Programming Android

Programming Android

and Masumi Nakamura

Zigurd Mednieks, Laird Dornin, G. Blake

Meike, Beijing Cambridge Farnham Köln

Sebastopol Tokyo

Programming Android
by Zigurd Mednieks, Laird Dornin, G. Blake Meike, and Masumi Nakamura

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

July 2011: First Edition.

Editors: Andy Oram and Brian Jepson
Production Editor: Adam Zaremba
Copyeditor: Audrey Doyle
Technical Editors: Vijay S. Yellapragada and Johan van der Hoeven
Proofreader: Sada Preisch
Indexer: Lucie Haskins
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

Printing History:

ISBN: 978-1-449-38969-7

[LSI]

1310671393

Table of Contents

Preface . . . . . .

. . . . . xiii Part I. Tools and Basics

1. Your Toolkit . . . . . . 3 Installing the Android SDK and Prerequisites 3 The Java Development Kit (JDK) 4 The Eclipse Integrated Development Environment (IDE) 5 The Android SDK 7 Adding Build Targets to the SDK 8 The Android Development Toolkit (ADT) Plug-in for Eclipse 9 Test Drive: Confirm That Your Installation Works 12 Making an Android Project 12 Making an Android Virtual Device (AVD) 16 Running a Program on an AVD 19 Running a Program on an Android Device 20 Troubleshooting SDK Problems: No Build Targets 21 Components of the SDK 21 The Android Debug Bridge (adb) 21 The Dalvik Debug Monitor Server (DDMS) 21 Components of the ADT Eclipse Plug-in 23 Android Virtual Devices 25 Other SDK Tools 26 Keeping Up-to-Date 28 Keeping the Android SDK Up-to-Date

28 Keeping Eclipse and the ADT Plug-in Up-to-Date 29 Keeping the JDK Up-to-Date 29 Example Code 30 SDK Example Code 30 Example Code from This Book 30 On Reading Code 32

2. Java for Android . . . . . 33 Android Is Reshaping Client-Side Java 33
The Java Type System 34
Primitive Types 34 Objects and Classes 35 Object Creation 35 The Object Class and Its Methods 37 Objects, Inheritance, and Polymorphism 39 Final and Static Declarations 41 Abstract Classes 45 Interfaces 46 Exceptions 48 The Java Collections Framework 52 Garbage Collection 55
Scope 56 Java Packages 56 Access Modifiers and Encapsulation 57
Idioms of Java Programming 59 Type Safety in Java 59 Using Anonymous Classes 62 Modular Programming in Java 65 Basic Multithreaded Concurrent Programming in Java 68 Synchronization and Thread Safety 68 Thread Control with wait() and notify() Methods 71 Synchronization and Data Structures 73

3. The Ingredients of an Android Application . . . . . 75 Traditional Programming Models Compared to Android 75 Activities, Intents, and Tasks 77 Other Android Components 78
Service 79 Content Providers 79 BroadcastReceiver 82
Static Application Resources and Context 82 Application Manifests 83 A Typical Source Tree 84 Initialization Parameters in AndroidManifest.xml 84
Resources 87 The Android Application Runtime Environment 88 The Dalvik VM 89 Zygote: Forking a New Process 89 Sandboxing: Processes and Users 89 Component Life Cycles 90

The Activity Life Cycle 90 Packaging an Android Application: The .apk File 92 On Porting Software to Android 93

4. Getting Your Application into Users' Hands . . . . . 95 Application Signing 95 Public Key Encryption and Cryptographic Signing 95 How Signatures Protect Software Users, Publishers, and
Secure Communications 97 Signing an Application 98 Placing an Application for Distribution in the Android Market 105 Becoming an Official Android Developer 106 Uploading Applications in the Market 106 Getting Paid 107 Google Maps API Keys 108 Specifying API-Level

Compatibility 109 Compatibility with Many Kinds of Screens 109 Testing for Screen Size Compatibility 110 Resource Qualifiers and Screen Sizes 110

5. Eclipse for Android Software Development . . . . . 111 Eclipse Concepts and Terminology 112 Plug-ins 112 Workspaces 113 Java Environments 114 Projects 115 Builders and Artifacts 115 Extensions 115 Associations 117 Eclipse Views and Perspectives 117 The Package Explorer View 118 The Task List View 118 The Outline View 119 The Problems View 120 Java Coding in Eclipse 120 Editing Java Code and Code Completion 120 Refactoring 121 Eclipse and Android 122 Preventing Bugs and Keeping Your Code Clean 122 Static Analyzers 123 Applying Static Analysis to Android Code 127 Limitations of Static Analysis 130 Eclipse Idiosyncrasies and Alternatives 130

6. Effective Java for Android . . . . . 133 The Android Framework 133 The Android Libraries 133 Extending Android 135 Organizing Java Source 140 Concurrency in Android 142 AsyncTask and the UI Thread 143 Threads in an Android Process 154 Serialization 156 Java Serialization 157 Parcelable 159 Classes That Support Serialization 162 Serialization and the Application Life Cycle 163

Part II. About the Android Framework

7. Building a View . . . . . 167 Android GUI Architecture 167 The Model 167 The View 168 The Controller 169 Putting It Together 169 Assembling a Graphical Interface 171 Wiring Up the Controller 176 Listening to the Model 178 Listening for Touch Events 183 Listening for Key Events 186 Alternative Ways to Handle Events 187 Advanced Wiring: Focus and Threading 189 The Menu 193

8. Fragments and Multiplatform Support . . . . . 197 Creating a Fragment 198 Fragment Life Cycle 201 The Fragment Manager 202 Fragment Transactions 203 The Compatibility Package 208

9. Drawing 2D and 3D Graphics . . . . . 211 Rolling Your Own Widgets 211 Layout 212 Canvas Drawing 217

Drawables 228 Bitmaps 232 Bling 234 Shadows, Gradients, and Filters 237 Animation 238 OpenGL Graphics 243

10. Handling and Persisting Data . . . . . 247 Relational Database Overview 247 SQLite 248 The SQL Language 248

SQL Data Definition Commands 249 SQL Data Manipulation Commands 252 Additional Database Concepts 254 Database Transactions 255 Example Database Manipulation Using sqlite3 255

SQL and the Database-Centric Data Model for Android Applications 258 The Android Database Classes 259 Database Design for Android Applications 260

Basic Structure of the SimpleVideoDbHelper Class 261 Using the Database API: MJAndroid 264 Android and Social Networking 264 The Source Folder (src) 265 Loading and Starting the Application 267 Database Queries and Reading Data from the Database 267 Modifying the Database 271

Part III. A Skeleton Application for Android

11. A Framework for a Well-Behaved Application . . . . . 279 Visualizing Life Cycles 280 Visualizing the Activity Life Cycle 280 Visualizing the Fragment Life Cycle 292 The Activity Class and Well-Behaved Applications 295 The Activity Life Cycle and the User Experience 296 Life Cycle Methods of the Application Class 296 A Flowing and Intuitive User Experience Across Activities 299 Multitasking in a Small-Screen Environment 299 Tasks and Applications 299 Specifying Launch and Task Behavior 300

12. Using Content Providers . . . . . 305 Understanding Content Providers 306 Implementing a Content Provider 307 Browsing Video with Finch 308 Defining a Provider Public API 309 Defining the CONTENT_URI 310 Creating the Column Names 312 Declaring Column Specification Strings 312 Writing and Integrating a Content Provider 314 Common Content Provider Tasks 314 File Management and Binary Data 316 Android MVC and Content Observation 318

A Complete Content Provider: The SimpleFinchVideoContentProvider

Code 319 The SimpleFinchVideoContentProvider Class and Instance Variables 319 Implementing the onCreate Method 321 Implementing the getType Method 322 Implementing the Provider API 322 Determining How Often to Notify Observers 327 Declaring Your Content Provider 327

13. Exploring Content Providers . . . . . 329 Developing RESTful Android Applications 330 A “Network MVC” 331 Summary of Benefits 333 Code Example: Dynamically Listing and Caching YouTube Video Content 334 Structure of the Source Code for the Finch YouTube Video Example 335 Stepping Through the Search Application 336 Step 1: Our UI Collects User Input 337 Step 2: Our Controller Listens for Events 337 Step 3: The Controller Queries the Content Provider with a managedQuery on the Content Provider/Model 338 Step 4: Implementing the RESTful Request 338 Constants and Initialization 338 Creating the Database 339 A Networked Query Method 339 insert and ResponseHandlers 352 File Management: Storing Thumbnails 353

Part IV. Advanced Topics

14. Multimedia . . . . . 359 Audio and Video 359 Playing Audio and Video 360 Audio Playback 361 Video Playback 363 Recording Audio and Video 364 Audio Recording 365 Video Recording 368 Stored Media Content 369

15. Location and Mapping . . . . . 371 Location-Based Services 372 Mapping 373 The Google Maps Activity 373 The MapView and MapActivity 374 Working with MapViews 375 MapView and MyLocationOverlay Initialization 375 Pausing and Resuming a MapActivity 378 Controlling the Map with Menu Buttons 379 Controlling the Map with the Keypad 381 Location Without Maps 382 The Manifest and Layout Files 382 Connecting to a Location Provider and Getting Location Updates 383 Updating the Emulated Location 386

16. Sensors, NFC, Speech, Gestures, and Accessibility . . . . .

. . . . **391** Sensors 391 Position 393 Other Sensors 395 Near Field Communication (NFC) 396 Reading a Tag 396 Writing to a Tag 403 P2P Mode 405 Gesture Input 406 Accessibility 407

17. Communication, Identity, Sync, and Social Media . . . . 411 Account Contacts 411 Authentication and Synchronization 414 Authentication 415

Synchronization 422 Bluetooth 429 The Bluetooth Protocol Stack 429 Bluez: The Linux Bluetooth Implementation 431 Using Bluetooth in Android Applications 431

18. The Android Native Development Kit (NDK) . . . . 445 Native Methods and JNI Calls 446 Conventions on the Native Method Side 446 Conventions on the Java Side 447 The Android NDK 448 Setting Up the NDK Environment 448 Compiling with the NDK 448 JNI, NDK, and SDK: A Sample App 449 Android-Provided Native Libraries 451 Building Your Own Custom Library Modules 453 Native Activities 456

Index . . . . . .

Preface

The purpose of this book is to enable you to create well-engineered Android applica tions that go beyond the scope of small example applications.

This book is for people coming to Android programming from a variety of backgrounds. If you have been programming iPhone or Mac OS applications in Objective-C, you will

find coverage of Android tools and Java language features relevant to Android pro

gramming that will help you bring your knowledge of mobile application development to Android. If you are an experienced Java coder, you will find coverage of Android application architecture that will enable you to use your Java expertise in this newly vibrant world of client Java application development. In short, this is a book for people with some relevant experience in object-oriented languages, mobile applications, REST applications, and similar disciplines who want to go further than an introductory book or online tutorials will take them.

How This Book Is Organized

We want to get you off to a fast start. The chapters in the first part of this book will step you through using the SDK tools so that you can access example code in this book and in the SDK, even as you expand your knowledge of SDK tools, Java, and database design. The tools and basics covered in the first part might be familiar enough to you that you would want to skip to Part II where we build foundational knowledge for developing larger Android applications.

The central part of this book is an example of an application that uses web services to deliver information to the user—something many applications have at their core. We present an application architecture, and a novel approach to using Android's frame work classes that enables you to do this particularly efficiently. You will be able to use this application as a framework for creating your own applications, and as a tool for learning about Android programming.

In the final part of this book, we explore Android APIs in specific application areas: multimedia, location, sensors, and communication, among others, in order to equip you to program applications in your specific area of interest.

By the time you reach the end of this book, we want you to have gained knowledge beyond reference material and a walk-through of examples. We want you to have a point of view on how to make great Android applications.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
Indicates new terms, URLs, email addresses, filenames, and file extensions

Constant width
Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords

Constant width bold
Shows commands or other text that should be typed literally by the user

Constant width italic
Shows text that should be replaced with user-supplied values or by values deter mined by context

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Programming Android* by

Zigurd

Mednieks, Laird Dornin, G. Blake Meike, and Masumi Nakamura. Copyright 2011 O'Reilly Media, Inc., 978-1-449-38969-7."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

> Safari Books Online is an on-demand digital library that lets you easily search more than 7,500 technology and creative reference books and vid eos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, down load chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other pub lishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreilly.com/catalog/9781449389697

To comment or ask technical questions about this book, send

email to: *bookquestions@oreilly.com*

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

The authors have adapted portions of this book from their previously released title, *Android Application Development* (O'Reilly).

Drafts of this book were released on the O'Reilly Open Feedback Publishing System (OFPS) in order to get your feedback on whether and how we are meeting the goals for this book. We are very grateful for the readers who participated in OFPS, and we owe them much in correcting our errors and improving our writing. Open review of drafts will be part of future editions, and we welcome your views on every aspect of this book.

PART I
Tools and Basics

Part I shows you how to install and use your tools, what you need to know about Java to write good Android code, and how to design and use SQL databases, which are central to the Android application model, persistence system, and implementation of key design patterns in Android programs.

CHAPTER 1
Your Toolkit

This chapter shows you how to install the Android software development kit (SDK) and all the related software you're likely to need. By the end, you'll be able to run a simple "Hello World" program on an emulator. Windows, Mac OS X, and Linux sys tems can all be used for Android application development. We will load the software, introduce you to the tools in the SDK, and point you to sources of example code.

Throughout this book, and especially in this chapter, we refer to instructions available on various websites for installing and updating the tools you will use for creating An droid programs. The most important place to find information and links to tools is the Android Developers site:

http://developer.android.com

Our focus is on guiding you through installation, with explanations that will help you understand how the parts of Android and its developer tools fit together, even as the details of each part change.

The links cited in this book may change over time. Descriptions and updated links are posted on this book's website. You can find a link to the website on this book's catalog page. You may find it convenient to have the book's website open as you read so that you can click through links on the site rather than entering the URLs printed in this book.

Installing the Android SDK and Prerequisites

Successfully installing the Android SDK requires two other software systems that are not part of the Android SDK: the Java Development Kit (JDK) and the Eclipse integrated development environment (IDE). These two systems are not delivered as part of the Android SDK because you may be using them for purposes outside of Android software development, or because they may already be installed on your system, and redundant installations of these systems can cause version clashes.

The Android SDK is compatible with a range of recent releases of the JDK and the Eclipse IDE. Installing the current release of each of these tools will usually be the right choice. The exact requirements are specified on the System Requirements page of the Android Developers site: *http://developer.android.com/sdk/requirements.html*.

One can use IDEs other than Eclipse in Android software development, and informa tion on using other IDEs is provided in the Android documentation at *http://developer .android.com/guide/developing/other-ide.html*. We chose Eclipse as the IDE covered in this book because Eclipse supports the greatest number of Android SDK tools and other plug-ins, and Eclipse is the most widely used Java IDE, but IntelliJ IDEA is an alternative many Java coders prefer.

The Java Development Kit (JDK)

If your system has an up-to-date JDK installed, you won't need to install it again. The JDK provides tools, such as the Java compiler, used by IDEs and SDKs for developing Java programs. The JDK also contains a Java Runtime Environment (JRE), which en ables Java programs, such as Eclipse, to run on your system.

If you are using a Macintosh running a version of Mac OS X supported by the Android SDK, the JDK is already installed.

If you are using Ubuntu Linux, you can install the JDK using the package manager, through the following command:

```
sudo apt-get install sun-java6-jdk
```

If this command reports that the JDK package is not available, you may need to enable the "partner" repositories using the Synaptic Package Manager utility in the System→Administration menu. The "partner" re positories are listed on the Other Software tab after you choose Set

tings→Repositories.

This is one of the very few places in this chapter where you will see a version number, and it appears here only because it can't be avoided. The version number of the JDK is in the package name. But, as with all other software mentioned in this chapter, you should refer to up-to-date online documentation to determine the version you will need.

If you are a Windows user, or you need to install the JDK from Oracle's site for some other reason, you can find the JDK at *http://www.oracle.com/technetwork/java/javase/ downloads/index.html*.

