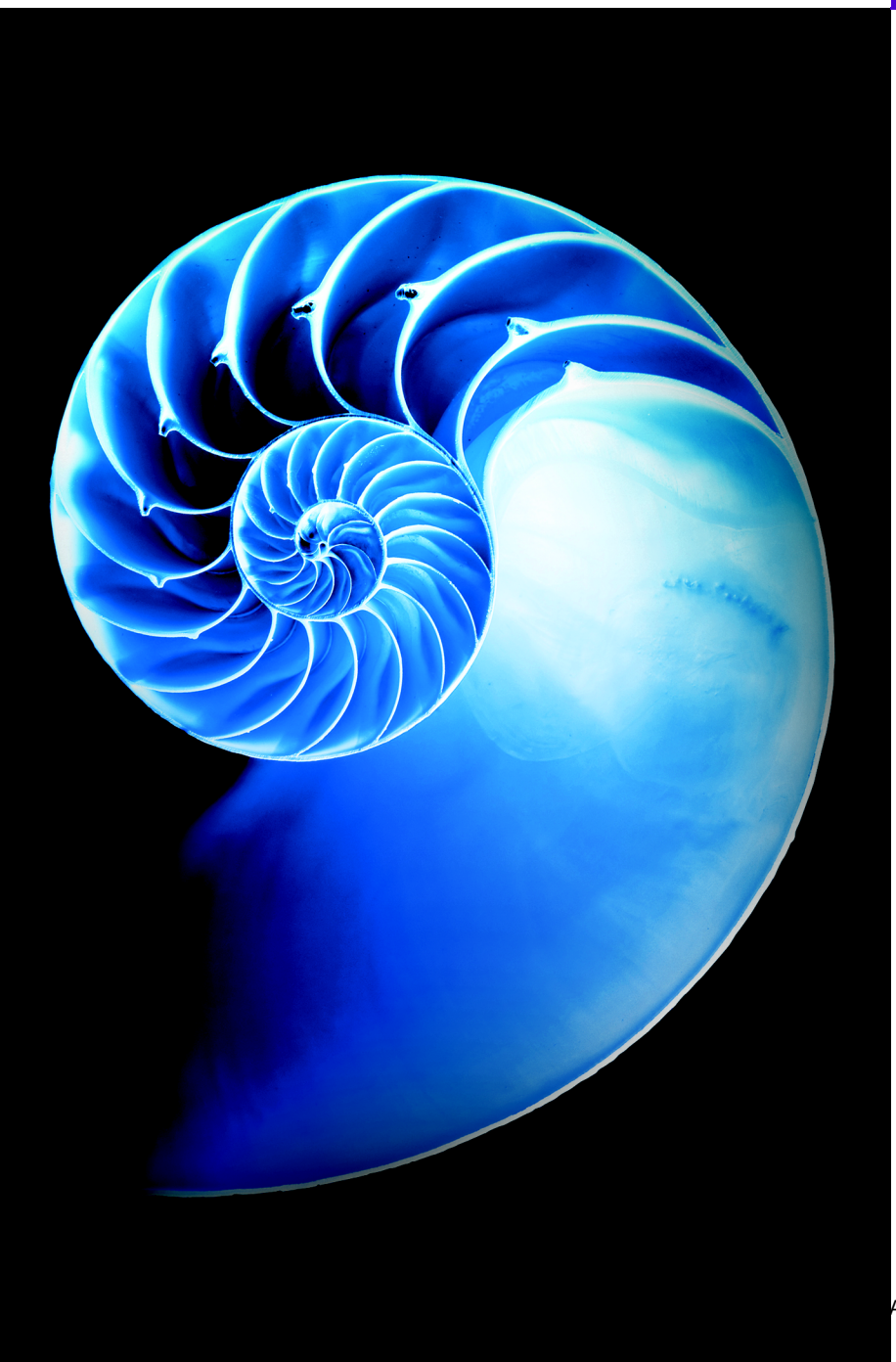
37

Additional Java 9 Topics



Objectives

In this chapter you'll:

- Briefly recap the Java 9 features we've already covered.
- Understand Java's new version numbering scheme.
- Use the new regular-expression `Matcher` methods `appendReplacement`, `appendTail`, `replaceFirst`, `replaceAll` and `results`.
- Use the new `Stream` methods `takeWhile` and `dropWhile` and the new `iterate` overload.
- Learn about the Java 9 JavaFX and other GUI and graphics enhancements.
- Use modules in JShell.
- Overview the Java 9 security-related changes and other Java 9 features.
- Become aware of the capabilities no longer available in JDK 9 and Java 9.
- Become aware of packages, classes and methods proposed for removal from future Java versions.

