Part I

Getting Started with Java Programming for Android Developers

web extras

Visit `www.dummies.com` for great *For Dummies* content online.

In this part . . .

- ✓ Downloading the software
- ✓ Installing Java and Android
- ✓ Testing Android apps on your computer

Chapter 1

All about Java and Android

In This Chapter

- The consumer's view of the Android ecosystem
- The ten-cent tour of Java and Android technologies

Until the mid-2000s, the word *android* represented a mechanical, humanlike creature — a root'n-toot'n officer of the law with built-in machine guns or a hyperlogical space traveler who can do everything except speak using contractions. And then in 2005, Google purchased Android, Inc. — a 22-month old company creating software for mobile phones. That move changed everything.

In 2007, a group of 34 companies formed the Open Handset Alliance. Its task is "to accelerate innovation in mobile and offer consumers a richer, less expensive, and better mobile experience"; its primary project is *Android,* an open, free operating system based on the Linux operating system kernel.

Though HTC released the first commercially available Android phone near the end of 2008, in the United States the public's awareness of Android and its potential didn't surface until early 2010.

As I sit and write in mid-2013, Mobile Marketing Watch reports more than 50 billion downloads from the Google Play app store.[1] Android developers earned more from their apps in the first half of 2013 than in all of 2012. And according to *Forbes*, Google paid approximately $900 million to Android developers during the 12-month period starting in mid-2012.[2] The pace is accelerating.

[1]See `www.mobilemarketingwatch.com/google-play-tops-50-billion-app-downloads-34516/`.

[2]See `www.forbes.com/sites/tristanlouis/2013/08/10/how-much-do-average-apps-make/`.

The Consumer Perspective

A consumer considers the alternatives:

- **Possibility #1: No mobile phone.**

  Advantages: Inexpensive; no interruptions from callers.

  Disadvantages: No instant contact with friends and family; no calls to services in case of emergencies.

- **Possibility #2: A feature phone.**

  This type of mobile phone isn't a smartphone. Though no official rule defines the boundary between feature phone and smartphone, a feature phone generally has an inflexible menu of Home screen options compared with a smartphone's "desktop" of downloaded apps.

  Advantage: Less expensive than a smartphone.

  Disadvantages: Less versatile than a smartphone, not nearly as cool as a smartphone, and nowhere near as much fun as a smartphone.

- **Possibility #3: An iPhone.**

  Advantages: Great-looking graphics.

  Disadvantages: Little or no flexibility with the single-vendor iOS operating system; only a handful of models to choose from.

- **Possibility #4: A Windows phone, a BlackBerry, or another non-Android, non-Apple smartphone**

  Advantage: Having a smartphone without having to belong to a crowd.

  Disadvantage: The possibility of owning an orphan product when the smartphone wars come to a climax.

- **Possibility #5: An Android phone**

  Advantages: Using a popular, open platform with lots of industry support and powerful market momentum; writing your own software and installing it on your own phone (without having to post the software on a company's website); publishing software without having to face a challenging approval process.

  Disadvantages: Security concerns when using an open platform; dismay when iPhone users make fun of your phone.

For me, Android's advantages far outweigh its possible disadvantages. And you're reading a paragraph from *Java Programming For Android Developers For Dummies*, so you're likely to agree with me.