The Downloads page will automatically detect your system and offer to download the correct version. The installer you download is an executable file. Run the executable installer file to install the JDK.

To confirm that the JDK is installed correctly, issue this command from the command line (Terminal on Linux and Mac; Command Prompt on Windows):

```
javac -version
```

If the javac command is not in your PATH, you may need to add the *bin* directory in the JDK to your path manually.

It should display the version number corresponding to the version of the JDK you installed. If you installed revision 20 of the Java 6 JDK, the command would display:

```
javac 1.6.0_20
```

Depending on the current version of the JDK available when you read this, version numbers may differ from what you see here.

If it is unclear which JRE you are running, or if you think you have the wrong JRE running on a Debian-derived Linux system, such as Ubuntu, you can use the following command to display the available JREs and select the right one:

```
sudo update-alternatives --config java
```

The Eclipse Integrated Development Environment

(IDE)

Eclipse is a general-purpose technology platform. It has been applied to a variety of uses in creating IDEs for multiple languages and in creating customized IDEs for many specialized SDKs, as well as to uses outside of software development tools, such as providing a Rich Client Platform (RCP) for Lotus Notes and a few other applications.

Eclipse is usually used as an IDE for writing, testing, and debugging software, especially Java software. There are also several derivative IDEs and SDKs for various kinds of Java software development based on Eclipse. In this case, you will take a widely used Eclipse package and add a plug-in to it to make it usable for Android software development. Let's get that Eclipse package and install it.

Eclipse can be downloaded from *http://www.eclipse.org/downloads*.

You will see a selection of the most commonly used Eclipse packages on this page. An Eclipse "package" is a ready-made collection of Eclipse modules that make Eclipse better suited for certain kinds of software development. Usually, Eclipse users start with one of the Eclipse packages available for download on this page and customize it with plug-ins, which is what you will do when you add the Android Development Tools

(ADT) plug-in to your Eclipse installation. The System Requirements article on the Android Developers site lists three choices of Eclipse packages as a basis for an Eclipse installation for Android software development:

- Eclipse Classic (for Eclipse 3.5 or later)
- Eclipse IDE for Java Developers
- Eclipse for RCP/Plug-in Developers

Any of these will work, though unless you are also developing Eclipse plug-ins, choos ing either Classic or the Java Developers package (EE or Standard) makes the most sense. The authors of this book started with the Java EE Developers package ("EE" stands for Enterprise Edition), and screenshots of Eclipse used in this book reflect that choice.

The Eclipse download site will automatically determine the available system-specific downloads for your system, though you may have to choose between 32 and 64 bits to match your operating system. The file you download is an archive. To install Eclipse, open the archive and copy the *eclipse* folder to your home folder. The executable file for launching Eclipse on your system will be found in the folder you just extracted from the archive.

> We really mean it about installing Eclipse in your home folder (or an other folder you own), especially if you have multiple user accounts on your system. Do not use your system's package manager. Your Eclipse installation is one of a wide range of possible groupings of Eclipse plug

ins. In addition, you will probably further customize your installation of Eclipse. And Eclipse plug-ins and updates are managed separately from other software in your system.

For these reasons, it is very difficult to successfully install and use Eclipse as a command available to all users on your system, even if your system can do this from its package manager. To successfully complete an in stallation as it is described here, you must install Eclipse in a folder managed by one user, and launch it from this location.

If you are using Ubuntu or another Linux distribution, you should not install Eclipse from your distribution's repositories, and if it is currently installed this way, you must remove it and install Eclipse as described here. The presence of an "eclipse" package in the Ubuntu repositories is an inheritance from the Debian repositories on which Ubuntu is based. It is not a widely used approach to installing and using Eclipse, be cause most of the time, your distribution's repositories will have older versions of Eclipse.

To confirm that Eclipse is correctly installed and that you have a JRE that supports Eclipse, launch the executable file in the Eclipse folder. You may want to make a short cut to this executable file to launch Eclipse more conveniently. You should see the Welcome screen shown in Figure 1-1.

Eclipse is implemented in Java and requires a JRE. The JDK you previously installed provides a JRE. If Eclipse does not run, you should check that the JDK is correctly installed.

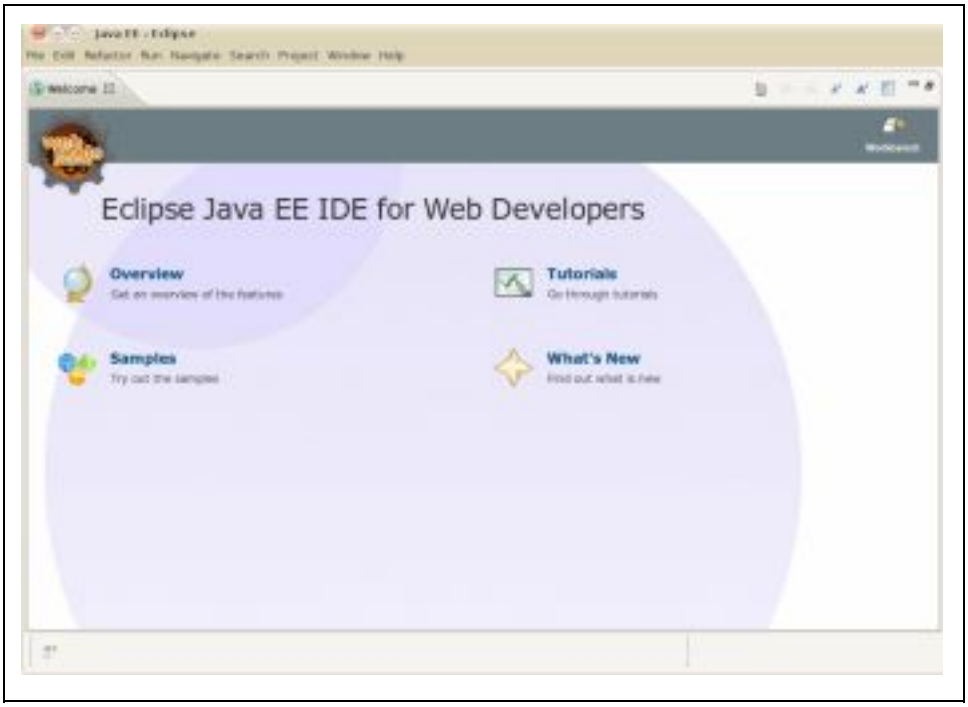

Figure 1-1. Welcome screen that you see the first time you run Eclipse

The Android SDK

With the JDK and Eclipse installed, you have the prerequisites for the Android SDK, and are ready to install the SDK. The Android SDK is a collection of files: libraries, executables, scripts, documentation, and so forth. Installing the SDK means down loading the version of the SDK for your platform and putting the SDK files into a folder in your home directory.

To install the SDK, download the SDK package that corresponds to your system from *http://developer.android.com/sdk/index.html*.

The download is an archive. Open the archive and extract the folder in the archive to your home folder.

If you are using a 64-bit version of Linux, you may need to install the ia32-libs package.

To check whether you need this package, try running the adb command (~/android-sdk-linux_*/platform-tools/adb). If your system reports that adb cannot be found (despite being right there in the *platform tools* directory) it likely means that the current version of adb, and pos

sibly other tools, will not run without the ia32-libs package installed. The command to install the ia32-libs package is:

```
sudo apt-get install ia32-libs
```

The SDK contains one or two folders for tools: one named *tools* and, starting in version 8 of the SDK, another called *platform-tools*. These folders need to be on your path, which is a list of folders your system searches for executable files when you invoke an executable from the command line. On Macintosh and Linux systems, setting the PATH environment variable is done in the *.profile* (Ubuntu) or *.bash_profile* (Mac OS X) file in your home directory. Add a line to that file that sets the PATH environment variable to include the *tools* directory in the SDK (individual entries in the list are separated by colons). For example, you could use the following line (but replace both instances of ~/android-sdk-*ARCH* with the full path to your Android SDK install):

```
export PATH=$PATH:~/android-sdk-ARCH/tools:~/android-sdk-ARCH/platform-tools
```

On Windows systems, click Start→right-click Computer, and choose Properties. Then click Advanced System Settings, and click the Environment Variables button. Double click the path system variable, and add the path to the folders by going to the end of this variable's value (do not change anything that's already there!) and adding the two paths to the end, separated by semicolons with no space before them. For example:

```
;C:\android-sdk-windows\tools;C:\android-sdk-windows\platform-tools
```

After you've edited your path on Windows, Mac, or Linux, close and reopen any Com mand Prompts or Terminals to pick up the new PATH setting (on Ubuntu, you may need to log out and log in unless your Terminal program is configured as a login shell).

Adding Build Targets to the SDK

Before you can build an Android application, or even create a project that would try to build an Android application, you must install one or more build targets. To do this, you will use the SDK and AVD Manager. This tool enables you to install packages in the SDK that will support multiple versions of the Android OS and multiple API levels.

Once the ADT plug-in is installed in Eclipse, which we describe in the next section, the SDK and AVD Manager can be invoked from within Eclipse. It can also be invoked from the command line, which is how we will do it here. To invoke the SDK and AVD Manager from the command line, issue this command:

```
android
```


The screenshot in Figure 1-2 shows the SDK and AVD Manager, with all the

available SDK versions selected for installation.

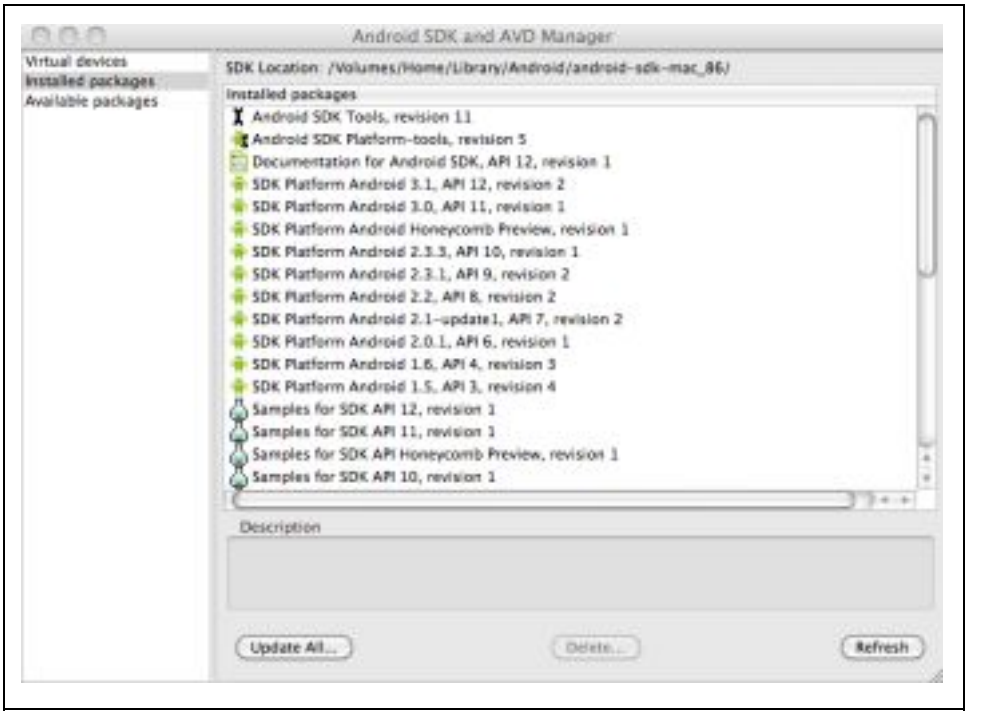

Figure 1-2. The SDK and AVD Manager, which enables installation of Android API levels

The packages labeled “SDK platform” support building applications compatible with different Android API levels. You should install, at a minimum, the most recent (highest numbered) version, but installing all the available API levels, and all the Google API add-on packages, is a good choice if you might someday want to build applications that run on older Android versions. You should also install, at a minimum, the most recent versions of the example applications package. You must also install the Android SDK Platform-Tools package.

The Android Development Toolkit (ADT) Plug-in for Eclipse

Now that you have the SDK files installed, along with Eclipse and the JDK, there is one more critical part to install: the Android Developer Toolkit (ADT) plug-in. The ADT plug-in adds Android-specific functionality to Eclipse.

Software in the plug-in enables Eclipse to build Android applications, launch the An droid emulator, connect to debugging services on the emulator, edit Android XML files, edit and compile Android Interface Definition Language (AIDL) files, create Android application packages (*.apk* files), and perform other

Android-specific tasks.

Using the Install New Software Wizard to download and install the ADT plug-in

You start the Install New Software Wizard by selecting Help→Install New Software (Figure 1-3). To install the ADT plug-in, type this URL into the Work With field and press Return or Enter: **https://dl-ssl.google.com/android/eclipse/** (see Figure 1-4).

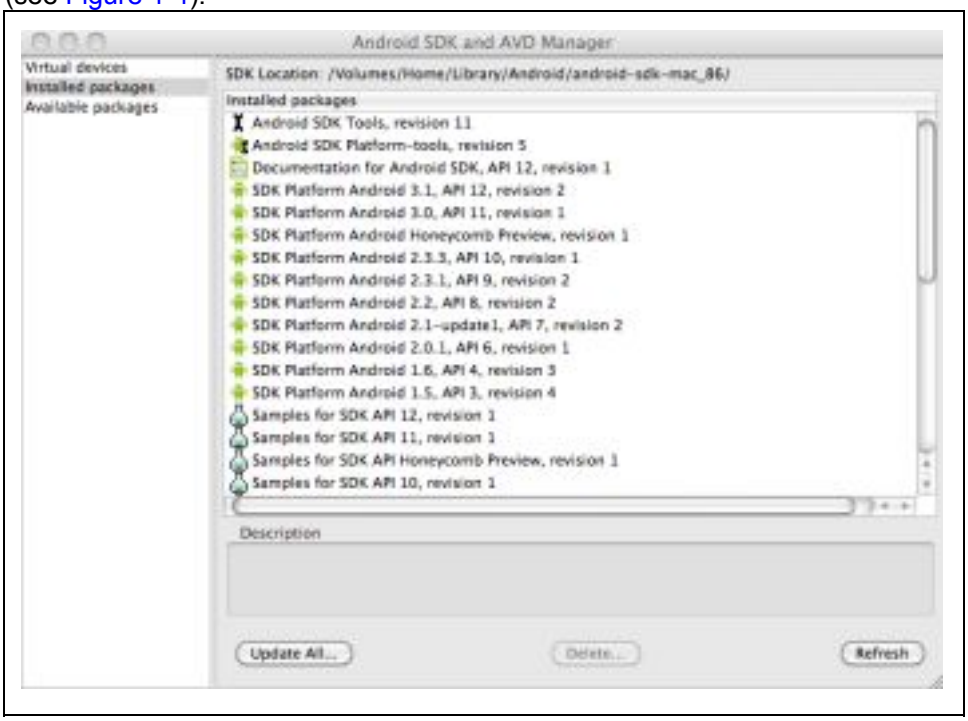

Figure 1-3. The Eclipse Add Site dialog

> More information on installing the ADT plug-in using the Install New Software Wizard can be found on the Android Developers site, at *http://developer.android.com/sdk/eclipse-adt.html#downloading*.
>
> Eclipse documentation on this wizard can be found on the Eclipse documentation site, at *http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.platform.doc.user/tasks/tasks-124.htm*.

Once you have added the URL to the list of sites for acquiring new plug-ins, you will see an entry called Developer Tools listed in the Available Software list.

Select the Developer Tools item by clicking on the checkbox next to it, and click on the Next button. The next screen will ask you to accept the license for this software. After you accept and click Finish, the ADT will be installed. You will have to restart Eclipse to complete the installation.

