Jason Lee

Java 9 Programming Blueprints

Implement new features such as modules, the process handling API, REPL, and many more to build end-to-end applications in Java 9

Packt

Java 9 Programming Blueprints

Implement new features such as modules, the process handling API, REPL, and many more to build end-to-end applications in Java 9

Jason Lee

BIRMINGHAM - MUMBAI

Java 9 Programming Blueprints

First published: July 2017

Production reference: 1250717

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78646-019-6

www.packtpub.com

Credits

Author
Jason Lee

Reviewer
Dionisios Petrakopoulos

Commissioning Editor
Kunal Parikh

Acquisition Editor
Chaitanya Nair

Content Development Editor
Lawrence Veigas

Technical Editor
Abhishek Sharma

Copy Editor
Zainab Bootwala

Project Coordinator
Prajakta Naik

Proofreader
Safis Editing

Indexer
Rekha Nair

Graphics
Abhinash Sahu

Production Coordinator
Nilesh Mohite

About the Author

Jason Lee has been writing software professionally for over 20 years, but his love for computers started over a decade earlier, in the fourth grade, when his dad brought home a Commodore 64. He has been working with Java for almost his entire career, with the last 12+ years focused primarily on Enterprise Java. He has written in-house web applications and libraries, and also worked on large, more public projects, such as the JavaServer Faces reference implementation Mojarra, GlassFish, and WebLogic Server.

Jason is currently the President of the Oklahoma City Java Users Group, and is an occasional speaker at conferences. Ever the technology enthusiast, his current interests include cloud computing, mobile development, and emerging JVM languages.

Apart from work, Jason enjoys spending time with his wife, Angela, and his two sons, Andrew and Noah. He is active in the music ministry of his local church, and enjoys reading, running, martial arts, and playing his bass guitar.

Everyone told me that writing a book is hard, and they weren't kidding! There's no way I could have done this without the love and support of my beautiful wife, Angela, who was patient and supportive during all of my late nights and long weekends, and was kind enough to read through every last page, helping me clean things up.

My two awesome sons, Andrew and Noah, also deserve huge thanks. There were certainly many nights when I was locked away in my office instead of spending time with you. I appreciate your understanding and patience during this project, and I hope this is something we can all be proud of together.

Angela, Andrew, and Noah, this is for you. I love you all!

About the Reviewer

Dionisios Petrakopoulos has worked in several companies using different technologies and programming languages, such as C, C++, Java SE, Java EE, and Scala, as a senior software engineer for the past 15 years. His main interest is the Java ecosystem and the various facets of it. His other area of interest is information security, especially cryptography. He holds a BSc degree in computer science and an MSc degree in information security, both from Royal Holloway, University of London. He is also the technical reviewer of the book *Learning Modular Java Programming* by Packt.

I would like to thank my wife Anna for her support and love.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/178646019X`.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

The world has been waiting for Java 9 for a long time. More specifically, we've been waiting for the Java Platform Module System, and Java 9 is finally going to deliver it. If all goes as planned, we'll finally have true isolation, giving us, potentially, smaller JDKs and more stable applications. That's not all that Java 9 is offering of course; there is a plethora of great changes in the release, but that's certainly the most exciting. That said, this book is not a book about the module system. There are plenty of excellent resources that can give you a deep dive into the Java Platform Module System and its many implications. This book, though, is a much more practical look at Java 9. Rather than discussing the minutiae of the release, as satisfying as that can be, what we'll do over the next few hundred pages is look at different ways all of the great changes in recent JDK releases--especially Java 9--can be applied in practical ways.

When we're done, you'll have ten different projects that cover a myriad of problem areas, from which you can draw usable examples as you work to solve your own unique challenges.

What this book covers

`Chapter 1`, *Introduction*, gives a quick overview of the new features in Java 9, and also covers some of the major features of Java 7 and 8 as well, setting the stage for what we'll be using in later chapters.

`Chapter 2`, *Managing Process in Java*, builds a simple process management application (akin to Unix's top command), as we explore the new OS process management API changes in Java 9.

`Chapter 3`, *Duplicate File Finder*, demonstrates the use of the New File I/O APIs in an application, both command line and GUI, that will search for and identify duplicate files. Technologies such as file hashing, streams, and JavaFX are heavily used.

`Chapter 4`, *Date Calculator*, shows a library and command-line tool to perform date calculations. We will see Java 8's Date/Time API exercised heavily.

`Chapter 5`, *Sunago - A Social Media Aggregator*, shows how one can integrate with third-party systems to build an aggregator. We'll work with REST APIs, JavaFX, and pluggable application architectures.

`Chapter 6`, *Sunago - An Android Port*, sees us return to our application from `Chapter 5`, *Sunago - A Social Media Aggregator*.

`Chapter 7`, *Email and Spam Management with MailFilter*, builds a mail filtering application, explaining how the various email protocols work, then demonstrates how to interact with emails using the standard Java email API--JavaMail.

`Chapter 8`, *Photo Management with PhotoBeans*, takes us in a completely different direction when we build a photo management application using the NetBeans Rich Client Platform.

`Chapter 9`, *Taking Notes with Monumentum*, holds yet another new direction. In this chapter, we build an application--and microservice--that offers web-based note-taking similar to several popular commercial offerings.

`Chapter 10`, *Serverless Java*, moves us into the cloud as we build a Function as a Service system in Java to send email and SMS-based notifications.

`Chapter 11`, *DeskDroid - A Desktop Client for Your Android Phone*, demonstrates a simple approach for a desktop client to interact with an Android device as we build an application to view and send text messages from our desktop.

`Chapter 12`, *What's Next?*, discusses what the future might hold for Java, and also touches upon two recent challengers to Java's preeminence on the JVM--Ceylon and Kotlin.

What you need for this book

You need the Java Development Kit (JDK) 9, NetBeans 8.2 or newer, and Maven 3.0 or newer. Some chapters will require additional software, including Scene Builder from Gluon and Android Studio.

Who this book is for

This book is for beginner to intermediate developers who are interested in seeing new and varied APIs and programming techniques applied in practical examples. Deep understanding of Java is not required, but a basic familiarity with the language and its ecosystem, build tools, and so on is assumed.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The Java architects have introduced a new file, `module-info.java`, similar to the existing `package-info.java` file, found at the root of the module, for example at `src/main/java/module-info.java`."