The Many Faces of Android

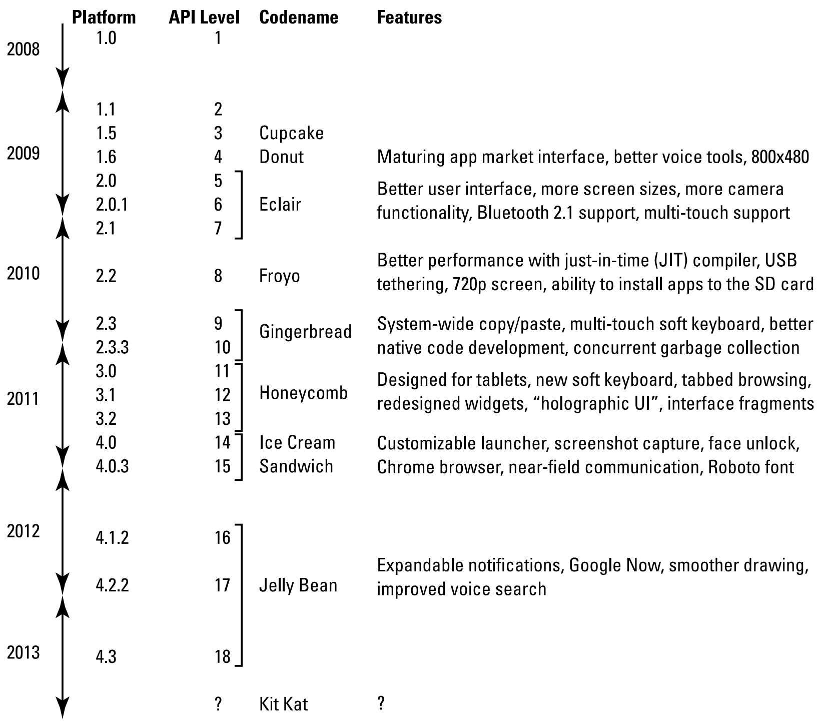
Version numbers can be tricky. My PC's model number is T420s. When I download the users' guide, I download one guide for any laptop in the T400 series. (No guide specifically addresses the T420, let alone the T420s.) But when I have driver problems, knowing that I have a T420s isn't good enough. I need drivers that are specific to my laptop's seven-digit model number. The moral to this story: What constitutes a "version number" depends on who's asking for the number.

With that in mind, you can see a history of Android versions in Figure 1-1.

A few notes on Figure 1-1 are in order:

- **The platform number is of interest to the consumer and to the company that sells the hardware.**

  If you're buying a phone with Android 4.2.2, for example, you might want to know whether the vendor will upgrade your phone to Android 4.3.



	Platform	API Level	Codename	Features
2008	1.0	1		
	1.1	2		
	1.5	3	Cupcake	
2009	1.6	4	Donut	Maturing app market interface, better voice tools, 800x480
	2.0	5	Eclair	Better user interface, more screen sizes, more camera functionality, Bluetooth 2.1 support, multi-touch support
	2.0.1	6		
	2.1	7		
2010	2.2	8	Froyo	Better performance with just-in-time (JIT) compiler, USB tethering, 720p screen, ability to install apps to the SD card
	2.3	9	Gingerbread	System-wide copy/paste, multi-touch soft keyboard, better native code development, concurrent garbage collection
	2.3.3	10		
	3.0	11	Honeycomb	Designed for tablets, new soft keyboard, tabbed browsing, redesigned widgets, "holographic UI", interface fragments
2011	3.1	12		
	3.2	13		
	4.0	14	Ice Cream Sandwich	Customizable launcher, screenshot capture, face unlock, Chrome browser, near-field communication, Roboto font
	4.0.3	15		
2012	4.1.2	16	Jelly Bean	Expandable notifications, Google Now, smoother drawing, improved voice search
	4.2.2	17		
2013	4.3	18		
		?	Kit Kat	?

Figure 1-1: Versions of Android.

- **The API level (also known as the SDK version) is of interest to the Android app developer.**

  For example, the word `MATCH_PARENT` has a specific meaning in Android API Levels 8 and higher. You might type `MATCH_PARENT` in code that uses API Level 7. If you do (and if you expect `MATCH_PARENT` to have that specific meaning), you'll get a nasty-looking error message.

  You can read more about the Application Programming Interface (API) in Chapter 2. For more information about the use of Android's API levels (SDK versions) in your code, see Chapter 4.

- **The code name is of interest to the creators of Android.**

  A *code name* refers to the work done by the creators of Android to bring Android to the next level. Picture Google's engineers working for months behind closed doors on Project Cupcake, and you'll be on the right track.

An Android version may have variations. For example, plain-old Android 2.2 has an established set of features. To plain-old Android 2.2 you can add the Google APIs (thus adding Google Maps functionality) and still be using platform 2.2. You can also add a special set of features tailored for the Samsung Galaxy Tab.

As a developer, your job is to balance portability with feature-richness. When you create an app, you specify a target Android version and a minimum Android version. (You can read more about this topic in Chapter 4.) The higher the version, the more features your app can have. But on the flip side, the higher the version, the fewer devices that can run your app.

