APPENDIX B

Java Versioning Scheme

In this chapter, you will learn:

- About the old and new Java release models
- What the JDK versioning scheme had been before JDK9
- About the new JDK versioning scheme in JDK9 and its updates in JDK10
- How to parse a JDK version string using the `Runtime.Version` class
- About the formats used to print the Java version string using different options with the `java` command

Java Release Models

Up to Java 9, which was released in September 2017, Java used a feature-driven release model with a new feature release every two years. In this model, every feature release contained one or more major features. Several times, a new feature release, for example, Java 8 and Java 9, was delayed by several months—sometimes over a year—because the development of the major features was not complete. This release model had a major drawback that small features had to wait for a long time to be available to developers.

Java has switched to a new time-based release model in which it will have:

- A feature release every six months
- An update release every three months
- A long-term support release every three years

In the new time-based release model, you will get a new feature release every six months, starting from March 2018, for example, Java 10 in Match 2018, Java 11 in September 2018, Java 12 in March 2019, and so on. A feature release may contain small and/or big additions to the Java language, JVM, or APIs. Development of a new feature is a continuous process, which is not linked to any future feature release until the feature is completely developed. Once a new feature is completely developed, it is merged to the upcoming feature release. This way, a new feature will not delay any upcoming release.

Starting from April 2018, you will get a new update release every three months in April, July, October, and January. An update release will contain only security fixes, and bugs fixes for newer releases.

Starting from September 2018, you will get a long-term support (LTS) feature release, which will be supported through updates for at least three years.

© Kishori Sharan 2018
K. Sharan, *Java Language Features*, https://doi.org/10.1007/978-1-4842-3348-1

What Is a Version String?

Each release of Java is identified by a unique string, which is known as the *version string*. For example, the version string, "1.8", "9", and "10" identify three different Java releases. The Java version string is important for several reasons. If you are developing a Java application, you want to use a Java version based on features supported by Java, support available in your organization, and how long the Java version will be supported by your vendor. Until Java 9, Java used version strings, which were like "1.7", "1.8.0_162", etc. Java 9 introduced a new version string scheme and Java 10 extended the version string scheme introduced in Java 9. The subsequent sections give you an overview of all the version string schemes. It explains how to print them on the command line and use them in programs.

Old Version String Scheme

Before JDK9, the version string used to be of the following form:

```
$n.$feature.$maintenance_$update-$identifier
```

Here,

- `$n` is always `1`.
- `$feature` is an integer incremented sequentially starting from `1` and it indicates a feature release that contains new functionalities.
- `$maintenance` is an integer to indicate a maintenance release, which contains engineering focused bug fixes. It is always zero for a feature release.
- `$update` is a two-digit integer with a leading zero if needed. It indicates a customer focused bug fixes. `$update` is not numbered sequentially.
- `$identifier` is used to identify a milestone before general availability (GA) of a product such as `ea` for early-access build, `beta` for beta release, build number, etc. Identifier is not part of the version string for GA releases.

The following are a few examples of the JDK version strings before JDK9:

- 1.8.0 is a GA feature release.
- 1.8.0_162 is an update release (with an update number 162) for the feature release 1.8.0.
- build 1.8.0_162-b12 is an update release (with an update number 162) for the feature release 1.8.0 with a build identifier of b12.

Before JDK9, the JDK versioning scheme was not intuitive to developers and not easy for programs to parse. Looking at two JDK versions, you could not tell the subtle differences between them. It was hard to answer a simple question: Which release contains the most recent all security fixes 1.7.0_55 or 1.7.0_60? The answer is not the obvious one, which you would have guessed: 1.7.0_60. The releases contain the same security fixes. What is the difference between releases named JDK 8 Update 66, 1.8.0_66, and JDK 8u66? They represent the same release. It was necessary to understand the versioning scheme in detail before you could determine the details contained in a version string. For example, after the initial release of JDK 1.5.0, the update release numbers in multiple of 20 meant different types of updates than the odd update numbers. The explanation in this section has not given you enough information to decipher the JDK version string scheme completely. Refer to the following web pages for more details on the JDK version string schemes before JDK9.