Outline

37.1 Introduction
37.2 Recap: Java 9 Features Covered in Earlier Chapters
37.3 New Version String Format
37.4 Regular Expressions: New `Matcher` Class Methods
37.4.1 Methods `appendReplacement` and `appendTail`
37.4.2 Methods `replaceFirst` and `replaceAll`
37.4.3 Method `results`
37.5 New `Stream` Interface Methods
37.5.1 Stream Methods `takeWhile` and `dropWhile`
37.5.2 Stream Method `iterate`
37.5.3 Stream Method `ofNullable`
37.6 Modules in JShell
37.7 JavaFX 9 Skin APIs
37.8 Other GUI and Graphics Enhancements
37.8.1 Multi-Resolution Images
37.8.2 TIFF Image I/O
37.8.3 Platform-Specific Desktop Features
37.9 Security Related Java 9 Topics
37.9.1 Filter Incoming Serialization Data
37.9.2 Create PKCS12 Keystores by Default
37.9.3 Datagram Transport Layer Security (DTLS)
37.9.4 OCSP Stapling for TLS
37.9.5 TLS Application-Layer Protocol Negotiation Extension
37.10 Other Java 9 Topics
37.10.1 Indify String Concatenation
37.10.2 Platform Logging API and Service
37.10.3 Process API Updates
37.10.4 Spin-Wait Hints
37.10.5 UTF-8 Property Resource Bundles
37.10.6 Use CLDR Locale Data by Default
37.10.7 Elide Deprecation Warnings on Import Statements
37.10.8 Multi-Release JAR Files
37.10.9 Unicode 8
37.10.10 Concurrency Enhancements
37.11 Items Removed from the JDK and Java 9
37.12 Items Proposed for Removal from Future Java Versions
37.12.1 Enhanced Deprecation
37.12.2 Items Likely to Be Removed in Future Java Versions
37.12.3 Finding Deprecated Features
37.12.4 Java Applets
37.13 Wrap-Up

37.1 Introduction

Just before we published this book, Java Specification Request (JSR) 379: Java SE 9 was released as a draft at:

```
http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec-01/java-se-9-spec.html
```

The JSR details the

- features included in Java 9,
- features that have been removed from Java 9, and
- features that are proposed for removal from future Java versions.

Once this JSR is approved as final it will be posted at:

```
https://www.jcp.org/en/jsretail?id=379
```

This JSR is a must read for any Java 9 developer. It gives a high-level overview of the breadth and depth of Java 9 and provides links to all the key JEPs and JSRs.

In any new version of a language there are items of immediate benefit to most programmers, items of interest to some programmers and narrow-purpose, specialty topics that limited numbers of developers will use. We divided this chapter into several groups:

- A recap of the Java 9 features we covered in earlier chapters.
- Live-code examples and discussions of additional functionality that will be useful to a wider audience.
- A brief overview of specialty features with references to where you can learn more.
- A list of features removed from JDK 9 and Java 9.
- A list of features proposed for removal from future Java versions.

Developers should, of course, avoid features in the last two groups in new development, and replace uses of those features in old code as it's migrated to Java 9.

37.2 Recap: Java 9 Features Covered in Earlier Chapters

Here we list the Java 9 features already covered in the book and where you can find each:

- Underscore (_) is no longer a valid identifier (Section 2.2). This is one of several features of JEP 213: Milling Project Coin (`http://openjdk.java.net/jeps/213`).
- Mentioned enhancements to `SecureRandom` (Section 6.8) per JEP 273 (`http://openjdk.java.net/jeps/273`).
- As of Java 9, the compiler now issues a warning if you attempt to access a `static` class member through an instance of the class (Section 8.11).
- Introduced `private` interface methods (Section 10.11), another feature of JEP 213: Milling Project Coin.
- Mentioned the new Stack-Walking API (Section 11.7) from JEP 259 (`http://openjdk.java.net/jeps/259`).
- Mentioned that effectively `final AutoCloseable` variables can now be used in `try`-with-resources statements (Section 11.12), another feature of JEP 213: Milling Project Coin.
- Overviewed new JavaFX 9 features and other GUI and graphics enhancements (Section 13.8).
- Mentioned Java 9's more compact `String` representation (Section 14.3), per JEP 254 (`http://openjdk.java.net/jeps/254`).
- Presented the new convenience factory methods for creating read-only collections (Section 16.14), per JEP 269 (`http://openjdk.java.net/jeps/269`).
- Chapter 25, Introduction to JShell: Java 9's REPL for Interactive Java, presented detailed, example-driven coverage of the JDK's new `jshell` tool.
- Chapter 36, Java Platform Module System, presented detailed example-driven coverage of Java 9's new module system.

37.3 New Version String Format

Prior to Java 9, JDK versions were numbered `1.X.0_`*updateNumber* where `X` was the major Java version. For example,

- Java 8's current JDK version number is `jdk1.8.0_121` and
- Java 7's final JDK version number was `jdk1.7.0_80`.

This numbering scheme has changed. JDK 9 initially will be known as `jdk-9`. Future minor version updates will add new features, and security updates will fix security holes. These updates will be reflected in the JDK version numbers. For example, in `9.1.3`:

- `9`—is the major Java version number
- `1`—is the minor version update number and
- `3`—is the security update number.