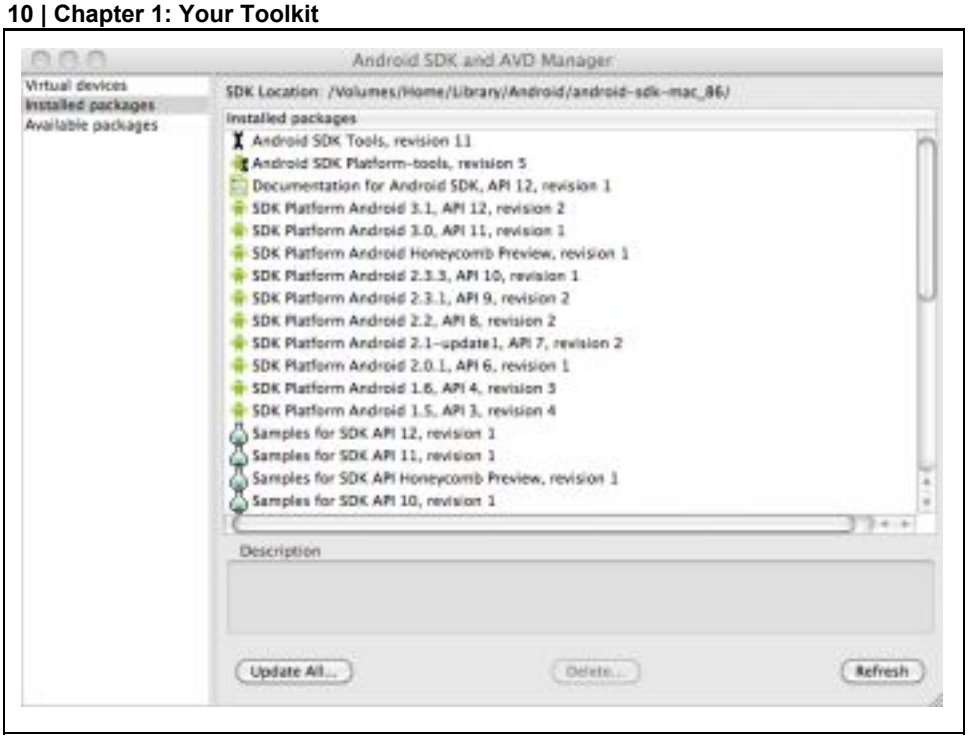

Figure 1-4. The Eclipse Install New Software dialog with the Android Hierarch Viewer plug-in shown as available

Configuring the ADT plug-in

One more step, and you are done installing. Once you have installed the ADT plug-in, you will need to configure it. Installing the plug-in means that various parts of Eclipse now contain Android software development-specific dialogs, menu commands, and other tools, including the dialog you will now use to configure the ADT plug-in. Start the Preferences dialog using the Window→Preferences (Linux and Windows) or Eclipse→Preferences (Mac) menu option. Click the item labeled Android in the left pane of the Preferences dialog.

The first time you visit this section of the preferences, you'll see a dialog asking if you want to send some usage statistics to Google. Make your choice and click Proceed.

A dialog with the Android settings is displayed next. In this dialog, a text entry field labeled “SDK location” appears near the top. You must enter the path to where you put the SDK, or you can use the file browser to select the directory, as shown in Fig ure 1-5. Click Apply. Note that the build targets you installed, as described in “Adding Build Targets to the SDK” on page 8, are listed here as well.

Figure 1-5. Configuring the SDK location into the Eclipse ADT plug-in using the Android Preferences dialog

Your Android SDK installation is now complete.

Test Drive: Confirm That Your Installation

Works

If you have followed the steps in this chapter, and the online instructions referred to here, your installation of the Android SDK is now complete. To confirm that everything you installed so far works, let's create a simple Android application.

Making an Android Project

The first step in creating a simple Android application is to create an Android project. Eclipse organizes your work into "projects," and by designating your project as an

Android project, you tell Eclipse that the ADT plug-in and other Android tools are going to be used in conjunction with this project.

Reference information and detailed online instructions for creating an Android project can be found at *http://developer.android.com/guide/developing/eclipse-adt.html*.

Start your new project with the File→New→Android Project menu command. Locate the Android Project option in the New Project dialog (it should be under a section named Android). Click Next, and the New Project dialog appears as shown in Fig ure 1-6.

To create your Android project, you will provide the following information:

Project name
: This is the name of the project (not the application) that appears in Eclipse. Type **TestProject**, as shown in Figure 1-6.

Workspace
: A workspace is a folder containing a set of Eclipse projects. In creating a new project, you have the choice of creating the project in your current workspace, or specifying a different location in the filesystem for your project. Unless you need to put this project in a specific location, use the defaults ("Create new project in workspace" and "Use default location").

Target name
: The Android system images you installed in the SDK are shown in the build target list. You can pick one of these system images, and the corresponding vendor, plat form (Android OS version number), and API level as the target for which your application is built. The platform and API level are the most important parameters here: they govern the Android platform library that your application will be com piled with, and the API

level supported—APIs with a higher API level than the one you select will not be available to your program. For now, pick the most recent Android OS version and API level you have installed.

Application name

This is the application name the user will see. Type **Test Application**.

Package name

The package name creates a Java package namespace that uniquely identifies pack ages in your application, and must also uniquely identify your whole Android ap plication among all other installed applications. It consists of a unique domain name—the application publisher's domain name—plus a name specific to the ap plication. Not all package namespaces are unique in Java, but the conventions used for Android applications make namespace conflicts less likely. In our example we used com.oreilly.testapp, but you can put something appropriate for your domain

Figure 1-6. The New Android Project dialog

here (you can also use com.example.testapp, since example.com is a domain name reserved for examples such as this one).

Activity

An *activity* is a unit of interactive user interface in an Android application, usually corresponding to a group of user interface objects occupying the entire screen. Optionally, when you create a project you can have a skeleton activity created for you. If you are creating a visual application (in contrast with a service, which can be "headless"—without a visual UI), this is a convenient way to create the activity the application will start with. In this example, you should create an activity called TestActivity.

Minimum SDK version

The field labeled Min SDK Version should contain an integer corresponding to the minimum SDK version required by your application, and is used to initialize the uses-sdk attribute in the application's manifest, which is a file that stores applica tion attributes. See "The Android Manifest Editor" on page 24. In most cases, this should be the same as the API level of the build target you selected, which is displayed in the rightmost column of the list of build targets, as shown in Figure 1-6.

Click Finish (not Next) to create your Android project, and you will see it listed in the left pane of the Eclipse IDE as shown in Figure 1-7.

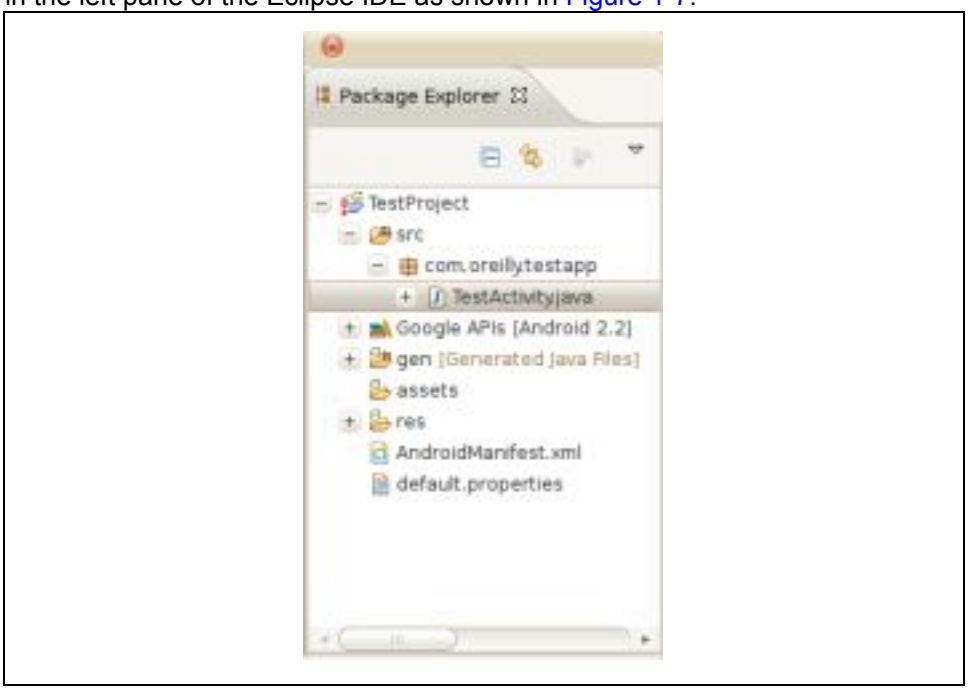

Figure 1-7. The Package Explorer view, showing the files, and their components, that are part of the project

If you expand the view of the project hierarchy by clicking the "+" (Windows) or tri angle (Mac and Linux) next to the project name, you will see the various parts of an Android project. Expand the *src* folder and you will see a Java package with the name you entered in the wizard. Expand that package and you will see the Activity class created for you by the wizard. Double-click that, and you will see the Java code of your first Android program:

```
package com.oreilly.demo.pa.ch01.testapp;

import android.app.Activity;
import android.os.Bundle;
import com.oreilly.demo.pa.ch01.R;

public class TestActivity extends Activity {
/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
}

}
```

If you've been following along and see the same thing on your computer, your SDK installation is probably working correctly. But let's make sure, and explore the SDK just a bit further, by running your first program in an emulator and on an Android device if you have one handy.

Making an Android Virtual Device (AVD)

The Android SDK provides an emulator, which emulates a device with an ARM CPU running an Android operating system (OS), for running Android programs on your PC. An Android Virtual Device (AVD) is a set of parameters for this emulator that configures it to use a particular system image—that is, a particular version of the Android operating

system—and to set other parameters that govern screen size, memory size, and other emulated hardware characteristics. Detailed documentation on AVDs is available at *http://developer.android.com/guide/developing/tools/avd.html*, and detailed documentation on the emulator is found here: *http://developer.android.com/guide/developing/ tools/emulator.html*.

Because we are just validating that your SDK installation works, we won't go into depth on AVDs, much less details of the emulator, just yet. Here, we will use the Android SDK and AVD Manager (see Figure 1-8) to set up an AVD for the purpose of running the program we just created with the New Android Project Wizard.

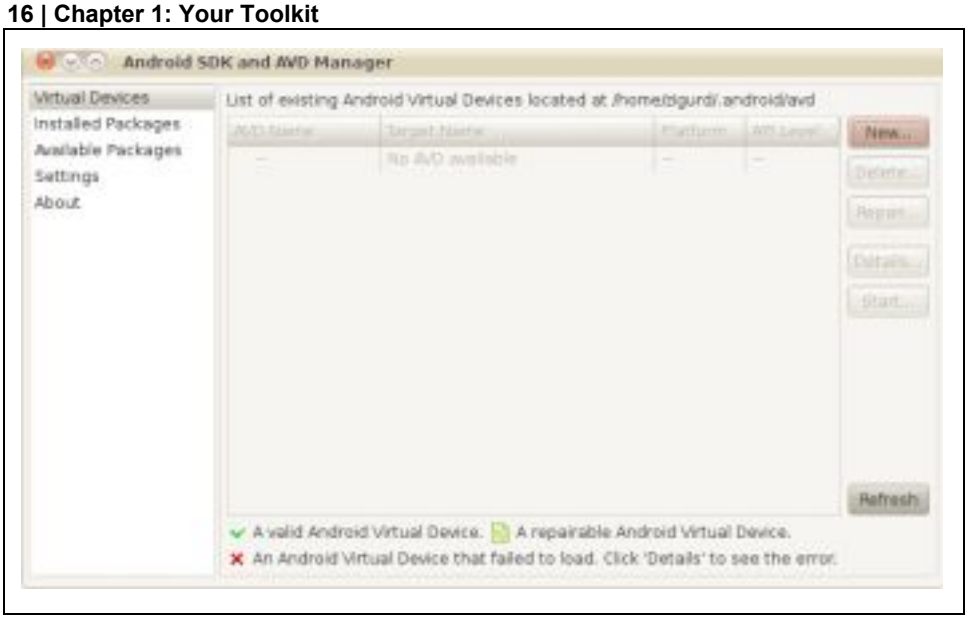

Figure 1-8. The SDK and AVD Manager

You will need to create an AVD with a system image that is no less recent than

the target specified for the project you created. Click the New button. You will now see the Create New Android Virtual Device (AVD) dialog, shown in Figure 1-9, where you specify the parameters of your new AVD.

This screen enables you to set the parameters of your new AVD:

Name

This is the name of the AVD. You can use any name for an AVD, but a name that indicates which system image it uses is helpful.

Target

The Target parameter sets which system image will be used in this AVD. It should be the same as, or more recent than, the target you selected as the build target for your first Android project.

SD Card

Some applications require an SD card that extends storage beyond the flash mem ory built into an Android device. Unless you plan to put a lot of data in SD card storage (media files, for example) for applications you are developing, you can create a small virtual SD card of, say, 100 MB in size, even though most phones are equipped with SD cards holding several gigabytes.

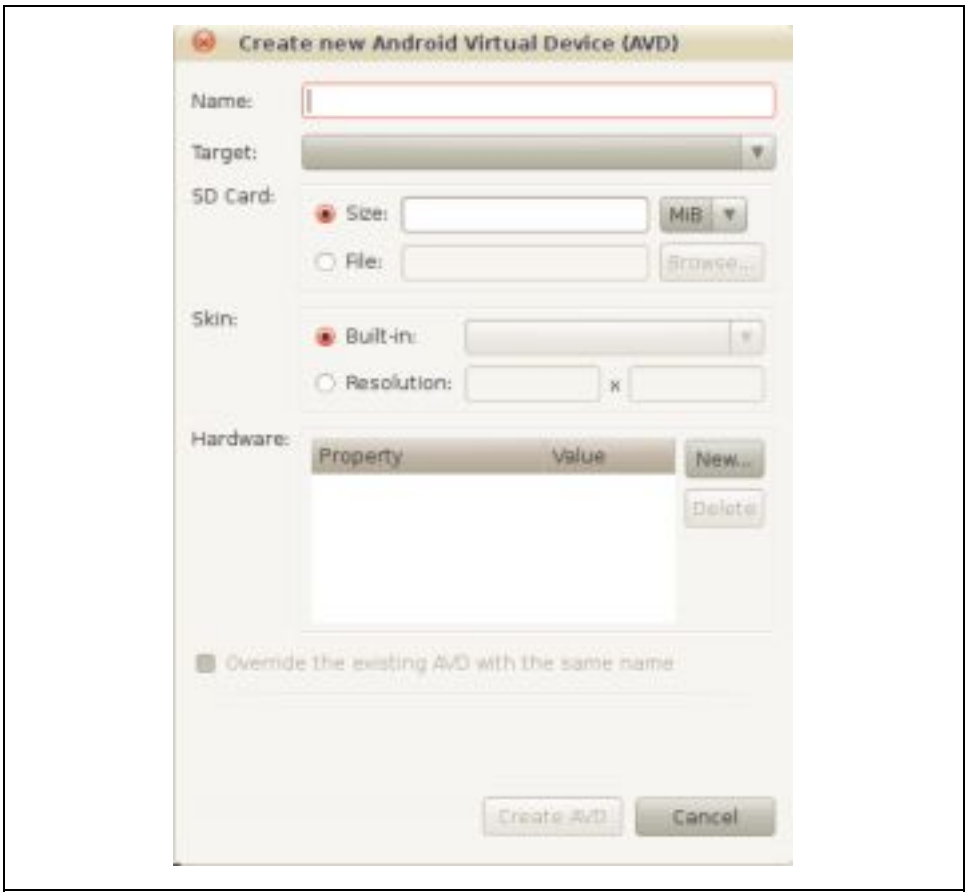

Figure 1-9. Creating a new AVD

Skin

The “skin” of an AVD mainly sets the screen size. You won’t need to change the default for the purpose of verifying that your SDK installation works, but a variety of emulators with different screen sizes is useful to check that your layouts work across different devices.

Hardware

The Hardware field of an AVD configuration enables you to set parameters indi cating which optional hardware is present. You won’t need to change the defaults for this project.

Fill in the Name, Target, and SD Card fields, and create a new AVD by clicking the Create AVD button. If you have not created an AVD with a system image that matches or is more recent than the target you specified for an Android project, you won’t be able to run your program.

Running a Program on an AVD

Now that you have a project that builds an application, and an AVD with a system image compatible with the application's build target and API level requirements, you can run your application and confirm that the SDK produced, and is able to run, an Android application.

