Table of Contents

Introduction

Welcome to *Java for Android*.

This book is for you if you want to learn Java and specialize in Android application development. This book consists of two parts. Part I is focused on Java and Part II explains how to build Android applications effectively.

The Java chapters in this book do not teach you every Java technology there is. (It is impossible to cram everything into a single volume anyway, and that's why most Java titles are focused on one technology.) Rather, they cover the most important Java programming topics that you need to master to be able to learn other technologies yourself. In particular, Part I offers all the three subjects that a professional Java programmer must be proficient in:

- Java as a programming language;
- Object-oriented programming (OOP) with Java;
- Java core libraries.

What makes structuring an effective Java course difficult is the fact that the three subjects are interdependent. On the one hand, Java is an OOP language, so its syntax is easier to learn if you already know about OOP. On the other hand, OOP features such as inheritance, polymorphism, and data encapsulation, are best taught if accompanied by real-world examples. Unfortunately, understanding real-world Java programs requires knowledge of the Java core libraries.

Because of such interdependence, the three main topics are not grouped into three isolated parts. Instead, chapters discussing a major topic and chapters teaching another are interwoven. For example, before explaining polymorphism, this book makes sure that you are familiar with certain Java classes so that real-world examples can be given. In addition, because a language feature such as generics cannot be explained effectively without the comprehension of a certain set of classes, it is covered after the discussion of the supporting classes.

There are also cases whereby a topic can be found in two or more places. For instance, the **for** statement is a basic language feature that should be discussed in an early chapter. However, **for** can also be used to iterate over a collection of objects, a feature that should only be given after the Collections Framework is taught. Therefore, **for** is first presented in Chapter 3, "Statements" and then revisited in Chapter 11, "The Collections Framework."

Part II starts by introducing the Android framework and the tools a Java programmer needs to start developing apps. It then proceeds with essential topics in Android programming, including the Android user interface, bitmap and graphics processing, animation, audio/video recording, and asynchronous tasks.

The rest of this introduction presents a high-level overview of Java, an introduction to OOP, a brief description of each chapter, and instructions for installing the Java software.

Java, the Language and the Technology

Java is not only an object-oriented programming language, it is also a set of technologies that make software development more rapid and resulting applications more robust and secure. For years Java has been the technology of choice because of the benefits it offers:

- platform independence
- ease of use
- complete libraries that speed up application development
- security
- scalability
- extensive industry support

Sun Microsystems introduced Java in 1995 and Java—even though it had been a general-purpose language right from the start—was soon well known as the language for writing applets, small programs that run inside web browsers and add interactivity to static websites. The growth of the Internet had much to contribute to the early success of Java.

Having said that, applets were not the only factor that made Java shine. The other most appealing feature of Java was its platform-independence promise, hence the slogan "Write Once, Run Anywhere." What this means is the very same program you write will run on Windows, Unix, Mac, Linux, and other operating systems. This was something no other programming language could do. At that time, C and C++ were the two most commonly used languages for developing serious applications. Java seemed to have stolen their thunder since its first birthday.

That was Java version 1.0.

In 1997, Java 1.1 was released, adding significant features such as a better event model, Java Beans, and internationalization to the original.

Java 1.2 was launched in December 1998. Three days after it was released, the version number was changed to 2, marking the beginning of a huge marketing campaign that started in 1999 to sell Java as the "next generation" technology. Java 2 was sold in four flavors: the Standard Edition (J2SE), the Enterprise Edition (J2EE), the Micro Edition (J2ME), and Java Card (that never adopted "2" in its brand name).

The next version released in 2000 was 1.3, hence J2SE 1.3. 1.4 came two years later to make J2SE 1.4. J2SE version 1.5 was released in 2004. However, the name Java 2 version 1.5 was then changed to Java 5.

On November 13, 2006, a month before the official release date of Java 6, Sun Microsystems announced that it had open-sourced Java. Java SE 6 was the first Java release for which Sun Microsystems had invited outside developers to contribute code and help fix bugs. True that the company had in the past accepted contributions from non-employees, like the work of Doug Lea on multithreading, but this was the first time Sun had posted an open invitation. The company admitted that they had limited resources, and outside contributors would help them cross the finish line sooner.

In May 2007 Sun released its Java source code to the OpenJDK community as free software. IBM, Oracle and Apple later joined OpenJDK.