A block of code is set as follows:

```
module com.steeplesoft.foo.intro {
  requires com.steeplesoft.bar;
  exports com.steeplesoft.foo.intro.model;
  exports com.steeplesoft.foo.intro.api;
}
```

Any command-line input or output is written as follows:

```
$ mvn -Puber install
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "In the **New Project** window, we select **Maven** then **NetBeans Application**."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.
To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.
If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Java-9-Programming-Blueprints`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

aownloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/Java9ProgrammingBlueprints_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Introduction

In the process of erecting a new building, a set of blueprints helps all related parties communicate--the architect, electricians, carpenters, plumbers, and so on. It details things such as shapes, sizes, and materials. Without them, each of the subcontractors would be left guessing as to what to do, where to do it, and how. Without these blueprints, modern architecture would be almost impossible.

What is in your hands--or on the screen in front of you--is a set of blueprints of a different sort. Rather than detailing exactly how to build your specific software system, as each project and environment has unique constraints and requirements, these blueprints offer examples of how to build a variety of Java-based systems, providing examples of how to use specific features in the **Java Development Kit**, or **JDK**, with a special focus on the new features of Java 9 that you can then apply to your specific problem.

Since it would be impossible to build an application using only the new Java 9 features, we will also be using and highlighting many of the newest features in the JDK. Before we get too far into what that entails, then, let's take a brief moment to discuss some of these great new features from recent major JDK releases. Hopefully, most Java shops are already on Java 7, so we'll focus on version 8 and, of course, version 9.

In this chapter, we will cover the following topics:

- New features in Java 8
- New features in Java 9
- Projects

New features in Java 8

Java 8, released on March 8, 2014, brought arguably two of the most significant features since Java 5, released in 2004--lambdas and streams. With functional programming gaining popularity in the JVM world, especially with the help of languages such as Scala, Java adherents had been clamoring for more functional-style language features for several years. Originally slated for release in Java 7, the feature was dropped from that release, finally seeing a stable release with Java 8.

While it can be hoped that everyone is familiar with Java's lambda support, experience has shown that many shops, for a variety of reasons, are slow to adopt new language versions and features, so a quick introduction might be helpful.

Lambdas

The term lambda, which has its roots in lambda calculus, developed by Alonzo Church in 1936, simply refers to an anonymous function. Typically, a function (or method, in more proper Java parlance), is a statically-named artifact in the Java source:

```
public int add(int x, int y) {
  return x + y;
}
```

This simple method is one named `add` that takes two `int` parameters as well as returning an `int` parameter. With the introduction of lambdas, this can now be written as follows:

```
(int x, int y) → x + y
```

Or, more simply as this:

```
(x, y) → x + y
```

This abbreviated syntax indicates that we have a function that takes two parameters and returns their sum. Depending on where this lambda is used, the types of the parameters can be inferred by the compiler, making the second, even more concise format possible. Most importantly, though, note that this method is no longer named. Unless it is assigned to a variable or passed as a parameter (more on this later), it can not be referenced--or used--anywhere in the system.

This example, of course, is absurdly simple. A better example of this might be in one of the many APIs where the method's parameter is an implementation of what is known as a **Single Abstract Method** (**SAM**) interface, which is, at least until Java 8, an interface with a single method. One of the canonical examples of a SAM is `Runnable`. Here is an example of the pre-lambda `Runnable` usage:

```
Runnable r = new Runnable() {
  public void run() {
    System.out.println("Do some work");
  }
};
Thread t = new Thread(r);
t.start();
```

With Java 8 lambdas, this code can be vastly simplified to this:

```
Thread t = new Thread(() ->
  System.out.println("Do some work"));
t.start();
```

The body of the `Runnable` method is still pretty trivial, but the gains in clarity and conciseness should be pretty obvious.

While lambdas are anonymous functions (that is, they have no names), Java lambdas, as is the case in many other languages, can also be assigned to variables and passed as parameters (indeed, the functionality would be almost worthless without this capability). Revisiting the `Runnable` method in the preceding code, we can separate the declaration and the use of `Runnable` as follows:

```
Runnable r = () {
  // Acquire database connection
  // Do something really expensive
};
Thread t = new Thread(r);
t.start();
```

This is intentionally more verbose than the preceding example. The stubbed out body of the `Runnable` method is intended to mimic, after a fashion, how a real-world `Runnable` may look and why one may want to assign the newly-defined `Runnable` method to a variable in spite of the conciseness that lambdas offer. This new lambda syntax allows us to declare the body of the `Runnable` method without having to worry about method names, signatures, and so on. It is true that any decent IDE would help with this kind of boilerplate, but this new syntax gives you, and the countless developers who will maintain your code, much less noise to have to parse when debugging the code.

Any SAM interface can be written as a lambda. Do you have a comparator that you really only need to use once?

```
List<Student> students = getStudents();
students.sort((one, two) -> one.getGrade() - two.getGrade());
```

How about `ActionListener`?

```
saveButton.setOnAction((event) -> saveAndClose());
```

Additionally, you can use your own SAM interfaces in lambdas as follows:

```
public <T> interface Validator<T> {
  boolean isValid(T value);
}
cardProcessor.setValidator((card)
card.getNumber().startsWith("1234"));
```

One of the advantages of this approach is that it not only makes the consuming code more concise, but it also reduces the level of effort, such as it is, in creating some of these concrete SAM instances. That is to say, rather than having to decide between an anonymous class and a concrete, named class, the developer can declare it inline, cleanly and concisely.

In addition to the SAMs Java developers have been using for years, Java 8 introduced a number of functional interfaces to help facilitate more functional style programming. The Java 8 Javadoc lists 43 different interfaces. Of these, there are a handful of basic function **shapes** that you should know of, some of which are as follows:

`BiConsumer<T,U>`	This represents an operation that accepts two input arguments and returns no result
`BiFunction<T,U,R>`	This represents a function that accepts two arguments and produces a result
`BinaryOperator<T>`	This represents an operation upon two operands of the same type, producing a result of the same type as the operands
`BiPredicate<T,U>`	This represents a predicate (Boolean-valued function) of two arguments
`Consumer<T>`	This represents an operation that accepts a single input argument and returns no result
`Function<T,R>`	This represents a function that accepts one argument and produces a result

`Predicate<T>`	This represents a predicate (Boolean-valued function) of one argument
`Supplier<T>`	This represents a supplier of results

There are a myriad of uses for these interfaces, but perhaps the best way to demonstrate some of them is to turn our attention to the next big feature in Java 8--Streams.

Streams

The other major addition to Java 8, and, perhaps where lambdas shine the brightest, is the new **Streams API**. If you were to search for a definition of Java streams, you would get answers that range from the somewhat circular **a stream of data elements** to the more technical **Java streams are monads**, and they're probably both right. The Streams API allows the Java developer to interact with a stream of data elements via a **sequence of steps**. Even putting it that way isn't as clear as it could be, so let's see what it means by looking at some sample code.

Let's say you have a list of grades for a particular class. You would like to know what the average grade is for the girls in the class. Prior to Java 8, you might have written something like this:

```
double sum = 0.0;
int count = 0;
for (Map.Entry<Student, Integer> g : grades.entrySet()) {
  if ("F".equals(g.getKey().getGender())) {
    count++;
    sum += g.getValue();
  }
}
double avg = sum / count;
```

We initialize two variables, one to store the sums and one to count the number of hits. Next, we loop through the grades. If the student's gender is female, we increment our counter and update the sum. When the loop terminates, we then have the information we need to calculate the average. This works, but it's a bit verbose. The new Streams API can help with that:

```
double avg = grades.entrySet().stream()
 .filter(e -> "F".equals(e.getKey().getGender())) // 1
 .mapToInt(e -> e.getValue()) // 2
 .average() // 3
 .getAsDouble(); //4
```

This new version is not significantly smaller, but the purpose of the code is much clearer. In the preceding pre-stream code, we have to play computer, parsing the code and teasing out its intended purpose. With streams, we have a clear, declarative means to express application logic. For each entry in the map do the following:

1. Filter out each entry whose `gender` is not `F`.
2. Map each value to the primitive int.
3. Average the grades.
4. Return the value as a double.

With the stream-based and lamba-based approach, we don't need to declare temporary, intermediate variables (grade count and total), and we don't need to worry about calculating the admittedly simple average. The JDK does all of the heavy-lifting for us.

The new java.time package

While lambdas and streams are extremely important game-changing updates, with Java 8, we were given another long-awaited change that was, at least in some circles, just as exciting: a new date/time API. Anyone who has worked with dates and times in Java knows the pain of `java.util.Calendar` and company. Clearly, you can get your work done, but it's not always pretty. Many developers found the API too painful to use, so they integrated the extremely popular Joda Time library into their projects. The Java architects agreed, and engaged Joda Time's author, Stephen Colebourne, to lead JSR 310, which brought a version of Joda Time (fixing various design flaws) to the platform. We'll take a detailed look at how to use some of these new APIs in our date/time calculator later in the book.

Default methods

Before turning our attention to Java 9, let's take a look at one more significant language feature: default methods. Since the beginning of Java, an interface was used to define how a class looks, implying a certain type of behavior, but was unable to implement that behavior. This made polymorphism much simpler in a lot of cases, as any number of classes could implement a given interface, and the consuming code treats them as that interface, rather than whatever concrete class they actually are.

One of the problems that have confronted API developers over the years, though, was how to evolve an API and its interfaces without breaking existing code. For example, take the `ActionSource` interface from the JavaServer Faces 1.1 specification. When the JSF 1.2 expert group was working on the next revision of the specification, they identified the need to add a new property to the interface, which would result in two new methods--the getters and setters. They could not simply add the methods to the interface, as that would break every implementation of the specification, requiring the maintainers of the implementation to update their classes. Obviously, this sort of breakage is unacceptable, so JSF 1.2 introduced `ActionSource2`, which extends `ActionSource` and adds the new methods. While this approach is considered ugly by many, the 1.2 expert group had a few choices, and none of them were very good.

With Java 8, though, interfaces can now specify a default method on the interface definition, which the compiler will use for the method implementation if the extending class does not provide one. Let's take the following piece of code as an example:

```
public interface Speaker {
  void saySomething(String message);
}
public class SpeakerImpl implements Speaker {
  public void saySomething(String message) {
    System.out.println(message);
  }
}
```

We've developed our API and made it available to the public, and it's proved to be really popular. Over time, though, we've identified an improvement we'd like to make: we'd like to add some convenience methods, such as `sayHello()` and `sayGoodbye()`, to save our users a little time. However, as discussed earlier, if we just add these new methods to the interface, we'll break our users' code as soon as they update to the new version of the library. Default methods allow us to extend the interface and avoid the breakage by defining an implementation:

```
public interface Speaker {
  void saySomething(String message);
  default public void sayHello() {
    System.out.println("Hello");
  }
  default public void sayGoodbye() {
    System.out.println("Good bye");
  }
}
```

Now, when users update their library JARs, they immediately gain these new methods and their behavior, without making any changes. Of course, to use these methods, the users will need to modify their code, but they need not do so until--and if--they want to.

New features in Java 9

As with any new version of the JDK, this release was packed with a lot of great new features. Of course, what is most appealing will vary based on your needs, but we'll focus specifically on a handful of these new features that are most relevant to the projects we'll build together. First up is the most significant, the Java Module System.

Java Platform Module System/Project Jigsaw

Despite being a solid, feature-packed release, Java 8 was considered by a fair number to be a bit disappointing. It lacked the much anticipated **Java Platform Module System** (**JPMS**), also known more colloquially, though not quite accurately, as Project Jigsaw. The Java Platform Module System was originally slated to ship with Java 7 in 2011, but it was deferred to Java 8 due to some lingering technical concerns. Project Jigsaw was started not only to finish the module system, but also to modularize the JDK itself, which would help Java SE scale down to smaller devices, such as mobile phones and embedded systems. Jigsaw was scheduled to ship with Java 8, which was released in 2014, but it was deferred yet again, as the Java architects felt they still needed more time to implement the system correctly. At long last, though, Java 9 will finally deliver this long-promised project.

That said, what exactly is it? One problem that has long haunted API developers, including the JDK architects, is the inability to hide implementation details of public APIs. A good example from the JDK of private classes that developers should not be using directly is the `com.sun.*/sun.*` packages and classes. A perfect example of this--of private APIs finding widespread public use--is the `sun.misc.Unsafe` class. Other than a strongly worded warning in Javadoc about not using these internal classes, there's little that could be done to prevent their use. Until now.

With the JPMS, developers will be able to make implementation classes public so that they may be easily used inside their projects, but not expose them outside the module, meaning they are not exposed to consumers of the API or library. To do this, the Java architects have introduced a new file, `module-info.java`, similar to the existing `package-info.java` file, found at the root of the module, for example, at `src/main/java/module-info.java`. It is compiled to `module-info.class`, and is available at runtime via reflection and the new `java.lang.Module` class.

So what does this file do, and what does it look like? Java developers can use this file to name the module, list its dependencies, and express to the system, both compile and runtime, which packages are exported to the world. For example, suppose, in our preceding stream example, we have three packages: `model`, `api`, and `impl`. We want to expose the models and the API classes, but not any of the implementation classes. Our `module-info.java` file may look something like this:

```
module com.packt.j9blueprints.intro {
  requires com.foo;
  exports com.packt.j9blueprints.intro.model;
  exports com.packt.j9blueprints.intro.api;
}
```

This definition exposes the two packages we want to export, and also declares a dependency on the `com.foo` module. If this module is not available at compile-time, the project will not build, and if it is not available at runtime, the system will throw an exception and exit. Note that the `requires` statement does not specify a version. This is intentional, as it was decided not to tackle the version-selection issue as part of the module system, leaving that to more appropriate systems, such as build tools and containers.

Much more could be said about the module system, of course, but an exhaustive discussion of all of its features and limitations is beyond the scope of this book. We will be implementing our applications as modules, though, so we'll see the system used--and perhaps explained in a bit more detail--throughout the book.

Those wanting a more in-depth discussion of the Java Platform Module System can search for the article, *The State of the Module System*, by Mark Reinhold.

Process handling API

In prior versions of Java, developers interacting with native operating system processes had to use a fairly limited API, with some operations requiring resorting to native code. As part of **Java Enhancement Proposal** (**JEP**) 102, the Java process API was extended with the following features (quoting from the JEP text):

- The ability to get the pid (or equivalent) of the current Java virtual machine and the pid of processes created with the existing API.
- The ability to enumerate processes on the system. Information on each process may include its pid, name, state, and perhaps resource usage.

- The ability to deal with process trees; in particular, some means to destroy a process tree.
- The ability to deal with hundreds of subprocesses, perhaps multiplexing the output or error streams to avoid creating a thread per subprocess.

We will explore these API changes in our first project, the Process Viewer/Manager (see the following sections for details).

Concurrency changes

As was done in Java 7, the Java architects revisited the concurrency libraries, making some much needed changes, this time in order to support the reactive-streams specification. These changes include a new class, `java.util.concurrent.Flow`, with several nested interfaces: `Flow.Processor`, `Flow.Publisher`, `Flow.Subscriber`, and `Flow.Subscription`.

REPL

One change that seems to excite a lot of people isn't a language change at all. It's the addition of a **REPL** (**Read-Eval-Print-Loop**), a fancy term for a language shell. In fact, the command for this new tool is `jshell`. This tool allows us to type or paste in Java code and get immediate feedback. For example, if we wanted to experiment with the Streams API discussed in the preceding section, we could do something like this:

```
$ jshell
|  Welcome to JShell -- Version 9-ea
|  For an introduction type: /help intro

jshell> List<String> names = Arrays.asList(new String[]{"Tom", "Bill",
"Xavier", "Sarah", "Adam"});
names ==> [Tom, Bill, Xavier, Sarah, Adam]

jshell> names.stream().sorted().forEach(System.out::println);
Adam
Bill
Sarah
Tom
Xavier
```

This is a very welcome addition that should help Java developers rapidly prototype and test their ideas.

Projects

With that brief and high-level overview of what new features are available to use, what do these blueprints we'll cover look like? We'll build ten different applications, varying in complexity and kind, and covering a wide range of concerns. With each project, we'll pay special attention to the new features we're highlighting, but we'll also see some older, tried and true language features and libraries used extensively, with any interesting or novel usages flagged. Here, then, is our project lineup.

Process Viewer/Manager

We will explore some of the improvements to the process handling APIs as we implement a Java version of the age old Unix tool--**top**. Combining this API with JavaFX, we'll build a graphical tool that allows the user to view and manage processes running on the system.

This project will cover the following:

- Java 9 Process API enhancements
- JavaFX

Duplicate File Finder

As a system ages, the chances of clutter in the filesystem, especially duplicated files, increases exponentially, it seems. Leveraging some of the new File I/O libraries, we'll build a tool to scan a set of user-specified directories to identify duplicates. Pulling JavaFX back out of the toolbox, we'll add a graphical user interface that will provide a more user-friendly means to interactively process the duplicates.

This project will cover the following:

- Java File I/O
- Hashing libraries
- JavaFX

Date Calculator

With the release of Java 8, Oracle integrated a new library based on a redesign of Joda Time, more or less, into the JDK. Officially known as JSR 310, this new library fixed a longstanding complaint with the JDK--the official date libraries were inadequate and hard to use. In this project, we'll build a simple command-line date calculator that will take a date and, for example, add an arbitrary amount of time to it. Consider the following piece of code for example:

```
$ datecalc "2016-07-04 + 2 weeks"
2016-07-18
$ datecalc "2016-07-04 + 35 days"
2016-08-08
$ datecalc "12:00CST to PST"
10:00PST
```

This project will cover the following:

- Java 8 Date/Time APIs
- Regular expressions
- Java command-line libraries

Social Media Aggregator

One of the problems with having accounts on so many social media networks is keeping tabs on what's happening on each of them. With accounts on Twitter, Facebook, Google+, Instagram, and so on, active users can spend a significant amount of time jumping from site to site, or app to app, reading the latest updates. In this chapter, we'll build a simple aggregator app that will pull the latest updates from each of the user's social media accounts and display them in one place. The features will include the following:

- Multiple accounts for a variety of social media networks:
    - Twitter
    - Pinterest
    - Instagram
- Read-only, rich listings of social media posts
- Links to the appropriate site or app for a quick and easy follow-up
- Desktop and mobile versions

This project will cover the following:

- REST/HTTP clients
- JSON processing
- JavaFX and Android development

Given the size and scope of this effort, we'll actually do this in two chapters: JavaFX in the first, and Android in the second.

Email filter

Managing email can be tricky, especially if you have more than one account. If you access your mail from more than one location (that is, from more than one desktop or mobile app), managing your email rules can be trickier still. If your mail system doesn't support rules stored on the server, you're left deciding where to put the rules so that they'll run most often. With this project, we'll develop an application that will allow us to author a variety of rules and then run them via an optional background process to keep your mail properly curated at all times.

A sample `rules` file may look something like this:

```
[
  {
    "serverName": "mail.server.com",
    "serverPort": "993",
    "useSsl": true,
    "userName": "me@example.com",
    "password": "password",
    "rules": [
       {"type": "move",
           "sourceFolder": "Inbox",
           "destFolder": "Folder1",
           "matchingText": "someone@example.com"},
        {"type": "delete",
           "sourceFolder": "Ads",
           "olderThan": 180}
     ]
  }
]
```

This project will cover the following:

- JavaMail
- JavaFX
- JSON Processing
- Operating System integration
- File I/O

JavaFX photo management

The Java Development Kit has a very robust assortment of image handling APIs. In Java 9, these were augmented with improved support for the TIFF specification. In this chapter, we'll exercise this API in creating an image/photo management application. We'll add support for importing images from user-specified locations into the configured official directory. We'll also revisit the duplicate file finder and reuse some of the code developed as a part of the project to help us identify duplicate images.

This project will cover the following:

- The new `javax.imageio` package
- JavaFX
- NetBeans Rich Client Platform
- Java file I/O

A client/server note application

Have you ever used a cloud-based note-taking application? Have you wondered what it would take to make your own? In this chapter, we'll create such an application, with complete front and backends. On the server side, we'll store our data in the ever popular document database, MongoDB, and we'll expose the appropriate parts of the business logic for the application via REST interfaces. On the client side, we'll develop a very basic user interface in JavaScript that will let us experiment with, and demonstrate how to use, JavaScript in our Java project.

This project will cover the following:

- Document databases (MongoDB)
- JAX-RS and RESTful interfaces
- JavaFX
- JavaScript and Vue 2

Serverless Java

Serverless, also known as **function as a service** (**FaaS**), is one of the hottest trends these days. It is an application/deployment model where a small function is deployed to a service that manages almost every aspect of the function--startup, shutdown, memory, and so on, freeing the developer from worrying about such details. In this chapter, we'll write a simple serverless Java application to see how it might be done, and how you might use this new technique for your own applications.

This project will cover the following:

- Creating an Amazon Web Services account
- Configuring AWS Lambda, Simple Notification Service, Simple Email Service, and DynamoDB
- Writing and deploying a Java function

Android desktop synchronization client

With this project, we'll change gears a little bit and focus specifically on a different part of the Java ecosystem: Android. To do this, we'll focus on a problem that still plagues some Android users--the synchronization of an Android device and a desktop (or laptop) system. While various cloud providers are pushing us to store more and more in the cloud and streaming that to devices, some people still prefer to store, for example, photos and music directly on the device for a variety of reasons, ranging from cost for cloud resources to unreliable wireless connectivity and privacy concerns.

In this chapter, we'll build a system that will allow users to synchronize music and photos between their devices and their desktop or laptop. We'll build an Android application that provides the user interface to configure and monitor synchronization from the mobile device side as well as the Android Service that will perform the synchronization in the background, if desired. We will also build the related components on the desktop--a graphical application to configure and monitor the process from the desktop as well as a background process to handle the synchronization from the desktop side.

This project will cover the following:

- Android
- User interfaces
- Services
- JavaFX
- REST

Getting started

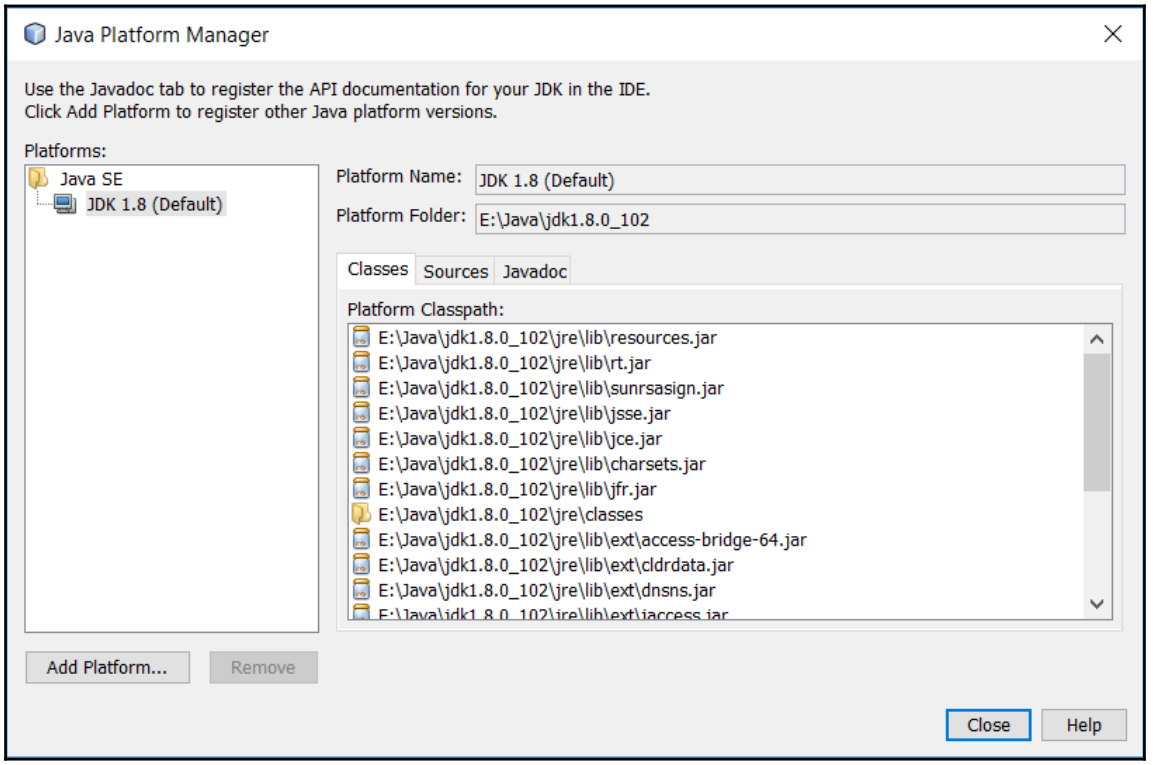
We have taken a quick look at some of the new language features we will be using. We have also seen a quick overview of the projects we will be building. One final question remains: what tools will we be using to do our work?

The Java ecosystem suffers from an embarrassment of riches when it comes to development tools, so we have much to choose from. The most fundamental choice facing us is the build tool. For our work here, we will be using Maven. While there is a strong and vocal community that would advocate Gradle, Maven seems to be the most common build tool at the moment, and seems to have more robust, mature, and native support from the major IDEs. If you do not have Maven already installed, you can visit `http://maven.apache.org` and download the distribution for your operating system, or use whatever package management system is supported by your OS.