To run your application, right-click on the project you created and, in the context menu that pops up, select Run As→Android Application.

If the AVD you created is compatible with the application you created, the AVD will start, the Android OS will boot on the AVD, and your application will start. You should see your application running in the AVD, similarly to what is shown in Figure 1-10.

Figure 1-10. The application you just created, running in an AVD

If you have more than one compatible AVD configured, the Android Device Chooser dialog will appear and ask you to select among the AVDs that are already running, or among the Android devices attached to your system, if any,

or to pick an AVD to start. Figure 1-11 shows the Android Device Chooser displaying one AVD that is running, and one that can be launched.

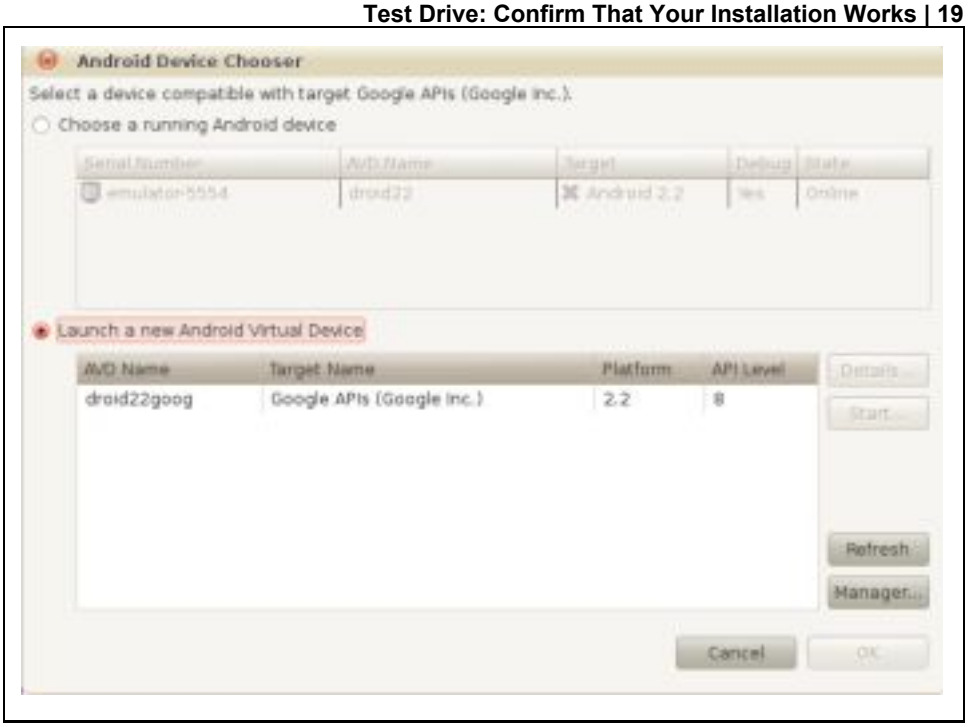

Figure 1-11. The Android Device Chooser

Running a Program on an Android Device

You can also run the program you just created on most Android devices.

You will need to connect your device to your PC with a USB cable, and, if needed, install a driver, or set permissions to access the device when connected via USB.

System-specific instructions for Windows, along with the needed driver, are available at *http://developer.android.com/sdk/win-usb.html*.

If you are running Linux, you will need to create a “rules” file for your Android device. If you are running Mac OS X, no configuration is required.

Detailed reference information on USB debugging is here: *http://developer.android.com/ guide/developing/device.html*.

You will also need to turn on USB debugging in your Android device. In most cases, you will start the Settings application, select Applications and then

Development, and then you will see an option to turn USB debugging on or off.

If an AVD is configured or is running, the Android Device Chooser will appear, dis playing both the Android device you have connected and the AVD.

Select the device, and the Android application will be loaded and run on the

device.

Troubleshooting SDK Problems: No Build Targets

If you are unable to make a new project or import an example project from the SDK, you may have missed installing build targets into your SDK. Reread the instructions in "Adding Build Targets to the SDK" on page 8 and make sure the Android pane in the Preferences dialog lists build targets as installed in your SDK, as shown in Figure 1-5.

Components of the SDK

The Android SDK is made of mostly off-the-shelf components, plus some purpose-built components. In many cases, configurations, plug-ins, and extensions adapt these com ponents to Android. The Android SDK is a study in the efficient development of a modern and complete SDK. Google took this approach in order to bring Android to market quickly. You will see this for yourself as you explore the components of the Android SDK. Eclipse, the Java language, QEMU, and other preexisting platforms, tools, and technologies comprise some of the most important parts of the Android SDK.

In creating the simple program that confirms that your SDK installation is correct, you have already used many of the components of the SDK. Here we will identify and describe the components of the SDK involved in creating your program, and other parts you have yet to use.

The Android Debug Bridge (adb)

adb is a program that enables you to control both emulators and devices, and to run a shell in order to execute commands in the environment of an emulator or device. adb is especially handy for installing and removing programs from an emulator or device. Documentation on adb can be found at *http://developer.android.com/guide/developing/ tools/adb.html*.

The Dalvik Debug Monitor Server (DDMS)

The Dalvik Debug Monitor Server (DDMS) is a traffic director between the single port that Eclipse (and other Java debuggers) looks for to connect to a

Java Virtual Machine (JVM) and the several ports that exist for each Android device or virtual device, and for each instance of the Dalvik virtual machine (VM) on each device. The DDMS also provides a collection of functionality that is accessible through a standalone user in terface or through an interface embedded in Eclipse via the ADT plug-in.

When you invoke the DDMS from the command line, you will see something similar to the window shown in Figure 1-12.

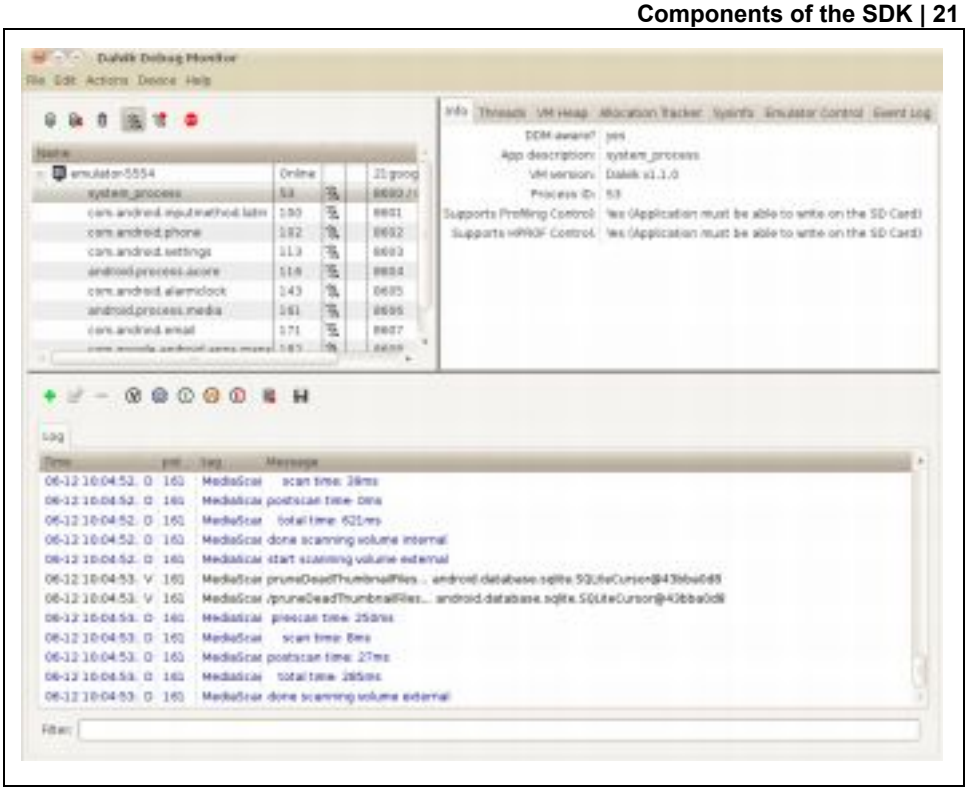

Figure 1-12. The Dalvik Debug Monitor running standalone

The DDMS's user interface provides access to the following:

A list of devices and virtual devices, and the VMs running on those devices In the upper-left pane of the DDMS window, you will see listed the Android

devices you have connected to your PC, plus any AVDs you have running. Listed under each device or virtual device are the tasks running in Dalvik VMs.

VM information

Selecting one of the Dalvik VMs running on a device or virtual device causes in formation about that VM to be displayed in the upper-right pane.

Thread information

Information for threads within each process is accessed through the "Threads" tab in the upper-right pane of the DDMS window.

Filesystem explorer

You can explore the filesystem on a device or virtual device using the DDMS file system explorer, accessible through the "File explorer" menu item in the Devices menu. It displays the file hierarchy in a window similar to the one shown in Fig ure 1-13.

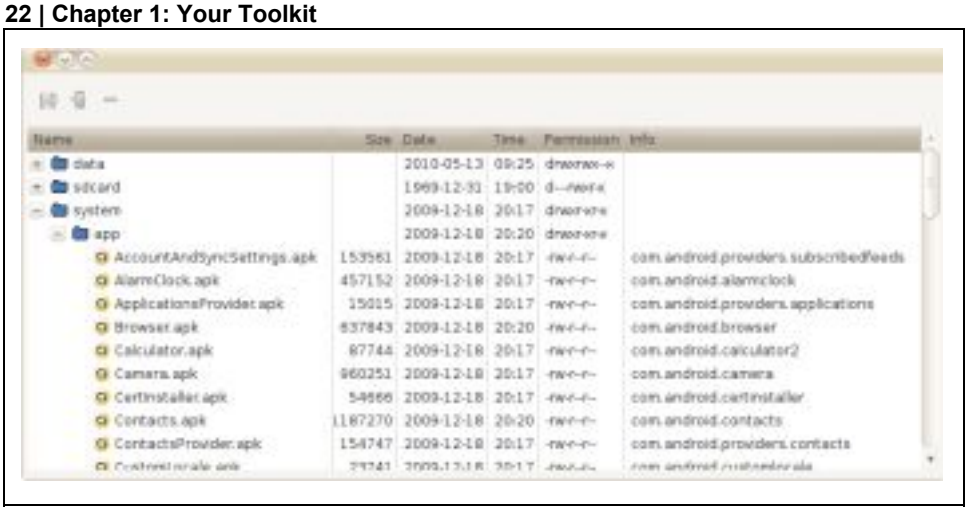

Figure 1-13. The DDMS file system explorer

Simulating phone calls

The Emulator Control tab in the upper-right pane of the DDMS window enables you to "fake" a phone call or text message in an emulator.

Screen capture

The "Screen capture" command in the Device menu fetches an image of the current screen from the selected Android device or virtual device.

Logging

The bottom pane of the DDMS window displays log output from processes on the selected device or virtual device. You can filter the log output by selecting a filter from among the buttons on the toolbar above the logging pane.

Dumping state for devices, apps, and the mobile radio

A set of commands in the Device menu enables you to command the device or virtual device to dump state for the whole device, an app, or the mobile radio.

Detailed documentation on the DDMS is available at *http://developer.android.com/ guide/developing/tools/ddms.html*.

Components of the ADT Eclipse Plug-in

Eclipse enables you to create specific project types, including several kinds of Java projects. The ADT plug-in adds the ability to make and use Android projects. When you make a new Android project, the ADT plug-in creates the project file hierarchy and all the required files for the minimal Android project to be correctly built. For Android projects, the ADT plug-in enables Eclipse to apply components of the ADT plug-in to editing, building, running, and debugging that project.

In some cases, components of the SDK can be used with Eclipse or in a standalone mode. But, in most of the Android application development cases covered in this book, the way these components are used in or with Eclipse will be the most relevant.

The ADT plug-in has numerous separate components, and, despite the connotations of a “plug-in” as a modest enhancement, it’s a substantial amount of software. Here we will describe each significant part of the ADT plug-in that you will encounter in using Eclipse for developing Android software.

The Android Layout Editor

Layouts for user interfaces in Android applications can be specified in XML. The ADT plug-in adds a visual editor that helps you to compose and preview Android layouts. When you open a layout file, the ADT plug-in automatically starts this editor to view and edit the file. Tabs along the bottom of the editing pane enable you to switch between the visual editor and an XML editor.

In earlier versions of the Android SDK, the Android Layout Editor was too limited to be of much use. Now, though, you should consider using visual editing of Android layouts as a preferred way of creating layouts. Automating the specification of layouts makes it more likely that your layouts will work on the widest range of Android devices.

The Android Manifest Editor

In Android projects, a manifest file is included with the project’s software and

resources when the project is built. This file tells the Android system how to install and use the software in the archive that contains the built project. The manifest file is in XML, and the ADT plug-in provides a specialized XML editor to edit the manifest.

Other components of the ADT Eclipse plug-in, such as the application builders, can also modify the manifest.

XML editors for other Android XML files

Other Android XML files that hold information such as specifications for menus, or resources such as strings, or that organize graphical assets of an application, have spe cialized editors that are opened when you open these files.

Building Android apps

Eclipse projects are usually built automatically. That means you will normally not en counter a separate step for turning the source code and resources for a project into a deployable result. Android requires Android-specific steps to build a file you can deploy to an Android emulator or device, and the ADT plug-in provides the software that executes these steps. For Android projects, the result of building the project is an *.apk* file. You can find this file for the test project created earlier in this chapter in the *bin* subfolder of the project's file hierarchy in your Eclipse workspace.

The Android-specific builders provided in the ADT plug-in enable you to use Java as the language for creating Android software while running that software on a Dalvik VM that processes its own bytecodes.

Running and debugging Android apps

When you run or debug an Android project from within Eclipse, the *.apk* file for that project is deployed and started on an AVD or Android device, using the ADB and DDMS to communicate with the AVD or device and the Dalvik runtime environment that runs the project's code. The ADT plug-in adds the components that enable Eclipse to do this.

The DDMS

In "The Dalvik Debug Monitor Server (DDMS)" on page 21 we described the Dalvik Debug Monitor and how to invoke the DDMS user interface from the command line. The DDMS user interface is also available from within Eclipse. You can access it by using the Window→Open Perspective→DDMS command in the Eclipse menu. You can also access each view that makes up the DDMS perspective separately by using the Window→Show View menu and selecting, for example, the LogCat view.

Android Virtual Devices

AVDs are made up of QEMU-based emulators that emulate the hardware of an Android device, plus Android system images, which consist of Android software built to run on the emulated hardware. AVDs are configured by the SDK and AVD Manager, which sets parameters such as the size of emulated storage devices and screen dimensions, and which enables you to specify which Android system image will be used with which emulated device.

AVDs enable you to test your software on a broader range of system characteristics than you are likely to be able to acquire and test on physical devices. Because QEMU based hardware emulators, system images, and the parameters of AVDs are all inter changeable parts, you can even test devices and system images before hardware is available to run them.

QEMU

QEMU is the basis of AVDs. But QEMU is a very general tool that is used in a wide range of emulation systems outside the Android SDK. While you will configure QEMU indirectly, through the SDK and AVD Manager, you may someday need to tweak em ulation in ways unsupported by the SDK tools, or you may be curious about the capa bilities and limitations of QEMU. Luckily, QEMU has a large and vibrant developer and user community, which you can find at *http://www.qemu.org*.

The SDK and AVD Manager

QEMU is a general-purpose emulator system. The Android SDK provides controls over the configuration of QEMU that make sense for creating emulators that run Android system images. The SDK and AVD Manager provides a user interface for you to control QEMU-based Android virtual devices.

Other SDK Tools

In addition to the major tools you are likely to use in the normal course of most devel opment projects, there are several other tools in the SDK, and those that are used or invoked directly by developers are described here. Still more components of the SDK are listed in the Tools Overview article in the Android documentation found at *http://developer.android.com/guide/developing/tools/index.html*.