So 9.2.5 would indicate the version of Java 9 for which there have been two minor version updates and five total security updates (across major and minor versions). For additional details, see JEP 223:

```
http://openjdk.java.net/jeps/223
```

37.4 Regular Expressions: New `Matcher` Class Methods

Java SE 9 adds several new `Matcher` method overloads—**`appendReplacement`**, **`appendTail`**, **`replaceFirst`**, **`replaceAll`** and **`results`** (Fig. 37.1).

```
1  // Fig. 37.1: MatcherMethods.java
2  // Java 9's new Matcher methods.
3  import java.util.regex.Matcher;
4  import java.util.regex.Pattern;
5
6  public class MatcherMethods {
7     public static void main(String[] args) {
8        String sentence = "a man a plan a canal panama";
9
10       System.out.printf("sentence: %s%n", sentence);
11
12       // using Matcher methods appendReplacement and appendTail
13       Pattern pattern = Pattern.compile("an"); // regex to match
14
15       // match regular expression to String and replace
16       // each match with uppercase letters
17       Matcher matcher = pattern.matcher(sentence);
18
19       // used to rebuild String
20       StringBuilder builder = new StringBuilder();
21
22       // append text to builder; convert each match to uppercase
23       while (matcher.find()) {
24          matcher.appendReplacement(
25             builder, matcher.group().toUpperCase());
26       }
27
28       // append the remainder of the original String to builder
29       matcher.appendTail(builder);
30       System.out.printf(
31          "%nAfter appendReplacement/appendTail: %s%n", builder);
32
```

Fig. 37.1 | Java 9's new Matcher methods. (Part 1 of 2.)

```
33          // using Matcher method replaceFirst
34          matcher.reset(); // reset matcher to its initial state
35          System.out.printf("%nBefore replaceFirst: %s%n", sentence);
36          String result = matcher.replaceFirst(m -> m.group().toUpperCase());
37          System.out.printf("After replaceFirst: %s%n", result);
38
39          // using Matcher method replaceAll
40          matcher.reset(); // reset matcher to its initial state
41          System.out.printf("%nBefore replaceAll: %s%n", sentence);
42          result = matcher.replaceAll(m -> m.group().toUpperCase());
43          System.out.printf("After replaceAll: %s%n", result);
44
45          // using method results to get a Stream<MatchResult>
46          System.out.printf("%nUsing Matcher method results:%n");
47          pattern = Pattern.compile("\\w+"); // regular expression to match
48          matcher = pattern.matcher(sentence);
49          System.out.printf("The number of words is: %d%n",
50             matcher.results().count());
51
52          matcher.reset(); // reset matcher to its initial state
53          System.out.printf("Average characters per word is: %f%n",
54             matcher.results()
55                    .mapToInt(m -> m.group().length())
56                    .average().orElse(0));
57       }
58    }
```

```
sentence: a man a plan a canal panama

After appendReplacement/appendTail: a mAN a plAN a cANal pANama

Before replaceFirst: a man a plan a canal panama
After replaceFirst: a mAN a plan a canal panama

Before replaceAll: a man a plan a canal panama
After replaceAll: a mAN a plAN a cANal pANama

Using Matcher method results:
The number of words is: 7
Average characters per word is: 3.000000
```

Fig. 37.1 | Java 9's new Matcher methods. (Part 2 of 2.)

37.4.1 Methods appendReplacement and appendTail

The new `Matcher` method overloads `appendReplacement` (lines 24–25) and `appendTail` (line 29) are used with `Matcher` method `find` (line 23) and a `StringBuilder` in a loop to iterate through a `String` and replace every-regular expression match with a specified `String`. At the end of the process, the `StringBuilder` contains the original `String`'s contents updated with the replacements. Lines 13–26 proceed as follows:

- Line 13 creates a `Pattern` to match—in this case, the literal characters `"an"`.
- Line 17 creates a `Matcher` object for the `String sentence` (declared in line 8). This will be used to locate the `Pattern` `"an"` in `sentence`.

- Line 20 creates the `StringBuilder` in which the results will be placed.
- Line 23 uses `Matcher` method `find`, to locate an occurrence of `"an"` in the original `String`.
- If a match is found, method `find` returns `true`, and line 24 calls `Matcher` method `appendReplacement` to replace `"an"` with `"AN"`. The method's second argument calls `Matcher` method `group` to get a `String` representing the set of characters that matched the regular expression (in this case, `"an"`). We then convert the matching characters to uppercase. Method `appendReplacement` then appends to the `StringBuilder` in the first argument all of characters up to the match in the original `String`, followed by the replacement specified in the second argument. Then, the loop-continuation condition attempts to `find` another match in the original `String`, starting from the first character *after* the preceding match.
- When method `find` returns `false`, the loop terminates and line 29 uses `Matcher` method `appendTail` to append the remaining characters of the original `String` sentence to the `StringBuilder`.

At the end of this process for the original `String` `"a man a plan a canal panama"`, the `StringBuilder` contains `"a mAN a plAN a cANal pANama"`.

37.4.2 Methods `replaceFirst` and `replaceAll`

`Matcher` method overloads `replaceFirst` (line 36) and `replaceAll` (line 42) replace the first match or all matches in a `String`, respectively, using a `Function` that receives a `MatchResult` and returns a replacement `String`. Lines 36 and 42 implement interface `Function` with lambdas that group the matching characters and convert them to uppercase `String`s. Lines 34 and 40 call `Matcher` method `reset` so that the subsequent calls to `replaceFirst` and `replaceAll` begin searching for matches from the first character in `sentence`.