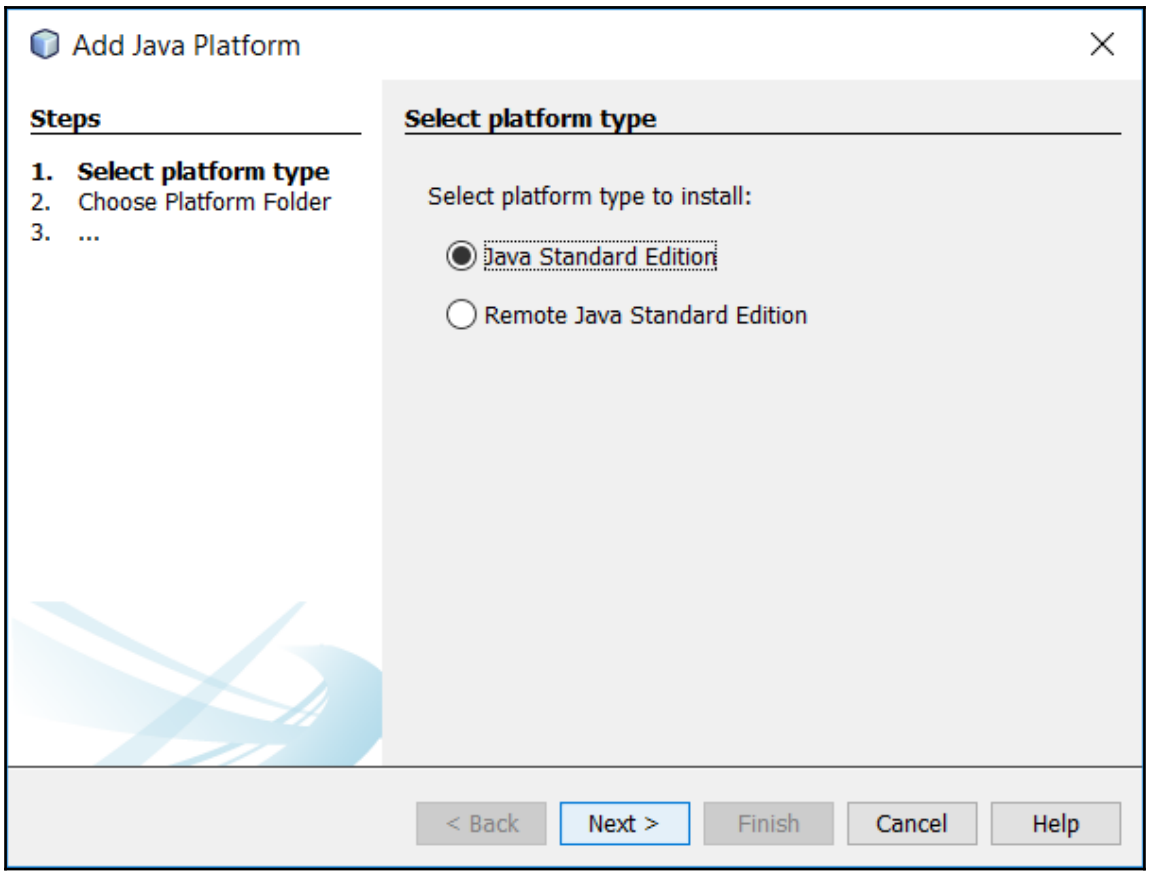
For the IDE, all screenshots, directions, and so forth will be using NetBeans--the free and open source IDE from Oracle. There are, of course, proponents of both IntelliJ IDEA and Eclipse, and they're both fine choices, but NetBeans offers a complete and robust development out-of-the-box, and it's fast, stable, and free. To download NetBeans, visit `http://netbeans.org` and download the appropriate installer for your operating system. Since we are using Maven, which IDEA and Eclipse both support, you should be able to open the projects presented here in the IDE of your choice. Where steps are shown in the GUI, though, you will need to adjust for the IDE you've chosen.

At the time of writing, the latest version of NetBeans is 8.2, and the best approach for using it to do Java 9 development is to run the IDE on Java 8, and to add Java 9 as an SDK. There is a development version of NetBeans that runs on Java 9, but, as it is a development version, it can be unstable from time to time. A stable NetBeans 9 should ship at roughly the same time as Java 9 itself. In the meantime, we'll push forward with 8.2:

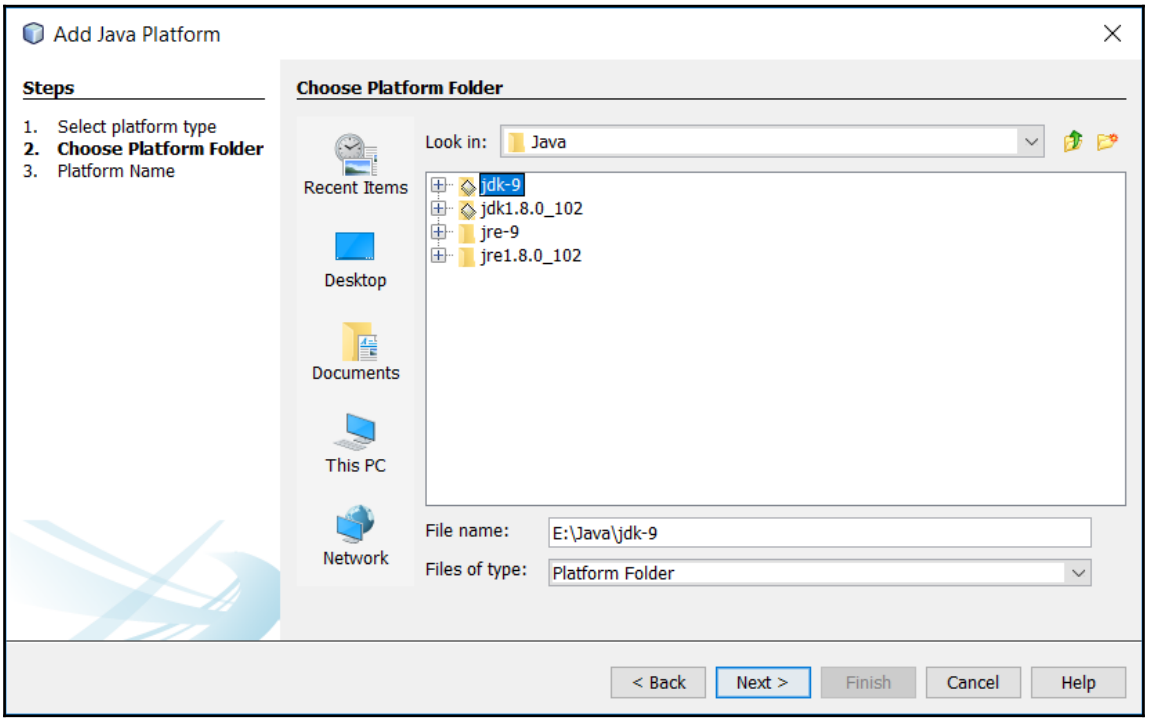
1. To add Java 9 support, we will need to add a new Java platform, and we will do that by clicking on **Tools** | **Platforms**.
2. This will bring up the **Java Platform Manager** screen:



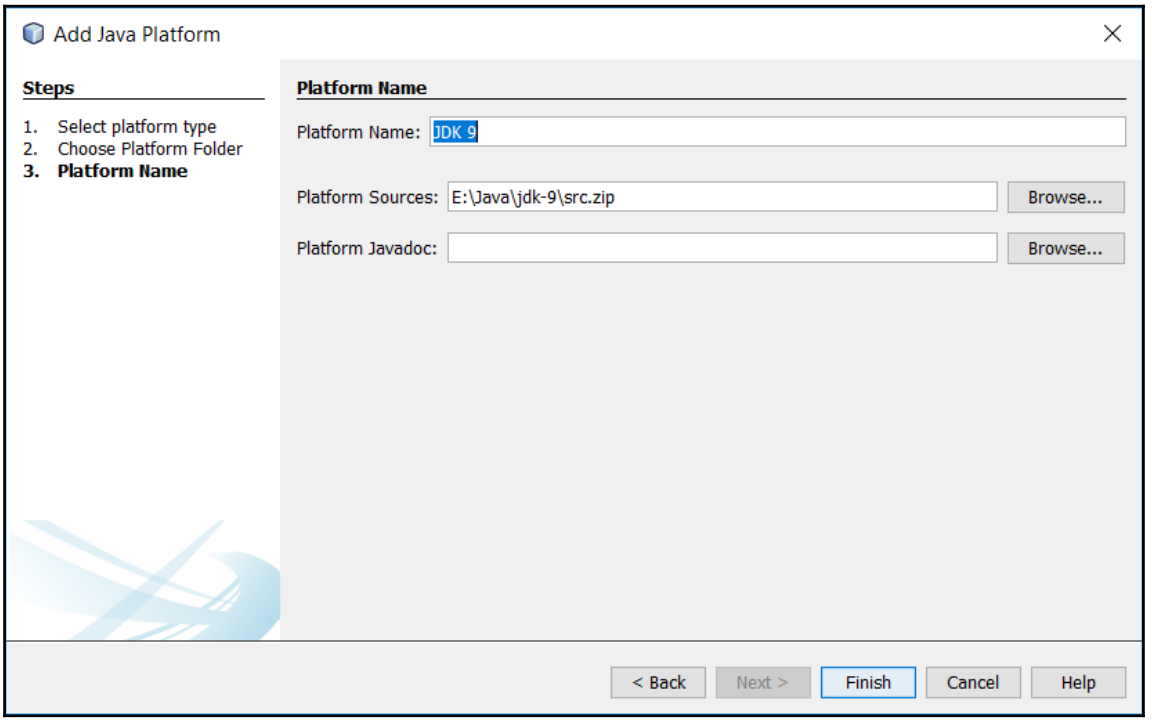
3. Click on **Add Platform...** on the lower left side of your screen.



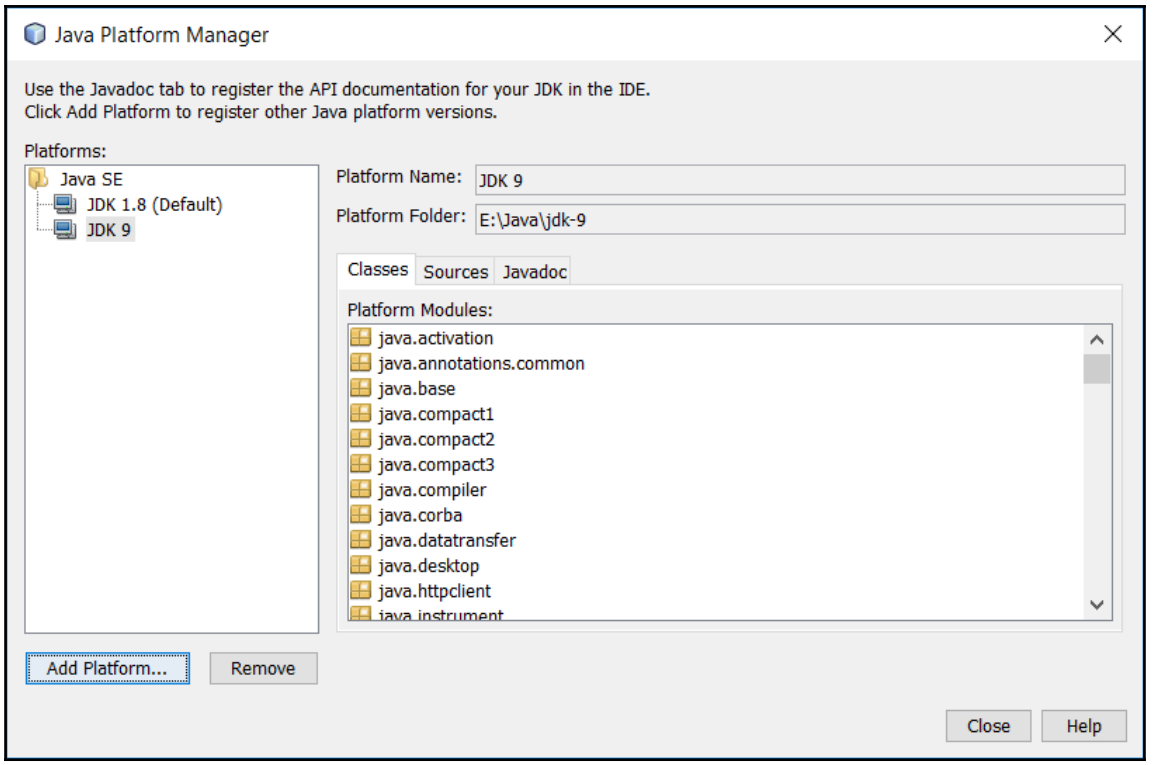
4. We want to add a **Java Standard Edition** platform, so we will accept the default and click on **Next**.



5. On the **Add Java Platform** screen, we will navigate to where we've installed Java 9, select the JDK directory, and click on **Next**.



6. We need to give the new Java Platform a name (NetBeans defaults to a very reasonable JDK 9) so we will click on **Finish** and can now see our newly added Java 9 option.



With the project SDK set, we're ready to take these new Java 9 features for a spin, which we'll start doing in `Chapter 2`, *Managing Processes in Java*.

If you do run NetBeans on Java 9, which should be possible by the time this book is published, you will already have Java 9 configured. You can, however, use the preceding steps to configure Java 8, should you need that version specifically.

Summary

In this chapter, we've taken a quick look at some of the great new features in Java 8, including lambdas, streams, the new date/time package, and default methods. From Java 9, we took a quick look at the Java Platform Module System and Project Jigsaw, the process handling APIs, the new concurrency changes, and the new Java REPL. For each, we've discussed the what and why, and looked at some examples of how these might affect the systems we write. We've also taken a look at the types of project we'll be building throughout the book and the tools we'll be using.

Before we move on, I'd like to restate an earlier point--every software project is different, so it is not possible to write this book in such a way that you can simply copy and paste large swathes of code into your project. Similarly, every developer writes code differently; the way I structure my code may be vastly different from yours. It is important, then, that you keep that in mind when reading this book and not get hung up on the details. The purpose here is not to show you the one right way to use these APIs, but to give you an example that you can look at to get a better sense of how they might be used. Learn what you can from each example, modify things as you see fit, and go build something amazing.

With all of that said, let's turn our attention to our first project, the Process Manager, and the new process handling APIs.

2
Managing Processes in Java

With a very quick tour through some of the big new features of Java 9, as well as those from a couple of previous releases, let's turn our attention to applying some of these new APIs in a practical manner. We'll start with a simple process manager.

While having your application or utility handle all of your user's concerns internally is usually ideal, occasionally you need to run (or **shell out to**) an external program for a variety of reasons. From the very first days of Java, this was supported by the JDK via the `Runtime` class via a variety of APIs. Here is the simplest example:

```
Process p = Runtime.getRuntime().exec("/path/to/program");
```

Once the process has been created, you can track its execution via the `Process` class, which has methods such as `getInputStream()`, `getOutputStream()`, and `getErrorStream()`. We have also had rudimentary control over the process via `destroy()` and `waitFor()`. Java 8 moved things forward by adding `destroyForcibly()` and `waitFor(long, TimeUnit)`. Starting with Java 9, these capabilities will be expanded. Quoting from the **Java Enhancement Proposal** (**JEP**), we see the following reasons for this new functionality:

Many enterprise applications and containers involve several Java virtual machines and processes and have long-standing needs that include the following:

- *The ability to get the pid (or equivalent) of the current Java virtual machine and the pid of processes created with the existing API.*
- *The ability to enumerate processes on the system. Information on each process may include its pid, name, state, and perhaps resource usage.*
- *The ability to deal with process trees, in particular, some means to destroy a process tree.*
- *The ability to deal with hundreds of sub-processes, perhaps multiplexing the output or error streams to avoid creating a thread per sub-process.*

In this chapter, we'll build a simple process manager application, akin to **Windows Task Manager** or *nix's top. There is, of course, little need for a process manager written in Java, but this will be an excellent avenue for us to explore these new process handling APIs. Additionally, we'll spend some time with other language features and APIs, namely, JavaFX and `Optional`.

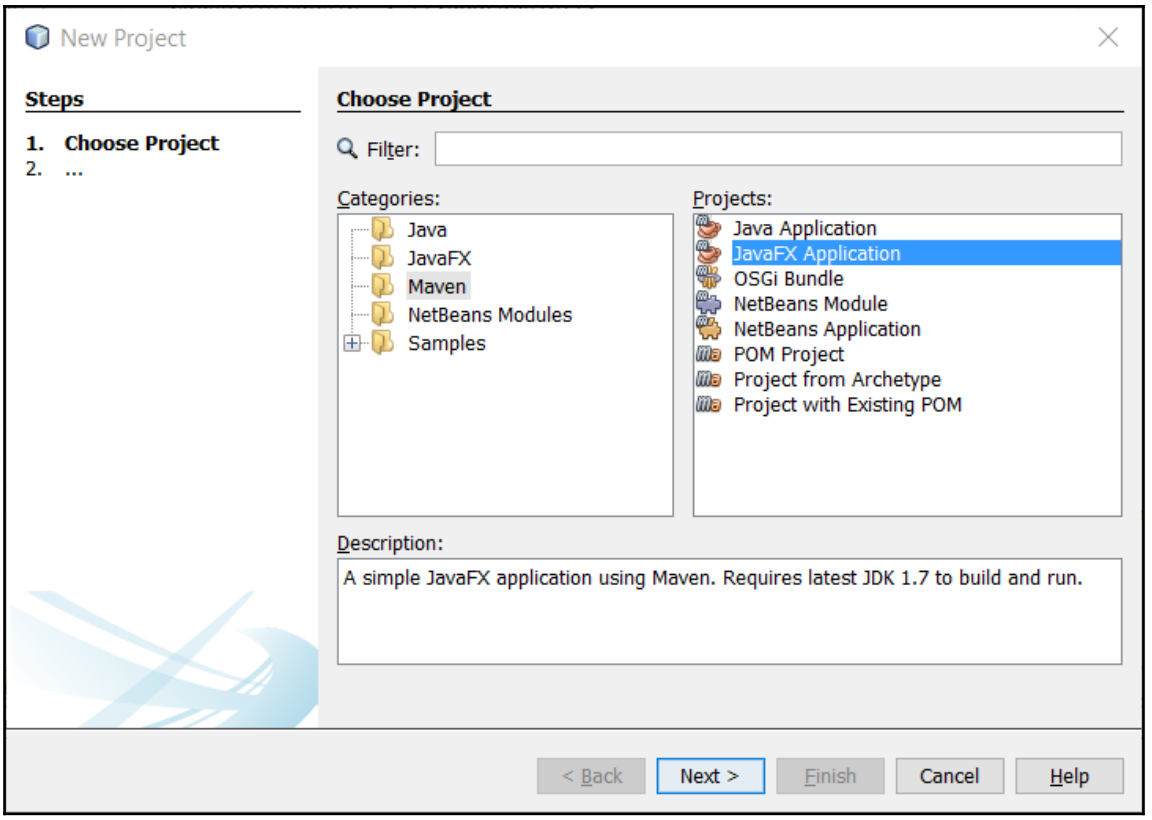
The following topics are covered in this chapter:

- Creating the project
- Bootstrapping the application
- Defining the user interface
- Initializing the user interface
- Adding menus
- Updating the process list

With that said, let's get started.

Creating a project

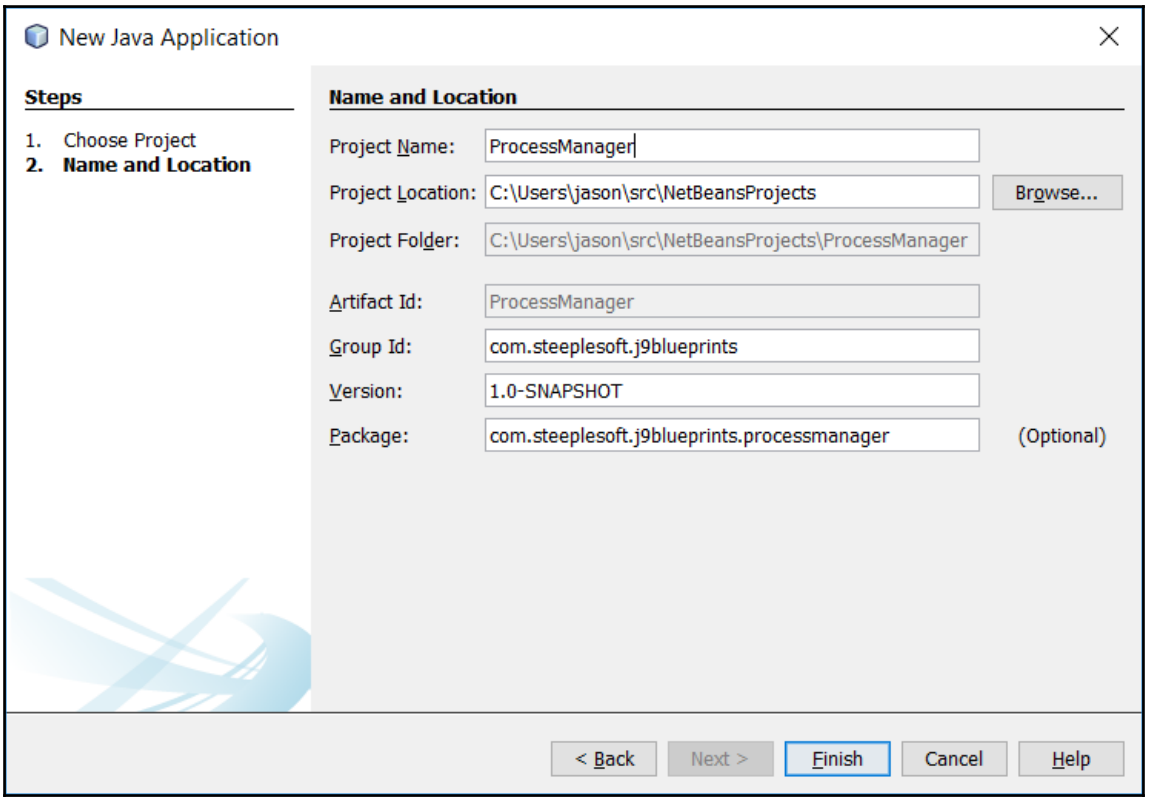
Typically speaking, it is much better if a build can be reproduced without requiring the use of a specific IDE or some other proprietary tool. Fortunately, NetBeans offers the ability to create a Maven-based JavaFX project. Click on **File** | **New Project** and select `Maven`, then **JavaFX Application**:



Next, perform the following steps:

1. Click on **Next**.
2. Enter **Project Name** as `ProcessManager`.
3. Enter **Group ID** as `com.steeplesoft`.
4. Enter **Package** as `com.steeplesoft.processmanager`.
5. Select **Project Location**.
6. Click on **Finish**.

Consider the following screenshot as an example:



Once the new project has been created, we need to update the Maven `pom` to use Java 9:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.6.1</version>
      <configuration>
        <source>9</source>
        <target>9</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Now, with both NetBeans and Maven configured to use Java 9, we're ready to start coding.

Bootstrapping the application

As noted in the introduction, this will be a JavaFX-based application, so we'll start by creating the skeleton for the application. This is a Java 9 application, and we intend to make use of the Java Module System. To do that, we need to create the module definition file, `module-info.java`, which resides in the root of our source tree. This being a Maven-based project, that would be `src/main/java`:

```
module procman.app {
  requires javafx.controls;
  requires javafx.fxml;
}
```

This small file does a couple of different things. First, it defines a new `procman.app` module. Next, it tells the system that this module `requires` two JDK modules: `javafx.controls` and `javafx.fxml`. If we did not specify these two modules, then our system, which we'll see below, would not compile, as the JDK would not make the required classes and packages available to our application. These modules are part of the standard JDK as of Java 9, so that shouldn't be an issue. However, that may change in future versions of Java, and this module declaration will help prevent runtime failures in our application by forcing the host JVM to provide the module or fail to start. It is also possible to build custom Java runtimes via the **J-Link** tool, so missing these modules is still a possibility under Java 9. With our module configured, let's turn to the application.

The emerging standard directory layout seems to be something like `src/main/java/<module1>`, `src/main/java/<module2>`, and so on. At the time of writing this book, while Maven can be coaxed into such a layout, the plugins themselves, while they do run under Java 9, do not appear to be module-aware enough to allow us to organize our code in such a manner. For that reason, and for the sake of simplicity, we will treat one Maven module as one Java module and maintain the standard source layout for the projects.

The first class we will create is the `Application` descendant, which NetBeans created for us. It created the `Main` class, which we renamed to `ProcessManager`:

```
public class ProcessManager extends Application {
  @Override
  public void start(Stage stage) throws Exception {
    Parent root = FXMLLoader
     .load(getClass().getResource("/fxml/procman.fxml"));
    Scene scene = new Scene(root);
    scene.getStylesheets().add("/styles/Styles.css");
    stage.setTitle("Process Manager");
```