- `http://www.oracle.com/technetwork/java/javase/versioning-naming-139433.html`
- `http://www.oracle.com/technetwork/java/javase/overview/jdk-version-number-scheme-1918258.html`

New Version String Scheme

JDK9 attempted to standardize the JDK versioning scheme, so it could be easily understood by humans, easily parsed by programs, and follow the industry-standard versioning scheme.

JDK9 added a static nested class named `Runtime.Version`, which represents a version string for an implementation of the Java SE platform. The class can be used to represent, parse, validate, and compare version strings.

A *version string* consists of the following four elements in order. Only the first element is mandatory:

- Version number (`$vnum`)
- Pre-release information (`$pre`)
- Build information (`$build`)
- Additional information (`$opt`)

The following regular expression defines the format for a version string:

```
$vnum(-$pre)?(\+($build)?(-$opt)?)?
```

A *short version string* consists of a version number optionally followed by pre-release information:

```
$vnum(-$pre)?
```

You can have a version string as short as `"9"` and `"10"`, which contains only the feature release number, and as big as `"10.0.1.2-ea+40-20180201.07.36am"`, which contains all parts of a version string. I made this big version string up to show you all the elements of a version string. Typically, a pre-release (with `-ea` in the version string) does not have update and patch releases. The subsequent sections explain all parts and their sub-parts of the version string in details.

Version Number

A version number is a sequence of elements separated by a period. It can be of an arbitrary length. Its format is as follows:

```
^[1-9][0-9]*(((\.0)*\.[1-9][0-9]*)*)*$
```

A version number may consist of one to four elements or more, which are as follows. The fifth and subsequent elements do not have any specific meanings; JDK vendors can use them to identify vendor specific patches. The format of a version number is as follows:

```
$feature.$interim.$update.$patch(.$additionalInfo)
```

The `$feature` element was called `$major` in JDK9. It is a feature-release counter, which is increased by 1 for every feature release. A feature release represents a major version of a JDK release and contains new features. For example, the feature release for JDK10 is 10 and for JDK11 is 11. When the feature number is incremented, all other parts in the version number are removed. For example, if you have a version number 10.0.2.1, the new version number will be 11 when the feature counter is incremented from 10 to 11. With a six-months release model, `$feature` will be incremented by 1 in March and September every year, starting from March 2018. Therefore, the March 2018 release is JDK10, the September 2018 release is JDK11, the March 2019 release is JDK12, and so on.

The `$interim` element was called `$minor` in JDK9. It represents a non-feature release that contains compatible bug fixes and enhancements. An interim release will not contain any incompatible changes, feature removals, or any changes to standard APIs. The value for `$interim`, if present, is always zero because in a six-months release model there will be no interim releases. In the future, the release model may change to start including interim releases and, in that case, `$interim` will be used.

The `$update` element was called `$security` in JDK9. It is incremented for compatible update releases that fix security issues, regressions, and bugs in newer features.

The `$patch` element represents an emergency patch release containing fixes for critical issues.

The following rules apply to a version number:

- All elements must be an unsigned integer.
- Only `$feature` is mandatory.
- Elements of a version number cannot contain leading zeros. For example, the second update version of the JDK10 is 10.0.2, not 10.0.02.
- Trailing elements cannot be zero. That is, you cannot have a version number as 10.0.0.0. It can be 10, 10.0.1, 10.0.0.1, and so on.
- When an element is increment, all subsequent elements are removed. Suppose the current JDK version is 10.0.1.4. If the update counter is incremented from 1 to 2, the patch counter 4 will be reset to zero and hence removed, making the version number as 10.0.2. If the feature counter is incremented to 11, all other counters will be removed and the version number will change from 10.0.1.4 to 11.
- Two version numbers are compared using their respective counters from left to right. For example, 10.0.4.1 is greater than 10.0.3.5, but less than 10.0.4.2.