The Developer Perspective

Android is a multifaceted beast. When you develop for the Android platform, you use many toolsets. This section gives you a brief rundown.

Java

James Gosling of Sun Microsystems created the Java programming language in the mid-1990s. (Sun Microsystems has since been bought by Oracle.) Java's meteoric rise in use stemmed from the elegance of the language and its well-conceived platform architecture. After a brief blaze of glory with applets and the web, Java settled into being a solid, general-purpose language with a special strength in servers and middleware.

In the meantime, Java was quietly seeping into embedded processors. Sun Microsystems was developing Java Mobile Edition (Java ME) for creating small apps to run on mobile phones. Java became a major technology in Blu-ray disc players. So the decision to make Java the primary development language for Android apps is no big surprise.

An *embedded processor* is a computer chip that is hidden from the user as part of a special-purpose device. The chips in cars are now embedded processors, and the silicon that powers the photocopier at your workplace is an embedded processor. Pretty soon, the flower pots on your windowsill will probably have embedded processors.

Figure 1-2 describes the development of new Java versions over time. Like Android, each Java version has several names. The *product version* is an official name that's used for the world in general, and the *developer version* is a number that identifies versions so that programmers can keep track of them. (In casual conversation, developers use all kinds of names for the various Java versions.) The *code name* is a more playful name that identifies a version while it's being created.

Year	Product Version	Developer Version	Codename	Features
1995	(Beta)			
1996	**JDK* 1.0**	1.0		
1997	**JDK 1.1**	1.1		Inner classes, Java Beans, reflection
1998	**J2SE* 1.2**	1.2	Playground	Collections, Swing classes for creation of GUI interfaces
1999				
2000	**J2SE 1.3**	1.3	Kestrel	Java Naming and Directory Interface (JNDI)
2001				
2002	**J2SE 1.4**	1.4	Merlin	New I/O, regular expressions, XML parsing
2003				
2004	**J2SE 5.0***	1.5	Tiger	Generic types, annotations, enum types, varargs, enhanced
2005				for statement, static imports, new concurrency classes
2006	**Java SE* 6**	1.6	Mustang	Scripting language support, performance enhancements
2007				
2008				
2009				
2010				
2011	**Java SE 7**	1.7	Dolphin	Strings in switch statement, catching multiple exceptions
2012				try statement with resources, integration with JavaFX
2013	**Java SE 8**	1.8		Lambda expressions

Figure 1-2: Versions of Java.

The asterisks in Figure 1-2 mark changes in the formulation of Java product-version names. Back in 1996, the product versions were *Java Development Kit 1.0* and *Java Development Kit 1.1*. In 1998, someone decided to christen the product *Java 2 Standard Edition 1.2*, which confuses everyone to this day. At the time, anyone using the term *Java Development Kit* was asked to use *Software Development Kit* (SDK) instead.

In 2004 the *1.* business went away from the platform version name, and in 2006 Java platform names lost the *2* and the *.0*.

By far the most significant changes for Java developers came about in 2004. With the release of J2SE 5.0, the overseers of Java made changes to the language by adding new features — features such as generic types, annotations, varargs, and the enhanced `for` statement.

CROSS-REFERENCE

To see Java annotations in action, go to Chapter 10. For examples of the use of generic types, varargs, and the enhanced for statement, see Chapter 12.

REMEMBER

If you compare Figures 1-1 and 1-2, you might notice that Android entered the scene when Java was in version Java SE 6. As a result, Java is frozen at version 6 for Android developers. When you develop an Android app, you can use J2SE 5.0 or Java SE 6. You cannot use Java SE 7 with strings in its `switch` statements or use Java SE 8 with its lambda expressions. But that's okay: As an Android developer, you probably won't miss these features.

XML

If you find View Source among your web browser's options one day and decide to use it, you'll see a bunch of HyperText Markup Language (HTML) tags. A *tag* is some text, enclosed in angle brackets, that describes something about its neighboring content.

For example, to create boldface type on a web page, a web designer writes

```
<b>Look at this!</b>
```

The `b` tags in angle brackets turn boldface type on and off.