37.4.3 Method `results`

The new `Matcher` method `results` (lines 50 and 54) returns a stream of `MatchResult`s. In lines 47–50, we use the regular expression `\w+` to match sequences of word characters then simply `count` the matches to determine the number of words in `sentence`. After resetting the `Matcher` (line 52), lines 54–56 use a stream to map each word to its `int` number of characters (via `mapToInt`), then calculate the average length of each word using `IntStream` method `average`.

37.5 New `Stream` Interface Methods

Java 9 adds several new `Stream` methods—**`takeWhile`**, **`dropWhile`**, **`iterate`** and **`ofNullable`** (Fig. 37.2). All but `ofNullable` are also available in the numeric streams like `IntStream`.

```
1  // Fig. 37.2: StreamMethods.java
2  // Java 9's new stream methods takeWhile, dropWhile, iterate
3  // and ofNullable.
```

Fig. 37.2 | Java 9's new stream methods `takeWhile`, `dropWhile`, `iterate` and `ofNullable`. (Part 1 of 3.)

```
4  import java.util.stream.Collectors;
5  import java.util.stream.IntStream;
6  import java.util.stream.Stream;
7
8  public class StreamMethods {
9     public static void main(String[] args) {
10       int[] values = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
11
12       System.out.printf("Array values contains: %s%n",
13          IntStream.of(values)
14                   .mapToObj(String::valueOf)
15                   .collect(Collectors.joining(" ")));
16
17       // take the largest stream prefix of elements less than 6
18       System.out.println("Demonstrating takeWhile and dropWhile:");
19       System.out.printf("Elements less than 6: %s%n",
20          IntStream.of(values)
21                   .takeWhile(e -> e < 6)
22                   .mapToObj(String::valueOf)
23                   .collect(Collectors.joining(" ")));
24
25       // drop the largest stream prefix of elements less than 6
26       System.out.printf("Elements 6 or greater: %s%n",
27          IntStream.of(values)
28                   .dropWhile(e -> e < 6)
29                   .mapToObj(String::valueOf)
30                   .collect(Collectors.joining(" ")));
31
32       // use iterate to generate stream of powers of 3 less than 10000
33       System.out.printf("%nDemonstrating iterate:%n");
34       System.out.printf("Powers of 3 less than 10,000: %s%n",
35          IntStream.iterate(3, n -> n < 10_000, n -> n * 3)
36                   .mapToObj(String::valueOf)
37                   .collect(Collectors.joining(" ")));
38
39       // demonstrating ofNullable
40       System.out.printf("%nDemonstrating ofNullable:%n");
41       System.out.printf("Number of stream elements: %d%n",
42          Stream.ofNullable(null).count());
43       System.out.printf("Number of stream elements: %d%n",
44          Stream.ofNullable("red").count());
45    }
46 }
```

```
Array values contains: 1 2 3 4 5 6 7 8 9 10
Demonstrating takeWhile and dropWhile:
Elements less than 6: 1 2 3 4 5
Elements 6 or greater: 6 7 8 9 10

Demonstrating iterate:
Powers of 3 less than 10,000: 3 9 27 81 243 729 2187 6561
```

Fig. 37.2 | Java 9's new stream methods `takeWhile`, `dropWhile`, `iterate` and `ofNullable`. (Part 2 of 3.)

```
Demonstrating ofNullable:
Number of stream elements: 0
Number of stream elements: 1
```

Fig. 37.2 | Java 9's new stream methods `takeWhile`, `dropWhile`, `iterate` and `ofNullable`. (Part 3 of 3.)

37.5.1 Stream Methods `takeWhile` and `dropWhile`

Lines 19–30 demonstrate methods `takeWhile` and `dropWhile`, which based on a `Predicate` include or omit stream elements, respectively. These methods are meant for use on ordered streams. Unlike `filter`, which processes all of the stream's elements, each of these new methods process elements only until its `Predicate` argument becomes `false`.

The stream pipeline in lines 19–23 takes `int`s from the beginning of the stream while each `int` is less than 6. The predicate returns `true` only for the first five stream elements—as soon as the `Predicate` returns `false`, the remaining elements of the original stream are ignored. For the five elements that remain in the stream, we map each to a `String` and returns a `String` containing the elements separated by spaces.

The stream pipeline in lines 26–30 drops `int`s from the beginning of the stream while each `int` is less than 6. The resulting stream contains the elements beginning with the first one that was 6 or greater. For the elements that remain in the stream, we map each element to a `String` and collect the results into a `String` containing the elements separated by spaces.

Error-Prevention Tip 37.1

Invoke `takeWhile` and `dropWhile` only on ordered streams. If these methods are called on an unordered stream, the stream may return any subset of the matching elements, including none at all, thus giving you potentially unexpected results.

Performance Tip 37.1

According to the `Stream` interface documentation, you may encounter performance issues for the `takeWhile` and `dropWhile` methods on ordered parallel pipelines. For more information, see `http://download.java.net/java/jdk9/docs/api/java/util/stream/Stream.html`.