Pre-Release Information

The `$pre` element in a version string is a pre-release identifier, such as `ea` for an early-access release, `snapshot` for a pre-release snapshot, and `internal` for a developer internal build. It is optional. If it is present, it is prefixed with a hyphen (`-`) and it must be alphanumeric matching the regular expression `([a-zA-Z0-9]+)`. The version string, `"10-ea"`, contains 10 as a version number and `ea` as a pre-release identifier.

Build Information

The `$build` element in a version string is a build number incremented for each promoted build. It is optional. It is reset to 1 when any part of the version number is incremented. If it is present, it is prefixed with a plus sign (+) and it must match the regular expression `(0|[1-9][0-9]*)`. The version string, `"10-ea+40"`, contains 40 as build number.

Additional Information

The `$opt` element in a version string contains additional build information such as date and time of an internal build. It is optional. It is alphanumeric and can contain hyphens and periods. If it is present, it is prefixed with a hyphen (`-`) and it must match the regular expression `([-a-zA-Z0-9\.]+)`. If `$build` is absent, you need to prefix `$opt` with a plus sign followed by a hyphen (`+-`). For example, in `"10-ea+40-2018-02-19"`, `$build` is 40 and `$opt` is `"2018-02-19"`; in `"9+-123"`, `$pre` and `$build` are absent and `$opt` is `123`. The version string, `"10-ea+40-20180219.07.36am"`, embeds the date and time of the release in its additional information element.

Parsing Old and New Version Strings

Before JDK9, JDK releases had either been Limited Update releases that include new functionality and non-security fixes, or Critical Patch Updates that only include fixes for security vulnerabilities. The version string included the version number including the update number and the build number. Limited Update releases were numbered in multiples of 20. Critical Patch Updates used odd numbers, which were calculated by adding multiples of five to the prior Limited Update and, when needed, adding one to keep the resulting number odd. An example is 1.8.0_31-b13, which is update 31 of the major version 8 of the JDK. Its build number is 13. Note that prior to JDK9, the version string always started with 1.

Your existing code that parses the version string to get the feature version of a JDK release may fail in JDK9 and JDK10, depending on the logic it used. For example, if the logic looked for the feature version at the second element by skipping the first, which used to be always 1 before JDK9, the logic will fail. For example, if it returned 8 from 1.8.0, now it will return 0 from 10.0.1.0 where you would expect 10.

Changes to System Properties

In JDK9, the values returned for the system properties that contains the JDK version string have changed. Table B-1 contains the list of those system properties and the format of their values. `$vstr`, `$vnum`, and `$pre` in the table mean version string, version number, and pre-release information, respectively.

Table B-1. *System Properties and Their Changed Values in JDK9*

System Property Name	Value
`java.version`	`$vnum(\-$pre)?`
`java.runtime.version`	`$vstr`
`java.vm.version`	`$vstr`
`java.specification.version`	`$vnum`
`java.vm.specification.version`	`$vnum`

JDK10 added two new system properties:

- `java.version.date`
- `java.vendor.version`

The `java.version.date` system property contains the GA date of the current JDK release in ISO-8601 `YYYY-MM-DD` format. For early access releases, this is the planned GA date for the release. This system property lets you figure out how old your JDK is. Its value is also printed when you print the JDK version using `java --version` and other commands.

The `java.vendor.version` system property is a vendor-specific product version string. It is optional and assigned by the JDK vendor. Its value, if set, must match the regular expression, `\p{Graph}+`, which allows one or more alphanumeric and punctuation characters. In the OpenJDK, it is set to the `$year.$month`, which indicates the year and month of the JDK release. For example, in JDK10, it is set to 18.3, which means March 2018 release of the JDK. Its value is also printed when you print the JDK version using `java --version` and other commands.

Using the Runtime.Version Class

JDK9 added a static nested class called `Runtime.Version` whose instances represent version strings. The class does not have a public constructor. You can obtain an instance of this class in the following two ways:

- Using the static `version()` method of the `Runtime` class.
- Using the static `parse(String vstr)` method of the `Runtime.Version` class.