The *M* in HTML stands for *Markup* — a general term describing any extra text that annotates a document's content. When you annotate a document's content, you embed information about the content into the document itself. For example, in the previous line of code, the content is `Look at this!` The markup (information about the content) consists of the tags `<b>` and `</b>`.

The HTML standard is an outgrowth of Standard Generalized Markup Language (SGML), an all-things-to-all-people technology for marking up documents for use by all kinds of computers running all kinds of software and sold by all kinds of vendors.

In the mid-1990s, a working group of the World Wide Web Consortium (W3C) began developing the eXtensible Markup Language, commonly known as *XML.* The working group's goal was to create a subset of SGML for use in transmitting data over the Internet. They succeeded. XML is now a well-established standard for encoding information of all kinds.

CROSS-REFERENCE

For an overview of XML, see the sidebar that describes it in Chapter 4.

Java is good for describing step-by-step instructions, and XML is good for describing the way things are (or the way they should be). A Java program says, "Do this and then do that." In contrast, an XML document says, "It's this way and it's that way." Android uses XML for two purposes:

- **To describe an app's data**

  An app's XML documents describe the layout of the app's screens, the translations of the app into one or more languages, and other kinds of data.

- **To describe the app itself**

  Every Android app has an `AndroidManifest.xml` file, an XML document that describe features of the app. A device's operating system uses the `AndroidManifest.xml` document's contents to manage the running of the app.

  For example, an app's `AndroidManifest.xml` file describes code that the app makes available for use by other apps. The same file describes the permissions that the app requests from the system. When you begin installing a new app, Android displays these permissions and asks for your permission to proceed with the installation. (I don't know about you, but I always read this list of permissions carefully. Yeah, right!)

CROSS-REFERENCE

For more information about the `AndroidManifest.xml` file, see Chapter 4.

Concerning XML, I have bad news and good news. The bad news is that XML isn't always easy to compose. At best, writing XML code is boring. At worst, writing XML code is downright confusing. The good news is that automated software tools compose most the world's XML code. As an Android programmer, the software on your development computer composes much of your app's XML code. You often tweak the XML code, read part of the code for information from its source, make minor changes, and compose brief additions. But you hardly ever create XML documents from scratch.

Linux

An *operating system* is a big program that manages the overall running of a computer or a device. Most operating systems are built in layers. An operating system's outer layers are usually in the user's face. For example, both Windows and Macintosh OS X have standard desktops. From the desktop, the user launches programs, manages windows, and does other important things.

An operating system's inner layers are (for the most part) invisible to the user. While the user plays Solitaire, for example, the operating system juggles processes, manages files, keeps an eye on security, and generally does the kinds of things that the user shouldn't have to micromanage.

At the deepest level of an operating system is the system's kernel. The *kernel* runs directly on the processor's hardware and does the low-level work required to make the processor run. In a truly layered system, higher layers accomplish work by making calls to lower layers. So an app with a specific hardware request sends the request (directly or indirectly) through the kernel.

The best-known, best-loved general purpose operating systems are Windows, Macintosh OS X (which is really Unix), and Linux. Both Windows and Mac OS X are the properties of their respective companies. But Linux is open source. That's one reason why your TiVo runs Linux and why the creators of Android based their platform on the Linux kernel.

As a developer, your most intimate contact with the Android operating system is via the command line, also known as the *Linux shell.* The shell uses commands such as `cd` to change to a directory, `ls` to list a directory's files and subdirectories, `rm` to delete files, and many others.

Google's Android Market has plenty of free terminal apps. A *terminal* app's interface is a plain-text screen on which you type Linux shell commands. And by using one of Android's developer tools, the Android Debug Bridge, you can issue shell commands to an Android device via your development computer. If you like getting your virtual hands dirty, the Linux shell is for you.

From Development to Execution with Java

Before Java became popular, running a computer program involved one translation step. Someone (or something) translated the code that a developer wrote into more cryptic code that a computer could actually execute. But then Java came along and added an extra translation layer, and then Android added another layer. This section describes all those layers.

What is a compiler?