37.5.2 Stream Method `iterate`

In Section 4.8, we showed a `while` loop that calculated the powers of 3 less than 100. Lines 34–37 show how to use the new overload of `Stream` method `iterate` to generate a stream of `int`s containing the powers of 3 less than 10,000. The new overload takes as its arguments

- a seed value which becomes the stream's first element,
- a `Predicate` that determines when to stop producing elements, and
- a `UnaryOperator` that's invoked initially on the seed value, then on each prior value that iterate produces until the `Predicate` becomes `false`.

In this case, the seed value is 3, the `Predicate` indicates that iterate should continue producing elements while the last element produced is less than 10,000, and the `UnaryOperator` multiplies the prior element's value by 3 to produce the next element. Then we map

each element to a `String` and collect the results into a `String` containing the elements separated by spaces.

37.5.3 Stream Method `ofNullable`

The new `Stream` `static` method `ofNullable` receives a reference to an object and, if the reference is not `null`, returns a one-element stream containing the object; otherwise, it returns an empty stream. Lines 42 and 44 show mechanical examples demonstrating an empty stream and a one-element stream, respectively.

Method `ofNullable` typically would be used to ensure that a reference is not `null`, before performing operations in a stream pipeline. Consider a company employee database. A program could query the database to locate all the `Employees` in a given department and store them as a collection in a `Department` object referenced by the variable `department`. If the query were performed for a nonexistent department, the reference would be `null`. Rather than first checking whether `department` is `null`, then performing a task as in

```
if (department != null) {
   // do something
}
```

you can instead use code like the following:

```
Stream.ofNullable(department)
      .flatmap(Department::streamEmployees)
      ... // do something with each Employee
```

Here we assume that class `Deparment` contains a `public` method `streamEmployees` that returns a stream of `Employees`. If `department` is not `null`, the pipeline would `flatMap` the `Department` object into a stream of `Employees` for further processing. If department were `null`, `ofNullable` would return an empty stream, so the pipeline would simply terminate.

37.6 Modules in JShell

In Section 25.10, we demonstrated how to add your custom classes to the JShell classpath, so that you can then interact with them in JShell. Here we show how to do that with the `com.deitel.timelibrary` module from Section 36.4. For the purpose of this section, open a command window and change to the `TimeApp` folder in the `ch36` examples folder, then start `jshell`.

Adding a Module to the JShell Session

The `/env` command can specify the module path and the specific modules that JShell should load from that path. To add the `com.deitel.timelibrary` module, execute the following command:

```
jshell> /env -module-path jars -add-modules com.deitel.timelibrary
|  Setting new options and restoring state.

jshell>
```

The `-module-path` option indicates where the modules you wish to load are located (in this case the `jars` folder in the folder from which you executed JShell). The `-add-modules` option indicates the specific modules to load (in this case, `com.deitel.timelibrary`).

Importing a Class from a Module's Exported Package(s)

Once the module is loaded, you may `import` types from any of the module's exported packages. The following command `imports` the module's `Time1` class:

```
jshell> import com.deitel.timelibrary.Time1

jshell>
```

Using the Imported Class

At this point, you can use class `Time1`, just as you used other classes in Chapter 25. Create a `Time1` object,

```
jshell> Time1 time = new Time1()
time ==> 12:00:00 AM

jshell>
```

Next, inspect its members with auto-completion by typing `"time."` and pressing *Tab*:

```
jshell> time.
equals(                getClass()             hashCode()
notify()               notifyAll()
setTime(               toString()             toUniversalString()
wait(

jshell> time.
```

View just the members that begin with `"to"` by typing `"to"` then pressing *Tab*:

```
jshell> time.to
toString()             toUniversalString()

jshell> time.to
```

Finally, type `"U"` then press *Tab* to auto-complete `toUniversalString()`, then press *Enter* to invoke the method and assign the 24-hour-clock-format `String` to an implicitly declared variable:

```
jshell> time.toUniversalString()
$3 ==> "00:00:00"

jshell>
```

37.7 JavaFX 9 Skin APIs

In Chapter 22, JavaFX Graphics and Multimedia, we demonstrated how to format JavaFX objects using *Cascading Style Sheets (CSS)* technology which was originally developed for styling the elements in web pages. CSS allows you to specify *presentation* (e.g., fonts, spacing, sizes, colors, positioning) separately from the GUI's *structure* and *content* (layout containers, shapes, text, GUI components, etc.). If a JavaFX GUI's presentation is determined entirely by a style sheet (which specifies the rules for styling the GUI), you can simply swap in a new style sheet—sometimes called a *theme* or a *skin*—to change the GUI's appearance.

Each JavaFX control also has a *skin class* that determines its default appearance and how the user can interact with the control. In JavaFX 8, these skin classes were defined as *internal APIs*, but many developers extended these classes to create custom skins.

Portability Tip 37.1

Due to strong encapsulation, the JavaFX 8 internal skin APIs are no longer accessible in Java 9. If you created custom skins based on these pre-Java-9 APIs, your code will no longer compile in Java 9, and any existing compiled code will not run in the Java 9 JRE.

As part of Java 9 modularization, JavaFX 9 makes the skin classes `public` APIs in the `javafx.scene.control.skin` package, as described by JEP 253:

```
http://openjdk.java.net/jeps/253
```

The new skin classes are direct or indirect subclasses of class `SkinBase` (package `javafx.scene.control`). You can extend the appropriate skin class to customize the look-and-feel for a given type of control. You can then specify the fully qualified name of your skin class for a given control via the JavaFX CSS property `-fx-skin`.