The static `version()` method of the `Runtime` class returns a `Runtime.Version`, which represents the version of the JRE. The following snippet of code prints the current version of the JRE, which indicates that my JRE is an early access build 42 of the feature release 10:

```
Runtime.Version version = Runtime.version();
System.out.println(version);
```

```
10-ea+42
```

You can use the `parse()` method of the `Runtime.Version` class to parse a version string and obtain a `Runtime.Version`. The `parse()` method throws an `IllegalArgumentException` if the specified string is not a valid version string. The following snippet of code shows you how to use the `parse()` method:

```
// Parse a version string - "10.0.1-ea+42"
Runtime.Version version =  Runtime.Version.parse("10.0.1-ea+42");
System.out.println(version);
```

```
10.0.1-ea+42
```

The following methods in the `Runtime.Version` class return elements of a version string. Method names are intuitive enough to guess the type of element they return.

- `int feature()`
- `int interim()`
- `int update()`
- `int patch()`

- `Optional<String> pre()`
- `Optional<Integer> build()`
- `Optional<String> optional()`

■ **Tip** The `major()`, `minor()`, and `security()` methods in the `Runtime.Version` class have been deprecated in JDK10. Use the `feature()`, `interim()`, and `update()` methods of the class instead.

Notice that for the optional elements—`$pre`, `$build`, and `$opt`—the return type is `Optional`. For the optional `$interim`, `$update`, and `$patch` elements, the return type is `int`, not `Optional`, which will return zero if they are absent in the version string.

The version number in a version string may contain additional information after the fourth element. The `Runtime.Version` class does not contain a method to get the additional information directly. It contains a `version()` method that returns a `List<Integer>`, where the list contains all elements of the version number. The first four elements in the list are `$feature`, `$interim`, `$update`, and `$patch`, in order. The remaining elements are the additional version number information. The list contains at least one element, which is the feature version number (`$feature`).

The `Runtime.Version` class contains methods to compare two version strings for order and equality. You can compare them with or without the optional build information (`$opt`). Those comparison methods are as follows:

- `int compareTo(Runtime.Version v)`
- `int compareToIgnoreOptional(Runtime.Version v)`
- `boolean equals(Object v)`
- `boolean equalsIgnoreOptional(Object v)`

The expression `v1.compareTo(v2)` will return a negative integer, zero, or a positive integer if `v1` is less than, equal to, or greater than `v2`. The `compareToIgnoreOptional()` method works the same way as the `compareTo()` method, except that it ignores the optional build information while comparing. The `equals()` and `equalsIgnoreOptional()` methods compare two version strings for equality with and without the optional build information.

Which version strings represents the latest build: `10.0.1.1` or `10.0.1.1-ea`? The first one does not contain the pre-release element, whereas the second one does, so the first one is the latest build. Which version strings represents the latest build: `10.0.1.1` or `10.0.1.2-ea`? This time, the second one represents the latest build because the version number `10.0.1.1` is considered less than `10.0.1.2`. When the version number is greater, other elements in the version string are not compared. Here are the detailed rules for comparing two version strings:

- Comparison occurs in the following sequence: `$vnum`, `$pre`, `$build`, and `$opt`.
- Comparison begins by comparing each element of the version number. A zero is assumed if any parts in the version number is missing. In the string from the feature version number, if any element in first version string is greater than the corresponding element in the second version string, the first version string is greater. For example, 10.0.1.1 is less than 11.

- If two version strings are the same based on their version numbers, the pre-release elements are compared. A version string with a pre-release element is always less than a version string without one. If pre-release elements consist only of digits, numerical comparison is made; otherwise, they are compared lexicographically. A numeric pre-release element is considered less than a non-numeric one.
- A version without a build number is always less than the one with a build number. If both version strings have a build number, they are compared numerically.
- A version with additional information element (`$opt`) is considered greater than the one without a `$opt`. Otherwise, the values of `$opt` are compared lexicographically.

Listing B-1 contains a complete program to show how to extract all parts of the version string using the `Runtime.Version` class.