In 2010 Oracle acquired Sun.

Java 7, code-named Dolphin, was released in July 2011 and a result of open-source collaboration through OpenJDK. Java 8, the latest version, was released on March 18, 2014.

What Makes Java Platform Independent?

You must have heard of the terms "platform-independent" or "cross-platform," which means your program can run on multiple operating systems. This is one major factor that made Java popular. But, what makes Java platform independent?

In traditional programming, source code is compiled into executable code. This executable code can run only on the platform it is intended to run. In other words, code written and compiled for Windows will only run on Windows, code written in Linux will only run on Linux, and so on. This is depicted in Figure I.1.

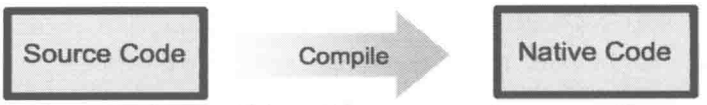

Figure I.1: Traditional programming paradigm

A Java program, on the other hand, is compiled to bytecode. You cannot run bytecode by itself because it is not native code. Bytecode can only run on a Java Virtual Machine (JVM). A JVM is a native application that interprets bytecode. By making the JVM available on many platforms, Sun transformed Java into a cross-platform language. As shown in Figure I.2, the very same bytecode can run on any operating system for which a JVM has been developed.

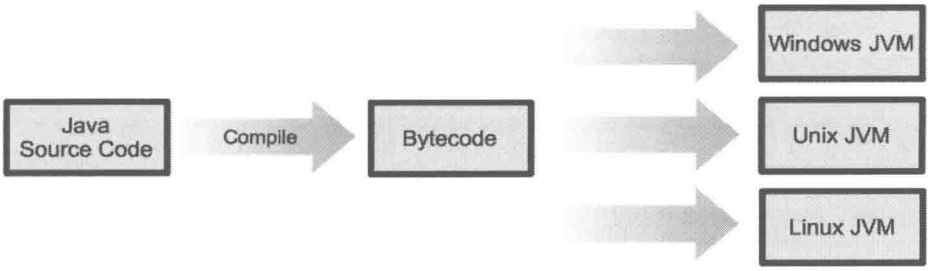

Figure I.2: Java programming model

Currently JVMs are available for Windows, Unix, Linux, Free BSD, and practically all other major operating systems in the world.

JDK, JRE, JVM, What's the Difference?

I mentioned that Java programs must be compiled. In fact, any programming language needs a compiler to be really useful. A compiler is a program that converts program source code to an executable format, either a bytecode, native code, or something else. Before you can start programming Java, you need to download a Java compiler. The Java compiler is a program named **javac**, which is short for Java compiler.

While **javac** can compile Java sources to bytecode, to run bytecode you need a Java Virtual Machine. In addition, because you will invariably use classes in the Java core libraries, you also need to download these libraries. The Java Runtime Environment (JRE) contains both a JVM and class libraries. As you may suspect, the JRE for Windows is different from that for Linux, which is different from the one for yet another operating system.

The Java software is available in two distributions:

- The JRE, which includes a JVM and the core libraries. This is good for running bytecode.
- The JDK, which includes the JRE plus a compiler and other tools. This is required software to write and compile Java programs.

To summarize, a JVM is a native application that runs bytecode. The JRE is an environment that includes a JVM and Java class libraries. The JDK includes the JRE plus other tools including a Java compiler.

The first version of the JDK is 1.0. The versions after that are 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, and 1.8. For minor releases, add another number to the version number. For instance, 1.7.1 is the first minor upgrade to version 1.7.

JDK 1.8 is better known as JDK 8. The version of the JRE included in a JDK is the same as the JDK. Therefore, JDK 1.8 contains JRE 1.8. The JDK is also often called the SDK (Software Development Kit).

In addition to the JDK, a Java programmer needs to download Java documentation that explains classes in the core libraries. You can download the documentation from the same URL that provides the JRE and the JDK.

Java 2, J2SE, J2EE, J2ME, Java 8, What Are They?