Hierarchy Viewer

The Hierarchy Viewer displays and enables analysis of the view hierarchy of the current activity of a selected Android device. This enables you to see and diagnose problems with your view hierarchies as your application is running, or to examine the view hi erarchies of other applications to see how they are designed. It also lets you examine a magnified view of the screen with alignment guides that help identify problems with layouts. Detailed information on the Hierarchy Viewer is available at *http://developer .android.com/guide/developing/tools/hierarchy-viewer.html*.

Layoutopt

Layoutopt is a static analyzer that operates on XML layout files and can diagnose some problems with Android layouts. Detailed information on Layoutopt is available at *http: //developer.android.com/guide/developing/tools/layoutopt.html*.

Monkey

Monkey is a test automation tool that runs in your emulator or device. You invoke this tool using another tool in the SDK: adb. Adb enables you to start a shell on an emulator or device, and Monkey is invoked from a shell, like this:

```
adb shell monkey --wait-dbg -p your.package.name 500
```

This invocation of Monkey sends 500 random events to the specified application (specified by the package name) after waiting for a debugger to be attached. Detailed information on Monkey can be found at *http://developer.android.com/guide/developing/ tools/monkey.html*.

sqlite3

Android uses SQLite as the database system for many system databases and provides APIs for applications to make use of SQLite, which is convenient for data storage and presentation. SQLite also has a command-line interface, and the sqlite3 command enables developers to dump database schemas and perform other operations on An droid databases.

These databases are, of course, in an Android device, or they are contained in an AVD, and therefore the sqlite3 command is available in the adb shell. Detailed directions for how to access the sqlite3 command line from inside the adb shell are available at *http://developer.android.com/guide/developing/tools/adb.html#shellcommands*. We in troduce sqlite3 in “Example Database Manipulation Using sqlite3” on page 255.

keytool

keytool generates encryption keys, and is used by the ADT plug-in to create temporary debug keys with which it signs code for the purpose of debugging. In most cases, you will use this tool to create a signing certificate for releasing your applications, as de scribed in “Creating a self-signed certificate” on page 99.

Zipalign

Zipalign enables optimized access to data for production releases of Android applica tions. This optimization must be performed after an application is signed for release, because the signature affects byte alignment. Detailed information on Zipalign is avail able at *http://developer.android.com/guide/developing/tools/zipalign.html*.

Draw9patch

A *9 patch* is a special kind of Android resource, composed of nine images, and useful when you want, for example, buttons that can grow larger without changing the radius of their corners. Draw9patch is a specialized drawing program for creating and pre viewing these types of resources. Details on draw9patch are available at *http://developer.android.com/guide/developing/tools/draw9patch.html*.

android

The command named android can be used to invoke the SDK and AVD Manager from the command line, as we described in the SDK installation instructions in “The Android SDK” on page 7. It can also be used to create an Android project from the command line. Used in this way, it causes all the project folders, the manifest, the build properties, and the ant script for building the project to be generated. Details on this use of the android command can be found at *http://developer.android.com/guide/developing/other-ide.html#CreatingAProject*.

Keeping Up-to-Date

The JDK, Eclipse, and the Android SDK each come from separate suppliers. The tools you use to develop Android software can change at a rapid pace. That is why, in this book, and especially in this chapter, we refer you to the Android Developers site for information on the latest compatible versions of your tools. Keeping your tools up-to date and compatible is a task you are likely to have to perform even as you learn how to develop Android software.

Windows, Mac OS X, and Linux all have system update mechanisms that keep your software up-to-date. But one consequence of the way the Android SDK is put together is that you will need to keep separate software systems up-to-date through separate mechanisms.

Keeping the Android SDK Up-to-Date

The Android SDK isn't part of your desktop OS, nor is it part of the Eclipse plug-in, and therefore the contents of the SDK folder are not updated by the OS or Eclipse. The SDK has its own update mechanism, which has a user interface in the SDK and AVD Manager. As shown in Figure 1-14, select Installed Packages in the left pane to show a list of SDK components installed on your system. Click on the Update All button to start the update process, which will show you a list of available updates.

Figure 1-14. Updating the SDK with the SDK and AVD Manager

Usually, you will want to install all available updates.

Keeping Eclipse and the ADT Plug-in Up-to-Date

While the SDK has to be updated outside of both your operating system and Eclipse, the ADT plug-in, and all other components of Eclipse, are updated using Eclipse's own update management system. To update all the components you have in your Eclipse environment, including the ADT plug-in,

use the “Check for Updates” command in the Help menu. This will cause the available updates to be displayed, as shown in Figure 1-15.

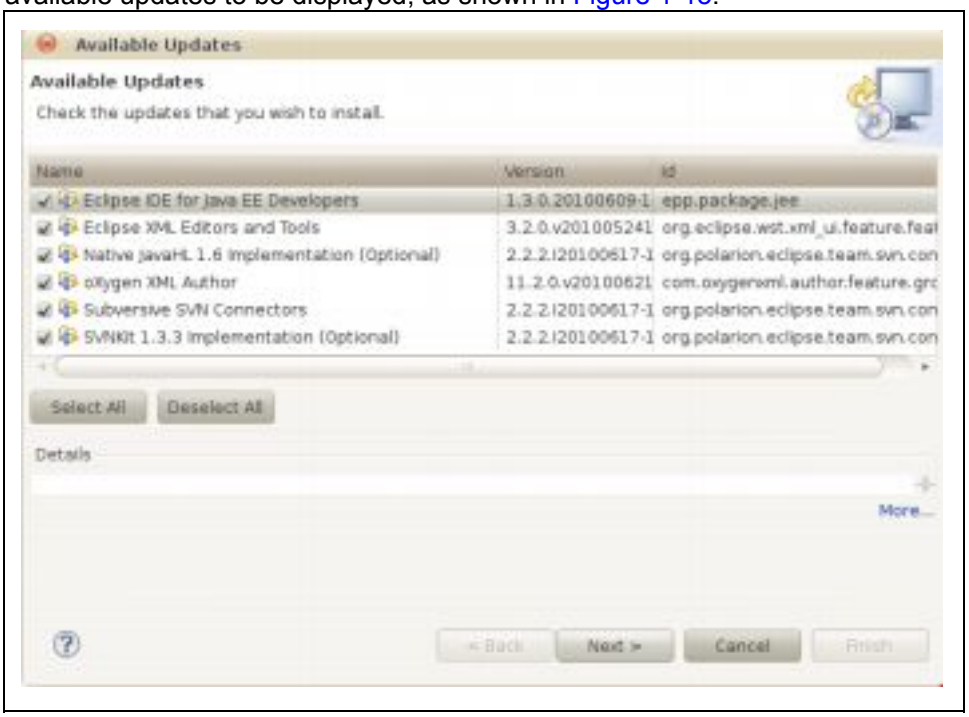

Figure 1-15. Updating Eclipse components and the ADT plug-in

Normally, you will want to use the Select All button to install all available updates. The updates you see listed on your system depend on what Eclipse modules you have in stalled and whether your Eclipse has been updated recently.

Keeping the JDK Up-to-Date

You won’t be updating Java as much as the SDK, ADT plug-in, and other Eclipse plug ins. Even if Java 7 has not been released by the time you read this, it is likely to happen soon enough to matter to Android developers. Before choosing to update the JDK, first check the System Requirements page of the Android Developers site at *http://developer.android.com/sdk/requirements.html*.

If an update is needed and you are using a Mac or Linux system, check the available updates for your system to see if a new version of the JDK is included. If the JDK was installed on your system by the vendor, or if you

installed it from your Linux distribu tion's repositories, updates will be available through the updates mechanism on your system.

Example Code

Having installed the Android SDK and tested that it works, you are ready to explore. Even if you are unfamiliar with the Android Framework classes and are new to Java, exploring some example code now will give you further confidence in your SDK in stallation, before you move on to other parts of this book.

SDK Example Code

The most convenient sample code comes with the SDK. You can create a new project based on the SDK samples, as shown in Figure 1-16. The sample you select appears in the left pane of the Eclipse window, where you can browse the files comprising the sample and run it to see what it does. If you are familiar with using IDEs to debug code, you may want to set some breakpoints in the sample code to see when methods get executed.

> In the dialog pictured in Figure 1-16, you must pick a build target before you pick a sample. Samples are organized by API level, and if you have not picked a build target, the drop-down list will be empty.

Each sample application that comes with the SDK corresponds to an article on the Android Developers site. More information about each sample can be found there. All of the samples are listed on the documentation page at *http://developer.android.com/ resources/samples/index.html*.

There are more than a dozen applications, one of which—the API demos application— is a sprawling exploration of most of the Android APIs. Creating a few projects based on these code samples will give you familiarity with how these programs work, and will help you understand what you will read in the upcoming chapters of this book, even if you don't fully understand what you are looking at yet.

Example Code from This Book

Example code from this book can be downloaded from the book's website at *http:// oreilly.com/catalog/0636920010364*.

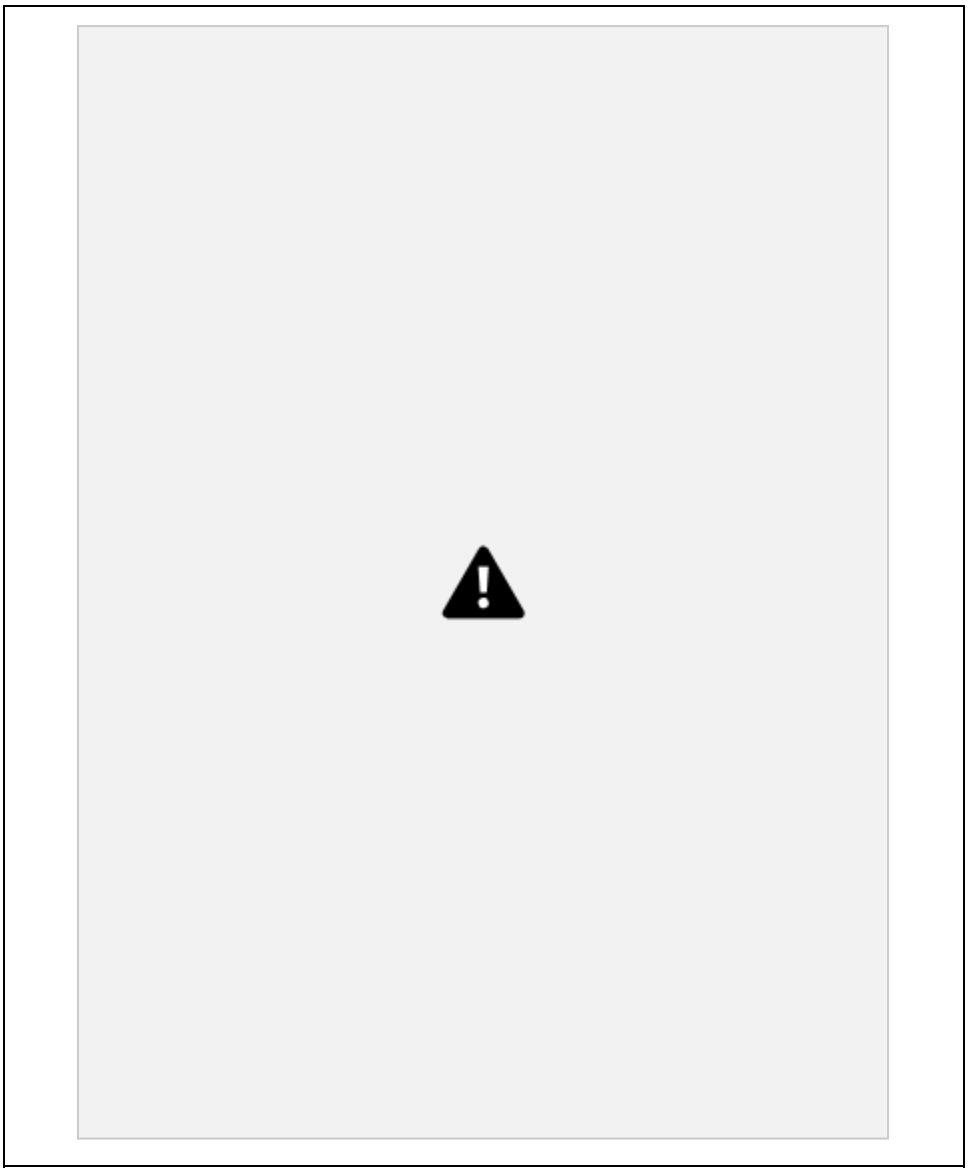

Figure 1-16. Creating a new project using example code from the SDK

On Reading Code

Good coders read a lot of code. The example code provided by the authors of this book is intended to be both an example of good Java coding and an example of how to use capabilities of the Android platform.

Some examples you will read fall short of what you will need for creating the best possible extensible and maintainable commercial software. Many example applications make choices that make sense if the coder's goal is to create an example in a single Java class. In many cases, Android applications are overgrown versions of example code, and they end up unreadable and unmaintainable. But that does not mean you should avoid reading examples that are more expedient than a large application should be.

The next chapter will explore the Java language, with the goal of giving you the ability to evaluate example code with good engineering and design practices in mind. We want you to be able to take examples and make them better, and to apply the ideas in ex amples to code you engineer to create high-quality products.

CHAPTER 2
Java for Android

We don't teach you Java in this book, but in this chapter we'll help you understand the special use of Java within Android. Many people can benefit from this chapter: students who have learned some Java but haven't yet stumbled over the real-life pro gramming dilemmas it presents, programmers from other mobile environments who have used other versions of Java but need to relearn some aspects of the language in the context of Android programming, and Java programmers in general who are new to Android's particular conventions and requirements.

If you find this chapter too fast-paced, pick up an introductory book on Java. If you follow along all right but a particular concept described in this chapter remains unclear to you, you might refer to the Java tutorial at *http://download.oracle.com/docs/cd/ E17409_01/javase/tutorial/index.html*.

Android Is Reshaping Client-Side Java

Android is already the most widely used way of creating interactive clients using

the Java language. Although there have been several other user interface class libraries for Java (AWT, SWT, Swing, J2ME Canvas, etc.), none of them have been as widely ac cepted as Android. For any Java programmer, the Android UI is worth learning just to understand what the future of Java UIs might look like.

The Android toolkit doesn't gratuitously bend Java in unfamiliar directions. The mobile environment is simply different. There is a much wider variety of display sizes and shapes; there is no mouse (though there might be a touch screen); text input might be triple-tap; and so on. There are also likely to be many more peripheral devices: motion sensors, GPS units, cameras, multiple radios, and more. Finally, there is the ever present concern about power. While Moore's law affects processors and memory (dou bling their power approximately every two years), no such law affects battery life. When processors were slow, developers used to be concerned about CPU speed and efficiency. Mobile developers, on the other hand, need to be concerned about energy efficiency.

This chapter provides a refresher for generic Java; Android-specific libraries are dis cussed in detail in Chapter 3.

The Java Type System

There are two distinct, fundamental types in the Java language: objects and primitives. Java provides type safety by enforcing static typing, which requires that every variable must be declared with its type before it is used. For example, a variable named i declared as type int (a primitive 32-bit integer) looks like this:

```
int i;
```

This mechanism stands in contrast to nonstatically typed languages where variables are only optionally declared. Though explicit type declarations are more verbose, they enable the compiler to prevent a wide range of programming errors—accidental vari able creation resulting from misspelled variable names, calls to nonexistent methods, and so on—from ever making it into running code. Details of the Java Type System can be found in the Java Language Specification.

Primitive Types

Java primitive types are not objects and do not support the operations associated with objects described later in this chapter. You can modify a primitive type only with a limited number of predefined operators: "+", "-", "&", "|", "=", and so on. The Java primitive types are:

boolean
: The values true or false

byte
: An 8-bit 2's-complement integer

short
: A 16-bit 2's-complement integer

int
: A 32-bit 2's-complement integer

long
: A 64-bit 2's-complement integer

char
: A 16-bit unsigned integer representing a UTF-16 code unit

float
: A 32-bit IEEE 754 floating-point number

double
: A 64-bit IEEE 754 floating-point number

Objects and Classes

Java is an object-oriented language and focuses not on its primitives but on objects— combinations of data, and procedures for operating on that data. A *class* defines the fields (data) and methods (procedures) that comprise an object. In Java, this definition —the template from which objects are constructed—is, itself, a particular kind of ob ject, a Class. In Java, classes form the basis of a type system that allows developers to describe arbitrarily complex objects with complex, specialized state and behavior.