A Java program (such as an Android application program) undergoes several translation steps between the time you write the program and the time a processor runs the program. One of the reasons is simple: Instructions that are convenient for processors to run are not convenient for people to write.

People can write and comprehend the code in Listing 1-1.

Listing 1-1: Java Source Code

```
public void checkVacancy(View view) {
    if (room.numGuests == 0) {
        label.setText("Available");
    } else {
        label.setText("Taken :-(");
    }
}
```

The Java code in Listing 1-1 checks for a vacancy in a hotel. You can't run the code in this listing without adding several additional lines. But here in Chapter 1, those additional lines aren't important. What's important is that, by staring at the code, squinting a bit, and looking past all its strange punctuation, you can see what the code is trying to do:

```
If the room has no guests in it,
    then set the label's text to "Available".
Otherwise,
    set the label's text to "Taken :-(".
```

The content of Listing 1-1 is *Java source code*.

The processors in computers, phones, and other devices don't normally follow instructions like the instructions in Listing 1-1. That is, processors don't follow Java source code instructions. Instead, processors follow cryptic instructions like the ones in Listing 1-2.

Listing 1-2: Java Bytecode

```
 0 aload_0
 1 getfield #19 <com/allmycode/samples/MyActivity/room
 Lcom/allmycode/samples/Room;>
 4 getfield #47 <com/allmycode/samples/Room/numGuests I>
 7 ifne 22 (+15)
10 aload_0
11 getfield #41 <com/allmycode/samples/MyActivity/label
 Landroid/widget/TextView;>
14 ldc #54 <Available>
```

(continued)

Listing 1-2 ***(continued)***

```
16 invokevirtual #56
   <android/widget/TextView/setText
   (Ljava/lang/CharSequence;)V>
19 goto 31 (+12)
22 aload_0
23 getfield #41 <com/allmycode/samples/MyActivity/label
 Landroid/widget/TextView;>
26 ldc #60 <Taken :-(>
28 invokevirtual #56
   <android/widget/TextView/setText
   (Ljava/lang/CharSequence;)V>
31 return
```

The instructions in Listing 1-2 aren't Java source code instructions. They're *Java bytecode instructions*. When you write a Java program, you write source code instructions (refer to Listing 1-1). After writing the source code, you run a program (that is, you apply a tool) to the source code. The program is a *compiler*: It translates your source code instructions into Java bytecode instructions. In other words, the compiler translates code that you can write and understand (again, refer to Listing 1-1) into code that a computer can execute (refer to Listing 1-2).

At this point, you might ask "What will I have to do to get the compiler running?" The one-word answer to your question is "Eclipse." All the translation steps described in this chapter come down to using Eclipse — a piece of software that you download for free using the instructions in Chapter 2. So when you read in this chapter about compiling and other translation steps, don't become intimidated. You don't have to repair an alternator in order to drive a car, and you won't have to understand how compilers work in order to use Eclipse.

No one (except for a few crazy developers in isolated labs in faraway places) writes Java bytecode. You run software (a compiler) to create Java bytecode. The only reason to look at Listing 1-2 is to understand what a hard worker your computer is.

If compiling is a good thing, compiling twice may be even better. In 2007, Dan Bornstein at Google created *Dalvik bytecode* — another way to represent instructions for processors to follow. (To find out where some of Bornstein's ancestors come from, run your favorite map application and look for Dalvik in Iceland.) Dalvik bytecode is optimized for the limited resources on a phone or a tablet device.

Listing 1-3 contains sample Dalvik instructions.

* To see the code in Listing 1-3, I used the Dedexer program. See `dedexer.sourceforge.net`.