Generally CSS is the easiest way to control the look of your JavaFX GUIs. For precise control over every aspect of a control, including the control's size, position, mouse and keyboard interactions and more, extend `SkinBase` or one of its many new control-specific subclasses in package `javafx.scene.control.skin`.

37.8 Other GUI and Graphics Enhancements

In addition to the changes mentioned in Section 13.8 and 37.7, JSR 379 includes enhanced image support and additional desktop integration features.

37.8.1 Multi-Resolution Images

Apps often display different versions of an image, based on a device's screen size and resolution. Java 9 adds support for multi-resolution images in which a single image actually represents a set of images and class `Graphics` (package `java.awt`) can choose the appropriate resolution to use, based on the device. For more information, visit:

```
http://openjdk.java.net/jeps/251
```

37.8.2 TIFF Image I/O

The Image I/O framework provides APIs for loading and saving images. The framework supports plug-ins for different image formats, with PNG and JPEG required to be supported on all Java implementations. As of Java 9, all implementations are also required to support the TIFF (also called TIF) format—macOS uses TIFF as one of its standard image formats and various other platforms also support it. For more information on the Image I/O framework, visit:

```
https://docs.oracle.com/javase/8/docs/technotes/guides/imageio/
```

For more information on the new TIFF support, visit:

```
http://openjdk.java.net/jeps/262
```

37.8.3 Platform-Specific Desktop Features

In Java 9, various internal APIs that were used for operating-system-specific desktop integration—such as interacting with the dock in macOS—are no longer accessible due to the module system's strong encapsulation. JEP 272 adds new public APIs to expose this capability for macOS and to provide similar capabilities for other operating systems (such as Windows and Linux). Other features that will be provided include

- login/logout and screen lock/unlock event listeners so a Java app can respond to those events
- getting the user's attention via the dock or task bar with blinking or bouncing app icons and
- displaying progress bars in a dock or task bar.

For more information, visit:

```
http://openjdk.java.net/jeps/272
```

37.9 Security Related Java 9 Topics

It's important for developers to be aware of Java security enhancements. In this section, we provide brief mentions of a few Java 9 security-related features and where you can learn more about each.

37.9.1 Filter Incoming Serialization Data

Java's **object serialization** mechanism enables programs to create **serialized objects**—sequences of bytes that include each object's data, as well as information about the object's type and the types of the object's data. After a serialized object has been output, it can be read into a program and **deserialized**—that is, the type information and bytes that represent the object are used to recreate the object in memory.

Deserialization has the potential for security problems. For example, if the bytes being deserialized are read from a network connection, an attacker could intercept the bytes and inject invalid data. If you do not validate the data after deserialization, it's possible that the object would be in an invalid state that could affect the program's execution. In addition, the deserialization mechanism enables any serialized object to be deserialized, provided that its type definition is available to the runtime. If the object being deserialized contains an array, an attacker potentially could inject an arbitrarily large number of elements, potentially using all of the app's available memory.

JEP 290, Filter Incoming Serialization Data:

```
http://openjdk.java.net/jeps/290
```

is a security enhancement to object serialization that enables programs to add filters that can restrict which types can be deserialized, validate array lengths and more.

37.9.2 Create PKCS12 Keystores by Default

A keystore maintains security certificates that are used in encryption. Java has used a custom keystore since Java 1.2 (1998). By default, Java 9 now uses the popular and extensible

PKCS12 keystore, which is more secure and will enable Java systems to interoperate with other systems that support the same standard. For more information, visit:

```
http://openjdk.java.net/jeps/229
```

37.9.3 Datagram Transport Layer Security (DTLS)

Datagrams provide a connectionless mechanism to communicate information over a network. Java 9 adds support for the Datagram Transport Layer Security (DTLS) protocol which provides secure communication via datagrams. For more information, visit:

```
http://openjdk.java.net/jeps/219
```

37.9.4 OCSP Stapling for TLS

X.509 security certificates are used in public-key cryptography. JEP 249 is a security and performance enhancement for checking whether an X.509 security certificate is still valid. For details, visit:

```
http://openjdk.java.net/jeps/249
```

37.9.5 TLS Application-Layer Protocol Negotiation Extension

This is a security enhancement to the `javax.net.ssl` package to enable applications to choose from a list of protocols for communicating with one another over a secure connection. For more details, visit

```
http://openjdk.java.net/jeps/244
```

37.10 Other Java 9 Topics

In this section, we provide brief mentions of various other features of JSR 379. At the time of this writing during Java 9's early access stage, only limited documentation was available to us. So we concentrated on the information from the JSRs and JEPs. In a few cases, we did not comment on certain new Java 9 features. These include:

- JEP 193: Variable Handles (`http://openjdk.java.net/jeps/193`),
- JEP 268: XML Catalogs (`http://openjdk.java.net/jeps/268`) and
- JEP 274: Enhanced Method Handles (`http://openjdk.java.net/jeps/274`).