In Java, as in most object-oriented languages, types may inherit from other types. A class that inherits from another is said to *subtype* or to be a *subclass* of its parent. The parent class, in turn, may be called the *supertype* or *superclass*. A class that has several different subclasses may be called the *base type* for those subclasses.

Both methods and fields have global scope within the class and may be visible from outside the object through a reference to an instance of the class.

Here is the definition of a very, very simple class with one field, ctr, and one method, incr:

```
public class Trivial {
 /** a field: its scope is the entire class */
 private long ctr;

 /** Modify the field. */
 public void incr() { ctr++; }
}
```

Object Creation

A new object, an instance of some class, is created by using the new keyword: `Trivial trivial = new Trivial();`

On the left side of the assignment operator "=", this statement defines a variable, named trivial. The variable has a type, Trivial, so only objects of type Trivial can be assigned to it. The right side of the assignment allocates memory for a new instance of the Trivial class and initializes the instance. The assignment operator assigns a reference to the newly created object to the variable.

It may surprise you to know that the definition of ctr, in Trivial, is perfectly safe despite the fact that it is not explicitly initialized. Java guarantees that it will be initialized to have the value 0. Java guarantees that all fields are automatically initialized at object creation: boolean is initialized to false, numeric primitive types to 0, and all object types (including Strings) to null.

> This applies only to object fields. Local variables must be initialized before they are referenced!

You can take greater control over the initialization of an object by adding a *construc tor* to its class definition. A constructor definition looks like a method except that it doesn't specify a return type. Its name must be exactly the name of the class that it constructs:

```
public class LessTrivial {
 /** a field: its scope is the entire class */
 private long ctr;

 /** Constructor: initialize the fields */
 public LessTrivial(long initCtr) { ctr = initCtr; }

 /** Modify the field. */
 public void incr() { ctr++; }
}
```

In fact, every class in Java has a constructor. The Java compiler automatically creates a constructor with no arguments, if no other constructor is specified. Further, if a con structor does not explicitly call some superclass constructor, the Java compiler will automatically add an implicit call to the superclass no-arg constructor as the very first statement. The definition of Trivial given earlier (which specifies no explicit construc tor), actually has a constructor that looks like this:

```
public Trivial() { super(); }
```

Since the LessTrivial class explicitly defines a constructor, Java does *not* implicitly add a default. That means that trying to create a LessTrivial object, with no arguments, will cause an error:

```
LessTrivial fail = new LessTrivial(); // ERROR!!
LessTrivial ok = new LessTrivial(18); // ... works
```

There are two concepts that it is important to keep separate: *no-arg constructor* and *default constructor*. A default constructor is the constructor that Java adds to your class, implicitly, if you don't define any other constructors. It happens to be a no-arg con structor. A no-arg constructor, on the other hand, is simply a constructor with no parameters. There is no requirement that a class have a no-arg constructor. There is no obligation to define one, unless you have a specific need for it.

> One particular case in which no-arg constructors are necessary deserves special attention. Some libraries need the ability to create new objects, generically, on your behalf. The JUnit framework, for instance, needs to be able to create new test cases, regardless of what they test. Libraries that marshal and unmarshal code to a persistent store or a network connection also need this capability. Since it would be pretty hard for these libraries to figure out, at runtime, the exact calling protocol for your particular object, they typically require a no-arg constructor.

If a class has more than one constructor, it is wise to cascade them, to make sure only a single copy of the code actually initializes the instance and that all other constructors

call it. For instance, as a convenience, we might add a no-arg constructor to the LessTrivial class, to accommodate a common case:

```
public class LessTrivial {
 /** a field: its scope is the entire class */
 private long ctr;

 /** Constructor: init counter to 0 */
 public LessTrivial() { this(0); }

 /** Constructor: initialize the fields */
 public LessTrivial(long initCtr) { ctr = initCtr; }

 /** Modify the field. */
 public void incr() { ctr++; }
}
```

Cascading methods is the standard Java idiom for defaulting the values of some argu ments. All the code that actually initializes an object is in a single, complete method or constructor and all other methods or constructors simply call it. It is a particularly good idea to use this idiom with constructors that must make explicit calls to a superconstructor.

Constructors should be simple and should do no more work than is necessary to put an object into a consistent initial state. One can imagine, for instance, a design for an object that represents a database or network connection. It might create the connection, initialize it, and verify connectivity, all in the constructor. While this might seem en tirely reasonable, in practice it creates code that is insufficiently modular and difficult to debug and modify. In a better design, the constructor simply initializes the connec tion state as closed and leaves it to an explicit open method to set up the network.

The Object Class and Its Methods

The Java class Object—java.lang.Object—is the root ancestor of every class. Every Java object is an Object. If the definition of a class does not explicitly specify a super class, it is a direct subclass of Object. The Object class defines the default implemen tations for several key behaviors that are common to every object. Unless they are overridden by the subclass, the behaviors are inherited directly from Object.

The methods wait, notify, and notifyAll in the Object class are part of Java's concur rency support. They are discussed in "Thread Control with wait() and notify() Meth ods" on page 71.

The toString method is the way an object creates a string representation of itself. One interesting use of toString is string concatenation: any object can be concatenated to a string. This example demonstrates two ways to print the same message: they both execute identically. In both, a new instance of the Foo class is created, its toString method is invoked, and the result is concatenated with a literal string. The result is then printed:


```
System.out.println(
 "This is a new foo: " + new Foo());
System.out.println(
 "This is a new foo: ".concat((new Foo()).toString()));
```

The Object implementation of toString returns a not-very-useful string that is based on the location of the object in the heap. Overriding toString in your code is a good first step toward making it easier to debug.

The clone and finalize methods are historical leftovers. The Java runtime will call the finalize method only if it is overridden in a subclass. If a class explicitly defines finalize, though, it is called for an object of the class just before that object is garbage collected. Not only does Java not guarantee when this might happen, it actually can't guarantee that it will happen at all. In addition, a call to finalize can resurrect an object! This is tricky: objects are garbage-collected when there are no live references to them. An implementation of finalize, however, could easily *create* a new live reference, for instance, by adding the object being finalized to some kind of list! Because of this, the existence of a finalize method precludes the defining class from many kinds of opti mization.

There is little to gain and lots to lose in attempting to use finalize.

The clone method creates objects, bypassing their constructors. Although clone is defined on Object, calling it on an object will cause an exception unless the object implements the Cloneable interface. The clone method is an optimization that can be useful when object creation has a significant cost. While clever uses of clone may be necessary in specific cases, a copy constructor—one which takes an existing instance as its only argument—is much more straightforward and, in most cases, has negligible cost.

The last two Object methods, hashCode and equals, are the methods by which a caller can tell whether one object is "the same as" another.

The definition of the equals method in the API documentation for the Object class stipulates the contract to which every implementation of equals must adhere. A correct implementation of the equals method has the following attributes, and the associated statements must always be true:

reflexive

```
x.equals(x)
```

symmetric

```
x.equals(y) == y.equals(x)
```

transitive

```
(x.equals(y) && y.equals(z)) == x.equals(z)
```

consistent

If x.equals(y) is true at any point in the life of a program, it is always true, provided x and y do not change.

Getting this right is subtle and can be surprisingly difficult. A common error—one that violates reflexivity—is defining a new class that is sometimes equal to an existing class. Suppose your program uses an existing library that defines the class EnglishWeekdays. Suppose, now, that you define a class FrenchWeekdays. There is an obvious temptation to define an equals method for FrenchWeekdays that returns true when it compares one of the EnglishWeekdays to its French equivalent. Don't do it! The existing English class has no awareness of your new class and so will never recognize instances of your class as being equal. You've broken reflexivity!

hashCode and equals should be considered a pair: if you override either, you should override both. Many library routines treat hashCode as an optimized rough guess as to whether two objects are equal or not. These libraries first compare the hash codes of the two objects. If the two codes are different, they assume there is no need to do any more expensive comparisons because the objects are definitely different. The point of hash code computation, then, is to compute something very quickly that is a good proxy for the equals method.

Visiting every cell in a large array, in order to compute a hash code, is probably no faster than doing the actual comparison. At the other extreme, it would be very fast to return 0, always, from a hash code computation. It just wouldn't be very helpful.

Objects, Inheritance, and Polymorphism

Java supports *polymorphism*, one of the key concepts in object-oriented programming. A language is said to be polymorphic if objects of a single type can have different be havior. This happens when subtypes of a given class can be assigned to a variable of the base class type. An example will make this much clearer.

Subtypes in Java are declared through use of the extends keyword. Here is an example of inheritance in Java:

```
public class Car {
 public void drive() {
 System.out.println("Going down the road!");
 }
}

public class Ragtop extends Car {
 // override the parent's definition.
 public void drive() {
 System.out.println("Top down!");

 // optionally use a superclass method
 super.drive();

 System.out.println("Got the radio on!");
 }
}
```


Ragtop is a subtype of Car. We noted previously that Car is, in turn, a subclass of Object. Ragtop changes the definition of Car's drive method. It is said to *override* drive. Car and Ragtop are both of type Car (they are not both of type Ragtop!) and have different behaviors for the method drive.

We can now demonstrate polymorphic behavior:

```
Car auto = new Car();
auto.drive();
auto = new Ragtop();
auto.drive();
```

This code fragment will compile without error (despite the assignment of a Ragtop to a variable whose type is Car). It will also run without error and would produce the following output:

```
Going down the road!
```

```
Top down!
Going down the road!
Got the radio on!
```

The variable auto holds, at different times in its life, references to two different objects of type Car. One of those objects, in addition to being of type Car, is also of subtype Ragtop. The exact behavior of the statement auto.drive() depends on whether the variable currently contains a reference to the former or the latter. This is polymorphic behavior.

Like many other object-oriented languages, Java supports type casting to allow coercion of the declared type of a variable to be any of the types with which the variable is polymorphic:

```
Ragtop funCar;

Car auto = new Car();
funCar = (Ragtop) auto; //ERROR! auto is a Car, not a Ragtop!
auto.drive();


auto = new Ragtop();
Ragtop funCar = (Ragtop) auto; //Works! auto is a Ragtop
auto.drive();
```

While occasionally necessary, excessive use of casting is an indication that the code is missing the point. Obviously, by the rules of polymorphism, all variables could be declared to be of type Object, and then cast as necessary. To do that, however, is to abandon the value of static typing.

Java limits a method's arguments (its actual parameters) to objects of types that are polymorphic with its formal parameters. Similarly, methods return values that are polymorphic with the declared return type. For instance, continuing our automotive example, the following code fragment will compile and run without error:


```
public class JoyRide {
 private Car myCar;

 public void park(Car auto) {
 myCar = auto;
 }

 public Car whatsInTheGarage() {
 return myCar;
 }

 public void letsGo() {
 park(new Ragtop());
 whatsInTheGarage().drive();
 }
}
```

The method park is declared to take an object of type Car as its only parameter. In the method letsGo, however, it is called with an object of type Ragtop, a subtype of type Car. Similarly, the variable myCar is assigned a value of type Ragtop, and the method whatsInTheGarage returns it. The object is a Ragtop: if you call its drive method, it will tell you about its top and its radio. On the other hand, since it is also a Car, it can be used anywhere that one would use a Car. This subtype replacement capability is a key example of the power of polymorphism and how it works with type safety. Even at compile time, it is clear whether an object is compatible with its use, or not. Type safety enables the compiler to find errors, early, that might be much more difficult to find were they permitted to occur at runtime.

Final and Static Declarations

There are 11 modifier keywords that can be applied to a declaration in Java. These modifiers change the behavior of the declared object, sometimes in important ways. The earlier examples used a couple of them, public and private, without explanation: they are among the several modifiers that control scope and visibility. We'll revisit them in a minute. In this section, we consider two other modifiers that are essential to a complete understanding of the Java type system: final and static.

A final declaration is one that cannot be changed. Classes, methods, fields, parameters, and local variables can all be final.

When applied to a class, final means that any attempt to define a subclass will cause an error. The class String, for instance, is final because strings must be immutable (i.e., you can't change the content of one after you create it). If you think about it for a while, you will see that this can be *guaranteed* only if String cannot be subtyped. If it were possible to subtype the String class, a devious library could create a subclass of String, DeadlyString, pass an instance to your code, and change its value from "fred" to "'; DROP TABLE contacts;" (an attempt to inject rogue SQL into your system that

might wipe out parts of your database) immediately after your code had validated its contents!

When applied to a method, final means that the method cannot be overridden in a subclass. Developers use final methods to design for inheritance, when the supertype needs to make a highly implementation-dependent behavior available to a subclass and cannot allow that behavior to be changed. A framework that implemented a generic cache might define a base class CacheableObject, for instance, which the programmer using the framework subtypes for each new cacheable object type. In order to maintain the integrity of the framework, however, CacheableObject might need to compute a cache key

that was consistent across all object types. In this case, it might declare its computeCacheKey method final.

When applied to a variable—a field, a parameter, or a local variable—final means that the value of the variable, once assigned, may not change. This restriction is enforced by the compiler: it is not enough that the value *does not* change, the compiler must be able to prove that it *cannot* change. For a field, this means that the value must be assigned either as part of the declaration or in every constructor. Failure to initialize a final field at its declaration or in the constructor—or an attempt to assign to it any where else—will cause an error.

For parameters, final means that, within the method, the parameter value always has the value passed in the call. An attempt to assign to a final parameter will cause an error. Of course, since the parameter value is most likely to be a reference to some kind of object, it is possible that the object might change. The application of the keyword final to a parameter simply means that the parameter cannot be assigned.

> In Java, parameters are passed by value: the method arguments are new copies of the values that were passed at the call. On the other hand, most things in Java are references to objects and Java only copies the refer ence, not the whole object! References are passed by value!

A final variable may be assigned no more than once. Since using a variable without initializing it is also an error, in Java, a final variable must be assigned exactly once. The assignment may take place anywhere in the enclosing block, prior to use.

A static declaration belongs to the class in which it is described, not to an instance of that class. The opposite of static is *dynamic*. Any entity that is not declared static is implicitly dynamic. This example illustrates:

```
public class QuietStatic {
 public static int classMember;
 public int instanceMember;
}

public class StaticClient {
 public void test() {
 QuietStatic.classMember++;
```



```
 QuietStatic.instanceMember++; // ERROR!!

 QuietStatic ex = new QuietStatic();
 ex.classMember++; // WARNING!
 ex.instanceMember++;
 }
}
```

In this example, QuietStatic is the name of a class, and ex is a reference to an

instance of that class. The static member classMember is an attribute of the class; you can refer to it simply by qualifying it with the class name. On the other hand, instanceMember is a member of an *instance* of the class. An attempt to refer to it through the class reference causes an error. That makes sense. There are many different variables called instance Member, one belonging to each instance of QuietStatic. If you don't explicitly specify which one you are talking about, there's no way for Java to figure it out.

As the second pair of statements demonstrates, Java does actually allow references to class (static) variables through instance references. It is misleading, though, and con sidered a bad practice. Most compilers and IDEs will generate warnings if you do it.

The implications of static versus dynamic declarations can be subtle. It is easiest to understand the distinction for fields. Again, while there is exactly one copy of a static definition, there is one copy per instance of a dynamic definition. Static class members allow you to maintain information that is held in common by all members of a class. Here's some example code:

```
public class LoudStatic {
 private static int classMember;
 private int instanceMember;

 public void incr() {
 classMember++;
 instanceMember++;
 }

 @Override public String toString() {
 return "classMember: " + classMember
 + ", instanceMember: " + instanceMember;
 }

 public static void main(String[] args) {
 LoudStatic ex1 = new LoudStatic();
 LoudStatic ex2 = new LoudStatic();
 ex1.incr();
 ex2.incr();
 System.out.println(ex1);
 System.out.println(ex2);
 }
}
```

and its output:

```
classMember: 2, instanceMember: 1
classMember: 2, instanceMember: 1
```


The initial value of the variable classMember in the preceding example is 0. It is incre mented by each of the two different instances. Both instances now see a new value, 2. The value of the variable instanceMember also starts at 0, in each instance. On the other hand, though, each instance increments its own copy and sees the value of its own variable, 1.