Listing B-1. A VersionTest Class That Shows How to Use the Runtime.Version Class to Work with Version Strings

```
// VersionTest.java
package com.jdojo.java10.newfeatures;

import java.lang.Runtime.Version;
import java.util.stream.Collectors;

public class VersionTest {
    public static void main(String[] args) {
        // Have some version strings
        String[] versionStrings = {
            "10", "10.1", "10.0.1.2", "10.0.2.3.4", "10.0.0",
            "10.1.2-ea+153", "10+132", "10-ea+24-2018-01-23", "10+-123",
            "10.0.1-ea+132-2018-01-28.10.56.45am"};

        // Parse each version string and display its components
        for (String vstr : versionStrings) {
            try {
                Version version = Version.parse(vstr);

                // Get the additional version number element, which starts at 5th element
                // in the version number part in the version string
                String vnumAdditionalInfo = version.version()
                        .stream()
                        .skip(4)
                        .map(n -> n.toString())
                        .collect(Collectors.joining("."));

                System.out.printf("Version String=%s%n", vstr);
                System.out.printf("feature=%d, interim=%d, update=%d, patch=%d,"
                        + " additional info=%s,"
                        + " pre=%s, build=%s, optional=%s %n%n",
                        version.feature(),
                        version.interim(),
                        version.update(),
                        version.patch(),
                        vnumAdditionalInfo,
                        version.pre().orElse(""),
                        version.build().isPresent() ? version.build().get().toString() : "",
                        version.optional().orElse(""));
            } catch (Exception e) {
                System.out.printf("%s%n%n", e.getMessage());
            }
        }
    }
}
```

```
Version String=10
feature=10, interim=0, update=0, patch=0, additional info=, pre=, build=, optional=

Version String=10.1
feature=10, interim=1, update=0, patch=0, additional info=, pre=, build=, optional=

Version String=10.0.1.2
feature=10, interim=0, update=1, patch=2, additional info=, pre=, build=, optional=

Version String=10.0.2.3.4
feature=10, interim=0, update=2, patch=3, additional info=4, pre=, build=, optional=

Invalid version string: '10.0.0'

Version String=10.1.2-ea+153
feature=10, interim=1, update=2, patch=0, additional info=, pre=ea, build=153, optional=

Version String=10+132
feature=10, interim=0, update=0, patch=0, additional info=, pre=, build=132, optional=

Version String=10-ea+24-2018-01-23
feature=10, interim=0, update=0, patch=0, additional info=, pre=ea, build=24,
optional=2018-01-23

Version String=10+-123
feature=10, interim=0, update=0, patch=0, additional info=, pre=, build=, optional=123

Version String=10.0.1-ea+132-2018-01-28.10.56.45am
feature=10, interim=0, update=1, patch=0, additional info=, pre=ea, build=132,
optional=2018-01-28.10.56.45am
```

Printing Java Version Strings

You can use one of the following options with the `java` command to print the Java product version string. The output is printed to the standard output or standard error:

- `-version`: Prints the product version to the error stream and exits.
- `--version`: Prints the product version to the output stream and exits.
- `-showversion`: Prints the product version to the error stream and continues.
- `--show-version`: Prints the product version to the output stream and continues.
- `-fullversion`: Prints the product version to the error stream and exits.
- `--full-version`: Prints the product version to the output stream and exits.

These options differ in three ways: the output format, the destination of the output, and post-output behavior. I show their output format shortly. If an option continues, it means it prints the output to its destination and continues to execute other parts of the command, if any. For example, you can use the `-showversion` and `--show-version` options to print the product version and run a class, like so:

```
java --show-version --module-path dist --module test/com.jdojo.Test
```

The following is the list of the output formats of each of the options, where `\u0020` stands for the Unicode value for the space character. In the format specifier:

- The free-from text, such as `openjdk version`, will be printed as is for OpenJDK. When you use Oracle JDK, `openjdk version` is replaced with `java version`.
- `${LTS}` expands to"`\u0020LTS` if the first three characters in the `$opt` part of the version string are `LTS`.
- `${JVV}` expands to `\u0020${java.vendor.version}` if the `java.vendor.version` system property is defined.

```
C:\> java -version
openjdk version \"${java.version}\" ${java.version.date}${LTS}
${java.runtime.name}${JVV} (build ${java.runtime.version})
${java.vm.name}${JVV} (build ${java.vm.version}, ${java.vm.info})

C:\> java --version
openjdk ${java.version} ${java.version.date}${LTS}
${java.runtime.name}${JVV} (build ${java.runtime.version})
${java.vm.name}${JVV} (build ${java.vm.version}, ${java.vm.info})

C:\> java -showversion <other-options-if-any>
openjdk version \"${java.version}\" ${java.version.date}${LTS}
${java.runtime.name}${JVV} (build ${java.runtime.version})
${java.vm.name}${JVV} (build ${java.vm.version}, ${java.vm.info})
[Output of other options, if used...]

C:\> java --show-version <other-options-if-any>
openjdk ${java.version} ${java.version.date}${LTS}
```

```
${java.runtime.name}${JVV} (build ${java.runtime.version})
${java.vm.name}${JVV} (build ${java.vm.version}, ${java.vm.info})
[Output of other options, if used...]

C:\> java -fullversion
openjdk full version \"${java.runtime.version}\"

C:\> java --full-version
openjdk ${java.runtime.version}
```

The following is the output of using the `--version` option with the `java` command in Java 10 early access build 42:

```
C:\>java --version
openjdk 10-ea 2018-03-20
OpenJDK Runtime Environment 18.3 (build 10-ea+42)
OpenJDK 64-Bit Server VM 18.3 (build 10-ea+42, mixed mode)
```

Summary

Java has adopted a new time-based release model in which it will have a feature release every six months, an update release every three months, and a long-term support (LTS) release every three years. New feature releases will happen in March and September, starting from March 2018 with Java 10. Update releases will happen in January, April, July, and October. LTS releases will happen every three years starting from September 2018.

The non-intuitive versioning scheme for the JDK was revamped in JDK9. A JDK version string consists of the following four elements in order (only the first one is mandatory): a version number, pre-release information, build information, and additional information. The regular expression, `"$vnum(-$pre)?(\+($build)?(-$opt)?)?"`, defines the format for a version string. In JDK9, `$vum` consisted of four parts: `$major.$minor.$security(.$additionalInfo)`. From JDK10, `$vnum` consists of five parts: `$feature.$interim.$update.$patch(.$additionalInfo)`. The `$feature`, `$interim`, and `$update` elements in `$vnum` in JDK10 were called `$major`, `$minor`, and `$security` in JDK9.

A short version string consists of only the first two elements—a version number optionally followed by pre-release information. You can have a version string as short as `"10"`, which contains only the major version number and as big as `"10.0.1.1-ea+54-20180210.07.36am"`, which contains all elements of a version string.

JDK9 added a static nested class called `Runtime.Version` whose instances represent JDK version strings. The class does not have a public constructor. You can get the current version of Java using the `version()` static method of the `Runtime` class. The method returns a `Runtime.Version`. You can use the `parse(String vstr)` static method of the `Runtime.Version` class to parse a version string to obtain a `Runtime.Version`. The `parse()` method may throw a runtime exception if the version string is `null` or invalid. The class contains several methods to get different parts of the version.

JDK10 added two new system properties named `java.version.date` and `java.vendor.version`. The `java.version.date` system property contains the GA date of the current JDK release in ISO-8601 `YYYY-MM-DD` format. The `java.vendor.version` system property is a vendor-specific product version string. It is optional and assigned by the JDK vendor.

The `Runtime.Version` class contains several methods to compare and retrieve different parts of a version string. For example, you can use the `feature()` method to get the `$feature` element for a version string. The `major()`, `minor()`, and `security()` methods in the `Runtime.Version` class have been deprecated in JDK10; use the `feature()`, `interim()`, and `update()` methods of the class instead.