Sun Microsystems has done a great deal promoting Java. Part of its marketing strategy was to coin the name Java 2, which was basically JDK 1.2. There were three editions of Java 2:

- Java 2 Platform, Standard Edition (J2SE). J2SE is basically the JDK. It also serves as the foundation for technologies defined in J2EE.
- Java 2 Platform, Enterprise Edition (J2EE). It defines the standard for developing component-based multi-tier enterprise applications. Features include Web services support and development tools (SDK).
- Java 2 Platform, Micro Edition (J2ME). It provides an environment for applications that run on consumer devices, such as mobile phones, personal digital assistants (PDAs), and TV set-top boxes. J2ME includes a JVM and a limited set of class libraries.

Name changes occurred in version 5. J2SE became *Java Platform, Standard Edition 5* (Java SE 5). Also, the 2 in J2EE and J2ME was dropped. The current version of the enterprise edition is *Java Platform, Enterprise Edition 7* (Java EE 7). J2ME is now called *Java Platform, Micro Edition* (Java ME, without a version number).

Unlike the first versions of Java that were products of Sun, starting from version 1.4, Java SE is a set of specifications that define features that need to be implemented. The software itself is called a reference implementation. Oracle, IBM, and others work together through OpenJDK to provide the Java reference implementation and reference implementations for the next versions of Java.

Java EE 6 and 7 are also sets of specifications that include technologies such as servlets, JavaServer Pages, JavaServer Faces, Java Messaging Service, etc. To develop and run Java EE 6 or 7 applications, you need a Java EE 6 or 7 application server. Anyone can implement a Java EE application server, which explains the availability of various application servers in the market, including some open source ones. Here are examples of Java EE 6 and 7 application servers:

- Oracle WebLogic (previously BEA WebLogic)
- IBM WebSphere
- GlassFish

- JBoss
- Jonas
- Apache Geronimo

The complete list can be found here.

```
http://www.oracle.com/technetwork/java/javaee/overview/compatibility
-jsp-136984.html
```

JBoss, GlassFish, Jonas, and Geronimo are open source application servers. They have different licenses, though, so make sure you read them before you decide to use the products.

The Java Community Process (JCP) Program

Java's continuous dominance as the technology of choice owes much to Sun's strategy to include other industry players in determining the future of Java. This way, many people feel that they also own Java. Many large corporations, such as IBM, Oracle, Nokia, Fujitsu, etc, invest heavily in Java because they too can propose a specification for a technology and put forward what they want to see in the next version of a Java technology. This collaborative effort takes the form of the JCP Program. The URL of its Web site is http://www.jcp.org.

Specifications produced by the JCP Program are known as Java Specification Requests (JSRs). For example, JSR 336 specifies Java SE 7.

An Overview of Object-Oriented Programming

Object-oriented programming (OOP) works by modeling applications on real-world objects. Three principles of OOP are encapsulation, inheritance, and polymorphism.

The benefits of OOP are real. These are the reason why most modern programming languages, including Java, are object-oriented (OO). I can even cite two well-known examples of language transformation to support OOP: The C language evolved into C++ and Visual Basic was upgraded into Visual Basic.NET.

This section explains the benefits of OOP and provides an assessment of how easy or hard it is to learn OOP.

The Benefits of OOP

The benefits of OOP include easy code maintenance, code reuse, and extendibility. These benefits are presented in more detail below.

1. **Ease of maintenance**. Modern software applications tend to be very large. Once upon a time, a "large" system comprised a few thousand lines of code. Now, even those consisting of one million lines are not considered that large. When a system gets larger, it starts giving its developers problems. Bjarne Stroustrup, the father of C++, once said something like this. A small program can be written in anything, anyhow. If you don't quit easily, you'll make it work, at the end. But a large program is a different story. If you don't use techniques of "good programming," new errors will emerge as fast as you fix the old ones.
   The reason for this is there is interdependency among different parts of a large program. When you change something in some part of the program, you may not realize how the change might affect other parts. OOP makes it

easy to make applications modular, and modularity makes maintenance less of a headache. Modularity is inherent in OOP because a class, which is a template for objects, is a module by itself. A good design should allow a class to contain similar functionality and related data. An important and related term that is used often in OOP is coupling, which means the degree of interaction between two modules. Loosely coupling among parts make code reuse—another benefit of OOP—easier to achieve.