Static class and method definitions are similar in that, in both cases, a static object is visible using its qualified name, while a dynamic object is visible only through an in stance reference. Beyond that, however, the differences are trickier.

One significant difference in behavior between statically and dynamically declared methods is that statically declared methods cannot be overridden in a subclass. The following, for instance, fails to compile:

```
public class Star {
 public static void twinkle() { }
}

public class Arcturus extends Star {
 public void twinkle() { } // ERROR!!
}

public class Rigel {
 // this one works
 public void twinkle() { Star.twinkle(); }
}
```

There is very little reason to use static methods in Java. In early implementations of Java, dynamic method dispatch was significantly slower than static dispatch. Devel opers used to prefer static methods in order to "optimize" their code. In Android's just in-time-compiled Dalvik environment, there is no need for this kind of optimization anymore. Excessive use of static methods is usually an indicator of bad architecture.

The difference between statically and dynamically declared classes is the subtlest. Most of the classes that comprise an application are static. A typical class is declared and defined at the *top level*—outside any enclosing block. Implicitly, all such declarations are static. Most other declarations, on the other hand, take place within the enclosing block of some class and are, by default, dynamic. Whereas most fields are dynamic by default and require a modifier to be static, most classes are static.

> A *block* is the code between two curly braces: { and }. Anything—var iables, types, methods, and so on—defined within the block is visible within the block and within lexically nested blocks. Except within the special block defining a class, things defined within a block are not visi ble outside the block.

This is, actually, entirely consistent. According to our description of static—something that belongs to the class, not to an instance of that class—top-level declarations should

be static (they belong to no class). When declared within an enclosing block,

however— for example, inside the definition of a top-level class—a class definition is also dynamic by default. In order to create a dynamically declared class, just define it inside another class.

This brings us to the difference between a static and a dynamic class. A dynamic class has access to instance members of the enclosing class (since it belongs to the instance). A static class does not. Here's some code to demonstrate:

```
public class Outer {
 public int x;

 public class InnerOne {
 public int fn() { return x; }
 }

 public static class InnerTube {
 public int fn() {
 return x; // ERROR!!!
 }
 }
}

public class OuterTest {
 public void test() {
 new Outer.InnerOne(); // ERROR!!!
 new Outer.InnerTube();
 }
}
```

A moment's reflection will clarify what is happening here. The field x is a member of an instance of the class Outer. In other words, there are lots of variables named x, one for each runtime instance of Outer. The class InnerTube is a part of the class Outer, but not of any *instances* of Outer. It has no way of identifying an x. The class InnerOne, on the other hand, because it is dynamic, belongs to an instance of Outer. You might think of a separate class InnerOne for each instance of Outer (though this is not, actually, how it is implemented). Consequently, InnerOne has access to the members of the instance of Outer to which it belongs.

OuterTest demonstrates that, as with fields, we can use the static inner definition (in this case, create an instance of the class) simply by using its qualified name. The dy namic definition is useful, however, only in the context of an instance.

Abstract Classes

Java permits a class declaration to entirely omit the implementation of one or more methods by declaring the class and unimplemented methods to be abstract:

```
public abstract class TemplatedService {

 public final void service() {
```

```
// subclasses prepare in their own ways
prepareService();
// ... but they all run the same service
runService()
}

public abstract void prepareService();

private final void runService() {
// implementation of the service ...
}
}

public class ConcreteService extends TemplatedService {
void prepareService() {
// set up for the service
}
}
```

An abstract class cannot be instantiated. Subtypes of an abstract class must either pro vide definitions for all the abstract methods in the superclass or must, themselves, be declared abstract.

As hinted in the example, abstract classes are useful in implementing the common template pattern, which provides a reusable piece of code that allows customization at specific points during its execution. The reusable pieces are implemented as an abstract class. Subtypes customize the template by implementing the abstract methods.

For more information on abstract classes, see the Java tutorial at *http://download.oracle .com/javase/tutorial/java/landl/abstract.html*.

Interfaces

Other languages (e.g., C++, Python, and Perl) permit a capability known as multiple implementation inheritance, whereby an object can inherit implementations of meth ods from more than one parent class. Such inheritance hierarchies can get pretty com plicated and behave in unexpected ways (such as inheriting two field variables with the same name from two different superclasses). Java's developers chose to trade the power of multiple inheritance for simplicity. Unlike the mentioned languages, in Java a class may extend only a single superclass.

Instead of multiple implementation inheritance, however, Java provides the ability for a class to inherit from several types, using the concept of an *interface*. Interfaces provide a way to define a type without defining its implementation. You can think of interfaces as abstract classes with all abstract methods. There is no limit on the number of inter faces that a class may implement.

Here's an example of a Java interface and a class that implements it:

```
public interface Growable {
 // declare the signature but not the implementation
```



```
 void grow(Fertilizer food, Water water);
}

public interface Eatable {
 // another signature with no implementation
 void munch();
}

/**
 * An implementing class must implement all interface methods
 */
public class Beans implements Growable, Eatable {

 @Override
 public void grow(Fertilizer food, Water water) {
 // ...
 }

 @Override
 public void munch() {
 // ...
 }
}
```

Again, interfaces provide a way to define a type distinct from the implementation of that type. This kind of separation is common even in everyday life. If you and a colleague are trying to mix mojitos, you might well divide tasks so that she goes to get the mint. When you start muddling things in the bottom of the glass, it is irrelevant whether she drove to the store to buy the mint or went out to the backyard and picked it from a shrub. What's important is that you have mint.

As another example of the power of interfaces, consider a program that needs to display a list of contacts, sorted by email address. As you would certainly expect, the Android runtime libraries contain generic routines to sort objects. Because they are generic, however, these routines have no intrinsic idea of what ordering means for the instances of any particular class. In order to use the library sorting routines, a class needs a way to define its own ordering. Classes do this in Java using the interface Comparable.

Objects of type Comparable implement the method compareTo. One object accepts an other, similar object as an argument and returns an integer that indicates whether the argument object is greater than, equal to, or less than the target. The library routines can sort anything that is Comparable. A program's Contact type need only be Compara ble and implement compareTo to allow contacts to be sorted:

```
public class Contact implements Comparable<Contact> {
 // ... other fields
```

```
private String email;

public Contact(
// other params...
String emailAddress)
{
```



```
// ... init other fields from corresponding params
email = emailAddress;
}

public int compareTo(Contact c) {
return email.compareTo(c.email);
}
}

public class ContactView {
// ...

private List<Contact> getContactsSortedByEmail(
List<Contact> contacts)
{
// getting the sorted list of contacts
// is completely trivial
return Collections.sort(contacts);
}

// ...
}
```

Internally, the Collections.sort routine knows only that contacts is a list of things of type Comparable. It invokes the class's compareTo method to decide how to order them.

As this example demonstrates, interfaces enable the developer to reuse generic routines that can sort any list of objects that implement Comparable. Beyond this simple example, Java interfaces enable a diverse set of programming patterns that are well described in other sources. We frequently and highly recommend the excellent *Effective Java* by Joshua Bloch (Prentice Hall).

Exceptions

The Java language uses *exceptions* as a convenient way to handle unusual conditions. Frequently these conditions are errors.

Code trying to parse a web page, for instance, cannot continue if it cannot read the page from the network. Certainly, it is possible to check the results of the attempt to read and proceed only if that attempt succeeds, as shown in this example:

```
public void getPage(URL url) {
String smallPage = readPageFromNet(url);
```

```
if (null != smallPage) {
Document dom = parsePage(smallPage);
if (null != dom) {
NodeList actions = getActions(dom);
if (null != action) {
// process the action here...
}
}
}
}
```


Exceptions make this more elegant and robust:

```
public void getPage(URL url)
 throws NetworkException, ParseException, ActionNotFoundException {
 String smallPage = readPageFromNet(url);
 Document dom = parsePage(smallPage);
 NodeList actions = getActions(dom);
 // process the action here...
}

  public String readPageFromNet(URL url) throws NetworkException {
// ...
public Document parsePage(String xml) throws ParseException {
// ...
public NodeList getActions(Document doc) throws ActionNotFoundException { //
...
```

In this version of the code, each method called from `getPage` uses an exception to im mediately short-circuit all further processing if something goes wrong. The methods are said to *throw* exceptions. For instance, the `getActions` method might look some thing like this:

```
public NodeList getActions(Document dom)
 throws ActionNotFoundException
{
 Object actions = xPathFactory.newXPath().compile("//node/@action")
.evaluate(dom, XPathConstants.NODESET);
 if (null == actions) {
 throw new ActionNotFoundException("Action not found");
 }
 return (NodeList) actions;
}
```

When the `throw` statement is executed, processing is immediately interrupted and re sumes at the nearest *catch* block. Here's an example of a try-catch block:

```
for (int i = 0; i < MAX_RETRIES; i++) {
 try {
 getPage(theUrl);
 break;
 }
 catch (NetworkException e) {
 Log.d("ActionDecoder", "network error: " + e);
 }
```

```
}
```

This code retries network failures. Note that it is not even in the same method, readPageFromNet, that threw the NetworkException. When we say that processing re sumes at the “nearest” try-catch block, we’re talking about an interesting way that Java delegates responsibility for exceptions.

If there is no try-catch block surrounding the throw statement within the method, a thrown exception makes it seem as though the method returns immediately. No further statements are executed and no value is returned. In the previous example, for instance, none of the code following the attempt to get the page from the network needs to concern itself with the possibility that the precondition—a page was read—was not met. The method is said to have been *terminated abruptly* and, in the example, control returns to getActions. Since getActions does not contain a try-catch block ei ther, it is terminated abruptly too. Control is passed back (up the stack) to the caller.

In the example, when a NetworkException is thrown, control returns to the first state ment inside the example catch block, the call to log the network error. The exception is said to have been *caught* at the first catch statement whose argument type is the same type, or a supertype, of the thrown exception. Processing resumes at the first statement in the catch block and continues normally afterward.

In the example, a network error while attempting to read a page from the network will cause both ReadPageFromNet and getPage to terminate abruptly. After the catch block logs the failure, the for loop will retry getting the page, up to MAX_RETRIES times.

It is useful to have a clear understanding of the root of the Java exception class tree, shown in Figure 2-1.

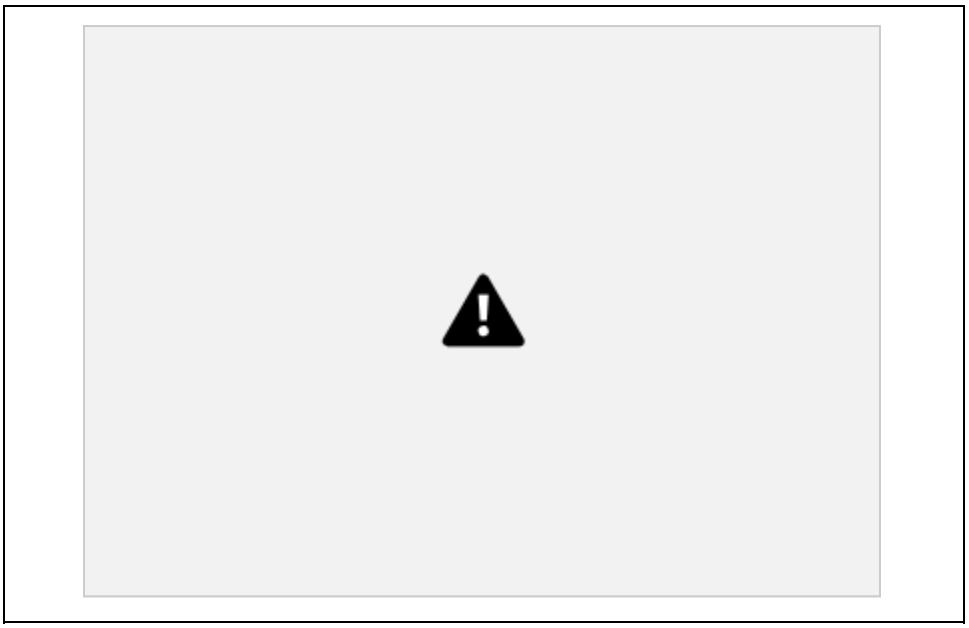

Figure 2-1. Exception base classes

All exceptions are subclasses of Throwable. There is almost never any reason to make reference to Throwable in your code. Think of it as just an abstract base class with two

subclasses: Error and Exception. Error and its subclasses are reserved for problems with the Dalvik runtime environment itself. While you can write code that appears to catch an Error (or a Throwable), you cannot, in fact, catch them. An obvious example of this, for instance, is the dreaded OOME, the OutOfMemoryException error. When the Dalvik system is out of memory, it may not be able to complete execution of even a single opcode! Writing tricky code that attempts to catch an OOME and then to release some block of preallocated memory might work—or it might not. Code that tries to catch Throwable or Error is absolutely whistling in the wind.

Java requires the signature of a method to include the exceptions that it throws. In the previous example, getPage declares that it throws three exceptions, because it uses three methods, each of which throws one. Methods that call getPage must, in turn, declare all three of the exceptions that getPage throws, along with any others thrown by any other methods that it calls.

As you can imagine, this can become onerous for methods far up the call tree. A top level method might have to declare tens of different kinds of exceptions, just because it calls methods that throw them. This problem can be mitigated by

creating an ex ception tree that is congruent to the application tree. Remember that a method needs only to declare supertypes for all the exceptions it throws. If you create a base class named MyApplicationException and then subclass it to create MyNetworkException and MyUIException for the networking and UI subsystems, respectively, your top-layer code need only handle MyApplicationException.

Really, though, this is only a partial solution. Suppose networking code somewhere way down in the bowels of your application fails, for instance, to open a network con nection. As the exception bubbles up through retries and alternatives, at some point it loses any significance except to indicate that "something went wrong." A specific da tabase exception, for instance, means nothing to code that is trying to prepopulate a phone number. Adding the exception to a method signature, at that point, is really just a nuisance: you might as well simply declare that all your methods throw Exception.

RuntimeException is a special subclass of Exception. Subclasses of RuntimeException are called *unchecked* exceptions and do not have to be declared. This code, for instance, will compile without error:

```
public void ThrowsRuntimeException() {
 throw new RuntimeException();
}
```

There is considerable debate in the Java community about when to use and when not to use unchecked exceptions. Obviously, you could use only unchecked exceptions in your application and never declare any exception in any of your method signatures. Some schools of Java programming even recommend this. Using checked exceptions, however, gives you the chance to use the compiler to verify your code and is very much in the spirit of static typing. Experience and taste will be your guide.

The Java Collections Framework

The Java Collections Framework is one of Java's most powerful and convenient tools. It provides objects that represent collections of objects: lists, sets, and maps. The in terfaces and implementations that comprise the library are all to be found in the java.util package.

There are a few legacy classes in java.util that are historic relics and are not truly part of the framework. It's best to remember and avoid them. They are Vector, Hashtable, Enumeration, and Dictionary.

Collection interface types

Each of the five main types of object in the Collections Library is represented by an interface:

Collection

This is the root type for all of the objects in the Collection Library. A Collection is a group of objects, not necessarily ordered, not necessarily addressable, possibly containing duplicates. You can add and remove things from it, get its size, and iterate over it (more on iteration in a moment).

List

A List is an ordered collection. There is a mapping between the integers 0 and length–1 and the objects in the list. A List may contain duplicates. You can do anything to a List that you can do to a Collection. In addition, though, you can map an element to its index and an index to an element with the get and indexOf methods. You can also change the element at a specific index with the add(index, e) method. The iterator for a List returns the elements in order.

Set

A Set is an unordered collection that does not contain duplicates. You can do anything to a Set that you can do to a Collection. Attempting to add an element to a Set that already contains it, though, does not change the size of the Set.

Map

A Map is like a list except that instead of mapping integers to objects it maps a set of key objects to a collection of value objects. You can add and remove key-value pairs from the Map, get its size, and iterate over it, just like any other collection. Examples of maps might include mapping words to their definitions, dates to events, or URLs to cached content.

