overhead garbage collector, also referred to as a no-op garbage collector. You might be able to guess what that means. It doesn't reclaim memory at all after the space has been allocated.

Although this behavior sounds a bit strange at first, it definitely makes sense to apply Epsilon in certain use cases. Examples include short-lived programs such as command-line applications that execute only a single command and then shut down the JVM, or scenarios in which you want to performance-test a program.

To enable Epsilon for a program, use the VM options `-XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC`. For more information, see JEP 318, the enhancement proposal document.

We round out the discussion of this section with an overview of the security improvements in Java 11.

TLS 1.3 and Support for Cryptographic Algorithms

Security is of the utmost importance to any software system. With the latest incarnation of Transport Layer Security (TLS), released in August 2018, HTTPS performance has become faster and safer than ever. Java 11 was able to incorporate support for the TLS 1.3 protocol right off the bat. It's important to note, however, that TLS 1.3 does not support all of the features offered by the new TLS protocol. You can find a list of the supported features in JEP 332 (http://openjdk.java.net/jeps/332).

So, how do you go about using TLS 1.3 with Java 11? Well, the API hasn't actually changed. Java 11 only adds new constants for protocols and cipher suites. That makes migrating existing application code to the new version of TLS extremely easy.

The following code example should give you a rough idea on the usage of TLS 1.3. The code creates a `ServerSocket` to represent a server instance running on port 8080. Additionally, it enables the TLS 1.3 protocol and provides one of the new cipher suites that comes with the latest version of the TLS specification, as illustrated here:

```
try {
    SSLServerSocket socket = (SSLServerSocket)
            SSLServerSocketFactory.getDefault()
            .createServerSocket(8080);
    socket.setEnabledProtocols(new String[]
            { "TLSv1.3" });                        ❶
    socket.setEnabledCipherSuites(new String[]
            { "TLS_AES_256_GCM_SHA384" });         ❷

    try {
        new Thread(() ->
                System.out.println("Server started on port "
                + socket.getLocalPort())).start();
    } finally {
        if (socket != null && !socket.isClosed()) {
            socket.close();
        }
    }
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

❶ Enables the TLS 1.3 protocol.

❷ Enables one of the new cipher suites in TLS 1.3.

Summary and References to Further Reading

This concludes our tour of the latest features and changes in Java 11. Java 11 might not come with a "killer feature," but there are good reasons for you to upgrade. You can find most of the code used in this report in the GitHub repository "What's new in Java 11", implemented as test cases. If you want to dive into the nitty-gritty, check out the JEP list. The following summary provides a high-level recap of the changes offered with Java 11.

Oracle JDK and OpenJDK converged functionally

> The main difference is that the Oracle JDK will require users to purchase a license for production environments. Java now follows a six-month release cycle for major versions. Additionally, you can expect two patch releases between each major version for fixing critical bugs. Java 11 is an LTS release. Paying customers of Oracle JDK will receive additional minor and patch releases. The OpenJDK continues to be a free-of-charge, open source distribution.

Latest features and API enhancements

> Probably the biggest feature in Java 11 is the HTTP client implementation, an API for HTTP communication that can meet the demands of modern application development. We looked at the ability to execute single-file Java programs without the need for compiling the code. The functionality is helpful if you want to quickly try out Java code snippets for prototyping purposes. The Java team added convenience methods to the standard modules. As a result, developers can do away with relying on external libraries that implement similar helper methods. Java 11 also adopts Unicode 10, nest-based access control, and the open source profiling tools Java Flight Recorder and Java Mission Control.

Removed APIs and deprecations

> There are a number of removed APIs and Java EE modules in Java 11, and that has a great impact when you upgrade a project from an earlier Java version. But there are other options for external libraries that can take their place. It has been a long time coming, but with Java 11, Applets and Java Web Start are finally out. Decoupling JavaFX from the JDK is going to be helpful to ensure consistent and periodic releases of Java. A number of functionalities have also been marked for removal in future versions.

Performance and security

> Java 11 has made the default garbage collector more efficient and has introduced two new experimental garbage collectors: ZGC, which is suited for memory-intensive applications, and Epsilon, which targets programs with a predictable, short-term memory footprint. Java 11 supports the latest version of the Transport Layer Security protocol, TLS 1.3, making HTTPS communication faster and safer than ever.

Although it might require some work to integrate, I think the new features, changes, and optimizations in Java 11 are worth the effort because they facilitate a more efficient programming style and provide improved performance and security in production.