37.10.1 Indify String Concatenation

JEP 280, Indify String Concatenation, is a behind-the-scenes enhancement to `javac` that's geared to improving `String` concatenation performance in the future. The goal is to enable such performance enhancements to be developed and added to future Java implementations *without* having to modify the bytecodes `javac` produces. For more information, visit:

```
http://openjdk.java.net/jeps/280
```

37.10.2 Platform Logging API and Service

Developers commonly use logging frameworks for tracking information that helps them with debugging, maintenance and evolution of their systems, analytics, detecting security

breaches and more. JEP 264, Platform Logging API and Service, adds a logging API for use by platform classes in the `java.base` module. Developers can then implement a service provider that routes logging messages to their preferred logging framework. For more information, visit

```
http://openjdk.java.net/jeps/264
```

37.10.3 Process API Updates

Java 9 includes enhancements to the APIs that enable Java programs to interact with operating-system-specific processes without having to use platform-specific native code written in C or C++. Some enhancements include access to a process's ID, arguments, start time, total CPU time and name, and terminating and monitoring processes from Java apps. For more information, visit:

```
http://openjdk.java.net/jeps/102
```

37.10.4 Spin-Wait Hints

Section 23.7 introduced a multithreading technique in which a thread that's waiting to aquire a lock on an object uses a loop to determine whether the lock is available and, if not, waits. Each time the thread is notified to check again, the loop repeats this process until the lock is acquired. This technique is known as a spin-wait loop. Java 9 adds a new API that enables such a loop to notify the JVM that it is a spin-wait loop. On some hardware platforms, the JVM can use this information to improve performance and reduce power consumption (especially crucial for battery-powered mobile devices). For more information, visit:

```
http://openjdk.java.net/jeps/285
```

37.10.5 UTF-8 Property Resource Bundles

Class `ResourceBundle` (package `java.util`) enables programs to load locale-specific information, such as `String`s in different spoken languages. This technique is commonly used to localize apps for users in different regions. Java 9 upgrades class `ResourceBundle` to support resources that are encoded in UTF-8 format (`https://en.wikipedia.org/wiki/UTF-8`). For more information, visit:

```
http://openjdk.java.net/jeps/226
```

37.10.6 Use CLDR Locale Data by Default

CLDR—the Unicode Common Locale Data Repository (`http://cldr.unicode.org`)—is an extensive repository of locale-specific information that developers can use when internationalizing their apps. Data in the repository includes information on

- date, time, number and currency formatting
- translations for the names of spoken languages, countries, regions, months, days, etc.
- language-specific information like capitalization, gender rules, sorting rules, etc.
- country information, and more.

CLDR support was included with Java 8, but is now the default in Java 9. For more information, visit:

```
http://openjdk.java.net/jeps/252
```

37.10.7 Elide Deprecation Warnings on Import Statements

Many company coding guidelines require code to compile without warnings. In JDK 8, if you imported a deprecated type or statically imported a deprecated member of a type, the compiler would issue warnings, even if those types or members were never used in your code. Java allows you to prevent deprecation warnings in your code via the `@SuppressWarnings` annotation, but this cannot be applied to `import` declarations. For this reason, it was not possible to prevent certain compile-time warnings. JDK 9 no longer produces such warnings on `import` declarations. For more information, visit:

```
http://openjdk.java.net/jeps/211
```

37.10.8 Multi-Release JAR Files

Even with Java 9's release, many people and organizations will continue using older versions of Java—some for many years. In one session at the 2016 JavaOne conference, attendees were asked which Java versions they were using. Several developers indicated their companies were still using versions as old as Java 1.4, which was released more than 15 years ago.

Library vendors often support multiple Java versions. Prior to Java 9, this required providing separate JAR files specific to each Java version. JDK 9 provides support for multi-release JAR files—a single JAR may contain multiple versions of the same class that are geared to different Java verions. In addition, these multi-release JAR files may contain module descriptors for use with the Java Platform Module System (Chapter 36). For more information, visit:

```
http://openjdk.java.net/jeps/238
```

37.10.9 Unicode 8

Java 9 supports the latest version of the Unicode Standard (`unicode.org`)—Unicode 8. Appropriate changes have been made to classes `String` and `Character`, as well as several other classes dependent on Unicode. For more details, visit:

```
http://openjdk.java.net/jeps/267
```

37.10.10 Concurrency Enhancements

JEP 266, More Concurrency Updates, adds features in three categories:

- Support for reactive streams—a technique for asynchronous stream processing—via class `Flow` and its nested interfaces. For a reactive streams overview and links to various other resources, visit:

```
https://en.wikipedia.org/wiki/Reactive_Streams
```

- Various improvements that the Java team accumulated since Java 8.
- Additional methods in class `CompletableFuture` (listed below).