Iterator

An Iterator returns the elements of the collection from which it is derived, each exactly once, in response to calls to its next method. It is the preferred means for processing all the elements of a collection. Instead of:


```
for (int i = 0; i < list.size(); i++) {
 String s = list.get(i)
 // ...
}
```

the following is preferred:

```
for (Iterator<String> i = list.iterator(); i.hasNext();) {
 String s = i.next();
 // ...
}
```

In fact, the latter may be abbreviated, simply, as:

```
for (String s: list) {
 // ...
}
```

Collection implementation types

These interface types have multiple implementations, each appropriate to its own use case. Among the most common of these are the following:

ArrayList

An ArrayList is a list that is backed by an array. It is quick to index but slow to change size.

LinkedList

A LinkedList is a list that can change size quickly but is slower to index.

HashSet

A HashSet is a set that is implemented as a hash. add, remove, contains, and size all execute in constant time, assuming a well-behaved hash. A HashSet may contain (no more than one) null.

HashMap

A HashMap is an implementation of the Map interface that uses a hash table as its index. add, remove, contains, and size all execute in constant time, assuming a well behaved hash. It may contain a (single) null key, but any number of values may be null.

TreeMap

A TreeMap is an ordered Map: objects in the map are sorted according to their natural order if they implement the Comparable interface, or according

to a Comparator passed to the TreeMap constructor if they do not.

Idiomatic users of Java prefer to use declarations of interface types instead of declara tions of implementation types, whenever possible. This is a general rule, but it is easiest to understand here in the context of the collection framework.

Consider a method that returns a new list of strings that is just like the list of strings passed as its second parameter, but in which each element is prefixed with the string passed as the first parameter. It might look like this:

```
public ArrayList<String> prefixList(
 String prefix,
 ArrayList<String> strs)
{
 ArrayList<String> ret
 = new ArrayList<String>(strs.size());
 for (String s: strs) { ret.add(prefix + s); }
 return ret;
}
```

There's a problem with this implementation, though: it won't work on just any list! It will only work on an ArrayList. If, at some point, the code that calls this method needs to be changed from using an ArrayList to a LinkedList, it can no longer use the method. There's no good reason for that, at all.

A better implementation might look like this:

```
public List<String> prefix(
 String prefix,
 List<String> strs)
{
 List<String> ret = new ArrayList<String>(strs.size());
 for (String s: strs) { ret.add(prefix + s); }
 return ret;
}
```

This version is more adaptable because it doesn't bind the method to a particular implementation of the list. The method depends only on the fact that the parameter implements a certain interface. It doesn't care how. By using the interface type as a parameter it requires exactly what it needs to do its job—no more, no less.

In fact, this could probably be further improved if its parameter and return type were Collection.

Java generics

Generics in Java are a large and fairly complex topic. Entire books have been

written on the subject. This section introduces them in their most common setting, the Col lections Library, but will not attempt to discuss them in detail.

Before the introduction of generics in Java, it wasn't possible to statically type the contents of a container. One frequently saw code that looked like this:

```
public List makeList() {
 // ...
}

public void useList(List l) {
 Thing t = (Thing) l.get(0);
 // ...
```



```
}

// ...
useList(makeList());
```

The problem is obvious: useList has no guarantee that makeList created a list of Thing. The compiler cannot verify that the cast in useList will work, and the code might explode at runtime.

Generics solve this problem—at the cost of some significant complexity. The syntax for a generic declaration was introduced, without comment, previously. Here's a ver sion of the example, with the generics added:

```
public List<Thing> makeList() {
 // ...
}

public void useList(List<Thing> l) {
 Thing t = l.get(0);
 // ...
}

// ...
useList(makeList());
```

The type of the objects in a container is specified in the angle brackets (<>) that are part of the container type. Notice that the cast is no longer necessary in useList because the compiler can now tell that the parameter l is a list of Thing.

Generic type descriptions can get pretty verbose. Declarations like this are not uncommon:

```
Map<UUID, Map<String, Thing>> cache
 = new HashMap<UUID, Map<String, Thing>>();
```

Garbage Collection

Java is a garbage-collected language. That means your code does not manage memory. Instead, your code creates new objects, allocating memory, and then simply stops using those objects when it no longer needs them. The Dalvik

runtime will delete them and compress memory, as appropriate.

In the not-so-distant past, developers had to worry about long and unpredictable pe riods of unresponsiveness in their applications when the garbage collector suspended all application processing to recover memory. Many developers, both those that used Java in its early days and those that used J2ME more recently, will remember the tricks, hacks, and unwritten rules necessary to avoid the long pauses and memory fragmen tation caused by early garbage collectors. Garbage collection technology has come a long way since those days. Dalvik emphatically does not have these problems. Creating new objects has essentially no overhead. Only the most demandingly responsive of UIs—perhaps some games—will ever need to worry about garbage collection pauses.

Scope

Scope determines where variables, methods, and other symbols are visible in a program. Outside of a symbol's scope, the symbol is not visible at all and cannot be used. We'll go over the major aspects of scope in this section, starting with the highest level.

Java Packages

Java packages provide a mechanism for grouping related types together in a universally unique namespace. Such grouping prevents identifiers within the package namespace from colliding with those created and used by other developers in other namespaces.

A typical Java program is made up of code from a forest of packages. The standard Java Runtime Environment supplies packages like java.lang and java.util. In addition, the program may depend on other common libraries like those in the org.apache tree. By convention, application code—code you create—goes into a package whose name is created by reversing your domain name and appending the name of the program. Thus, if your domain name is androidhero.com, the root of your package tree will be com.androidhero and you will put your code into packages like com.androidhero.awesomeprogram and com.androidhero.geohottness.service. A typical package layout for an Android application might have a package for persistence, a package for the UI, and a package for application logic or controller code.

In addition to providing a unique namespace, packages have implications on member (field and method) visibility for objects in the same package. Classes in the same pack age may be able to see each other's internals in ways that are not available to classes outside the package. We'll return to this topic in a moment.

To declare a class as part of a package, use the package keyword at the top of

the file containing your class definition:

```
package your.qualifieddomainname.functionalgrouping
```

Don't be tempted to shortcut your package name! As surely as a quick, temporary implementation lasts for years, so the choice of a package name that is not guaranteed unique will come back to haunt you.

Some larger projects use completely different top-level domains to separate public API packages from the packages that implement those APIs. For example, the Android API uses the top-level package, android, and implementation classes generally reside in the package, com.android. Sun's Java source code follows a similar scheme. Public APIs reside in the java package, but the implementation code resides in the package sun. In either case, an application that imports an implementation package is clearly doing something fast and loose, depending on something that is not part of the public API.

While it is possible to add code to existing packages, it is usually considered bad form to do so. In general, in addition to being a namespace, a package is usually a single source tree, at least up as far as the reversed domain name. It is only convention, but

Java developers usually expect that when they look at the source for the package com.brashandroid.coolapp.ui, they will see all the source for the UI for CoolApp. Most will be surprised if they have to find another source tree somewhere with, for instance, page two of the UI.

The Android application framework also has the concept of a Package. It is different, and we'll consider it in Chapter 3. Don't confuse it with Java package names.

For more information on Java packages, see the Java tutorial at *http://download.oracle .com/javase/tutorial/java/package/packages.html*.

Access Modifiers and Encapsulation

We hinted earlier that members of a class have special visibility rules. Definitions in most Java blocks are lexically scoped: they are visible only within the block and its nested blocks. The definitions in a class, however, may be visible outside the block. Java supports publishing top-level members of a class—its methods and fields—to code in other classes, through the use of *access modifiers*. Access modifiers are keywords that modify the visibility of the declarations to which they are applied.

There are three access-modifying keywords in the Java language: public, protected, and private. Together they support four levels of access. While access

modifiers affect the visibility of a declaration from outside the class containing it, within the class, normal block scoping rules apply, regardless of access modification.

The private access modifier is the most restrictive. A declaration with private access is not visible outside the block that contains it. This is the safest kind of declaration because it guarantees that there are no references to the declaration, except within the containing class. The more private declarations there are in a class, the safer the class is.

The next most restrictive level of access is default or package access. Declarations that are not modified by any of the three access modifiers have default access and are visible only from other classes in the same package. Default access can be a very handy way to create state shared between objects, similar to the use of the friend declaration in C++.

The protected access modifier permits all the access rights that were permitted by de fault access but, in addition, allows access from within any subtype. Any class that extends a class with protected declarations has access to those declarations.

Finally, public access, the weakest of the modifiers, allows access from

anywhere.

Here's an example that will make this more concrete. There are four classes in two different packages here, all of which refer to fields declared in one of the classes, Accessible:

```
package over.here;

public class Accessible {
 private String localAccess;
 String packageAccess;
 protected String subtypeAccess;
 public String allAccess;

 public void test() {
 // all of the assignments below work:
 // the fields are declared in an enclosing
 // block and are therefore visible.
 localAccess = "success!!";
 packageAccess = "success!!";
 subtypeAccess = "success!!";
 allAccess = "success!!";
 }
```

```
}


package over.here;
import over.here.Accessible;

// this class is in the same package as Accessible
public class AccessibleFriend {

 public void test() {
 Accessible target = new Accessible();

 // private members are not visible
 // outside the declaring class
 target.localAccess = "fail!!"; // ERROR!!

 // default access visible within package
 target.packageAccess = "success!!";

 // protected access is superset of default
 target.subtypeAccess = "success!!";

 // visible everywhere
 target.allAccess = "success!!";
 }
}


package over.there;
import over.here.Accessible;

// a subtype of Accessible
// in a different package
public class AccessibleChild extends Accessible {
```



```
 // the visible fields from Accessible appear
 // as if declared in a surrounding block
 public void test() {
 localAccess = "fail!!"; // ERROR!!
 packageAccess = "fail!!"; // ERROR!!

 // protected declarations are
 // visible from subtypes
 subtypeAccess = "success!!";

 // visible everywhere
 allAccess = "success!!";
 }
}


package over.there;
import over.here.Accessible;

// a class completely unrelated to Accessible
```

```
public class AccessibleStranger {

 public void test() {
 Accessible target = new Accessible();
 target.localAccess = "fail!!"; // ERROR!!
 target.packageAccess = "fail!!"; // ERROR!!
 target.subtypeAccess = "success!!"; // ERROR!!

 // visible everywhere
 target.allAccess = "success!!";
 }
}
```

Idioms of Java Programming

Somewhere between getting the specifics of a programming language syntax right and good pattern-oriented design (which is language-agnostic), is idiomatic use of a lan guage. An idiomatic programmer uses consistent code to express similar ideas and, by doing so, produces programs that are easy to understand, make optimal use of the runtime environment, and avoid the “gotchas” that exist in any language syntax.

Type Safety in Java

A primary design goal for the Java language was programming safety. Much of the frequently maligned verbosity and inflexibility of Java, which is not present in languages such as Ruby, Python, and Objective-C, is there to make sure a compiler can guarantee that entire classes of errors will never occur at runtime.

Java’s static typing has proven to be valuable well beyond its own compiler. The ability for a machine to parse and recognize the semantics of Java code was a major force in the development of powerful tools like FindBugs and IDE refactoring tools.

Many developers argue that, especially with modern coding tools, these constraints are a small price to pay for being able to find problems immediately that might otherwise manifest themselves only when the code is actually deployed. Of course, there is also a huge community of developers who argue that they save so much time coding in a dynamic language that they can write extensive unit and integration tests and still come out ahead.

Whatever your position in this discussion, it makes a lot of sense to make the best possible use of your tools. Java’s static binding absolutely is a constraint. On the other hand, Java is a pretty good statically bound language. It is a lousy dynamic language. It is actually possible to do fairly dynamic things with Java by using its reflection and introspection APIs and doing a lot of type casting.

Doing so, except in very limited circumstances, is using the language and its runtime environment to cross-purposes. Your program is likely to run very slowly, and the Android tool chain won't be able to make heads or tails of it. Perhaps most important, if there are bugs in this seldom-used part of the platform, you'll be the first to find them. We suggest embracing Java's static nature—at least until there is a good, dynamic alternative—and taking every possible advantage of it.

Encapsulation

Developers limit the visibility of object members in order to create *encapsulation*. En capsulation is the idea that an object should never reveal details about itself that it does not intend to support. To return to the mojito-making example, recall that, when it comes time to make the cocktail, you don't care at all how your colleague got the necessary mint. Suppose, though, that you had said to her, "Can you get the mint? And, oh, by the way, while you are out there, could you water the rosebush?" It is no longer true that you don't care how your colleague produces mint. You now depend on the exact way that she does it.

In the same way, the interface (sometimes abbreviated as API) of an object consists of the methods and types that are accessible from calling code. By careful encapsulation, a developer keeps implementation details of an object hidden from code that uses it. Such control and protection produce programs that are more flexible and allow the developer of an object to change object implementation over time without causing ripple-effect changes in calling code.

Getters and setters

A simple, but common, form of encapsulation in Java involves the use of getter and setter methods. Consider a naive definition of a Contact class:


```
public class Contact {
 public String name;
 public int age;
 public String email;
}
```

This definition makes it necessary for external objects to access the fields of the class directly. For example:

```
Contact c = new Contact();
c.name = "Alice";
c.age = 13;
c.email = "alice@mymail.com";
```

It will take only a tiny amount of use in the real world to discover that contacts actually have several email addresses. Unfortunately, adding a multiple-address feature to the naive implementation requires updating every

single reference to Contact.email, in the entire program.

In contrast, consider the following class:

```
class Contact {
 private int age;
 private String name;
 private String email;

 Contact(int age, String name, String email) {
 this.age = age;
 this.name = name;
 this.email = email;
 }

 public int getAge() {
 return age;
 }

 public String getName() {
 return name;
 }

 public String getEmail() {
 return address;
 }
}
```

Use of the private access modifier prevents direct access to the fields of this version of the Contact class. Use of public getter methods provides the developer with the op portunity to change how the Contact object returns the name, age, or email address of the Contact. For example, the email address could be stored by itself, as in the preceding code or concatenated from a username and a hostname if that happened to be more convenient for a given application. Internally, the age could be held as an int or as an Integer. The class can be extended to support multiple email addresses without any change to any client.

Java does allow direct reference to fields and does not, like some languages, automat ically wrap references to the fields in getters and setters. In order to preserve encapsu lation, you must define each and every access method yourself. Most IDEs provide code generation features that will do this quickly and accurately.

Wrapper getter and setter methods provide future flexibility, whereas direct field access means that all code that uses a field will have to change if the type of that field changes, or if it goes away. Getter and setter methods represent a simple form of object encap sulation. An excellent rule of thumb recommends that all fields be either private or final. Well-written Java programs use this and other, more sophisticated forms of encapsulation in order to preserve adaptability in more complex programs.

Using Anonymous Classes

Developers who have experience working with UI development will be familiar with the concept of a callback: your code needs to be notified when something in the UI changes. Perhaps a button is pushed and your model needs to make a corresponding change in state. Perhaps new data has arrived from the network and it needs to be displayed. You need a way to add a block of code to a framework, for later execution on your behalf.

Although the Java language does provide an idiom for passing blocks of code, it is slightly awkward because neither code blocks nor methods are first-class objects in the language. There is no way, in the language, to obtain a reference to either.

You can have a reference to an instance of a class. In Java, instead of passing blocks or functions, you pass an entire class that defines the code you need as one of its methods. A service that provides a callback API will define its protocol using an interface. The service client defines an implementation of this interface and passes it to the framework.

Consider, for instance, the Android mechanism for implementing the response to a user keypress. The Android `View` class defines an interface, `OnKeyListener`, which, in turn, defines an `onKey` method. If your code passes an implementation of `OnKeyListener` to a `View`, its `onKey` method will be called each time the `View` processes a new key event.

The code might look something like this:

```
public class MyDataModel {
// Callback class
private class KeyHandler implements View.OnKeyListener {
                public boolean onKey(View v, int keyCode, KeyEvent event) {
handleKey(v, keyCode, event)
}
}

/** @param view the view we model */
public MyDataModel(View view) { view.setOnKeyListener(new KeyHandler()) }
```