2. **Reusability**. Reusability means that code that has previously been written can be reused by the code author and others who need the same functionality provided by the original code. It is not surprising, then, that an OOP language often comes with a set of ready-to-use libraries. In the case of Java, the language is accompanied by hundreds of class libraries or Application Programming Interfaces (APIs) that have been carefully designed and tested. It is also easy to write and distribute your own library. Support for reusability in a programming platform is very attractive because it shortens development time.
   One of the main challenges to class reusability is creating good documentation for the class library. How fast can a programmer find a class that provides the functionality he/she is looking for? Is it faster to find such a class or write a new one from scratch? Fortunately, Java core and extended APIs come with extensive documentation.
   Reusability does not only apply to the coding phase through the reuse of classes and other types; when designing an application in an OO system, solutions to OO design problems can also be reused. These solutions are called design patterns. To make it easier to refer to each solution, each pattern is given a name. The early catalog of reusable design patterns can be found in the classic book *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
3. **Extendibility**
   Every application is unique. It has its own requirements and specifications. In terms of reusability, sometimes you cannot find an existing class that provides the exact functionality that your application requires. However, you will probably find one or two that provide part of the functionality. Extendibility means that you can still use those classes by extending them to suit your need. You still save time, because you don't have to write code from scratch.
   In OOP, extendibility is achieved through inheritance. You can extend an existing class, add some methods or data to it, or change the behavior of methods you don't like. If you know the basic functionality that will be used in many cases, but you don't want your class to provide very specific functions, you can provide a generic class that can be extended later to provide functionality specific to an application.

Is OOP Hard?

Java programmers need to master OOP. As it happens, it does make a difference if you have programmed using a procedural language, such as C or Pascal. In the light of this, there is bad news and good news.

First the bad news.

Researchers have been debating the best way to teach OOP in school; some argue that it is best to teach procedural programming before OOP is introduced. In many curricula, we see that an OOP course can be taken when a student is nearing the final year of his/her university term.

More recent studies, however, argue that someone with procedural programming skill thinks in a paradigm very different from how OO programmers view and try to solve problems. When this person needs to learn OOP, the greatest struggle he/she faces is having to go through a paradigm shift. It is said that it takes six to 18 months to switch your mindset from procedural to object-oriented paradigms. Another study shows that students who have not learned procedural programming do not find OOP that difficult.

Now the good news.

Java qualifies as one of the easiest OOP languages to learn. For example, you do not need to worry about pointers, don't have to spend precious time solving memory leaks caused by failing to destroy unused objects, etc. On top of that, Java comes with very comprehensive class libraries with relatively very few bugs in their early versions. Once you know the nuts and bolts of OOP, programming with Java is really easy.

About This Book

The following presents the overview of each chapter.

Part I: Java

Chapter 1, "Your First Taste of Java" aims at giving you the feel of working with Java. This includes writing a simple Java program, compiling it using the **javac** tool, and running it using the **java** program. In addition, some advice on code conventions and integrated development environments is also given.

Chapter 2, "Language Fundamentals", teaches you the Java language syntax. You will be introduced to topics such as character sets, primitives, variables, operators, etc.

Chapter 3, "Statements", explains Java statements **for**, **while**, **do-while**, **if**, **if-else**, **switch**, **break**, and **continue**.

Chapter 4, "Objects and Classes," is the first OOP lesson in this book. It starts by explaining what a Java object is an how it is stored in memory. It then continues with a discussion of classes, class members, and two OOP concepts (abstraction and encapsulation). Some related topics, such as garbage collection and object comparison, are briefly discussed.

Chapter 5, "Core Classes" covers important classes in the Java core libraries: **java.lang.Object**, **java.lang.String**, **java.lang.StringBuffer** and **java.lang.StringBuilder**, wrapper classes, and **java.util.Scanner**. Boxing/unboxing and varargs are also taught. This is an important chapter because the classes explained in this chapter are some of the most commonly used classes in Java.

Chapter 6, "Inheritance" discusses an OOP feature that enables code extendibility. This chapter teaches you how to extend a class, affect the visibility of a subclass, override a method, and so forth.

Undoubtedly, error handling is an important feature of any programming language. As a mature language, Java has a very robust error handling mechanism that can help prevent bugs from creeping in. Chapter 7, "Error Handling" is a detailed discussion of this mechanism.

Chapter 8, "Numbers and Dates" deals with three issues when working with numbers and dates: parsing, formatting, and manipulation. This chapter introduces Java classes that can help you with these tasks.

Chapter 9, "Interfaces and Abstract Classes", explains that an interface is more than a class without implementation. An interface defines a contract between a service provider and a client. This chapter explains how to work with interfaces and abstract classes.