Listing 1-3: Dalvik Bytecode

```
.method public checkVacancy(Landroid/view/View;)V
.limit registers 4
; this: v2 (Lcom/allmycode/samples/MyActivity;)
; parameter[0] : v3 (Landroid/view/View;)
.line 30
    iget-object
    v0,v2,com/allmycode/samples/MyActivity.room
    Lcom/allmycode/samples/Room;
; v0 : Lcom/allmycode/samples/Room; , v2 :
    Lcom/allmycode/samples/MyActivity;
    iget    v0,v0,com/allmycode/samples/Room.numGuests I
; v0 : single-length , v0 : single-length
    if-nez    v0,l4b4
; v0 : single-length
.line 31
    iget-object
    v0,v2,com/allmycode/samples/MyActivity.label
    Landroid/widget/TextView;
; v0 : Landroid/widget/TextView; , v2 :
    Lcom/allmycode/samples/MyActivity;
    const-string    v1,"Available"
; v1 : Ljava/lang/String;
    invoke-virtual
    {v0,v1},android/widget/TextView/setText
    ; setText(Ljava/lang/CharSequence;)V
; v0 : Landroid/widget/TextView; , v1 : Ljava/lang/String;
l4b2:
.line 36
    return-void
l4b4:
.line 33
    iget-object
    v0,v2,com/allmycode/samples/MyActivity.label
    Landroid/widget/TextView;
; v0 : Landroid/widget/TextView; , v2 :
    Lcom/allmycode/samples/MyActivity;
    const-string    v1,"Taken :-("
; v1 : Ljava/lang/String;
    invoke-virtual
    {v0,v1},android/widget/TextView/setText ;
    setText(Ljava/lang/CharSequence;)V
; v0 : Landroid/widget/TextView; , v1 : Ljava/lang/String;
    goto    l4b2
.end method
```

When you create an Android app, Eclipse performs at least two compilations:

- **One compilation creates Java bytecode from your Java source files.** The source filenames have the `.java` extension; the Java bytecode filenames have the `.class` extension.

- **Another compilation creates Dalvik bytecode from your Java bytecode files.** Dalvik bytecode file names have the `.dex` extension.

But that's not all! In addition to its Java code, an Android app has XML files, image files, and possibly other elements. Before you install an app on a device, Eclipse combines all these elements into a single file — one with the `.apk` extension. When you publish the app on an app store, you copy that `.apk` file to the app store's servers. Then, to install your app, a user visits the app store and downloads your `.apk` file.

To perform the compilation from source code to Java bytecode, Eclipse uses a program named javac, also known as the Java compiler. To perform the compilation from Java bytecode to Dalvik code, Eclipse uses a program named dx (known affectionately as "the dx tool"). To combine all your app's files into one .apk file, Eclipse uses a program named apkbuilder.

What is a virtual machine?

In the section "What is a compiler?" earlier in this chapter, I make a big fuss about phones and other devices following instructions like the ones in Listing 1-3. As fusses go, it's a nice fuss. But if you don't read every fussy word, you may be misguided. The exact wording is ". . . processors follow cryptic instructions *like* the ones in Listing *blah-blah-blah.*" The instructions in Listing 1-3 are a lot like instructions that a phone or tablet can execute, but computers generally don't execute Java bytecode instructions, and phones don't execute Dalvik bytecode instructions. Instead, each kind of processor has its own set of executable instructions, and each operating system uses the processor's instructions in a slightly different way.

Imagine that you have two different devices: a smartphone and a tablet computer. The devices have two different kinds of processors: The phone has an ARM processor, and the tablet has an Intel Atom processor. (The acronym ARM once stood for Advanced RISC Machine. These days, *ARM* simply stands for ARM Holdings, a company whose employees design processors.) On the ARM processor, the *multiply* instruction is 000000. On an Intel processor, the *multiply* instructions are D8, DC, F6, F7, and others. Many ARM instructions have no counterparts in the Atom architecture, and many Atom instructions have no equivalents on an ARM processor. An ARM processor's instructions make no sense to your tablet's Atom processor, and an Atom processor's instructions would give your phone's ARM processor a virtual headache.

What's a developer to do? Does a developer provide translations of every app into every processor's instruction set?

No. Virtual machines create order from all this chaos. Dalvik bytecode is similar to the code in Listing 1-3, but Dalvik bytecode isn't specific to a single kind of processor or to a single operating system. Instead, a set of Dalvik bytecode instructions runs on any processor. If you write a Java program and compile that Java program into Dalvik bytecode, your Android phone can run the bytecode, your Android tablet can run the bytecode, and even your grandmother's supercomputer can run the bytecode. (To do this, your grandmother must install *Android-x86,* a special port of the Android operating system, on her Intel-based machine.)