New Methods of Class `CompletableFuture`

Section 23.14 introduced class `CompletableFuture`, which enables you to *asynchronously* execute `Runnable`s that perform tasks or `Supplier`s that return values. Java 9 enhances `CompletableFuture` with the following methods:

- `newIncompleteFuture`
- `defaultExecutor`
- `copy`
- `minimalCompletionStage`
- `completeAsync`
- `orTimeout`
- `completeOnTimeout`
- `delayedExecutor`
- `completedStage`
- `failedFuture`
- `failedStage`

For more information on the concurrency enhancements, visit:

```
http://openjdk.java.net/jeps/266
```

and see the online Java 9 documentation for `java.util.concurrent` (which includes class `CompletableFuture`'s methods) and related packages in the `java.base` module:

```
http://download.java.net/java/jdk9/docs/api/overview-summary.html
```

37.11 Items Removed from the JDK and Java 9

To help prepare the Java Platform for modularization, Java 9 removed several items from both the platform and its APIs. These are listed in JSR 379, Sections 8 and 9:

```
http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec-01/java-se-
   9-spec.html
```

Removed Platform Features

JSR 379 Section 8 lists the platform changes. These include removal of the Java extensions mechanism. Prior to Java 9, the extensions mechanism allowed you to place a library's JAR file in a special JRE folder to make the library available to all Java apps on that computer. Classes in that folder were guaranteed to load before app-specific classes, so this was sometimes used to upgrade libraries with newer versions. In Java 9, the extensions mechanism is replaced with *upgradeable modules*:

```
http://openjdk.java.net/projects/jigsaw/goals-reqs/03#upgradeable-
   modules
```

Upgradable modules are used primarily for standard technologies that evolve independently of the Java SE platform, but are bundled with the platform, such as JAXB—the Java Architecture for XML Binding. When a new JAXB version is released, its module can

be placed in the `java` command's **`--upgrade-module-path`**. The runtime will then use the new version, rather than the earlier version that was bundled with the platform.

Removed Methods

JSR 379 Section 9 lists methods that have been removed from various Java classes to help modularize the platform. According to the JSR, these methods were infrequently used, but keeping them would have required placing the packages of the `java.desktop` module into the `java.base` module, resulting in a much larger minimal runtime size. This would not make sense, because many apps do not require the `java.desktop` module's GUI and desktop integration capabilities.

37.12 Items Proposed for Removal from Future Java Versions

The Java Platform has been in use for more than 20 years. Over that time, some APIs have been deprecated in favor of newer ones—often to fix bugs, to improve security or simply because an improved API was added that rendered the prior ones obsolete. Yet, many deprecated APIs—some from as far back as Java 1.2, which was released in December 1998—have remained available in every new version of Java, mostly for backward compatibility.

37.12.1 Enhanced Deprecation

JEP 277

```
http://openjdk.java.net/jeps/277
```

adds new features to the `@Deprecated` annotation that enable developers to provide more information about deprecated APIs, including whether or not the API is scheduled to be removed in a future release. These enhanced annotations are now used throughout the Java 9 APIs and pointed out in the online API documentation to highlight features you should no longer use and that you should expect to be removed from future versions. For example, everything in the `java.applet` package is now deprecated (Section 37.12.4), so when you view the package's documentation at

```
http://download.java.net/java/jdk9/docs/api/java/applet/package-
   summary.html
```

you'll see deprecation notes in the package's description and for each type in the package. In addition, if you use the types in your code, you'll get warnings at compile time.

37.12.2 Items Likely to Be Removed in Future Java Versions

JSR 379, Section 10 lists the various packages, classes, fields and methods that are likely to be removed from future Java versions. The JSR indicates that these packages evolve separately from the Java SE Platform or are part of the Java EE Platform specification. According to the JSR, the classes, fields and methods proposed for removal typically do not work, are not useful or have been rendered obsolete by newer APIs.

37.12.3 Finding Deprecated Features

Each page in the online Java API documentation

```
http://download.java.net/java/jdk9/docs/api/overview-summary.html
```

now includes a **DEPRECATED** link so you can view the **Deprecated API** list containing the deprecated APIs:

```
http://download.java.net/java/jdk9/docs/api/deprecated-list.html
```

When you click a given item, its documentation generally mentions why it was deprecated and what you should use instead.

Error-Prevention Tip 37.2

Avoid using deprecated features in new code. Also, if you maintain or evolve legacy Java code, you should carefully study the Deprecated API list and consider replacing the listed items with the alternatives specified in the online Java documentation. This will help ensure that your code continues to compile and execute correctly in future Java versions.

37.12.4 Java Applets

As of Java 9 the Java Applet API is deprecated, per JEP 289 (`http://openjdk.java.net/jeps/289`). Previously this enabled Java to run in web browsers via a plug-in. Though this API has not been proposed for removal *yet*, it could be in a future Java version. Most popular web browsers removed Java plug-in support due to security issues.

37.13 Wrap-Up

In this chapter, we briefly recapped the Java 9 features covered in earlier chapters, then discussed various additional Java 9 topics. We presented the fundamentals of Java's new version numbering scheme. We demonstrated the new regular-expression `Matcher` methods `appendReplacement`, `appendTail`, `replaceFirst`, `replaceAll` and `results`. We also demonstrated the new `Stream` methods `takeWhile` and `dropWhile` and the new `iterate` overload. We discussed the Java 9 JavaFX changes, including the new `public` skin APIs and other GUI and graphics enhancements. You saw how to use modules in JShell.

We overviewed the Java 9 security-related changes and various other Java 9 features. We discussed the capabilities that are no longer available in JDK 9 and Java 9. Finally, we discussed the packages, classes and methods proposed for removal from future Java versions.