Chapter 10, "Enums" covers enum, a type added to Java since version 5.

Chapter 11, "The Collections Framework" shows how you can use the members of the **java.util** package to group objects and manipulate them.

Generics are a very important feature in Java and Chapter 12, "Generics" adequately explains this feature.

Chapter 13, "Input/Output" introduces the concept of streams and explains how you can use the four stream types in the **java.io** package to perform input-output operations. In addition, object serialization and deserialization are discussed.

Chapter 14, "Nested and Inner Classes" explains how you can write a class within another class and why this OOP feature can be very useful.

Polymorphism is one of the main pillars of OOP. It is incredibly useful in situations whereby the type of an object in not known at compile time. Chapter 15, "Polymorphism" explains this feature and provides useful examples.

Chapter 16, "Annotations" talks about one of the features in Java. It explains the standard annotations that come with the JDK, common annotations, meta-annotations, and custom annotations.

Today it is common for software applications to be deployable to different countries and regions. Such applications need to be designed with internationalization in mind. Chapter 17, "Internationalization" explores techniques that Java programmers can use.

Chapter 18, "Java Networking" deals with classes that can be used in network programming. A simple Web server application is presented to illustrate how to use these classes.

A thread is a basic processing unit to which an operating system allocates processor time, and more than one thread can be executing code inside a process. Chapter 19, "Java Threads," shows that in Java multithreaded programming is not only the domain of expert programmers.

Chapter 20, "The Concurrency Utilities" is the second chapter on multi-threaded programming. It discusses interfaces and classes that make writing multi-threaded programs easier.

Part II: Android

Chapter 21, "Introduction to Android" introduces the Android framework and contains instructions to download and install the tools for developing apps.

Chapter 22, "Your First Android Application" shows how easy it is to create an application using the ADT Bundle.

Chapter 23, "Activities" explains the activity and its lifecycle. The activity is one of the most important concepts in Android programming.

Chapter 24, "UI Components" covers the more important UI components, including widgets, Toast, and AlertDialog.

Chapter 25, "Layouts" shows how to lay out UI components in Android applications and use the built-in layouts available in Android.

Chapter 26, "Listeners" talks about creating a listener to handle events.

Chapter 27, "The Action Bar" shows how you can add items to the action bar and use it to drive application navigation.

Menus are a common feature in many graphical user interface (GUI) systems whose primary role is to provide shortcuts to certain actions. Chapter 28, "Menus" looks at Android menus closely.

Chapter 29, "ListView" explains about **ListView**, a view that shows a scrollable list of items and gets its data source from a list adapter.

Chapter 30, "GridVIew" covers the **GridView** widget, a view similar to the **ListView**. Unlike the **ListView**, however, the **GridView** displays its items in a grid.

Chapter 31, "Styles and Themes" discusses the two important topics directly responsible for the look and feel of your apps.

Chapter 32, "Bitmap Processing" teaches you how to manipulate bitmap images. The techniques discussed in this chapter are useful even if you are not writing an image editor application.

The Android SDK comes with a wide range of views that you can use in your applications. If none of these suits your need, you can create a custom view and draw on it. Chapter 33, "Graphics and Custom Views" shows how to create a custom view and draw shapes on a canvas.

Chapter 34, "Fragments" discusses fragments, which are components that can be added to an activity. A fragment has its own lifecycle and has methods that get called when certain phases of its life occur.

Chapter 35, "Multi-Pane Layouts" shows how you can use different layouts for different screen sizes, like that of handsets and that of tablets.

Chapter 36, "Animation" discusses the latest Animation API in Android called property animation. It also provides an example.

Chapter 37, "Preferences" teaches you how to use the Preference API to store application settings and read them back.

Chapter 38, "Working with Files" show how to use the Java File API in an Android application.

Chapter 39, "Working with the Database" discusses the Android Database API, which you can use to connect to an SQLite database. SQLite is the default relational database that comes with every Android device.

Chapter 40, "Taking Pictures" teaches you how to take still images using the built-in Camera application and the Camera API.

Chapter 41, "Making Videos" shows the two methods for providing video-making capability in your application, by using a built-in intent or by using the **MediaRecorder** class.

Chapter 42, "The Sound Recorder" shows how you can record audio.