REMEMBER

You never have to write or decipher Dalvik bytecode. Writing bytecode is the compiler's job. Deciphering bytecode is the virtual machine's job.

Both Java bytecode and Dalvik bytecode have virtual machines. With the Dalvik virtual machine, you can take a bytecode file that you created for one Android device, copy the bytecode to another Android device, and then run the bytecode with no trouble. That's one of the many reasons why Android has become popular quickly. This outstanding feature, which lets you run code on many different kinds of computers, is called *portability.*

Imagine that you're the Intel representative to the United Nations Security Council, as shown in Figure 1-3. The ARM representative is seated to your right, and the representative from Texas Instruments is to your left. (Naturally, you don't get along with either of these people. You're always cordial to one another, but you're never sincere. What do you expect? It's politics!) The distinguished representative from Dalvik is at the podium. The Dalvik representative speaks in Dalvik bytecode, and neither you nor your fellow ambassadors (ARM and Texas Instruments) understand a word of Dalvik bytecode.

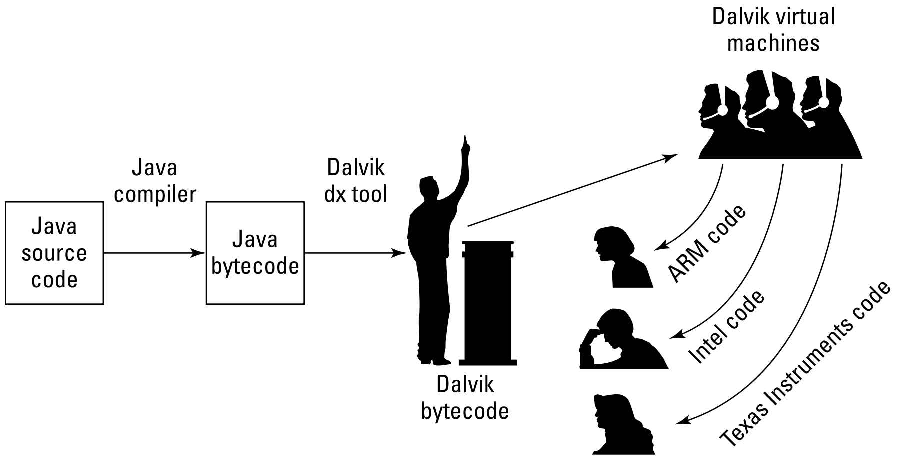


Figure 1-3: An imaginary meeting of the UN Security Council.

But each of you has an interpreter. Your interpreter translates from Dalvik bytecode to Intel instructions as the Dalvik representative speaks. Another interpreter translates from bytecode to "ARM-ese." And a third interpreter translates bytecode into "Texas Instruments-speak."

Think of your interpreter as a virtual ambassador. The interpreter doesn't really represent your country, but the interpreter performs one important task that a real ambassador performs: It listens to Dalvik bytecode on your behalf. The interpreter does what you would do if your native language were Dalvik bytecode. The interpreter, pretending to be the Intel ambassador, endures the boring bytecode speech, taking in every word and processing each one in some way or another.

You have an interpreter — a virtual ambassador. In the same way, an Intel processor runs its own bytecode-interpreting software. That software is the Dalvik virtual machine — a proxy, an errand boy, a go-between. The *Dalvik virtual machine* serves as an interpreter between Dalvik's run-anywhere bytecode and your device's own system. As it runs, the virtual machine walks your device through the execution of bytecode instructions. It examines your bytecode, bit by bit, and carries out the instructions described in the bytecode. The virtual machine interprets bytecode for your ARM processor, your Intel processor, your Texas Instruments chip, or whatever kind of processor you're using. That's a good thing. It's what makes Java code and Dalvik code more portable than code written in any other language.

Java, Android, and Horticulture

"You don't see the forest for the trees," said my Uncle Harvey. To which my Aunt Clara said "You don't see the trees for the forest." This argument went on until they were both too tired to discuss the matter.

As an author, I like to present both the forest and the trees. The "forest" is the broad overview, which helps you understand why you perform various steps. The "trees" are the steps themselves, getting you from Point A to Point B until you complete a task.

This chapter shows you the forest. The rest of this book shows you the trees.